J Supercomput (2010) 53: 247–268
DOI 10.1007/s11227-009-0292-7

Characterizing the basic synchronization and communication operations in Dual Cell-based Blades through CellStats

José L. Abellán · Juan Fernández · Manuel E. Acacio

Published online: 1 April 2009

Abstract The Cell Broadband Engine (Cell BE) is a heterogeneous chip-multiprocessor (CMP) architecture to offer very high performance, especially on game and multimedia applications. The singularity of its architecture, nine cores of two different types, along with the variety of synchronization and communication primitives offered to programmers, make the task of developing efficient applications very challenging. This situation gets even worse when dual Cell-based blade platforms are considered, where two separate Cells can be linked together through a dedicated high-speed interface. In this work, we present a characterization of the main synchronization and communication primitives provided to programmers in the context of a dual Cell-based blade under varying workloads through our *CellStats* tool. In particular, we focus on the DMA transfer mechanism, the mailboxes, the signals, the read-modify-write atomic operations, and the time taken by thread creation. Our performance results expose the bottlenecks and asymmetries of these platforms, which must be taken into account by programmers for choosing the most adequate primitives to improve the efficiency of their applications.

Keywords Cell Broadband Engine · CellStats · Synchronization primitives · Communication operations

J.L. Abellán (✉) · J. Fernández · M.E. Acacio
Dept. de Ingeniería y Tecnología de Computadores, University of Murcia, 30100 Murcia, Spain
e-mail: jl.abellan@ditec.um.es

J. Fernández
e-mail: juanf@ditec.um.es

M.E. Acacio
e-mail: meacacio@ditec.um.es

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 presented in applications [1]. This tendency that had been followed until very recently by most commercial microprocessor developers, has been replaced lately by others that try 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; they improve overall performance and are more energy efficient.

Nowadays, CMP (or chip-multiprocessor) 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 [2] and AMD [3]), servers (for example, the IBM Power6 [4] or Sun UltraSPARC T2 [5]) and even supercomputers (for example, the CPU chips used as building blocks in the IBM BlueGene/P [6]). Nowadays, among all contemporary CMP (or chip-multiprocessor) 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, the Cell BE is the heart of the Sony Playstation 3 (PS3).

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 [7]. 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 8) are called *Synergistic Processing Elements* or SPEs 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 which 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 [8].

The Cell BE provides programmers with a broad variety of communication and synchronization primitives between the threads that comprise parallel applications, which were evaluated in [9]. 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 describe *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), lists of DMA transfers, the read-modify-write atomic operations, the mailboxes, the signals and the time taken by thread creation. For the DMA transfers or lists of 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. Finally, for the read-modify-write atomic operations the remote memory locations reside in main memory. All the primitives can be evaluated for different number of intervening SPEs, and when applicable, with varying memory sizes and address ranges. 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 Sect. 2, we provide a revision of the architecture of the Cell BE and a dual Cell-based blade, and a description of some of the communication and synchronization primitives provided to programmers. Next, in Sect. 3, we introduce our *CellStats* tool, for characterizing these primitives. The results obtained after executing *CellStats* on a dual Cell-based blade are presented in Sect. 4. From the previous characterization, some recommendations to programmers are summarized in Sect. 5. Finally, Sect. 6 gives the main conclusions of the paper.

2 Dual cell-based blade

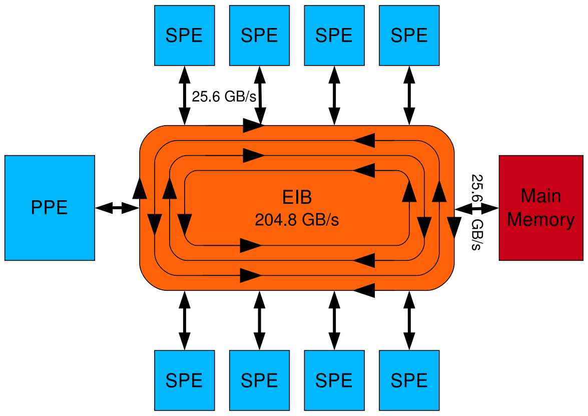
2.1 Architecture

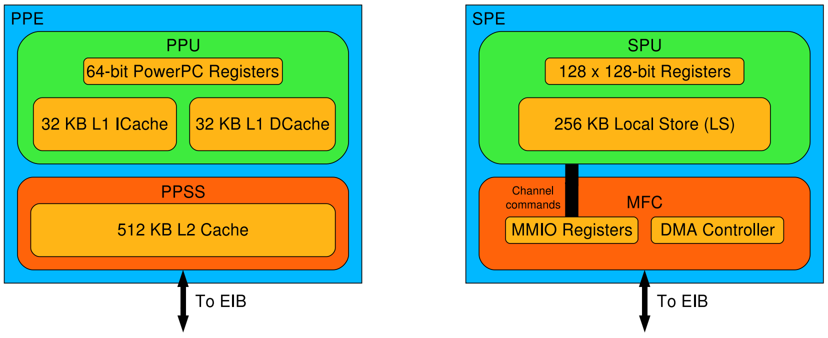
The Cell BE architecture [7, 10, 11] is a heterogeneous multicore chip composed of one general-purpose processor, called *PowerPC Processor Element* (PPE), eight specialized coprocessors, called *Synergistic Processing Elements* (SPEs), a high-speed memory interface controller, and an I/O interface, all integrated in a single chip. All these elements communicate through an internal high-speed *Element Interconnect Bus* (EIB) (see Fig. 1).

The latest version of the Cell BE processor, running at 3.2 GHz, has a theoretical peak performance of 204.8 Gflop/s (single precision) and 14.63 Gflop/s (double precision). The EIB supports a peak bandwidth of 204.8 Gbytes/s for intrachip 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/s to main memory. Finally, the I/O controller provides peak bandwidths of 25 Gbytes/s inbound and 35 Gbytes/s 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 (*Vector/SIMD Multimedia Extension*), a 32 KByte L1 instruction cache, a 32 KByte L1 data cache, and a 512 KByte L2 cache. The PPE is a dual issue, in-order execution, 2-way SMT processor. The PPE comprehends

Fig. 1 Cell broadband engine





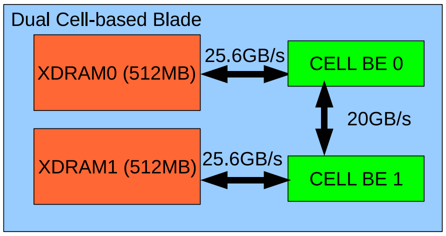
(a) PowerPC Processor Element. (b) Synergistic Processing Unit.

Fig. 2 Types of cores of a cell broadband engine

two different units, namely *PowerPC Processor Unit* (PPU) and *PowerPC Processor Storage Subsystem* (PPSS) (see Fig. 2(a)).

