16th Euromicro Conference on Parallel, Distributed and Network-Based Processing

CellStats: a Tool to Evaluate the Basic Synchronization and Communication Operations of the Cell BE

José L. Abellán Juan Fernández Manuel E. Acacio
Dept. de Ingeniería y Tecnología de Computadores
Facultad de Informática - Universidad de Murcia
30100 Murcia, Spain
{jl.abellan,juanf,meacacio}@ditec.um.es

Abstract

The Cell Broadband Engine (Cell BE) is a recent heterogeneous chip-multiprocessor (CMP) architecture jointly developed by IBM, Sony and Toshiba to offer very high performance, especially on game and multimedia applications. The significant number of processor cores that it contains (nine in its first generation), along with their heterogeneity (they are of two different types) and the variety of synchronization and communication primitives offered to programmers, make the task of developing efficient applications for the Cell BE very challenging. In this work, we present CellStats, a tool aimed at characterizing the performance of the main synchronization and communication primitives provided by the Cell BE under varying workloads. In particular, the current implementation of CellStats allows to evaluate the DMA transfer mechanism, the read-modify-write atomic operations, the mailboxes, the signals and the time taken by thread creation. As an example of application of CellStats, we present a characterization of the Cell BE incorporated into the PlayStation 3. From this characterization, we extract some recommendations that can help programmers to identify the most appropriate primitive under different assumptions.

1 Introduction

As the number of transistors in a single chip increases, following the well-known Moore's Law, it becomes harder and harder to translate such increased potential into effective computational power through the exploitation of just the instruction-level parallelism (ILP) that is present in applications [1]. This tendency, that had been followed until very recently by most commercial microprocessor developers, has been replaced lately by other that tries to exploit coarser grained parallelism (besides ILP), in particular thread-level parallelism (TLP). This new class of architecture has been named as CMP (or chip-multiprocessor) and integrates several processor cores in a single chip, allowing that as many threads as cores can be executed in parallel in a particular instant. Although in most cases each of the processor cores in chip-multiprocessors are lower frequency and simpler than their contemporary single-core, overall they improve performance and are more energy efficient.

Nowadays, CMP architectures are omnipresent and can be found in all market segments. In particular, they constitute the CPU of many embedded systems (for example, the last generation video game consoles), personal computers (for example, the latest developments from Intel and AMD), servers (the IBM Power5 or Sun UltraSPARC T1 among others) and even supercomputers (for example, the CPU chips used as building blocks in the IBM BlueGene/L). However, among all contemporary CMP architectures, there is one that is currently concentrating an enormous attention due to its architectural particularities and tremendous potential in terms of sustained performance: the Cell Broadband Engine (Cell BE from now on). The Cell BE is the result of a collaborative effort between IBM, Sony and Toshiba to develop a new microprocessor able to offer very high performance, especially on game and multimedia applications. In fact, it is the heart of the Sony PlayStation 3.

From the architectural point of view, the Cell BE can be classified as a heterogeneous CMP. In particular, the first generation of the chip integrates up to nine cores of two distinct types [8]. One of the cores, known as the *Power Processor Element* or PPE, is a 64-bit multithreaded Power-Architecture-compliant processor with two levels of on-chip cache that includes the vector multimedia extension (VMX) instructions. The main role of the PPE is to coordinate and supervise the tasks performed by the rest of cores. The remaining cores (a maximum of eight) are called *Syner-*

0-7695-3089-3/08 $25.00 © 2008 IEEE
DOI 10.1109/PDP.2008.49

gistic Processing Elements and provide the main computing power of the Cell BE. Each SPE is a RISC processor especially designed to accelerate media and streaming workloads that implements a new 128-bit SIMD instruction set. Each SPE includes a local memory for keeping instructions and data that is not coherent with the PPE main memory. Data transfers to and from the SPE local memories must be explicitly managed by using a DMA engine. Finally, all these cores are interconnected with memory using the *Element Interconnect Bus* or EIB.

The Cell BE provides programmers with a variety of primitives to manage communication and synchronization between the threads that comprise parallel applications. At the end, the performance achieved by the applications running on the Cell BE will depend in great extent on the ability of the programmer to select the most adequate primitives as well as their corresponding configuration values.

