Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

Scheduling streaming applications on a complex multicore platform

Tudor David,
Mathias Jacquelin,
Loris Marchal

July 2010

Research Report N° 2010-25

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : lip@ens-lyon.fr




Scheduling streaming applications on a complex multicore platform

Tudor David, Mathias Jacquelin, Loris Marchal

July 2010

Abstract

In this report, we consider the problem of scheduling streaming applications described by complex task graphs on a heterogeneous multi-core platform, the IBM QS 22 platform, embedding two STI Cell BE processor. We first derive a complete computation and communication model of the platform, based on comprehensive benchmarks. Then, we use this model to express the problem of maximizing the throughput of a streaming application on this platform. Although the problem is proven NP-complete, we present an optimal solution based on mixed linear programming. We also propose simpler scheduling heuristics to compute mapping of the application task-graph on the platform. We then come back to the platform, and propose a scheduling software to deploy streaming applications on this platform. This allows us to thoroughly test our scheduling strategies on the real platform. We thus show that we are able to achieve a good speed-up, either with the mixed linear programming solution, or using involved scheduling heuristics.

Keywords: Scheduling, multicore processor, streaming application, Cell processor.

Résumé

Dans ce rapport, nous nous intéressons au problème de l'ordonnancement d'une application de flux décrite par un graphe de tâche sur une plate-forme multi-cœur hétérogène, le QS 22 d'IBM, qui embarque deux processeurs Cell. Nous mettons d'abord au point un modèle de calcul et de communication complet de la plate-forme, en nous fondant sur des tests de performances extensifs. Ensuite, nous utilisons ce modèle pour exprimer le problème d'optimisation du débit d'une application de flux sur cette plate-forme. Nous montrons que ce problème est NP-complet, et présentons une solution optimale utilisant la programmation linéaire mixte. Nous proposons ensuite des heuristiques de placement plus simples pour déterminer comment distribuer les tâches de l'application sur la plate-forme. Nous présentons également un canevas logiciel pour exécuter une telle application sur cette plate-forme, qui permet de tester les différentes stratégies de placement proposées. Nous montrons ainsi que nous sommes capables d'atteindre des performances intéressantes, soit avec la stratégie utilisant la programmation linéaire, soit avec des heuristiques de placement élaborées.

Mots-clés: Ordonnancement, processeur multi-cœur, application de flux, processeur Cell.

1 Introduction

The last decade has seen the arrival of multi-core processors in every computer and electronic device, from the personal computer to the high-performance computing cluster. Nowadays, heterogeneous multi-core processor are emerging. Future processors are likely to embed several special-purpose cores –like networking or graphic cores– together with general cores, in order to tackle problems like heat dissipation, computing capacity or power consumption. Deploying an application on this kind of platform becomes a challenging task due to the increasing heterogeneity.

Heterogeneous computing platforms such as Grids have been available for a decade or two. However, the heterogeneity is now likely to exist at a much smaller scale, that is within a single machine, or even a single processor. Major actors of the CPU industry are already planning to include a GPU core to their multi-core processors [2]. Classical processors are also often provided with an accelerator (like GPUs, graphics processing units), or with processors dedicated to special computations (like ClearSpeed [8] or Mercury cards [22]), thus resulting in a heterogeneous platform. The best example is probably the IBM RoadRunner, the first supercomputer to break the petaflop barrier, which is composed of an heterogeneous collection of classical AMD Opteron processors and Cell processors.

The STI Cell BE processor is an example of such an heterogeneous architecture, since it embeds both a PowerPC processing unit, and up to eight simpler cores dedicated to vectorial computing. This processor has been used by IBM to design machines dedicated to high performance computing, like the Bladecenter QS 22. We have chosen to focus our study on this platform because the Cell processor is nowadays widely available and affordable, and it is to our mind a good example of future heterogeneous processors.

Deploying an application on such a heterogeneous platform is not an easy task, especially when the application is not purely data-parallel. In this work, we focus on applications that exhibit some regularity, so that we can design efficient static scheduling solutions. We thus concentrate our work on streaming applications. These applications usually concern multimedia stream processing, like video edition software, web radios or Video On Demand applications [28, 17]. However, streaming applications also exist in other domains, like real time data encryption applications, or routing software, which are for example required to manage mobile communication networks [27]. A stream is a sequence of data that have to go through several processing tasks. The application is generally structured as a directed acyclic task graph, ranging from a simple chain of tasks to a more complex structure, as illustrated in the following.

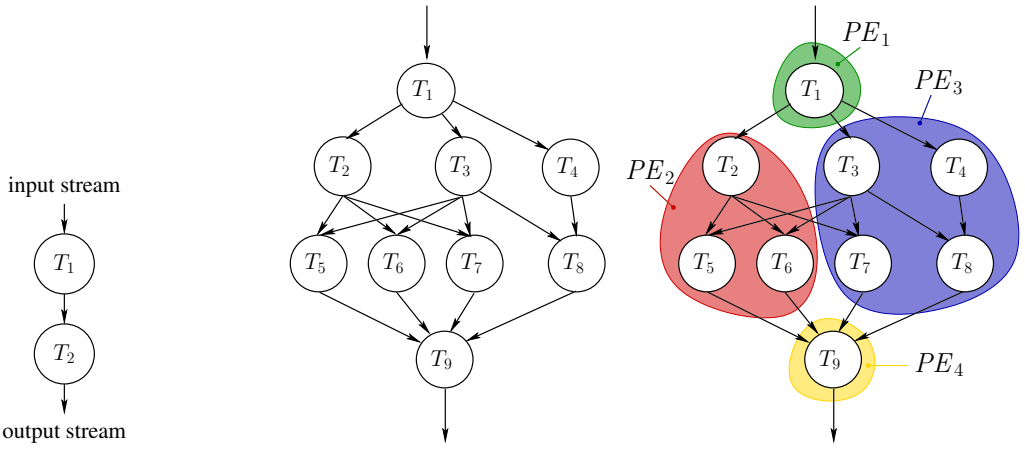
To process a streaming application on a heterogeneous platform, we have to decide which tasks will be processed onto which processing elements, that is, to find a mapping of the tasks onto the platform. This is a complex problem since we have to take platform heterogeneity, task computing requirements, and communication volume into account. The objective is to optimize the *throughput* of the application: for example in the case of a video stream, we are looking for a solution that maximizes the number of images processed per time-unit.

Several streaming solutions have already been developed or adapted for the Cell processor. DataCutter-Lite [16] is an adaptation of the DataCutter framework for the Cell processor, but it is limited to simple streaming applications described as linear chains, so it cannot deal with complex task graphs. StreamIt [15, 25] is a language developed to model streaming applications; a version of the Streamit compiler has been developed for the Cell processor, however it does not allow the user to specify the mapping of the application, and thus to precisely control the application. Some other frameworks allow to handle communications and are rather dedicated to matrix operations, like ALF (part of the IBM Software Kit for Multicore Acceleration [18]), Sequoia [11], CellSs [6] or BlockLib [1].

2 General framework and context

Streaming applications may be complex: for example, when organized as a task graph, a simple Vocoder audio filter can be decomposed in 140 tasks. All the data of the input stream must be processed by this task graph in a pipeline fashion. On a parallel platform, we have to decide which processing element will process each task. In order to optimize the performance of the application, which is usually evaluated using its throughput, we have take into account the computation capabilities of each resource, as well as the incurred communication overhead between processing elements. The platform we target in this study is a heterogeneous multi-core processor, which adds to the previous constraints a number of specific limitations (limited memory size on some nodes, specific communication constraints, etc.). Thus, the optimization problem corresponding to the throughput maximization is a complex duty. However, the typical run time of a streaming application is long: a video encoder is likely to run for at least several minutes, and probably several hours. Besides, we target a dedicated environment, with stable run time conditions. Thus, it is worth taking additional time to optimize the application throughput.