Each SPE is a 128-bit RISC processor designed for high-performance on streaming and data-intensive applications [12]. Each SPE consists of a *Synergistic Processing Unit* (SPU) and a *Memory Flow Controller* (MFC). The SPUs are 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. As opposed to the PPE, the SPEs do not have a private cache memory. In contrast, each SPU includes a 256 KByte LS memory to hold both instructions and data of SPU programs, that is, the SPUs cannot access main memory directly. The MFC contains a *DMA Controller* and a set of memory-mapped registers called *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

Fig. 3 Dual Cell-based Blade



channel commands, or the other SPEs (or even the PPE), which access the MFC through the MMIO registers (see Fig. 2(b)).

A dual Cell-based Blade is composed of two separate Cell BEs linked together through the EIB [7]. This results in a maximum theoretical computation performance of 409.6 Gflop/s for single-precision and 29.26 Gflop/s for double-precision, which is very interesting for emerging scientific, game, and multimedia applications. The main components of a dual Cell-based blade are shown in Fig. 3. In this architecture, the two Cell BEs operate in SMP mode with full cache and memory coherency. Main memory is split into two different modules, namely XDRAM0 and XDRAM1, that are attached to Cell0 and Cell1, respectively. In turn, the EIB is extended transparently across a high-speed coherent interface running at 20 Gbytes/s in each direction [8]. Each Cell BE processor includes 512 MBytes of XDR DRAM for a total of 1 GByte of main memory.

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 nonblocking 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*.

The 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 KByte. DMA transfers can be either blocking or nonblocking. The latter allow overlapping 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 2,048 DMA transfers, each one up to 16 KByte 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 [13]. 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. For us, atomic operations are simple read-modify-write 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. We refer the reader to [14, 15] for additional details. The PPE program can include extensions (e.g., `vec_ add`), to use its VMX unit; and library function calls [16], to manage threads and perform communication and synchronization operations (e.g., `spe_context_run`, `spe_in_mbox_write` and `spe_signal_write`). The SPE program follows an SPMD model (*Single Program Multiple Data*). It includes extensions [17], to execute SIMD instructions, and communication and synchronization operations (e.g., `spu_add`, `spu_read_in_mbox` and `spu_ mfcdma32`); and function calls to the SDK library [18], to carry out complex tasks of different nature (matrix, FFT, filters, etc.).

Programming of a dual Cell-based blade is equivalent to that of an independent Cell from a functional point of view. However, there are two important differences. First, dual Cell-based blades have 16 SPEs at programmer's disposal rather than 8 SPEs. This feature involves doubling the maximum theoretical performance but also making much more difficult to extract thread-level parallelism from applications. Second, from an architectural point of view, any operation crossing the Cell-to-Cell interface results in significantly less performance than those that stay on-chip (see Sect. 4). These facts must be taken into account by programmers to avoid unexpected and undesirable surprises when parallelizing applications for a dual Cell-based blade platform.

3 CellStats

3.1 Functionality

CellStats aims at characterizing all the main primitives (see Sect. 2.2) which probably any programmer has to use when developing applications on the Cell BE. Thus, our tool allows to evaluate: the thread creation (in Sect. 3.1.1); the mailboxes (in Sect. 3.1.2); the signals (in Sect. 3.1.3); the DMA transfers (in Sect. 3.1.4): GETs, PUTs and lists; and finally, the read-modify-write atomic operations (in Sect. 3.1.5). In more detail, for the mailboxes and signals, we distinguish the case in which the PPE and the SPEs are involved, and that in which two SPEs are the participants. For the DMA transfers or lists of DMA transfers, we consider both transfers between main memory and an SPE's LS, and between SPEs' LSs. For the read-modify-write atomic operations, the remote memory locations reside in main memory. All the primitives can be evaluated for different number of intervening SPEs, and when applicable, with varying buffer sizes and access patterns. Eventually, due to the fact that our study comprises a dual Cell-based blade, we have to deal with SPEs belonging to Cell0, Cell1, or even both when dealing with more than 8 SPEs, and DMA transfers to/from XDRAM0 and XDRAM1 memory modules (see Sect. 2.1).

From the user's point of view, *CellStats* is a command-line tool which admits parameters in function of the described primitives. In this way, our tool supports a different primitive depending on the `-f FUNCTION` parameter. Thus, `FUNCTION` can take the following values: `mkthread` for the thread creation; `mailbox` for the mailboxes; `signal` for the signals; regarding the atomic operations: `fetchadd/fetchsub` for adding/subtracting a remote memory word, `fetchinc/fetchdec` for increasing/decreasing a remote memory word, and `fetchset` for setting a remote memory word (remote memory locations refer to main memory); and finally the DMA transfers can be: `dmaget/dmaput` for transfers to/from LS and `dmalistget/dmalistput` which involve transfers by using lists. In addition, other parameters are needed in order to specify the number of involved SPEs (`-n NSPE`), the Cell which the SPEs belong to (`-c CELL`) and the XDRAM memory module where buffers are allocated (`-m XDRAM`) through the `NUMA` policy library.

All experiments are executed in a loop in order to report average statistics. In this way, the user must specify a number of iterations through the parameter `-i ITER`. In particular, all results shown in Sect. 4 are the arithmetic mean of one million repetitions, but also we have applied a warm-up of one thousand iterations before each experiment. To do that, we use the parameters `-w 1k` and `-i 1m`, respectively (numeric parameters admit the use of multipliers `k` and `m`). In light of our performance results, we would like to point out that there is a negligible variance, what implies reliable arithmetic means.

Next, we take a closer look to the above described operations with command-line examples of using our tool.

3.1.1 Thread creation

In a multithreaded application on the Cell BE, programmers have to create threads running on the PPE and also on the SPEs. A well-known programming model for this processor lies in dividing an application into independent tasks, and devoting a single PPE's thread to create and orchestrate all the SPEs' threads which run those tasks. In this context, measuring the time of creating SPEs' threads is a great asset to programmers in order to determine the overhead which it introduces in their applications. Thus, we have used an empty SPE's task that returns immediately once it is launched. Consequently, we measure 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.

3.1.2 Mailboxes

Once a task is spawned into an specific SPE, it commonly requires to perform a PPE-to-SPE or even an SPE-to-SPE synchronization during its execution. As explained in Sect. 2.2, there are primarily two primitives to implement synchronization mechanisms: mailboxes and signals. In this section, we deal with the former. To perform a synchronization, the PPE/SPE 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. In the former case, the PPE uses a runtime management library function involving a system call (`spe_write_in_mbox` [16]) which explains higher latency as we will explain in Sect. 4. Nevertheless, the PPE can also write directly into the corresponding SPE's MMIO register using a regular assignment. For example,

```
$ CellStats -f mailbox -n 1 -i 1m -c 0
$ CellStats -f mailbox -n 1 -i 1m -c 1
```

repeat the PPE-to-SPE synchronization cycle between the PPE and one SPE from Cell0 (SPE0 as default value) or from Cell1 (SPE8 as default value) one million times, respectively.

```
$ CellStats -f mailbox -n 2 -i 1m -c 0 -s
$ CellStats -f mailbox -n 2 -i 1m -c 1 -s
```

Do the same between two SPEs: both from Cell0 or one SPE from Cell0 and another SPE from Cell1, respectively. The `-s` parameter specifies that the operations will be performed between SPEs, and the `-n` parameter specifies the number of SPEs. SPE0 is always selected as default value and the remaining SPEs belong to Cell specified by the parameter `-c`.