In this work, we present *CellStats*, a tool aimed at characterizing the performance of the main synchronization and communication primitives provided by the Cell BE under varying workloads. In particular, the current implementation of CellStats allows to evaluate DMA transfers (GETs and PUTs), the read-modify-write atomic operations, the mailboxes, the signals and the time taken by thread creation. For the DMA transfers we consider both transfers between main memory and an SPE's LS, and between LSs. Similarly, for the signals and the mailboxes we distinguish the case in which the PPE and the SPEs are involved, and that in which two SPEs are the participants. Finally, for the read-modify-write atomic operations we consider that the remote memory locations always reside in main memory. Due to hardware constraints, it is not possible to perform these atomic operations on a memory location residing in a remote LS. All the primitives can be evaluated for different number of intervening SPEs, and when applicable, with varying memory sizes and address ranges. As an example of application of CellStats we present a characterization of the Cell BE that acts as CPU in the PlayStation 3. From this characterization, we extract some recommendations that can help programmers to identify the most appropriate primitive in different situations.

The rest of the paper is organized as follows. In Section 2 we provide a short revision of the architecture of the Cell BE and a description of some of the communication and synchronization primitives that it provides to programmers. Next, in Section 3 we introduce the details of CellStats. The results obtained after executing CellStats on the Cell BE included in the PlayStation 3 are presented in Section 4. Finally, Section 5 gives the main conclusions of the paper and some of the lessons learned.

2 Cell BE

2.1 Architecture

The Cell Broadband Engine (Cell BE) [8] is an heterogeneous multi-core chip composed of one general-purpose processor, called *PowerPC Processor Element* (PPE), eight specialized co-processors, called *Synergistic Processing Elements* (SPEs), a high-speed memory interface controller, and an I/O interface, all integrated in a single chip. All these elements communicate through an internal high-speed *Element Interconnect Bus* (EIB) (see Figure 1(a)).

The latest version of the Cell BE processor, running at 3.2 GHz, has a theoretical peak performance of 204.8 Gflops/second (single precision) and 14.63 Gflops/second (double precision). The EIB supports a peak bandwidth of 204.8 Gigabytes/second for intra-chip data transfers among the PPE, the SPEs, and the memory and the I/O interface controllers. The memory interface controller (MIC) provides a peak bandwidth of 25.6 GBytes/second to main memory. The I/O controller provides peak bandwidths of 25 GBytes/second inbound and 35 GBytes/second outbound.

The PPE is the main processor of the Cell BE, and is responsible for running the operating system and coordinating the SPEs. It is a traditional 64-bit PowerPC (PPC) processor core with a VMX unit, a 32 KB Ll instruction cache, a 32 KB L1 data cache, and a 512 KB L2 cache. The PPE is a dual issue, in-order execution, 2-way SMT processor. The PPE comprehends two different units, namely *PowerPC Processor Unit* (PPU) and *PowerPC Processor Storage Subsystem* (PPSS) (see Figure 1(b)).

Each SPE is a 128-bit RISC processor specifically designed for high-performance on streaming and data-intensive applications [3]. Each SPE consists of a *Synergistic Processing Unit* (SPU) and a *Memory Flow Controller* (MFC) (see Figure 1(c)). SPUs are a in-order processors with two pipelines and 128 128-bit registers. All SPU instructions are inherently SIMD operations that the proper pipeline can run at four different granularities: 16-way 8-bit integers, 8-way 16-bit integers, 4-way 32-bit integers or single-precision floating-point numbers, or 2-way 64-bit double-precision floating-point numbers. SPEs do not have a private cache memory. In contrast, SPUs include a 256 KB *Local Store* (LS) memory to hold both instructions and data of SPU programs, that is, SPUs cannot access main memory directly. The MFC contains a *DMA Controller* and a set of memory-mapped registers or *MMIO Registers*. Each SPU can write its MMIO registers though several *Channel Commands*. The DMA controller supports DMA transfers among the LSs and main memory. These operations can be issued by the owner SPE, which accesses the MFC through the channel commands, or the other SPEs (or even the PPE), which access the MFC through the MMIO registers.