In previous work, we have developed a framework to schedule large instance of similar jobs, known as steady-state scheduling [5]. It aims at maximizing the throughput, that is, the number of jobs processed per time-unit. In the case of a streaming application like a video filter, a job may well represent the complete processing of an image of the stream. Steady-state scheduling is able to deal with complex jobs, described as directed acyclic graphs (DAG) of tasks. The nodes of this graph are the tasks of the applications, denoted by $T_1, \ldots, T_n$, whereas edges represent dependencies between tasks as well as the data associated to the dependencies: $D_{k,l}$ is a data produced by task T_k and needed to process T_l. Figure 1(a) presents the task graph of a simplistic stream application: the stream goes through two successive filters. Applications may be much more complex, as depicted in Figure 1(b). Computing resources consist in several processing elements $PE_1, \ldots, PE_p$, connected by communication links. We usually adopt the general *unrelated* computation model: the processing time of a task T_k on a processing element PE_i, denoted by $w_{PE_i}(T_k)$ is not necessarily function of the processing element speed, since some processing elements may be faster for some tasks and slower for some others. Communication limitations are taken into account following one of the existing communication models, usually the one-port or bounded multiport models.



(a) Simple streaming application. (b) Other streaming application. (c) A possible mapping.

Figure 1: Applications and mapping.

The application consists of a single task graph, but all data sets of the stream must be processed following this task graphs. This results in a large number of copies, or instances, of each tasks. Consider the processing of the first data set (e.g., the first image of a video stream) for the application described in Figure 1(b). A possible *mapping* of all these tasks to processing elements is described on Figure 1(c), which details on which processing element each task will be computed. For the following data sets, there is two possibilities. A possibility is that all data sets use the same mapping, that it, on this example, all tasks T_1 will be processed by processing element PE_1, etc. In this case, a single mapping is used for all instances. Another possibility is that some instances may be processed using different mappings. This multiple-mapping solution allows to split the processing of T_1 among different processors, and thus can usually achieve a larger throughput. However, the control overhead for a multiple-mapping solution is much larger than for a single-mapping solution.

The problem of optimizing the steady-state throughput of an application on heterogeneous platform has been studied in the context of Grid computing. It has been proven that the general problem, with multiple mappings, is NP-complete on general graphs, but can be solved in polynomial time provided that the application graph has a limited depth [4]. Nevertheless, the control policy that would be needed by a multiple-mapping solution would be far to complex for a heterogeneous multi-core processor: each processor would need to process a large number of different tasks, store a large number of temporary files, and route messages to multiple destinations depending on their type and instance index. In particular, the limited memory of the Synergistic Processing Element makes it impossible to implement such a complex solution.

Thus, single-mapping solutions are more suited for the targeted platform. These solution have also been studied in the context of Grid computing. We have proven that the general solution is NP-complete, but can be computed using a mixed integer linear program; several heuristics have also been propose to compute an efficient mapping of the application onto the platform [12]. The present paper aims at studying the ability of adapting the former solution for heterogeneous multi-core processors, and illustrates this on the Cell processor and the QS 22. The task is challenging as it requires to solve the following problems:

- Derive a model of the platform that both allows to accurately predict the behavior of the application, and to compute an efficient mapping using steady-state scheduling techniques.
- As far as possible, adapt the solutions presented in [12] for the obtained model: optimal solution via mixed linear programming and heuristics.
- Develop a light but efficient scheduling software that allows to implement the proposed scheduling policies and test them in real life conditions.

3 Adaptation of the scheduling framework to the QS 22 architecture

In this section, we present the adaptation of the steady-state scheduling framework in order to cope with streaming applications on the Cell processor and the QS 22 platform. As outlined, the main changes concern the computing platform. We first briefly present the architecture, and we propose a model for this platform based on communication benchmarks. Then, we detail the application model and some specificities related to stream applications. Based on these models, we adapt the optimal scheduling policy from [12] using mixed linear programing.

3.1 Platform description and model

We detail here the multi-core processor used in this study, and the platform which embeds this processor. In order to adapt our scheduling framework to this platform, we need a communication model which is able to predict the time taken to perform a set of transfer between processing elements. As outlined below, no such model is available to the best of our knowledge. Thus, we perform some communications benchmarks on the QS 22, and we derive a model which is suitable for our study.

3.1.1 Description of the QS 22 architecture

The IBM Bladecenter QS 22 is a bi-processor platform embedding two Cell processors and up to 32 GB of DDR2 memory [23]. This platform offers a high computing power for a limited power consumption. The QS 22 has already been used for high performance computing: it is the core of IBM RoadRunner platform, leader of the Top500 from November 2008 to June 2009, the first computer to reach one petaflops [3].

As mentioned in the introduction, the Cell processor is a heterogeneous multi-core processor. It has jointly been developed by Sony Computer Entertainment, Toshiba, and IBM [19], and embeds the following components:

- **Power Processing Element (PPE) core.** This two-way multi-threaded core follows the Power ISA 2.03 standard. Its main role is to control the other cores, and to be used by the operating system due to its similarity with existing Power processors.
- **Synergistic Processing Elements (SPE) cores.** These cores constitute the main innovation of the Cell processor and are small 128-bit RISC processors specialized in floating point, SIMD operations. These differences induce that some tasks are by far faster when processed on a SPE, while some other tasks can be slower. Each SPE has its own local memory (called *local store*) of size $LS = 256$ kB, and can access other local stores and main memory only through explicit asynchronous DMA calls.
- **Main memory.** Only PPEs have a transparent access to main memory. The dedicated memory controller is integrated in the Cell processor and allows a fast access to the requested data. Since this memory is by far larger than the SPE's local stores, we do not consider its limited size as a constraint for the mapping of the application. The memory interface supports a total bandwidth of $bw = 25$ GB/s for read and writes combined.
- **Element Interconnect Bus (EIB).** This bus links all parts of the Cell processor to each other. It is composed of 4 unidirectional rings, 2 of them in each direction. The EIB has an aggregated bandwidth $BW = 204.8$ GB/s, and each component is connected to the EIB through a bidirectional interface, with a bandwidth $bw = 25$ GB/s in each direction. Several restrictions apply on the EIB and its underlying rings:
  - A transfer cannot be scheduled on a given ring if data has to travel more than halfway around that ring
  - Only one transfer can take place on a given portion of a ring at a time
  - At most 3 non-overlapping transfers can take place simultaneously on a ring.
- **FLEXIO Interfaces.** The Cell processor has two FLEXIO interfaces ($IOIF_0$ and $IOIF_1$). These interfaces are used to communicate with other devices. Each of these interfaces offers an input bandwidth of $bwio_{\text{in}} = 26$ GB/s and an output bandwidth of $bwio_{\text{out}} = 36.4$ GB/s.

Both Cell processors of the QS 22 are directly interconnected through their respective $IOIF_0$ interface.We will denote the first processor by $Cell_0$, and the second processor $Cell_1$. Each of these processors is connected to a bank of DDR memory; these banks are denoted by $Memory_0$ and $Memory_1$. A processor can access the other bank of memory, but then experiences a non-uniform memory access time.

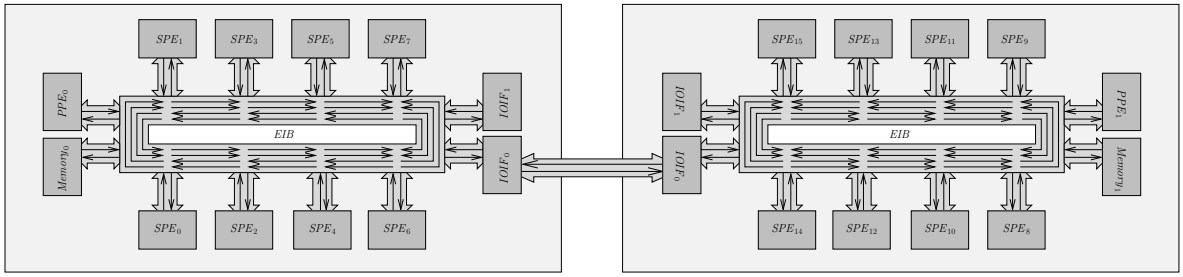


Figure 2: Schematic view of the QS 22.