3.1.3 Signals

As we mentioned in last section, the second method to perform a PPE-to-SPE or an SPE-to-SPE synchronization is based on using signals. The process is as follows. 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*). Like mailboxes, it is possible to write directly into the SPE's MMIO register instead of using the runtime management library function called `spe_write_signal` [16]. For example,

```
$ CellStats -f signal -n 1 -i 1m -c 0
$ CellStats -f signal -n 1 -i 1m -c 1
```

repeat the PPE-to-SPE synchronization cycle between the PPE and one SPE from Cell0 (SPE0 as default value) or from Cell1 (SPE8 as default value) one million times, respectively.

```
$ CellStats -f signal -n 2 -i 1m -c 0 -s
$ CellStats -f signal -n 2 -i 1m -c 1 -s
```

Do the same between two SPEs: both from Cell0 or one SPE from Cell0 and another SPE from Cell1, respectively.

3.1.4 DMA operations

Commonly, any SPE's task collects data from main memory into its LS, computes them, and writes them back to main memory. However, for the sake of efficiency, it is also common to transfer data to/from other LSs. These transfers are accomplished through DMA operations, as explained in Sect. 2.1. Because of this, we have studied DMAs between main memory and the local LS, and even also between a remote LS and the local LS. In current version of *CellStats*, these operations are blocking, meaning that a read of an empty channel command or a write to a full channel command causes the SPU to block until the operation is completed. When evaluating the performance of DMA operations, the size of the DMA buffer is an important factor. Hence, the Cell BE can handle buffers between 1, 2, 4, 8, or a multiple of 16 Bytes up to a maximum of 16 KByte. In this way, *CellStats* supports different buffer's sizes via the `-b SIZE` parameter. As we will explain in Sect. 4.4, shared or private buffers report different performance statistics. With a shared buffer, we mean a buffer on which several SPEs can perform the DMA operations, and on the other hand, a private buffer is that on which only one SPE (its owner) can perform the DMA operations. Besides, to know how the EIB behaves under high-load conditions, we do not only consider the case in which several SPEs perform DMA operations on a single shared buffer simultaneously, but also that in which every DMA operation uses its own private buffer. In the latter case, we define the distance between two consecutive private buffers, namely stride. To do that, the `-n` and `-N` parameters indicate whether the source buffer (GETs) or the destination buffer (PUTs) is shared or private. Like mailbox or signal operations, 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) and the parameter `-m` indicates if memory location resides in XDRAM0 or XDRAM1. Finally, the `-t STRIDE` parameter specifies the stride. For example,

```
$ CellStats -f dmaget -n 3 -i 1m -b 1k -c 0 -m 0
$ CellStats -f dmaput -n 6 -i 1m -b 1k -c 0 -m 1
```

create three or six threads from Cell0, that execute one million `dmaget` or `dmaput` operations, each over a shared 1024 Bytes buffer from XDRAM0 or XDRAM1, respectively.

```
$ CellStats -f dmaget -N 3 -b 1k -i 1m -t 2k -c 1 -m 0
$ CellStats -f dmaput -N 6 -b 1k -i 1m -t 2k -c 1 -m 1
```

create three or six threads from Cell1, that execute one million `dmaget` or `dmaput` operations, each over three or six different 1024-Bytes buffers separated by 2,048 memory locations from XDRAM0 or XDRAM1, respectively. Moreover, when using `-s` parameter:

```
$ CellStats -f dmaget -N 6 -b 32 -t 128 -i 1m -c 1 -m 1 -s
```

creates five threads from Cell1 and one from Cell0 (SPE 0 as default value), that execute one million `dmaget` operations, each over five different 32-Bytes buffers separated by 128 memory locations from SPE 0's LS.

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 -i 1m -c 1 -m 1
```

creates six threads from Cell1, that execute one million data transfers from one 1,024 Bytes shared memory buffer residing on XDRAM1. Therefore, each data transfer consists of 16 DMA operations of 64 Bytes each.

3.1.5 Atomic operations

Atomic operations are commonplace in multicore architectures in order to set sections in the code that have to be processed as an uninterrupted unit by a single core. In this work, these operations enable sequences of read-modify-write instructions on main memory locations in an atomic fashion. Like DMA operations, the atomic operations admit the same set of parameters: `-N SPEs`, `-m XDRAM`, `-b SIZE`, `-t STRIDE`, `-n` and `-N` parameters. However, these operations are performed only on buffer's sizes up to 128 Bytes due to hardware restrictions (PPE's cache memory line size is 128 Bytes [16]). In the current version of *CellStats*, these operations can only be performed in main memory locations. For example,

```
$ CellStats -f fetchadd -n 3 -i 1m -c 0 -m 0
$ CellStats -f fetchsub -n 6 -i 1m -c 0 -m 1
```

create three or six threads from Cell0, that execute one million `fetchadd` or `fetchsub` atomic operations, each over a shared 32-bit (default size) main memory variable in XDRAM0 or XDRAM1, respectively.