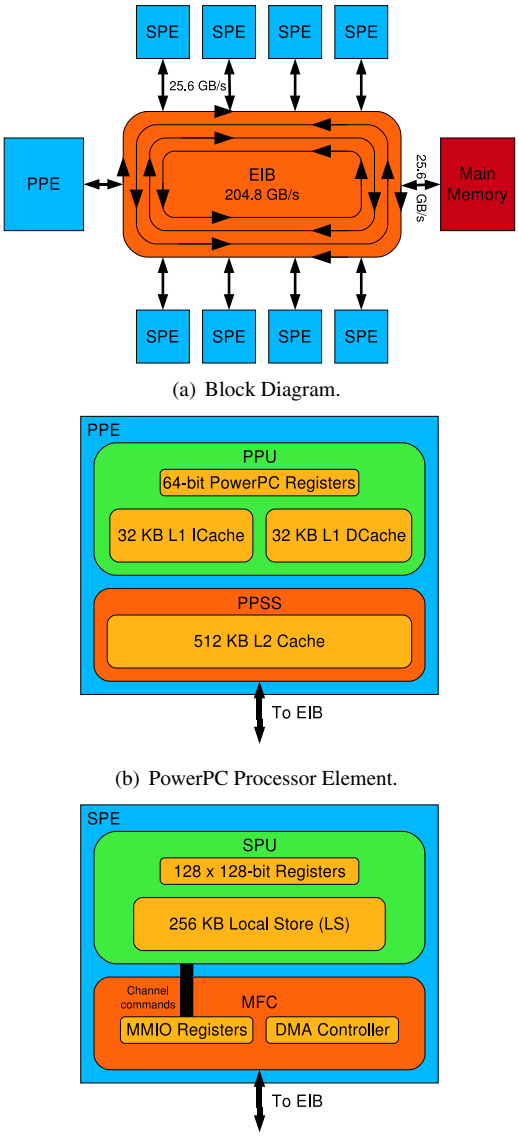


Figure 1. Cell BE Architecture.

2.2 Programming

The Cell BE has been specifically designed to exploit multiple levels of parallelism at the same time: (a) each SPE executes a different thread, (b) an SPE can overlap computation and communication by using non-blocking DMA operations, (c) SIMD instructions perform the very same operation on multiple data simultaneously, and (d) SPEs have two pipelines that can execute two instructions concurrently. Nevertheless, the main advantage also becomes the major drawback: Cell BE programming is as flexible as complex. Flexibility stems from the possibility to use a number of programming models depending on the application domain. Complexity is due to the fact that threads must communicate and synchronize across program execution. To do that, the PPE and the SPEs can use a variety of mechanisms provided by the Cell BE architecture: *DMA Transfers*, *Mailboxes*, *Signals* and *Atomic Operations*.

SPEs use DMA transfers to read from (GET) or write to (PUT) main memory. DMA transfer size must be 1, 2, 4, 8 or a multiple of 16 Bytes up to a maximum of 16 KB. DMA transfers can be either blocking or non-blocking. The latter allow to overlap computation and communication: there might be up to 128 simultaneous transfers between the eight SPE LSs and main memory. In addition, an SPE can issue a single command to perform a list of up to 2048 DMA transfers, each one up to 16 KB in size. In all cases, peak performance can be achieved when both the source and destination addresses are 128-Byte aligned and the size of the transfer is an even multiple of 128 Bytes [9].

Mailboxes are FIFO queues that support exchange of 32-bit messages among the SPEs and the PPE. Each SPE includes two outbound mailboxes, called *SPU Write Outbound Mailbox* and *SPU Write Outbound Interrupt Mailbox*, to send messages from the SPE; and a 4-entry inbound mailbox, called *SPU Read Inbound Mailbox*, to receive messages. Every mailbox is assigned a channel command and a MMIO register. The former allows the owner SPE to access the outbound mailboxes. The latter enables remote SPEs and the PPE to access the inbound mailbox.

In contrast, signals were designed with the only purpose of sending notifications to the SPEs. Each SPE has two 32-bit signal registers to collect incoming notifications. A signal register is assigned a MMIO register to enable remote SPEs and the PPE to send individual signals (*overwrite mode*) or combined signals (*OR mode*) to the owner SPE.

Read-modify-write atomic operations enable simple transactions on single words residing in main memory. For example, the *atomic_add_return* atomic operation adds a 32-bit integer to a word in main memory and returns its value before the addition.

Cell BE programming requires separate programs, written in C/C++, for the PPE and the SPEs, respectively. The PPE program can include extensions (e.g., `vec_add`), to use its VMX unit; and library function calls [7], to manage threads and perform communication and synchronization operations (e.g., `spe_create_thread`, `spe_write_in_mbox` and `spe_mfc_put`). The SPE program follows an SPMD model (*Single Program Multiple Data*). It includes extensions [4], to execute SIMD instructions, and communication and synchronization operations (e.g., `spu_add`, `spu_read_in_mbox` and `mfc_get`); and function calls to the SDK library [5], to carry out complex tasks of different nature (matrix, FFT, filters, etc.).