Since we are considering a bi-processor Cell platform, there are $n_P = 2$ PPE cores, denoted by PPE_0 and PPE_1. Furthermore, each Cell processor embedded in the QS 22 has eight SPEs, we thus consider $n_S = 16$ SPEs in our model, denoted by $SPE_0, \ldots, SPE_{15}$ ($SPE_0, \ldots, SPE_7$ belong to the first processor). To simplify the following computations, we gather all processing elements under the same notation PE_i, so that the set of PPEs is $\{PE_0, PE_1\}$, while $\{PE_2, \ldots, PE_{17}\}$ is the set of SPEs (again, PE_2 to PE_9 are the SPEs of the first processor). Let n be the total number of processing elements, i.e., $n = n_P + n_S = 18$. All processing elements and their interconnection are depicted on Figure 2. We have two classes of processing elements, which fall under the *unrelated* computation model: a PPE can be fast for a given task T_k and slow for another one T_l, while a SPE can be slower for T_k but faster for T_l. Each core owns a dedicated communication interface (a DMA engine for the SPEs and a memory controller for the PPEs), and communications can thus be overlapped with computations.

3.1.2 Benchmarks and model of the QS 22

We need a precise performance model for the communication within the Bladecenter QS 22. More specifically, we want to predict the time needed to perform any pattern of communication among the computing elements (PPEs and SPEs). However, we do not need this model to be precise enough to predict the behavior of each packet (DMA request, etc.) in the communications elements. We want a simple (thus tractable) and yet accurate enough model. To the best of our knowledge, there does not exists such a model for irregular communication patterns on the QS 22. Existing studies generally focus on the aggregated bandwidth obtained with regular patterns, and are limited to a single Cell processor [20]. This is why, in addition to the documentation provided by the manufacturer, we perform communication benchmarks. We start by simple scenarios with communication patterns involving only two communication elements, and then move to more complex scenarios to evaluate communication contention.

To do so, we have developed a communication benchmark tool, which is able to perform and time any pattern of communication: it launches (and pins) threads on all the computing elements involved in the communication pattern, and make them read/write the right amount of data from/to a distant memory or local store. For all these benchmarks, each test is repeated a number of times (usually 10 times), and only the average performance is reported. We report the execution times of

the experiments, either in nanoseconds (ns) or in cycles: since the time-base frequency of the QS 22 Bladecenter is 26.664 Mhz, a cycle is about 37.5 ns long.

In this section, we will first present the timing result for single communications. Then, since we need to understand communication contention, we present the results when performing several simultaneous communications. Thanks to these benchmarks, we are finally able to propose a complete and tractable model for communications on the QS 22.

Note that for any communication between two computing elements PE_i and PE_j, we have to choose between two alternatives: (i) PE_j reads data from PE_i local memory, or (ii) PE_i writes data into PE_j local memory. In order to keep our scheduling framework simple, we have chosen to implement only one of these alternatives. When performing the tests presented in this section, we have noticed a slightly better performance for read operations. Thus, we focus only on read operations: the following benchmarks are presented only for read operations, and we will use only these operations when implementing the scheduling framework. However, write operations would most of the time behave similarly.

Single transfer performances. First, we deal with the simplest kind of transfers: SPE to SPE data transfers. In this test, we choose a pair of SPE in the same Cell, and we make one read in the local store of the other, by issuing the corresponding DMA call. We present in Figure 3.1.2 the duration of such a transfer for various data sizes, ranging from 8 bytes, to 16kB (the maximum size for a single transfer). When the size of the data is small, the duration is 112 ns (3 cycles), which we consider as the latency for this type of transfers. For larger data sizes, the execution time increases quasi-linearly. For 16 kB, the bandwidth obtained (around 25 GB/s) is close the theoretical one.

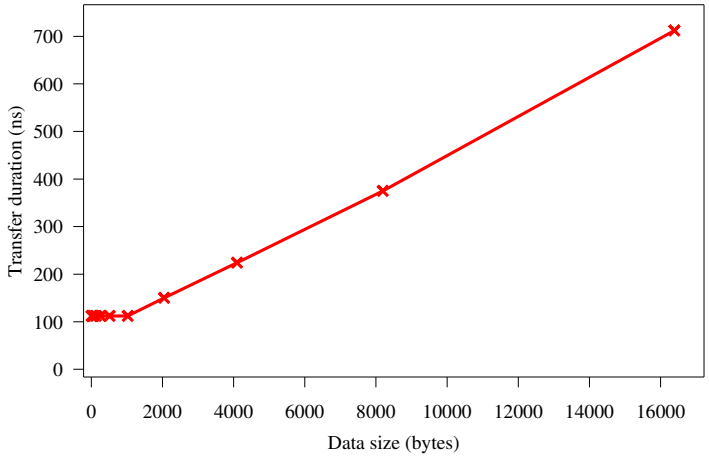


Figure 3: Data transfer time between two SPE from the same Cell

When performing the same tests among SPEs from both Cells of the QS 22, the results are not the same. The results can be summarized as follows: when a SPE from $Cell_0$ reads from a local store located in $Cell_1$, the average latency of the transfer is 444 ns, and the average bandwidth is 4.91 GB/s. When on the contrary, a SPE from $Cell_1$ reads some data in local store of $Cell_0$, then the average latency is 454 ns, and the average bandwidth is 3.38 GB/s (complete results for this benchmark may be found in Appendix A). This little asymmetry can explained by the fact that both Cells do not play the same role: the data arbiter and address concentrators of $Cell_0$ act as masters for communications involving both Cells, which explains why communications originating from different Cells are handled differently.

When we involve both Cells in the data transfer, the performances fall again. We make an SPE of $Cell_0$ read data from a local store in $Cell_1$, and measure a latency of 12 cycles (448 ns) and a bandwidth of 5 GB/s. For the converse scenario when a SPE of $Cell_1$ reads from a local store of $Cell_0$, then the latency is similar, but the bandwidth is only 3 GB/s. We notice that all communications initiated by $Cell_1$ takes longer, because the DMA request must go through the data arbiter on $Cell_0$ before being actually performed.

Finally, we have to study the special case of a PPE reading from a SPE's local store. This transfer is particular, since it involves multiple transfers: in order to perform the communication, the PPE adds an DMA instruction in the transfer queue of the corresponding SPE. Then, the DMA engine of the SPE reads this instruction and perform the copy into the main memory. Then, in order for the data to be loaded available for the PPE, it is loaded in its private cache. Due to this multiple transfers, and to the fact that DMA instructions issued by the PPE have a smaller priority than the local instructions, the latency of these communications is particularly large, about 300 cycles (11250 ns). With such a high latency, the effect of the size of the data on the transfer time is almost negligible, which makes it difficult to compute a maximal bandwidth.

Concurrent transfers. The previous benchmarks gives us an insight of the performances for a single transfer. However, when performing several transfers at the same time, other effect may appear: concurrent transfers may be able to reach a larger aggregated bandwidth, but each of them may also suffers from the contention and have its bandwidth reduced. In the following, we try to both exhibit the maximum bandwidth of each connecting element, and to understand how the available bandwidth is shared between concurrent flows.

In a first step, we try to see if the bandwidths measured for single transfers and presented above can be increased when using several transfers instead of one. For SPE-SPE transfers, this is not the case: when performing several read operations on different SPEs from the same local store (on the same Cell), the 25 GB/s limitation makes it impossible to go beyond the single transfer performances.

The Element Interconnect Bus (EIB) has a theoretical capacity of 204.8 GB/s. However, its structure consisting of two bi-directional rings makes certain communication pattern more efficient than others. For patterns which are made of short-distance transfers, and for which can be performed without overlap within one ring, we can almost get 200GB/s out of the EIB. On the other hand, if we concurrently schedule several transfers between elements that are far away from each other, the whole pattern cannot be scheduled on the rings without overlapping some transfers, and the bandwidth is reduced substantially.