```
$ CellStats -f fetchinc -N 3 -t 2k -i 1m -c 1 -m 0 -b 128
$ CellStats -f fetchset -N 6 -t 2k -i 1m -c 1 -m 1 -b 128
```

create three or six threads from Cell1, that execute one million `fetchinc` or `fetchset` atomic operations, each over three or six different 128-bytes main memory variables separated by 2,048 memory locations from XDRAM0 or XDRAM1, respectively.

3.2 Architecture

In order to have a clear understanding of how *CellStats* implements the functionality aforementioned, we explain its architecture succinctly. In this sense, our tool carries out the following steps to characterize all the primitives. 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* (duration of every *tick* for the dual Cell-based blade is 70 ns). 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.

4 Evaluation

4.1 Testbed

To develop *CellStats,* we used the IBM SDK v3.0 for the Cell BE architecture installed atop Fedora Core 7 on a regular PC [19]. This development kit includes a simulator, named Mambo, that allows programmers to execute binary files compiled for the Cell BE architecture. To obtain the experimental results, we installed the same development kit atop Fedora Core 6 on a dual Cell-based IBM BladeCenter QS20 blade which incorporates two 3.2 GHz Cell BEs v5.1, namely Cell0 and Cell1, with 1 GByte of main memory and a 40 Gbytes hard disk.

4.2 Thread creation

The average latency to launch each new thread is considerably high, around 1.68 ms. It is explained since this operation implies to perform the following three library calls from the PPE (see [14]): the `spe_context_create` call to create a context for the SPE thread which contains the persistent information about a logical SPE; the `spe_program_load` call to load the code of the SPE's tasks; and finally, the `spe_ context_run` call to execute the SPE context. As described in Sect. 3.1.1, this latency also includes the time needed to detect thread finalization.

4.3 Mailboxes and signals

In Table 1, the average latencies, measured in nanoseconds, for PPE-to-SPE synchronization using mailboxes or signals are shown. In both cases, the PPE can either invoke a system call (Mailbox-sc or Signal-sc) or write directly into the corresponding SPE's MMIO register (Mailbox or Signal). Besides, we consider that the selected SPE can be placed on either Cell for comparison (PPE-SPEc0 for Cell0 and PPE-SPEc1 for Cell1). As we can see, the latency is shorter when writing directly into the SPE's MMIO registers, as defined in file `cbe_mfc.h`, instead of using the runtime management library function calls `spe_write_signal` or `spe_write_in_mbox` [16]. In the former case, it is worth noting that the synchronization latency doubles when the destination SPE resides on Cell1 in both cases. In addition, Table 2 summarizes the average SPE-to-SPE synchronization latency, measured in

Table 1 Average latency for PPE-to-SPE synchronization

Primitive	PPE-SPEc0	PPE-SPEc1
Mailbox-sc	10,000.0	10,000.0
Mailbox	779.7	1,678.2
Signal-sc	18,000.0	18,000.0
Signal	503.8	1,182.3

Table 2 Average latency for SPE-to-SPE synchronization

Primitive	SPEc0-SPEc0	SPEc0-SPEc1
Mailbox	158.1	589.9
Signal	160.1	619.4

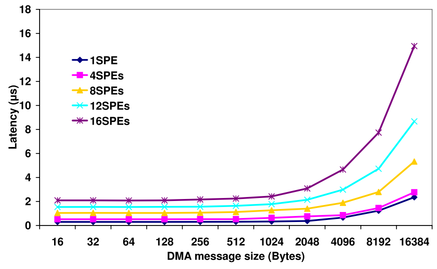
nanoseconds, using mailboxes or signals when both SPEs are located on the same Cell (SPEc0-SPEc0) or on different Cells (SPEc0-SPEc1), respectively. In the former case, the latency is almost four times shorter because the synchronization messages stay on-chip and do not need to cross the Cell-to-Cell interface.

4.4 DMA operations

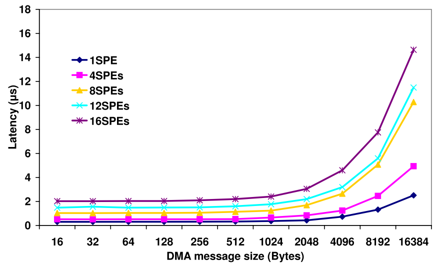
There are three different scenarios for data movement: data transfers between main memory and an SPE's LS (GETs), data transfers between an SPE's LS and main memory (PUTs) and data transfers between SPEs' LSs (MOVs).

In Figs. 4 and 5, latency and bandwidth figures for GETs and PUTs on shared buffers using XDRAM0 and XDRAM1 are shown. As we can see, two general trends can be identified. First, latency is constant for message sizes smaller than or equal to the cache line, that is 128 Bytes. Second, latency grows proportionally to the message size for messages larger than the cache line until the available bandwidth is exhausted. In addition, a more in depth analysis provides other interesting conclusions. Latency is constant, but proportional to the number of SPEs for message sizes up to 128 Bytes (see Figs. 4(a), 4(b), 5(a), and 5(b)) regardless of the XDRAM module. For messages larger than 128 Bytes, GETs and PUTs from or to XDRAM1 memory module report higher latency, because these operations must cross the Cell-to-Cell interface. In addition, GETs outperform PUTs under stress [13] which explains higher latencies in Figs. 5(a) and 5(b).

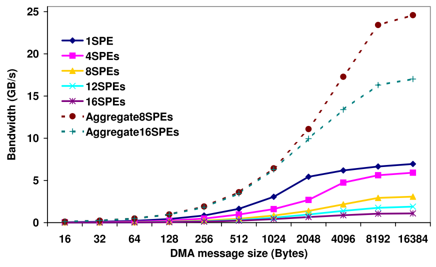