3 CellStats

3.1 Architecture

CellStats is a command-line tool which admits a number of parameters such as the operation to evaluate, the number of SPEs, the number of iterations and other operation-specific parameters. However, the process to launch, instruct and synchronize the threads is the same in all cases. First, the PPE marshals an structure called control block. The control block contains all the information needed by each SPE to complete the operation demanded by the user. Next, the PPE creates as many threads as specified by the user and synchronizes them using mailboxes. In turn, SPEs transfer the control block from main memory to their private LSs, report control block transfer completion to the PPE, and wait for PPE's approval to resume execution. Then, each SPE performs the task entrusted by the user in a loop. In order to measure the time to complete the loop, the SPE utilizes a register called *SPU_Decrementer* which decrements at regular intervals or *ticks*[1]. Upon completion of the loop, the SPE sends to the PPE the number of elapsed ticks through its outgoing mailbox. In this way, the PPE can compute not only the elapsed time from the go-ahead indication given to the SPEs, but also the time taken by each individual SPE to complete the task.

3.2 Functionality

CellStats performs a different task depending on the `-f FUNCTION` parameter specified by the user. `FUNCTION` can take the following values: `mkthread` for thread creation; `mailbox` or `signal` for PPE-to-SPE or SPE-to-SPE synchronization using mailboxes or signal, respectively; `dmaget/dmaput` for main memory to local LS/local LS to main memory or remote LS to local LS/local LS to remote LS data transfers through DMA operations; `dmalistget/dmalistput` for data transfers using lists of DMA operations; and, finally, `fetchadd/fetchsub` for adding/subtracting an integer to a remote memory word, `fetchinc/fetchdec` for increasing/decreasing a remote memory word, and `fetchset` for setting a remote memory word to a certain value[2]. Other common parameters are `-n NSPE` and `-i ITER`, that specify the number of involved SPEs, and the number of iterations for the loop (all results shown in this paper are the arithmetic mean of one million repetitions with a warm-up of one thousand iterations), respectively.[3] Next, we take a closer look to these operations emphasizing their particular parameters.

[1] Duration of every *tick* for the PlayStation 3 is 12.53 ns.
[2] Remote memory locations refer to main memory.
[3] Numeric parameters admit the use of multipliers `k` and `m`.

3.2.1 Thread creation

This operation measures the time to launch the threads that are executed by the SPEs. To do that, an empty task that returns immediately is used. Consequently, this operation measures not only the time to create the thread but also the time needed to detect its finalization. For example,

```
$ CellStats -f mkthread -n 1 -i 100k
```

creates a *thread* in an SPE one hundred thousand times and prints **1.68 ms** that is the average latency for such a task.

3.2.2 Mailboxes

This operation performs a PPE-to-SPE or an SPE-to-SPE synchronization using mailboxes. The PPE writes a message in the incoming mailbox (*SPU Read Inbound Mailbox*) of the receiver SPE. Next, the receiver SPE reads the message and replies with another message written to its outgoing mailbox (*SPU Write Outbound Mailbox*). When the initiator SPE/PPE reads the message, the synchronization process is complete. For example,

```
$ CellStats -f mailbox -n 1 -i 1m
```

repeats the PPE-to-SPE synchronization cycle between the PPE and SPE 0 (default value) one million times and prints **6 μs** that is the average round-trip latency for such an operation. Meanwhile,

```
$ CellStats -f mailbox -n 2 -i 1m -s
```

does the same between SPE 0 and SPE 1 (default values) and prints **160 ns**. In the latter case, the `-s` parameter specifies that the operations will be realized between as many SPEs as stated by the `-n` parameter. In the former case, the PPE uses the runtime management library function `spe_in_mbox_write` [7] involving a system call which explains the increased latency. Nevertheless, the PPE can also write directly into the corresponding SPE's MMIO register using a regular assignment. In this case, CellStats would print **362 ns** rather than 6 μs.