We illustrate this phenomenon by two examples. Consider the layout of the SPEs as depicted by Figure 2. In the first scenario, we perform the following transfers ($SPE_i \leftarrow SPE_j$ means "SPE_i reads from SPE_j"): $SPE_1 \leftarrow SPE_3$, $SPE_3 \leftarrow SPE_1$, $SPE_5 \leftarrow SPE_7$, $SPE_7 \leftarrow SPE_5$, $SPE_0 \leftarrow SPE_2$, $SPE_2 \leftarrow SPE_0$, $SPE_4 \leftarrow SPE_6$, and $SPE_6 \leftarrow SPE_4$. Then, the aggregated bandwidth is 200 GB/s, that is all transfers get their maximal bandwidth (25 GB/s). Then, we perform another transfer pattern: $SPE_0 \leftarrow SPE_7$, $SPE_7 \leftarrow SPE_0$, $SPE_1 \leftarrow SPE_6$, $SPE_6 \leftarrow SPE_1$, $SPE_2 \leftarrow SPE_5$, $SPE_5 \leftarrow SPE_2$, $SPE_3 \leftarrow SPE_4$, and $SPE_4 \leftarrow SPE_3$. In this case, the aggregated bandwidth is only 80 GB/s, because of the large overlap between all transfers. However, these cases are extreme ones; Figure 4 shows the distribution of the aggregated bandwidth when we run one hundred of random transfer patterns. The average bandwidth is 149 GB/s.

We have measured earlier that the bandwidth of a single transfer between the two Cells (through the FlexIO) was limited to either 4.91 GB/s or 3.38 GB/s depending on the direction of the transfer. When we aggregate several transfer through the FlexIO, a larger bandwidth can be obtained:

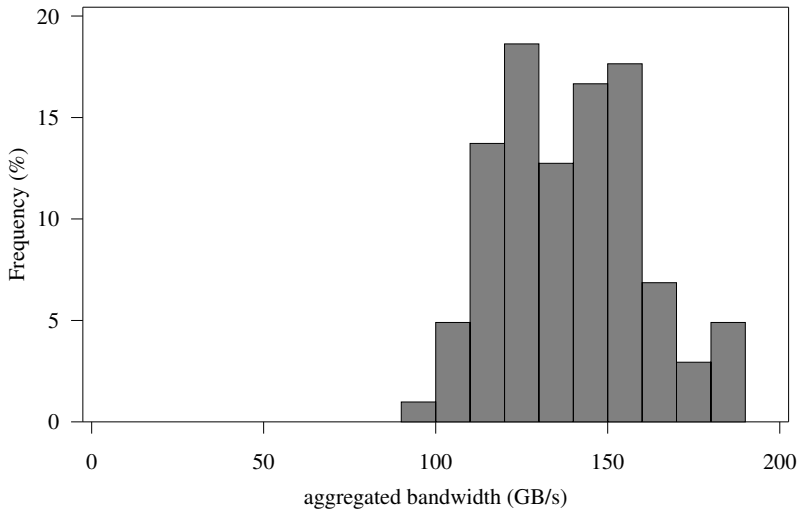


Figure 4: Bandwidth distribution

when several SPEs from $Cell_0$ reads from local stores on $Cell_1$, the maximal cumulated bandwidth is 13 GB/s, where as it is 11.5 GB/s for the converse scenario. Finally, we have measured the overall bandwidth of the FlexIO, when performing transfers in both direction. The cumulated bandwidth is only 19 GB/s, and it is interesting to note that all transfers initiated by $Cell_0$ gets an aggregated bandwidth of 10 GB/s, while all transfers initiated by $Cell_1$ gets 9 GB/s. Excepted for the last scenario, bandwidth is always shared quite fairly among the different flows.

Toward a communication model for the QS 22. We are now able to propose a communication model of the Cell processor and its integration in the Bladecenter QS 22. This model has to be a trade-off between accuracy and tractability. Our ultimate goal is to estimate the time needed to perform a whole pattern of transfers. The pattern is described by the amount of data exchanged between any pair of processing elements. Given two processing elements PE_i and PE_j, we denote by $data_{i,j}$ the size (in GB) of the data which is read by PE_j from PE_i's local memory. If PE_i is a SPE, its local memory is naturally its local store, and for the sake of simplicity, we consider that the local memory of a PPE is the main memory of the same Cell. Note that the quantity $data_{i,j}$ may well be zero if no communication happens between PE_i and PE_j. We denote by $\mathcal{T}$ the time needed to complete the whole transfer pattern.

We will use this model to optimize the schedule of a stream of task graphs, and this purpose has an impact on the model design. In our scheduling framework, we try to overlap communications by computations: a computing resource process a tasks while it receives the data for the next task, as outlined later when we discuss about steady-state scheduling. Thus, we are more concerned by the capacity of the interconnection network, and the bandwidth of different resources underlined above, than by the latencies. In the following, we approximate the total completion time $\mathcal{T}$ by the maximum occupation time of all communication resources. For sake of simplicity, we choose to express everything as linear formula of the size of the data. Of course, this choice limits the accuracy of our model; some behavior of the Cell are hard to model and do not follow linear laws. This is especially true in the case of multiple transfers. However, a linear model renders the optimization of the schedule tractable. A more complex model would result in more complex, thus harder, scheduling problem. Instead, we consider that a linear model is a very good trade-off between the simplicity of the solution

and its accuracy.

In the following, we denote by $chip(i)$ the index of the Cell where processing element PE_i lies ($chip(i)$ = 0 or 1):

$$chip(i) = \begin{cases} 0 & \text{if } i = 0 \text{ or } 2 \leq i \leq 9 \\ 1 & \text{if } i = 1 \text{ or } 10 \leq i \leq 17 \end{cases}$$

We first consider the capacity of the input port of every processing element (either PPE or SPE), which is limited to 25 GB/s:

$$\forall PE_i, \quad \sum_{j=0}^{17} \delta_{i,j} \times \frac{1}{25} \leq \mathcal{T} \tag{1}$$

The output capacity of the processing elements and the main memory is also limited to 25 GB/s.

$$\forall PE_i, \quad \sum_{j=0}^{17} \delta_{j,i} \times \frac{1}{25} \leq \mathcal{T} \tag{2}$$

When a PPE is performing a read operation, the maximum bandwidth of this transfer cannot exceed 2 GB/s.

$$\forall PE_i \text{ such that } 0 \leq i \leq 1), \quad \forall PE_j, \delta_{i,j} \times \frac{1}{2} \leq \mathcal{T} \tag{3}$$

The average aggregate capacity of the EIB of one Cell is limited to 149 GB/s.

$$\forall Cell_k, \quad \sum_{\substack{0 \leq i,j \leq 17 \text{ with} \\ chip(i)=k \text{ or } chip(j)=k}} \delta_{i,j} \times \frac{1}{149} \leq \mathcal{T} \tag{4}$$

When a SPE of $Cell_0$ reads from a local memory in $Cell_1$, the bandwidth of the transfer is to 4.91 GB/s.

$$\forall PE_i \text{ such that } chip(i) = 0, \quad \sum_{j,\ chip(j)=1} \delta_{j,i} \times \frac{1}{4.91} \leq \mathcal{T} \tag{5}$$

Similarly, we a SPE of $Cell_1$ reads from a local memory in $Cell_0$, the bandwidth is limited to 3.38 GB/s.

$$\forall PE_i \text{ such that } chip(i) = 1, \quad \sum_{j,\ chip(j)=0} \delta_{j,i} \times \frac{1}{3.38} \leq \mathcal{T} \tag{6}$$

On the whole, all processing elements of $Cell_0$ cannot read data from $Cell_1$ at a rate larger than 13 GB/s.

$$\sum_{\substack{PE_i, PE_j, \\ chip(i)=0 \text{and} chip(j)=1}} \delta_{j,i} \times \frac{1}{13} \leq \mathcal{T} \tag{7}$$

Similarly, all processing elements of $Cell_1$ cannot read data from $Cell_0$ at rate larger than 11.5 GB/s/

$$\sum_{\substack{PE_i, PE_j, \\ chip(i)=1 \text{and} chip(j)=0}} \delta_{j,i} \times \frac{1}{11.5} \leq \mathcal{T} \tag{8}$$

The overall capacity of the FlexIO between both Cells is limited to 19 GB/s.

$$\sum_{\substack{PE_i, PE_j \\ chip(i) \neq chip(j)}} \delta_{i,j} \times \frac{1}{19} \leq \mathcal{T} \tag{9}$$

We have tested this model on about one hundred random transfer patterns, comprising between 2 and 49 concurrent transfers. The accuracy of the model is presented on Figure 5, as the ratio between the theoretical transfer time and the experimental one. This figure shows that on average, the predicted communication time is close to the experimental one (the average absolute between the theoretical and the experimental time is 13%). We can also notice that our model is slightly pessimistic (the average ratio is 0.89). This is because the congestion constraints presented above correspond to scenarios where all transfers must go through the same interface, which is unlikely. In practice, communications are scattered among all communication units, so that the total time for communications is slightly less than what is predicted by the model. However, we choose to keep our conservative model, to prevent an excessive usage of the communications.