Moreover, Figs. 4(c), 4(d), 5(c), and 5(d) present the same results in terms of bandwidth achieved by each DMA operation. There are three important trends to be considered. Firstly, when 8 SPEs are involved, GETs on XDRAM0 memory module obtain an aggregate bandwidth of 24.6 Gbytes/s (close to the peak memory bandwidth), while GETs on XDRAM1 memory module reach an aggregate bandwidth of 13.6 Gbytes/s because of crossing the Cell-to-Cell interface, limiting the maximum achievable aggregate bandwidth. Secondly, when 16 SPEs are considered both Cells are involved, thus the figures report the benefits of transferring data from the closest XDRAM memory module (XDRAM0 for SPEs within Cell0 or XDRAM1 for SPEs within Cell1), and also report the drawback of going through the Cell-to-Cell



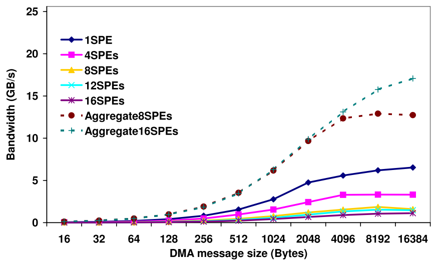
(a) GETs from XDRAM0 (Latency).



(b) GETs from XDRAM1 (Latency).

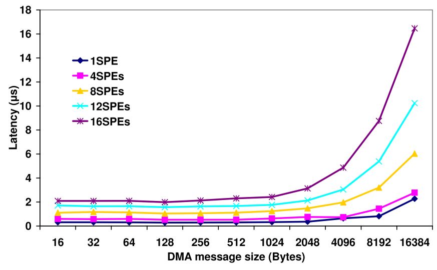


(c) GETs from XDRAM0 (Bandwidth).

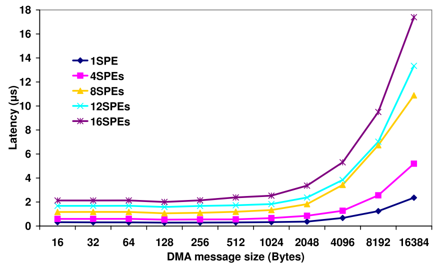


(d) GETs from XDRAM1 (Bandwidth).

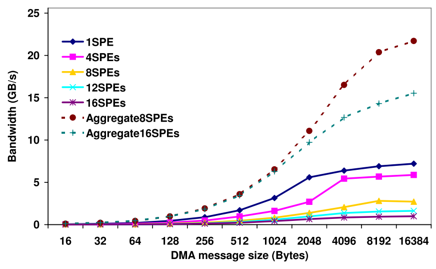
Fig. 4 Latency and bandwidth of DMA GETs on shared main memory buffers for a variable number of SPEs and packet sizes using Cell0 and Cell1



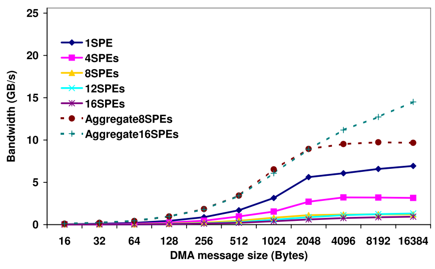
(a) PUTs to XDRAM0 (Latency).



(b) PUTs to XDRAM1 (Latency).

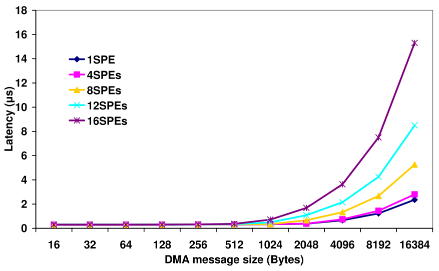


(c) PUTs to XDRAM0 (Bandwidth).

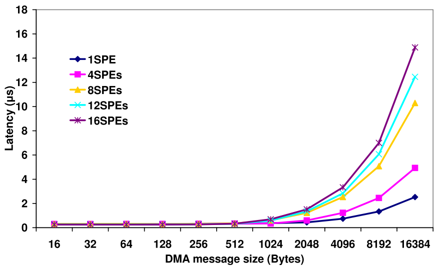


(d) PUTs to XDRAM1 (Bandwidth).

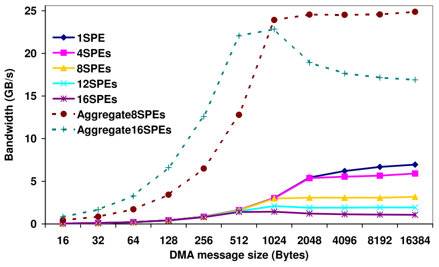
Fig. 5 Latency and bandwidth of DMA PUTs on shared main memory buffers for a variable number of SPEs and packet sizes using Cell0 and Cell1



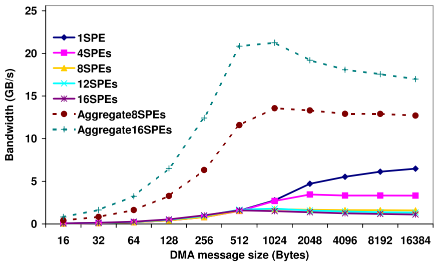
(a) GETs from XDRAM0 (Latency).



(b) GETs from XDRAM1 (Latency).



(c) GETs from XDRAM0 (Bandwidth).



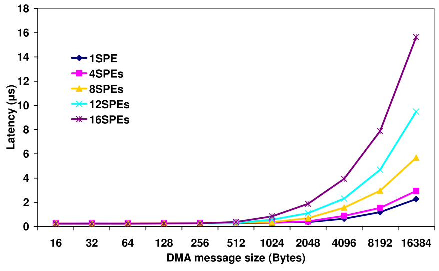
(d) GETs from XDRAM1 (Bandwidth).

Fig. 6 Latency and bandwidth of DMA GETs on private main memory buffers for a variable number of SPEs and packet sizes using Cell0 and Cell1

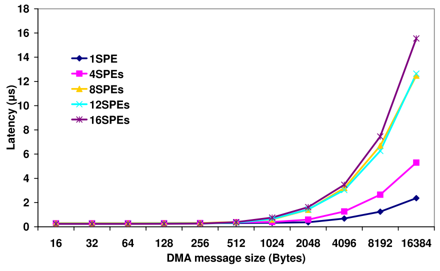
interface (XDRAM1 for SPEs within Cell0 and XDRAM0 for SPEs within Cell0). Finally, because of higher latencies in Figs. 5(a) and 5(b), PUTs reports lower bandwidths than GETs operations.

In Figs. 6 and 7, latency and bandwidth figures for GETs and PUTs on private buffers using XDRAM0 and XDRAM1 are shown (stride is larger than or equal to the cache line size in all cases). As we can see, the same trends are shown than figures with shared buffers. But a more in depth analysis provides the following conclusions. Latency is constant, around 300 ns, for message sizes up to 512 Bytes regardless of the XDRAM memory module is used (see Figs. 6(a), 6(b), 7(a), and 7(b)). Moreover, the aggregate bandwidth grows faster for message sizes up to 1 KByte because of exploiting simultaneous transfers to different buffers. After that, the aggregate bandwidth figures converge to the same values as for shared buffers.

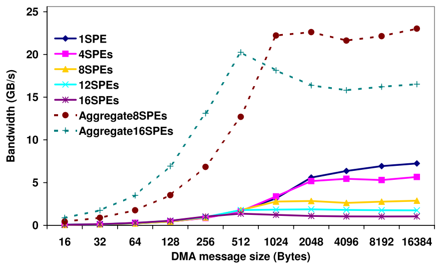
In turn, latency and bandwidth figures for MOVs using Cell0 and Cell1 are shown in Figs. 8 and 9. In particular, Figs. 8(a), 8(b), 9(a), and 9(b) correspond to DMA MOVs in Cell0, while Figs. 8(c), 8(d), 9(c), and 9(d) correspond to DMA MOVs between Cell0 and Cell1. In the former case, SPEs approach the maximum available bandwidth of the EIB-to-SPE interface. In the later case, the Cell-to-Cell interface bandwidth is the limiting factor. Nevertheless, the latency is much longer than expected resulting in an aggregate bandwidth shorter than that of GETs from XDRAM1 memory module.



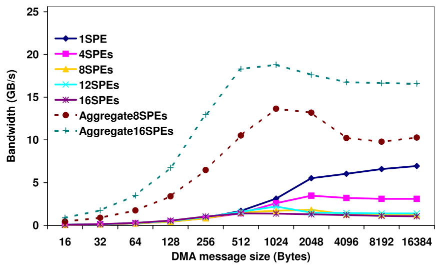
(a) PUTs to XDRAM0 (Latency).



(b) PUTs to XDRAM1 (Latency).

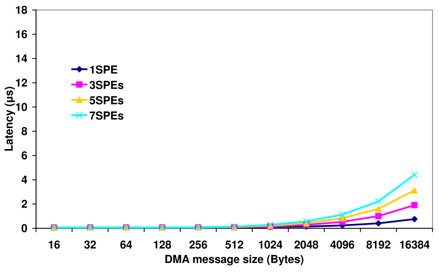


(c) PUTs to XDRAM0 (Bandwidth).

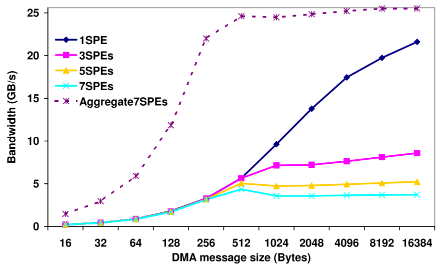


(d) PUTs to XDRAM1 (Bandwidth).

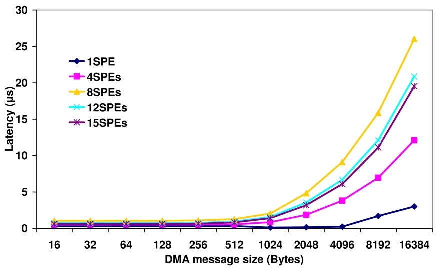
Fig. 7 Latency and bandwidth of DMA PUTs on private main memory buffers for a variable number of SPEs and packet sizes using Cell0 and Cell1



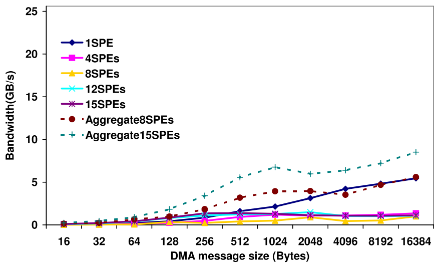
(a) Intra-Cell GETs (shared LS buffer).



(b) Intra-Cell GETs (shared LS buffer).

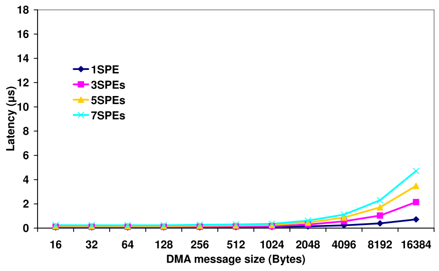


(c) Inter-Cell GETs (shared LS buffer).

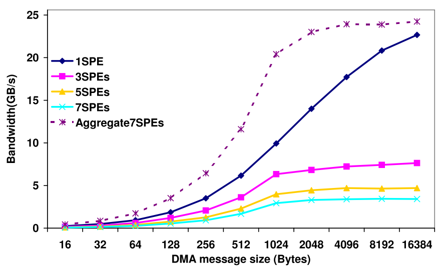


(d) Inter-Cell GETs (shared LS buffer).

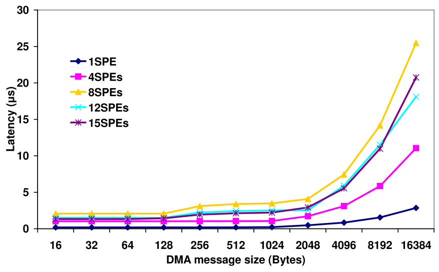
Fig. 8 Latency and bandwidth of GETs on shared LS buffers for a variable number of SPEs and packet sizes using a single Cell and both Cells



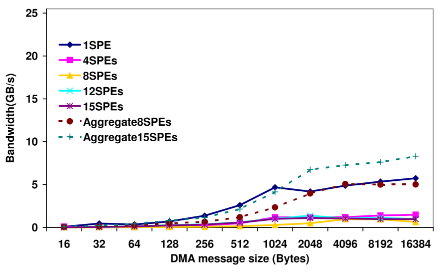
(a) Intra-Cell PUTs (shared LS buffer).



(b) Intra-Cell PUTs (shared LS buffer).



(c) Inter-Cell PUTs (shared LS buffer).



(d) Inter-Cell PUTs (shared LS buffer).

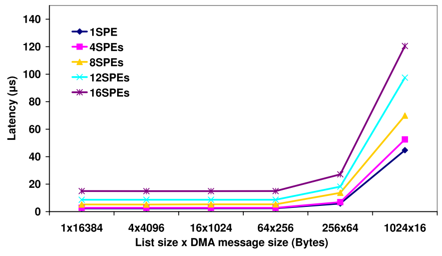