3.2.3 Signals

Unlike mailboxes, this operation performs a PPE-to-SPE or an SPE-to-SPE synchronization using signals. The initiator SPE/PPE signals the destination SPE by writing to the corresponding MMIO register (*SPU Signal Notification*). If the initiator is an SPE, the destination SPE signals in turn the source SPE, thus finishing the synchronization cycle. Otherwise, the destination SPE sends the reply to the PPE using its outgoing mailbox (*SPU Write Outbound Mailbox*). For example,

```
$ CellStats -f signal -n 2 -i 1m -s
```

repeats the SPE-to-SPE synchronization cycle between SPE 0 and SPE 1 (default values) one million times and prints **160 ns** (round-trip) that is the average latency for such an operation. In the meantime,

```
$ CellStats -f signal -n 1 -i 1m
```

does the same between the PPE and the SPE 0 (default value) and prints **28** μ**s** (round-trip). Again, writing directly into the SPE's MMIO register instead of using the runtime management library function call `spe_signal_write` [7] results in a reduced latency of **290 ns** rather than 28 μs.

3.2.4 Atomic operations

These operations enable sequences of read-modify-write instructions on main memory locations in an atomic fashion. They admit, other than the parameters already described in Section 3.2, the `-N SPEs` and `-t STRIDE` parameters. In both cases the SPEs would write to main memory. However, the memory location accessed would be shared or private with `-n` and `-N`, respectively. In the latter case, the `-t STRIDE` parameter specifies the distance, measured in Bytes, between two consecutive private variables. For example,

```
$ CellStats -f fetchadd -N 6 -t 2k -i 1m
```

creates six threads that execute one million `fetchadd` atomic operations, each over six different 32-bit main memory variables separated by 2048 memory locations, and prints **135 ns** that is the average latency for such an operation.

3.2.5 DMA operations

Data transfers between main memory and the local LS, and between a remote LS and the local LS, are achieved through DMA operations. The `-b SIZE` parameter specifies the DMA size, the `-n` and `-N` parameters indicate whether the source buffer (GETs) or the destination buffer (PUTs) is shared or private, and the `-s` parameter indicates whether the memory location is in main memory (if this parameter is omitted) or in SPE 0's LS (default value). The `-t STRIDE` parameter specifies the distance, measured in Bytes, between two consecutive private buffers. For example,

```
$ CellStats -f dmaget -n 6 -b 1k -i 1m
```

creates six threads that execute one million 1 KB GETs from a shared main memory buffer to their local LSs, while

```
$ CellStats -f dmaput -N 6 -b 2k -i 100k
  -t 512
```

creates six threads that execute one hundred thousand 2 KB PUTs to a private main memory buffers separated by 512 memory locations.

In addition, for lists of DMA operations, the `-l` parameter sets the number of elements of the list. For example,

```
$ CellStats -f dmalistget -N 6 -l 16
  -b 64 -t 256 -i 1m
```

creates six threads that execute one million data transfers from six private memory buffers, residing on SPE 0's LS (default value) and separated by 256 memory locations, to their private LSs. Each data transfer would consist of 16 DMA operations of 64 Bytes each.

4 Evaluation

In Section 3 we presented results in terms of average latency for some of the operations that the Cell BE offers to programmers (thread creation, mailboxes, signals and atomic operations). Now in this Section, we carry out a deeper performance analysis of the atomic operations, DMA operations and lists of DMA operations under varying workloads. At the light of the results reported in this work we finalize with some recommendations for Cell BE programmers on the PlayStation 3 (PS3).