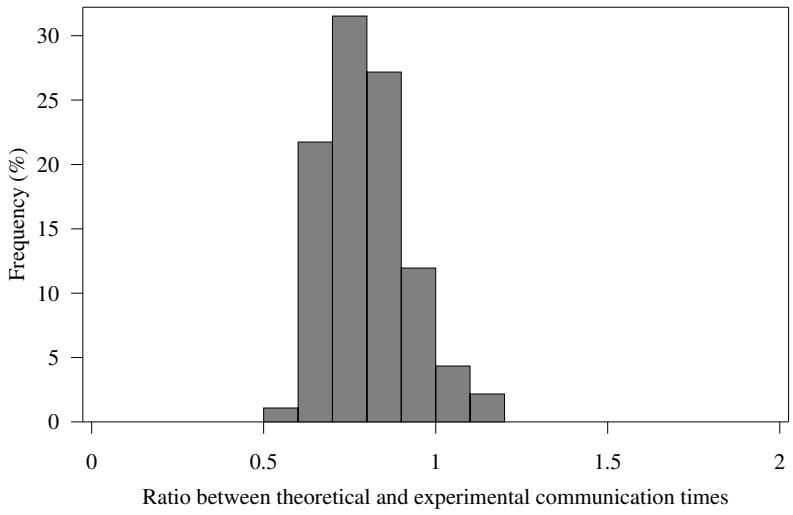


Figure 5: Model accuracy

3.1.3 Communications and DMA calls

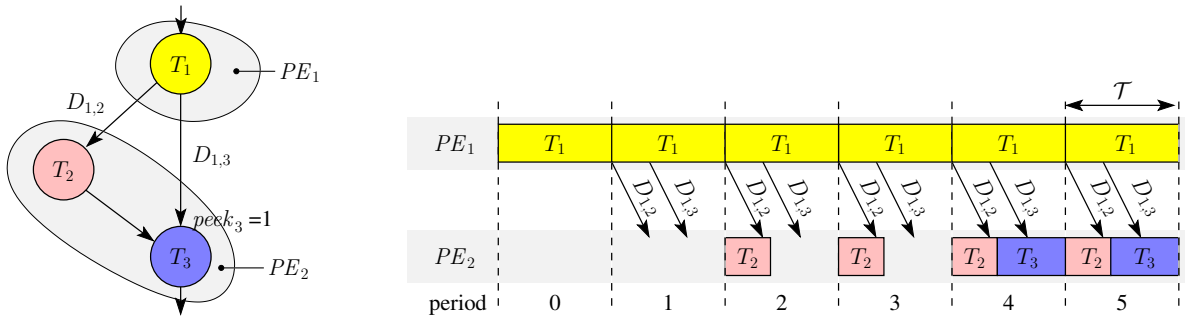
The Cell processor has very specific constraints, especially on communications between cores. Even if SPEs are able to receive and send data while they are doing some computation, they are not multi-threaded. The computation must be interrupted to initiate a communication (but the computation is resumed immediately after the initialization of the communication). Due to the absence of auto-interruption mechanism, the thread running on each SPE has regularly to suspend its computation and check the status of current DMA calls. Moreover, the DMA stack on each SPE has a limited size. A SPE can issue at most 16 simultaneous DMA calls, and can handle at most 8 simultaneous DMA calls issued by the PPEs. Furthermore, when building a steady-state schedule, we do not want to precisely order communications among processing elements. Indeed, such a task would require a lot of synchronizations. On the contrary, we assume that all the communications of a given period may happen simultaneously. These communications correspond to edges $D_{k,l}$ of the task graph when tasks T_k and T_l are not mapped on the same processing element. With the previous limitation on concurrent DMA calls, this induces a strong limitation on the mapping: each SPE is able to receive at most 16 different data, and to send at most 8 data to PPEs per period.

3.2 Mapping a streaming application on the Cell

Thanks to the model obtained in the previous section, we are now able to design an efficient strategy to map a streaming application on the target platform. We first recall how we model the application. We then detail some specificities of the implementation of a streaming application on the Cell processor.

3.2.1 Complete application model

As presented above, we target complex streaming applications, as the one depicted on Figure 1(b). These applications are commonly modeled with a Directed Acyclic Graph (DAG) $G_A = (V_A, E_A)$. The set V_A of nodes corresponds to tasks $T_1, \ldots, T_K$. The set E_A of edges models the dependencies between tasks, and the associated data: the edge from T_k to T_l is denoted by $D_{k,l}$. A data $D_{k,l}$, of size $data_{k,l}$ (in bytes), models a dependency between two task T_k and T_l, so that the processing of the ith instance of task T_l requires the data corresponding to the ith instance of data $D_{l,k}$ produced by T_k. Moreover, it may well be the case that T_l also requires the results of a few instances following the ith instance. In other words, T_l may need information on the near future (i.e., the next instances) before actually processing an instance. For example, this happens in video encoding softwares, when the program only encodes the difference between two images. We denote by $peek_k$ the number of such instances. More formally, instances $i, i+1, \ldots, i+peek_k$ of $D_{k,l}$ are needed to process the ith instance of T_l. This number of following instances is important not only when constructing the actual schedule and synchronizing the processing elements, but also when computing the mapping, because of the limited size of local memories holding temporary data.



(a) Application and mapping. $peek_1 = peek_2 = 0$ and $peek_3 = 1$

(b) Periodic schedule.

Figure 6: Mapping and schedule

Given an application, our goal is to determine the best mapping of the tasks onto the processing element. Once the mapping is chosen, a periodic schedule is automatically constructed as illustrated on Figure 6. After a few periods for initialization, each processing element enters a *steady state* phase. During this phase, a processing element in charge of a task T_k has to simultaneously perform three operations. First it has to process one instance of T_k. It has to send the result $D_{k,l}$ of the previous instance to the processing element in charge of each successor task T_l. Finally, it has to receive the data $D_{j,k}$ of the next instance from the processing element in charge of each predecessor task T_j. The exact construction of this periodic schedule is detailed in [4] for general mappings. In our case, the construction of the schedule is quite straightforward: a processing element PE_i in charge of a task T_k simply processes it as soon as its input data is available. In other words, as soon as PE_i has received the data for the current instance and potentially the $peek_k$ following ones. For the sake

of simplicity, we do not consider the precise ordering of communications within a period. On the contrary, we assume that all communications can happen simultaneously in one period as soon as the communication constraints expressed in the previous section are satisfied.

3.2.2 Determining buffer sizes

Since SPEs have only 256 kB of local store, memory constraints on the mapping are tight. We need to precisely model them by computing the exact buffer sizes required by the application.

Mainly for technical reasons, the code of the whole application is replicated in the local stores of SPEs (of limited size LS) and in the memory shared by PPEs. We denote by $code$ the size of the code deployed on each SPE, so that the available memory for buffers is $LS - code$. A SPE processing a task T_k has to devote a part of its memory to the buffers dedicated to hold incoming data $D_{j,k}$, as well as for outgoing data $D_{k,l}$. Note that both buffers have to be allocated into the SPE's memory even if one of the neighbor tasks T_j or T_l is mapped on the same SPE. In a future optimization, we could save memory by avoiding the duplication of buffers for neighbor tasks mapped on the same SPE.