Fig. 9 Latency and bandwidth of PUTs on shared LS buffers for a variable number of SPEs and packet sizes using a single Cell and both Cells

4.5 Lists of DMA operations

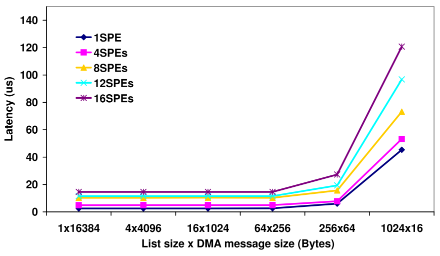
Finally, in Figs. 10 and 11 latency and aggregate bandwidth figures for lists of GETs and PUTs on shared buffers using XDRAM0 and XDRAM1 are shown. As we can see in X-axis, we adjust the size of lists and the size of transfer per list element to always transfer 16 kbytes.

In latency figures, three important trends can be identified. Firstly, the latency obtained up to 64 element per list is equal to the latency obtained with one 16 kbytes DMA transfer (without list mechanism). Secondly, when the number of elements per list is larger than 64, the latency grows quickly because of the overhead introduced by the list mechanism (see almost 140 µs for 1024 elements of 16 Bytes). Finally, the same trends under stress presented in Figs. 4(a) and 5(a) for GETs and PUTs, are illustrated in Figs. 10(a) and 10(c) for lists of GETs and PUTs.

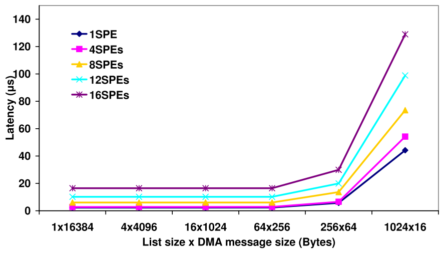
Regarding aggregate bandwidth figures there are three important conclusions. Firstly, as we expected, when the number of elements per list ranges from 1 to 64, the same trends as DMA transfers without list mechanism are presented. Secondly, Figs. 11(a) and 11(b) for lists of GETs, and 11(c) and 11(d) for lists of PUTs, illustrate the effects of crossing the Cell-to-Cell interface limiting the maximum achievable aggregate bandwidth. Finally, when the number of elements per list is larger than 64, the overhead introduced by the list mechanism reduces considerably the achievable aggregate bandwidth (almost 3 Gbytes/s in all 1024 × 16 Bytes experiments).



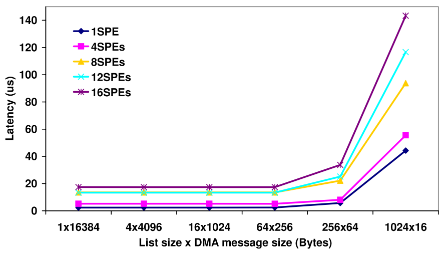
(a) Lists of GETs from XDRAM0.



(b) Lists of GETs from XDRAM1.

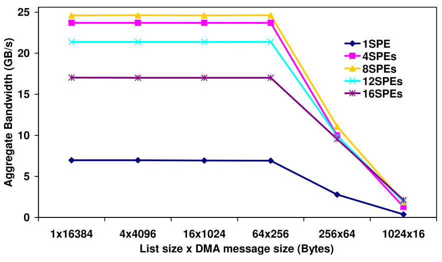


(c) Lists of PUTs from XDRAM0.

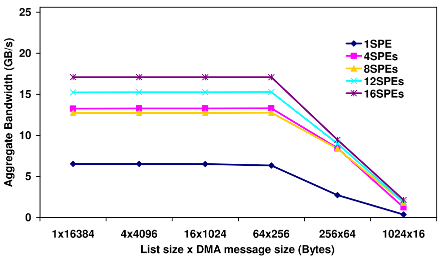


(d) Lists of PUTs from XDRAM1.

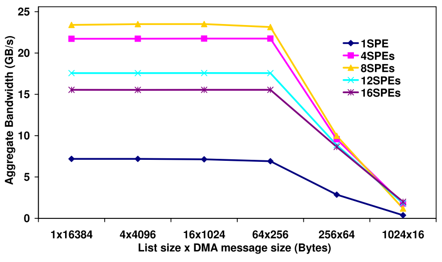
Fig. 10 Latency for lists of DMA operations (16 KByte packets) on shared main memory buffers for a variable number of SPEs and packet sizes using Cell0 and Cell1



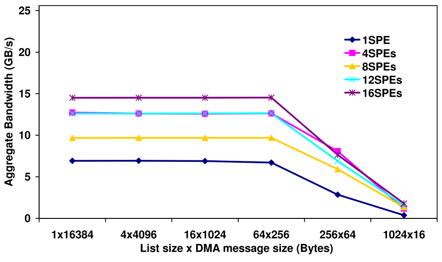
(a) Lists of GETs from XDRAM0.



(b) Lists of GETs from XDRAM1.

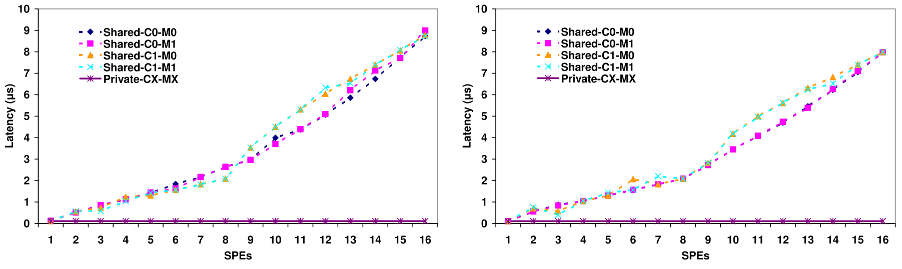


(c) Lists of PUTs from XDRAM0.



(d) Lists of PUTs from XDRAM1.

Fig. 11 Aggregate bandwidth for lists of DMA operations (16 KByte packets) on shared main memory buffers for a variable number of SPEs and packet sizes using Cell0 and Cell1



(a) Variable of 4-to-64 Bytes. (b) Variable of 128 Bytes.

Fig. 12 Latency of atomic operations on shared and separate variables (128-Bytes stride)

4.5.1 Atomic operations

The average latency of the `fetch&add` atomic operation is shown in Fig. 12. It depends on the size of the variable, the variable's memory location (XDRAM0 or XDRAM1), whether the SPEs belong to Cell0 or Cell1 and whether operations are on private variables or on a shared variable. For example, `Shared-C1-M1` means that a shared variable is used, the SPEs belong to Cell1 (if there are more than 8 SPEs, the remaining SPEs belong to the other Cell) and the variable's memory location is XDRAM1. In order to select the variable's memory location (XDRAM0 or XDRAM1), we have used the `NUMA` policy library. In addition, for `Private-CX-MX`, `X` means either Cell or either variable's memory location.

There are two types of figures: Figure 12(a) is for variables of 4-to-64 Bytes and Fig. 12(b) is for variables of 128 Bytes (size of PPE's cache line). In both figures, the latency remains constant, at approximately 111 ns, when the variable is privately accessed by the SPEs. However, the latency grows linearly, up to 9 µs (Fig. 12(a)) or 8 µs (Fig. 12(b)) when the variable is shared by all intervening SPEs. This is due to the fact that shared variables serialize the execution of atomic operations, but also in latter case it is used the variable's size of PPE's cache line. Results for the rest of the atomic operations are similar and, therefore, have been omitted. Notice that, the XDRAM memory module selected has negligible effect on performance results. Nevertheless, from 10 to 15 SPEs the `Shared-C1-MX` experiments are 1 µs greater than the others `Shared-C0-MX`.

5 Recommendations for Cell BE programmers

5.1 Thread creation

Due to the high overhead of thread creation, we should create threads with the utmost care. For that, programmers could create SPE threads at startup and keep them alive until the application finishes. In this way, instead of creating a thread per SPE task, each SPE could store the code of all tasks in its LS. Then the SPE thread receives task identifiers from the PPE, through mailboxes or signals, and finally the SPE executes the associate task. The main drawback of this method is the limited size of the LS (only 256 kbytes). However, in [20], the authors demonstrate that this approach is more efficient than creating as many threads as tasks in several orders of magnitude, and they implement an overlay-based technique to solve the LS limitation.