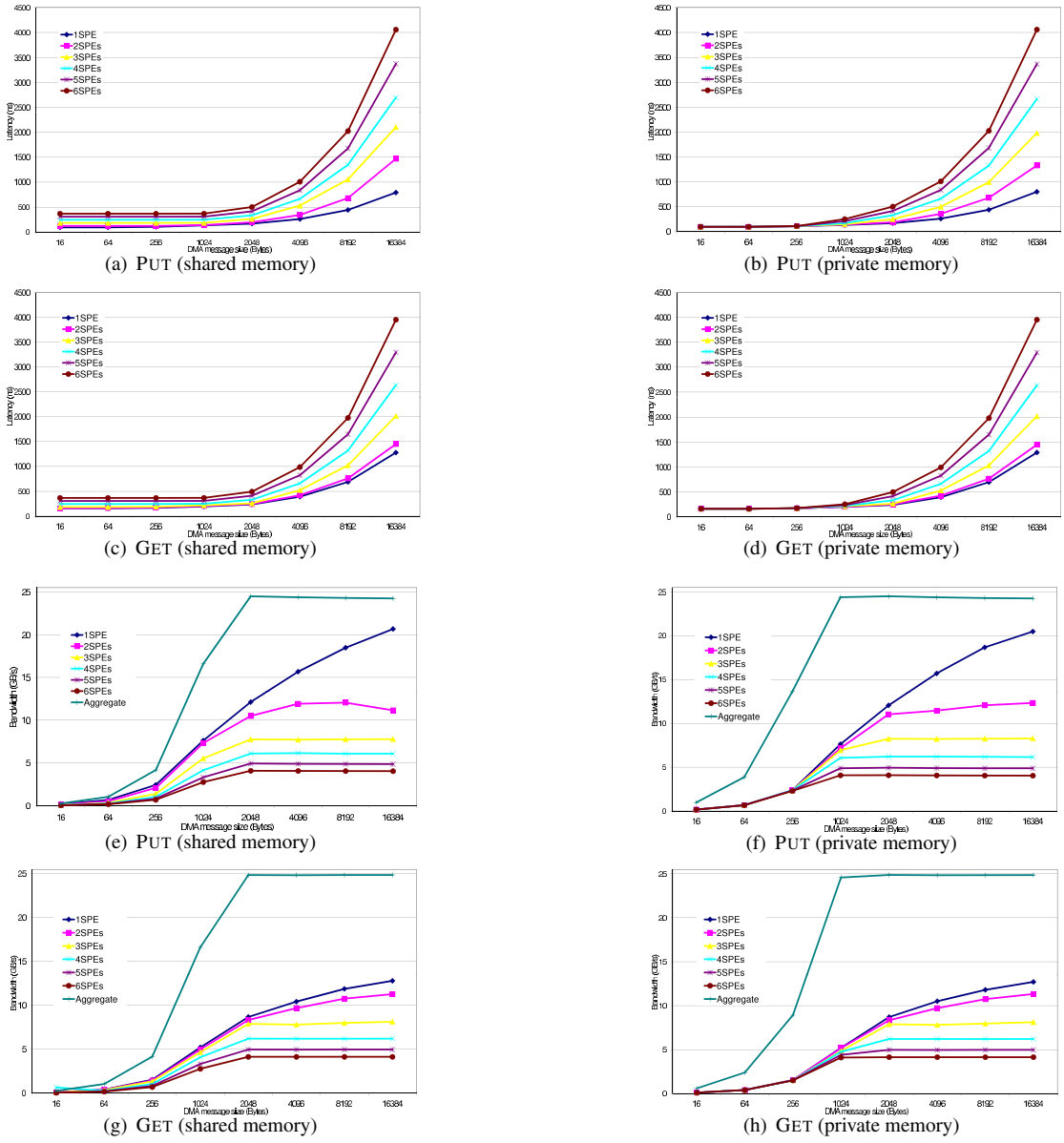
4.1 Testbed

To develop CellStats we used the IBM SDK v2.1 for the Cell BE architecture installed on top of Fedora Core 7 on a regular PC [6]. This development kit includes a simulator, named Mambo [2], that allows programmers to execute binary files compiled for the Cell BE architecture. To obtain the experimental results, we installed Fedora Core 6 on a PlayStation 3. Our console incorporates a 3.2 GHz Cell BE Rev. 5.1. (only 6 SPEs are enabled) with 200 MB of main memory and a 160 GB hard disk.

4.2 Results

4.2.1 Atomic Operations

Latency of the `fetchadd` atomic operation for a variable is shown in Figure 3. As we can see, latency remains constant, at approximately 135 ns, when the variable is private, but grows linearly, up to 1 μs for 6 SPEs, when the variable is shared. This is because shared variables serialize the execution of atomic operations. Results for the rest of the atomic operations mentioned in Section 3.2 are similar and, therefore, have been omitted for the sake of brevity.



(a) PUT (shared memory)

(b) PUT (private memory)

(c) GET (shared memory)

(d) GET (private memory)

(e) PUT (shared memory)

(f) PUT (private memory)

(g) GET (shared memory)

(h) GET (private memory)

Figure 2. Latency and bandwidth of DMA operations on shared memory and private memory for a variable number of SPEs and packet sizes.