As presented above, before computing an instance of a task T_k, a processing element has to receive all the corresponding data, that is the data $D_{j,k}$ produced by each predecessor task T_j, both for the current instance and for the $peek_k$ following instances. Moreover processing elements are not synchronized on the same instance. Thus, the results of several instances need to be stored during the execution. In order to compute the number of stored data, we first compute the index of the period in the schedule when the first instance of T_k is processed. The index of this period is denoted by $\mathit{firstPeriod}(T_k)$, and is expressed by:

$$\mathit{firstPeriod}(T_k) = \begin{cases} 0 & \text{if } T_k \text{ has no predecessor,} \\ \max\limits_{D_{j,k}} \left(\mathit{firstPeriod}(T_j)\right) + peek_k + 2 & \text{otherwise.} \end{cases}$$

All predecessors of an instance of task T_k are processed after $\max_{D_{j,k}} \left(\mathit{firstPeriod}(T_j)\right) + 1$ periods. We have also to wait for $peek_k$ additional periods if some following instances are needed. An additional period is added for the communication from the processing element handling the data, hence the result. By induction on the structure of the task graph, this allows to compute $\mathit{firstPeriod}$ for all tasks. For example, with the task graph and mapping described on Figure 6, we have $\mathit{firstPeriod}(1) = 0$, $\mathit{firstPeriod}(2) = 2$, and $\mathit{firstPeriod}(3) = 4$. Again, we could have avoided the additional period dedicated for communication when tasks are mapped on the same processor (e.g., we could have $\mathit{firstPeriod}(3) = 3$). However, we let this optimization as a future work to keep our scheduling framework simple.

Once the $\mathit{firstPeriod}(T_k)$ value of a task T_k is known, buffer sizes can be computed. For a given data $D_{k,l}$, the number of temporary instances of this data that have to be stored in the system if $\mathit{firstPeriod}(T_l) - \mathit{firstPeriod}(T_k)$. Thus, the size of the buffer needed to store this data is $temp_{k,l} = data_{k,l} \times (\mathit{firstPeriod}(T_l) - \mathit{firstPeriod}(T_k))$.

3.3 Optimal mapping through mixed linear programming

In this section, we present a mixed linear programming approach that allows to compute a mapping with optimal throughput. This study is adapted from [12], but takes into account the specific constraints of the QS 22 platform. The problem is expressed as a linear program where integer and rational variables coexist. Although the problem remains NP-complete, in practice, some software are

able to solve such linear programs [9]. Indeed, thanks to the limited number of processing elements in the QS 22, we are able to compute the optimal solution for task graphs of reasonable size (up to a few hundreds of tasks).

Our linear programming formulation makes use of both integer and rational variables. The integer variables are described below. They can only take values 0 or 1.

- α's variables which characterize where each task is processed: $\alpha_i^k = 1$ if and only if task T_k is mapped on processing element PE_i.
- β's variables which characterize the mapping of data transfers: $\beta_{i,j}^{k,l} = 1$ if and only if data $D_{k,l}$ is transferred from PE_i to PE_j (note that the same processing element may well handle both task if $i = j$).

Obviously, these variables are related. In particular, $\beta_{i,j}^{k,l} = \alpha_i^k \times \alpha_j^l$, but this redundancy allows us to express the problem as a set of linear constraints. The objective of the linear program is to minimize the duration $\mathcal{T}$ of a period, which corresponds to maximizing the throughput $\rho = 1/\mathcal{T}$. The constraints of the linear program are detailed below. Remember that processing elements $PE_0, \ldots, PE_{n_P-1}$ are PPEs whereas $PE_{n_P}, \ldots, PE_n$ are SPEs.

- α and β are integers.

$$\forall D_{k,l}, \forall PE_i \text{ and } PE_j, \qquad \alpha_i^k \in \{0,1\}, \beta_{i,j}^{k,l} \in \{0,1\} \tag{10}$$

- Each task is mapped on one and only one processing element.

$$\forall T_k, \qquad \sum_{i=0}^{n-1} \alpha_i^k = 1 \tag{11}$$

- The processing element computing a task holds all necessary input data.

$$\forall D_{k,l}, \forall j, 0 \leq j \leq n-1, \qquad \sum_{i=0}^{n-1} (\beta_{i,j}^{k,l}) \geq \alpha_j^l \tag{12}$$

- A processing element can send the output data of a task only if it processes the corresponding task.

$$\forall D_{k,l}, \forall i, 0 \leq i \leq n-1, \qquad \sum_{j=0}^{n-1} (\beta_{i,j}^{k,l}) \leq \alpha_i^k \tag{13}$$

- The computing time of each processing element (PPE or SPE) is no larger that $\mathcal{T}$.

$$\forall i, 0 \leq i < n_P, \qquad \sum_{T_k} (\alpha_i^k w_{PPE}(T_k)) \leq \mathcal{T} \tag{14}$$

$$\forall i, n_P \leq i < n, \qquad \sum_{T_k} (\alpha_i^k w_{SPE}(T_k)) \leq \mathcal{T} \tag{15}$$

- All temporary buffers allocated on the SPEs fit into their local stores.

$$\forall i, n_P \leq i < n, \qquad \sum_{T_k} \left(\alpha_i^k \left(\sum_{D_{k,l}} temp_{k,l} + \sum_{D_{l,k}} temp_{l,k} \right) \right) \leq LS - code \tag{16}$$

- A SPE can perform at most 16 simultaneous incoming DMA calls, and at most eight simultaneous DMA calls are issued by PPEs on each SPE.

$$\forall j, n_P \leq j < n, \qquad \sum_{0 \leq i < n, i \neq j} \sum_{D_{k,l}} \beta_{i,j}^{k,l} \leq 16 \tag{17}$$

$$\forall i, n_P \leq i < n, \qquad \sum_{0 \leq j < n_P} \sum_{D_{k,l}} \beta_{i,j}^{k,l} \leq 8 \tag{18}$$

- The amount of data communicated among processing elements during one period can be deduced from β.

$$\forall PE_i \text{ and } PE_j, \qquad \delta_{i,j} = \sum_{D_{k,l}} \beta_{i,j}^{k,l} data_{k,l} \tag{19}$$

Using this definition, Equations (1), (2), (3), (4), (5), (6), (7), (8), and (9) ensure that all communications are performed within period $\mathcal{T}$.

The size of the linear program depends on the size of the task graphs: it counts $n\,|V_A|+n^2\,|E_A|+1$ variables.

We denote $\rho_{opt} = 1/\mathcal{T}_{opt}$, where $\mathcal{T}_{opt}$ is the value of $\mathcal{T}$ in any optimal solution of the linear program. This linear program computes a mapping of the application that reaches the maximum achievable throughput. By construction, α is a valid mapping, and all possible mappings can be described as α and β variables, which obey the constraints of the linear program.

4 Low-complexity heuristics

For large task graphs, solving the linear program may not be possible, or may take a very long time. This is why we propose in this section heuristics with lower complexity to find a mapping of the task graph on the QS 22. As explained in Section 3.2.1, the schedule which takes into account precedence constraints, naturally derives from the mapping. We start with straightforward greedy mapping algorithms, and then move to more involved strategies.

We recall a few useful notations for this section: $w_{PPE}(T_k)$ denotes the processing time of task T_k on any PPE, while $w_{SPE}(T_k)$ is its processing time on a SPE. For data dependency $D_{k,l}$ between tasks T_k and T_l, we have computed $temp_{k,l}$, the number of temporary data that must be stored in steady state. Thus, we can compute the overall buffer capacity needed for a given task T_k, which corresponds to the buffers for all incoming and outgoing data: $buffers[k] = \sum_{j \neq k}(temp_{j,k} + temp_{k,j})$.

4.1 Communication-unaware load-balancing heuristic

The first heuristic is a greedy load-balancing strategy, which is only concerned by computation, and does not take communication into account. Usually, such algorithms offer reasonable solutions while having a low complexity.

This strategy first consider SPEs, since they hold the major part of the processing power of the platform. The tasks are sorted according to their affinity with SPEs, and the tasks with larger affinity are load-balanced among SPEs. Tasks that do not fit in the local store of SPEs are mapped on PPEs. After this first step, some PPE might be underutilized. In this case, we then move some tasks which have affinity with PPEs from SPEs to PPEs, until the load is globally balanced. This heuristic will be referred to as GREEDY in the following, and is detailed in Algorithm 1.