5.2 Mailboxes and signals

In light of latency results, it is clear that programmers should use direct writes to the SPEs' MMIO registers through library functions which do not involve a system call (see Sect. 4.3), and they should try to reduce inter-Cell communications. To clarify the last recommendation, we show a real example from [21]. In that work, we implement three common collective operations: barrier, reduce and broadcast. For each of them, we explore several alternatives ranging from the naive approach to well-known algorithms coming from the cluster arena. In particular, we focus on two versions of the All-To-One barrier implemented by means of signals operations (we consider the case in which 16 SPEs participate in a barrier). The first one is a naive approach in which all SPEs perform an SPE-to-SPE synchronization with an specific SPE from Cell0, namely root. The second approach is more sophisticated since two SPEs, belonging to each Cell, are set as roots. In this case, each root gathers intra-Cell signals, sends a signal to the other root and finally, sends intra-Cell signals to complete the synchronization. Consequently, only two signals are sent through the inter-Cell interface. As a result, latency of the first approach is around 2700 ns, whereas the second one takes only around 700 ns.

5.3 DMA operations

To optimize their applications, multicore programmers should design several data layouts of their applications, and choose the best mapping according to their target architectures [20]. In case of the Cell BE, this situation becomes much harder than conventional multicores due to the explicit control of DMA transfers and its heterogeneity. In this sense, we believe that our evaluation is of paramount importance, because programmers must tackle with the tradeoff between computing granularity vs. message size when designing their data layouts. Therefore, to obtain the maximum peak performance, they should take into account that: private buffers are more efficient than shared buffers, but only up to 1 kB, because for messages larger than 1 kB the latency is identical in both cases; for list of DMA transfers, programmers should be aware of the overhead of the list mechanism, for instance when the number of elements per list is larger than 64 to transfer 16 kbytes; finally programmers should be aware of the Cell-to-Cell interface, which determines the maximum achievable bandwidth, and also the asymmetries that arise when memory locations are in the furthest XDRAM memory module. Hence, programmers should use the `NUMA` policy library in order to control memory locations to always transfer data from the closest XDRAM memory module.

5.4 Atomic operations

Similarly to DMA operations, whenever possible, programmers should use private buffers because latency grows linearly with the number of involved SPEs when shared buffers are used. However, it depends on the tradeoff between performance and programming complexity. For instance, atomic operations on a shared buffer are commonly used in the context of dynamic load-balancing [22, 23]. This is due to the fact that they introduce negligible overhead with respect to the granularity of the load to balance. Moreover, this approach is much easier to program than using private buffers because it does not need to gather partial results from private buffers.

6 Conclusions

The Cell BE is a heterogeneous multicore processor specifically designed for emerging scientific, game and multimedia applications. Programming the Cell BE is a great challenge, because it involves dealing with multiple threads and with a number of communication and synchronization primitives. This situation gets even worse when dual Cell-based blades are considered, what means up to 16 SPEs at programmer's disposal and asymmetries when crossing the Cell-to-Cell interface. Hence, programming is as flexible as complex because efficient programs relies primarily on the ability to select the most appropriate primitive as well as their corresponding configuration values. In this context, we have performed a characterization of those primitives on a dual Cell-based blade platform. To do so, we have developed a command-line tool, namely *CellStats*, which supports different configurations per primitive in order to explore its performance (i.e., average latency or bandwidth) under varying workloads. As a result of our work, we highlight the real potential of the Cell BE but also we give some recommendations that can help programmers identify the most appropriate primitive in different situations.