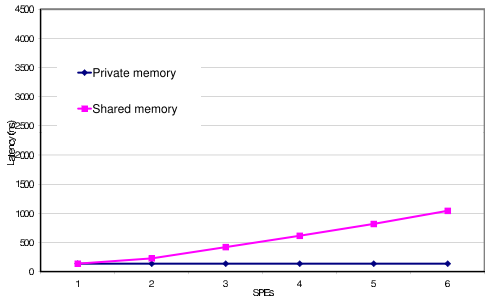


Figure 3. Latency of `fetchadd` on a shared variable and separate variables (128-Bytes stride).

4.2.2 DMA Operations

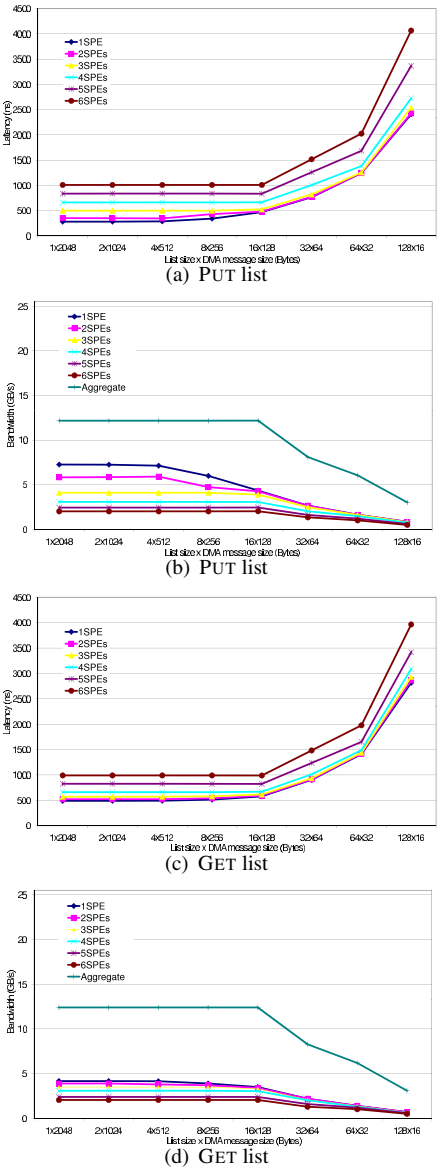
Latency and bandwidth of PUT and GET operations, using shared memory and private memory, are shown in Figure 2. For PUT operations, latency on private memory is smaller than latency on shared memory up to 1 KB messages because multiple memory accesses can take place in parallel. For messages larger than 1 KB, latency is almost identical in both cases. As we can see in Figures 2(e) and 2(f), the limiting factor is now the EIB bandwidth. GET operations follow the same trends that PUT operations. Nevertheless, their latency is slightly higher because PUT operations complete as soon as they go through the main memory interface. Additionally, results for data transfers between pairs of LSs show that the maximum achievable bandwidth is close to the theoretical limit and have been omitted for the sake of brevity.

4.2.3 Lists of DMA Operations

Finally, Figures 4(a), 4(b), 4(c) and 4(d) exemplify the use of CellStats to analyze a specific configuration. In particular, with the following command line:

```
$ CellStats -f dmalistget -n 6 -l NLIST
  -b ELEMSIZE -i 1m
```

where $NLIST = 2^i (i = 0..7)$ and $ELEMSIZE = 2^i (i = 4..11)$, and satisfy that $NLIST \times ELEMSIZE = 2KB$, we can see how all the configurations until 32×64 obtain the same results to access a shared memory buffer of 2 KB using lists of PUTs and GETs. As shown, for this particular configuration, the maximum aggregate bandwidth that can be achieved for lists of DMAs is half the bandwidth previously reported for single DMAs due to their higher latency. This latency increase is due to the overhead introduced by the list mechanism.



(a) PUT list
(b) PUT list
(c) GET list
(d) GET list

Figure 4. Latency and bandwidth for lists of DMA operations (2 KB packets) on a shared memory buffer for a variable number of SPEs.

4.3 Recommendations for Cell BE programmers on PS3

Thread creation introduces an overhead of 1.68 ms. Consequently, programmers should avoid frequent creation of threads specially when dealing with fine-grained applications.

Mailbox-based and signal-based implementations of PPE-to-SPE synchronization are very slow (in the order of microseconds) when based on the runtime management li-

brary calls. Programmers should either deprecate these library calls in favor of direct writes to the SPEs' MMIO registers, or use atomic operations, DMA operations or a combination of both instead (in the order of nanoseconds).