4.2 Prerequisites for communication-aware heuristics

When considering complex task graphs, handling communications while mapping tasks onto processors is a hard task. This is especially true on the QS 22, which has several heterogeneous communication links, and even within each of its Cell processors. The previous heuristic ignore communications for the sake of simplicity. However, being aware of communications while mapping tasks onto processing element is crucial for performance. We present here a common framework to handle communications in our heuristics.

4.2.1 Partitioning the Cell in clusters of processing elements

As presented above, the QS 22 is made of several heterogeneous processing elements. In order to handle communications, we first simplify its architecture and aggregate these processing elements into coarse-grain groups sharing common characteristics.

Algorithm 1: GREEDY(G_A)

foreach T_k **do** $affinity(T_k) \leftarrow \frac{w_{SPE}(T_k)}{w_{PPE}(T_k)}$
foreach SPE_i **do**
  $workload[SPE_i] \leftarrow 0$
  $memload[SPE_i] \leftarrow 0$
foreach PPE_j **do**
  $workload[PPE_j] \leftarrow 0$
foreach T_k *in non decreasing order of affinity* **do**
  Find SPE_i such that $workload[SPE_i]$ is minimal and $memload[SPE_i] + buffers[k] \leq LS$
  if *there is such a* SPE_i **then**
    $mapping[T_k] \leftarrow SPE_i$ /* Map T_k onto SPE_i */
    $workload[SPE_i] \leftarrow workload[SPE_i] + w_{SPE}(T_k)$
    $memload[SPE_i] \leftarrow memload[SPE_i] + buffers[k]$
  else
    Find PPE_j such that $workload[PPE_j]$ is minimal
    $mapping[T_k] \leftarrow PPE_j$ /* Map T_k onto PPE_j */
    $workload[PPE_j] \leftarrow workload[PPE_j] + w_{PPE}(T_k)$
while *the maximum workload of PPEs is smaller than the maximum workload of SPEs* **do**
  Let SPE_i be the SPE with maximum workload
  Let PPE_j be the PPE with minimum workload
  Consider the list of tasks mapped on SPE_i, ordered by non-increasing order of affinity
  Find the first task T_k in this list such that
  $workload[PPE_j] + w_{PPE}(T_k) \leq workload[SPE_i]$ and
  $workload[SPE_i] - w_{SPE}(T_k)) \leq workload[SPE_i]$
  if *there is such a task* T_k **then**
    Move T_k from SPE_i to PPE_j:
    $mapping[T_k] \leftarrow PPE_j$
    $workload[PPE_j] \leftarrow workload[PPE_j] + w_{PPE}(T_k)$
    $workload[SPE_i] \leftarrow workload[SPE_i] - w_{SPE}(T_k)$
    $memload[SPE_i] \leftarrow memload[SPE_i] - buffers[k]$
  else
    get out of the *while* loop
return $mapping$

If we take a closer look on the QS 22, we can see that some of the communication links are likely to become bottlenecks. This is the case for the link between the PPEs and their respective set of SPEs, and for the link between both Cell chips. Based on this observation, the QS 22 can therefore be partitioned into four sets of processing elements, as shown in Figure 7. The communications within each set are supposed to be fast enough. Therefore, their optimization is not crucial for performance, and only the communications among the sets will be taken into account.

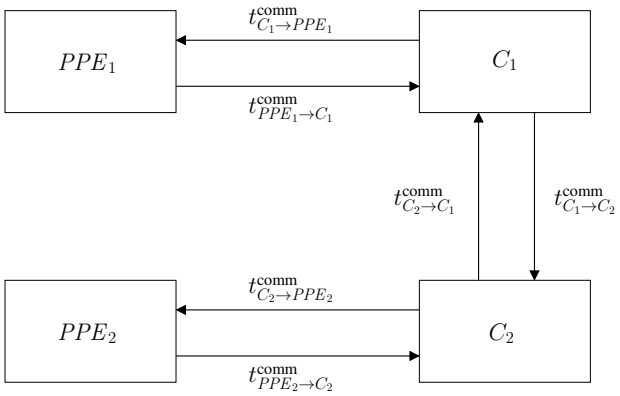


Figure 7: Partition of the processing elements within the Cell

In order to estimate the performance of any candidate mapping of a given task graph on this platform, it is necessary to evaluate both communication and computation times. We adopt a similar view as developed above for the design of the linear program. Given a mapping, we estimate the time taken by all computations and communications, and we compute the period as the maximum of all processing times.

As far as computations are concerned, we estimate the computation time t_P^{comp} of each part P of processing element as the maximal workload of any processing element within the set. The workload of a processing element PE is given by $\sum_{T_k \in S} w_{SPE}(T_k)$ in case of SPE, and by $\sum_{T_k \in S} w_{PPE}(T_k)$ in case of a PPE, where S is the set of tasks mapped on the processing element.

For communications, we need to estimate the data transfer times on every links interconnecting the sets of processing elements, as represented on Figure 7. For each unidirectional communication link between two parts P_1 and P_2, we define $t^{\text{comm}}_{P_1 \to P_2}$ as the time required to transfer every data going through that link. We adopt a linear cost model, as in the previous section. Hence, the communication time is equal to the amount of data divided by the bandwidth of the link. We rely on the previous benchmarks for the bandwidths:

- The links links between the two Cell chips have a bandwidth of 13 GB/s ($C_1 \to C_2$) and 11.5 GB/s ($C_2 \to C_1$), as highlighted in Equations (7) and (8);
- The links from a PPE to the accompanying set of SPEs ($PPE_1 \to C_1$ and $PPE_2 \to C_2$), which correspond to a read operation from the main memory, have a bandwidth of 25 GB/s (see Equation (2)).

The routing in this simplified platform is straightforward: for instance, the data transiting between PPE_1 and PPE_2 must go through links $PPE_1 \to C_1$, $C_1 \to C_2$ and finally $C_2 \to PPE_2$. Formally, when $D_{P_1 \to P_2}$ denotes the amount of data transferred from part P_1 to part P_2, the communication time of link $C_1 \to C_2$ is given by:

$$t^{\text{comm}}_{C_1 \to C_2} = \frac{D_{C_1 \to C_2} + D_{PPE_1 \to C_2} + D_{C_1 \to PPE_2} + D_{PPE_1 \to PPE_2}}{13}$$

4.2.2 Load-balancing procedure among the SPEs of a given Cell

All heuristics that will be introduced in the following need to know whether the load-balancing of a given task list across a set of SPEs is feasible or not, and what is the expected computation time. We thus propose a handy greedy mapping policy which balances the load on each SPE, and will be used in more complex heuristics. The provided task list L is first sorted using task weights $w_{SPE}(T_k)$. Then we map the heaviest task T_k on the least loaded SPE SPE_i, provided that it has enough memory left to host T_k. If there is no such SPE, we return an error. Otherwise, we repeat this step until every tasks are mapped onto a given SPE. This procedure is described in Algorithm 2.

Algorithm 2: CELLGREEDYSET(L, C_j)

Input: a list L of tasks, a set C_j of SPEs
Sort L in non increasing order of $w_{SPE}(T_k)$;
foreach $SPE_i \in C_j$ **do**
  $workload[SPE_i] \leftarrow 0$
  $memload[SPE_i] \leftarrow 0$
foreach T_k *in* L **do**
  Search SPE_i in C_j such that $workload[SPE_i]$ is minimal and $memload[SPE_i] + buffers[k] \leq LS$
  if *there is such a* SPE_i **then**
    $mapping[T_k] \leftarrow SPE_i$ /* Map T_k on SPE_i */
    $memload[SPE_i] \leftarrow memload[SPE_i] + buffers[k]$
    $workload[SPE_i] \leftarrow workload[SPE_i] + w_{SPE}(T_k)$
  else
    return $FAILURE$
return $mapping$

4.3 Clustering and mapping

A classical approach when scheduling a complex task graph on a parallel platform with both communication and computation costs is two use a two-step strategy. In a first set, the tasks are grouped into clusters without considering the platform. In a second step, these clusters are mapped onto the actual computation resources. In our case, the computing resources are the groups of processing elements determined in the previous sections: PPE_1, PPE_2, C_1, and C_2.