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

References

1. Agarwal V, Hrishikesh M, Keckler SW, Burger D (2000) Clock rate versus IPC: the end of the road for conventional microarchitectures. In: Proceedings of the 27th international symposium on computer architecture, Vancouver, British Columbia, Canada
2. Kanter D Intels next generation microarchitecture unveiled. Real World Technologies
3. Kanter D Inside Barcelona: AMDs next generation. Real World Technologies
4. Le HQ, Starke WJ, Fields JS, OConnell FP, Nguyen DQ, Ronchetti BJ, Sauer WM, Schwarz EM, Vaden MT. IBM POWER6 Microarchitecture. IBM J Res Dev 51(6)
5. Kanter D Niagara II: the hydra returns. Real World Technologies
6. Team IBG Overview of the IBM blue gene/P project. IBM J Res Dev
7. Kahle J, Day M, Hofstee H, Johns C, Maeurer T, Shippy D (2005) Introduction to the Cell multiprocessor. IBM J Res Dev 49(4/5):589–604
8. Ainsworth TW, Pinkston TM (2007) Characterizing the Cell EIB on-chip network. IEEE Micro 27(5):6–14
9. Abellán JL, Fernández J, Acacio ME (2008) Characterizing the basic synchronization and communication operations in dual cell-based blades. In: Proceedings of the 8th International Conference on Computational Science, Krákow, Poland
10. Chen T, Raghavan R, Dale J, Iwata E (2005) Cell broadband engine architecture and its first implementation. Tech. rep.

11. Flachs B et al (2005) A streaming processor unit for a cell processor. In: Proceedings of international solid state circuits conference, San Fransisco, CA, USA
12. Gschwind M, Hofstee HP, Flachs B, Hopkins M, Watanabe Y, Yamazaki T (2006) Synergistic processing in Cell's multicore architecture. IEEE Micro 26(2):10–24
13. Kistler M, Perrone M, Petrini F (2006) Cell processor interconnection network: built for speed. IEEE Micro 25(3):2–15
14. IBM Systems and Technology Group (2007) Cell broadband engine programming tutorial version 2.1 (March 2007)
15. IBM Systems and Technology Group (2008) Programming the Cell broadband engine architecture: examples and best practices. http://www.redbooks.ibm.com/redbooks/pdfs/sg247575.pdf
16. IBM Systems and Technology Group (2007) SPE runtime management library version 2.2 (September 2007)
17. IBM Systems and Technology Group (2007) C/C++ language extensions for cell BroadBand engine architecture V2.4 (March 2007)
18. IBM Systems and Technology Group (2007) Cell broadband engine SDK libraries version 3.0 (October 2007)
19. IBM Systems and Technology Group (2007) Cell broadband engine software development toolkit (SDK) installation guide version 3.0 (October 2007)
20. Varbanescu AL, Sips H, Ross KA, Liu Q, Natsev AP, Smith JR, Liu L-K (2009) Evaluating application mapping scenarios on the Cell-B.E. Concurr Comput Pract Experience 21(1):85–100
21. Gaona E, Fernández J, Acacio ME (2009) Fast and efficient synchronization and communication collective primitives for dual Cell-based blades, Tech. rep.
22. Schiller A, Sutmann G, Yang L (2008) A fast wavelet based implementation to calculate Coulomb potentials on the Cell/B.E. In: Proceedings of the 10th IEEE international conference on high performance computing and communications, Dalian, China
23. Kang S, Bader D (2008) Optimizing discrete wavelet transform on the Cell broadband engine. In: Proceedings of the 12th annual high performance embedded computing workshop, Lexington, MA

José L. Abellán received the M.S. degree in Computer Science from the Univesidad de Murcia, Spain, in 2007. He joined the Computer Engineering Department, Universidad de Murcia, in 2007, where he is currently a Ph.D. student. His research interests include multicore architectures, parallel programming models, load balancing, and task scheduling.

Juan Fernández received the Ph.D. in Computer Science from the Universidad de Murcia, Spain, in 2005. He worked as a graduate research assistant at Los Alamos National Laboratory from 2001 to 2003, and as a postdoc at Pacific Northwest National Laboratory in 2006. He is currently an associate professor at the Computer Engineering Department of the Universidad de Murcia. His research interests include cluster computing, high-performance networking, multicore programming, and architecture.

Manuel E. Acacio received the M.S. and Ph.D. degrees in computer science from the Universidad de Murcia, Spain, in 1998 and 2003, respectively. He joined the Computer Engineering Department, Universidad de Murcia, in 1998, where he is currently an Associate Professor of computer architecture and technology. His research interests include prediction and speculation in multiprocessor memory systems, multiprocessor-on-a-chip architectures, power-aware cache-coherence protocol design, fault tolerance, and hardware transactional memory systems.