Mailbox or Signal-based implementation of SPE-to-SPE synchronization introduces a negligible overhead (around 160 ns). Therefore, programmers should convert serialized PPE-to-SPE synchronization cycles into signal-based SPE-to-SPE ones whenever possible.

For atomic operations, whenever possible, programmers should use private variables (residing on different cache memory lines) instead of shared ones in order to minimize latency (around 135 ns). Otherwise, programmers should keep in mind that Lat_{atomic} would be approximately 150 times `NSPE` ns.

DMA operations on private memory buffers incur smaller latency than DMA operations on shared memory buffers, where Lat_{shared} is approximately equal to $Lat_{private}$ plus 50 times `NSPE` ns. For latency-sensitive applications, message size can be increased up to 1 KB with no additional penalty for both private memory and shared memory PUTs and GETs. For memory-bounded applications, all the available bandwidth is exploited when messages larger than 1 KB or 2 KB are used for private memory and shared memory operations, respectively. In addition, if the SPE code requires access to non-contiguous main memory buffers, programmers should note that some configurations could provide better performance than others.

In general, CellStats may help programmers to evaluate latency and bandwidth of most common synchronization and communication building blocks for their parallel applications. In this way, they can assess *a priori* what is the most suitable implementation for a certain algorithm.

5 Conclusions

The large number of processor cores that the Cell BE contains (nine in its first generation), along with their heterogeneity (they are of two different types) and the variety of communication and synchronization primitives offered to programmers, make the task of developing efficient applications challenging. At the end, the performance achieved by the applications running on the Cell BE will depend in great extent on the ability of the programmer to select the most adequate communication and synchronization primitives as well as their corresponding configuration values.

In this work, we have presented *CellStats*, a tool aimed at characterizing the performance of the main synchronization and communication primitives provided by the Cell BE under varying workloads. In particular, the current implementation of CellStats allows to evaluate DMA transfers (GETs and PUTs), the read-modify-write atomic operations, the mailbox mechanism, the signals and the time taken by thread creation. For the DMA transfers we consider both transfers between main memory and an SPE's LS, and between LSs. Similarly, for the signals and mailboxes we distinguish the case in which the PPE and the SPEs are involved, and that in which two SPEs are the participants. All the primitives can be evaluated for different number of intervening SPEs, and when applicable, with varying memory sizes and address ranges. As an example of application of CellStats we have presented in Section 4 a characterization of the Cell BE that acts as CPU in the PlayStation 3. From this characterization, we have pointed out some recommendations that can help programmers to identify the most appropriate primitive in different situations.

Acknowledgments

This work has been jointly supported by the Spanish MEC and European Commission FEDER funds under grants "Consolider Ingenio-2010 CSD2006-00046" and "TIN2006-15516-C04-03".

References

[1] V. Agarwal, M. Hrishikesh, S. W. Keckler, and D. Burger. Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. In *Proceedings of 27th International Symposium on Computer Architecture*, Vancouver, British Columbia, Canada, June 2000.

[2] P. Bohrer, J. Peterson, M. Elnozahy, R. Rajamony, A. Gheith, R. Rockhold, C. Lefurgy, H. Shafi, T. Nakra, R. Simpson, E. Speight, K. Sudeep, E. V. Hensbergen, and L. Zhang. Mambo: a Full System Simulator for the PowerPC Architecture. *ACM SIGMETRICS Performance Evaluation Review*, 31(4):8–12, March 2005.

[3] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic Processing in Cell's Multicore Architecture. *IEEE Micro*, 26(2):10–24, March/April 2006.

[4] IBM Systems and Technology Group. *C/C++ Language Extensions for Cell BroadBand Engine Architecture V2.4*, March 2007.

[5] IBM Systems and Technology Group. *Cell Broadband Engine SDK Libraries Version 2.1*, March 2007.

[6] IBM Systems and Technology Group. *Cell Broadband Engine Software Development Toolkit (SDK) Installation Guide Version 2.1*, March 2007.

[7] IBM Systems and Technology Group. *SPE Runtime Management Library Version 2.1*, March 2007.

[8] J. Kahle, M. Day, H. Hofstee, C. Johns, T. Maeurer, and D. Shippy. Introduction to the Cell Multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589–604, July/September 2005.

[9] M. Kistler, M. Perrone, and F. Petrini. Cell Processor Interconnection Network: Built for Speed. *IEEE Micro*, 25(3):2–15, May/June 2006.