The classical greedy heuristic due to Sarkar [24] is used to build up task clusters. The rationale of this heuristic is to consider each expensive communication, and when it is interesting, to group the tasks involved in this communication. This is achieved by merging the clusters containing the source and destination tasks. The merging decision criterion is based on the makespan obtained for a single instance of the task graph. To estimate this makespan, Algorithm 3 assumes that each cluster is processed by a dedicated resource. The processing time of task T_k on a resource is set to $(w_{SPE}(T_k) + w_{PPE}(T_k))/2$, and that the communication bandwidth between these resources is set to $bw = 10\ GB/s$ (the available bandwidth between both Cell chips). In addition, when building clusters, Algorithm 4 bounds the memory footprint of each cluster so that it can be mapped onto any resource (PPE or set of SPEs). In a second step, this heuristic maps the clusters onto the real resources by load-balancing cluster loads across PPEs and sets of SPEs, as described in Algorithm 5.

Algorithm 3: $EPT(C)$: Estimated parallel time of the clustering C

foreach *task* T_k *in reverse topological order* **do**

$max^{out}_{local} = \max\Big(bl(T_m) \text{ such that } (T_k, T_m) \in E \text{ and } C(T_k) = C(T_m)\Big)$

$max^{out}_{remote} = \max\Big(bl(T_m) + \frac{data_{k,m}}{bw} \text{ such that } (T_k, T_m) \in E \text{ and } C(T_k) \neq C(T_m)\Big)$

$bl(T_k) = \frac{w_{PPE}(T_k) + w_{SPE}(T_k)}{2} + \max(max^{out}_{local}, max^{out}_{remote})$

foreach *task* T_k *in topological order* **do**

$max^{in}_{local} = \max\Big(tl(T_m) + \frac{w_{PPE}(T_m) + w_{SPE}(T_m)}{2} \text{ such that } (T_m, T_k) \in E \text{ and } C(T_m) = C(T_k)\Big)$

$max^{in}_{remote} = \max\Big(tl(T_m) + \frac{w_{PPE}(T_m) + w_{SPE}(T_m)}{2} + \frac{data_{m,k}}{bw} \text{ such that } (T_m, T_k) \in E \text{ and } C(T_m) \neq C(T_k)\Big)$

$tl(T_k) = \max(max^{in}_{local}, max^{in}_{remote})$

return $\max(bl(T_k) + tl(T_k))$

Algorithm 4: $CellClustering(G_A)$

In the initial clustering, each tasks of G_A has its own cluster:

foreach *task* T_k **do** $Clustering(T_k) = k$

compute $EPT(Clustering)$

$L \leftarrow$ list of edges of the task graph

Sort L by non-increasing communication weight

foreach *edge* (T_k, T_l) *in* L **do**

$newClustering \leftarrow Clustering$

Merge the clusters containing T_k and T_l:

foreach *task* T_i **do**

if $Clustering(T_i) = Clustering(T_l)$ **then**

$newClustering(T_i) \leftarrow Clustering(T_k)$

compute $EPT(newClustering)$

if $EPT(newClustering) \leq EPT(Clustering)$ *and the* CELLGREEDYSET *greedy heuristic successfully maps cluster* $newClustering(T_k)$ *on a set of SPEs* **then**

$Clustering \leftarrow newClustering$

Algorithm 5: $CellMapClusters(Clustering)$: Map clusters into computing resources

Consider a set of clusters $Clustering$ built with Algorithm 4
foreach *cluster* C *in* $Clustering$ **do**
  Compute the weight of cluster C: $w(c) = \sum_{T_k \in C} w_{PPE}(T_k) + w_{SPE}(T_k)$
Sort L by non-increasing weights
$load(PPE_1) \leftarrow 0$, $load(PPE_2) \leftarrow 0$
foreach *cluster* C *in* $Clustering$ **do** (we try to map C on each resource)
  $newload(PPE_1) \leftarrow load(PPE_1) + \sum_{T_k \in C} w_{PPE}(T_k)$
  $newload(PPE_2) \leftarrow load(PPE_2) + \sum_{T_k \in C} w_{PPE}(T_k)$
  Run CELLGREEDYSET on C_1 using the clusters already mapped to C_1 plus C, compute $newload(C_1)$ as the maximum load of all SPEs in C_1 (let $newload(C_1) \leftarrow \infty$ if CELLGREEDYSET fails)
  Do the same on C_2 to compute $newload(C_2)$
  Select the resource R such that $newload(R)$ is minimum
  Map C on resource R: $mapping[C] \leftarrow R$
  if R *is a PPE* **then** $load(R) \leftarrow newload(R)$
return $mapping$

4.4 Iterative refinement using DELEGATE

In this Section, we present an iterative strategy to build an efficient allocation. This method, called DELEGATE, is adapted from the heuristic introduced [12]. It consists in iteratively refining an allocation by moving some work from a highly loaded resource to a less loaded one, until the load is equally balanced on all resources. Here, resources can be either PPEs (PPE_1 or PPE_2) or set of SPEs (C_1 or C_2). In the beginning, all tasks are mapped on one of the PPE (PPE_1). Then, a (connected) subset of tasks is selected and its processing is delegated to another resource. The new mapping is selected if it respects memory constraints and improves the performance. This refinement procedure is then repeated until no more transfer is possible. If the selected resource for a move is a PPE, it is straightforward to compute the new processing time. If it is a set of SPEs, then the procedure CELLGREEDYSET is used to check memory constraints and compute the new processing time.

As in [12], at each step, a large number of moves are considered: for each task, all d-neighborhoods of this task are generated (with $d = 1, 2 \ldots d_{\max}$), and mapped onto each available resource. Among this large set of moves, we select the one with best performance. This heuristic needs a more involved way to compute the performance than simply using the period. Consider for example that both PPEs are equally loaded, but all SPEs are free. Then no move can directly decrease the period, but two moves are needed to decrease the load of both PPEs. Thus, for a given mapping, we compute all contributions to the period: the load of each resource, and the time needed for communications between any pair of resources (the period is the maximum of all contributions). The list of these contributions is sorted by non-increasing value. To compare two mappings, the one with the smallest contribution list in lexicographical order is selected.

Algorithm 6: $Eval(mapping)$

Compute the list $Score$ of every computation time and communication times of this initial solution :

$$Score = \{t^{\text{comp}}_{C_1}, t^{\text{comp}}_{C_2}, t^{\text{comp}}_{PPE_1}, t^{\text{comp}}_{PPE_2}, t^{\text{comm}}_{C_1 \to C_2}, t^{\text{comm}}_{C_2 \to C_1}, t^{\text{comm}}_{PPE_1 \to C_1}, t^{\text{comm}}_{C_1 \to PPE_1}, t^{\text{comm}}_{PPE_2 \to C_2}, t^{\text{comm}}_{C_2 \to PPE_2}\}$$

Sort $Score$ in non increasing order.
return $Score$

Algorithm 7: DELEGATE(G_A)

Map all tasks on PPE_1: **foreach** T_k **do** $mapping[T_k] \leftarrow PPE_1$
$move\_possible \leftarrow 1$
while $move\_possible$ **do**
  $best\_move \leftarrow \emptyset$
  foreach T_k **do**
    foreach *neighborhood* N *of* T_k **do**
      foreach $S \in \{PPE_1, PPE_2, C_1, C_2\}$ **do**
        $move \leftarrow mapping$
        $move[N] \leftarrow S$
        if S *is a set of SPEs and* CELLGREEDYSET *fails to map* N *on* S **then**
          Try another S
        if $Eval(move) <_{\text{lex}} Eval(best\_move)$ **then**
          $best\_move \leftarrow move$
  if $Eval(best\_move) <_{\text{lex}} Eval(mapping)$ **then**
    $mapping \leftarrow best\_move$
    $move\_possible \leftarrow 1$
  else
    $move\_possible \leftarrow 0$

5 Experimental validation

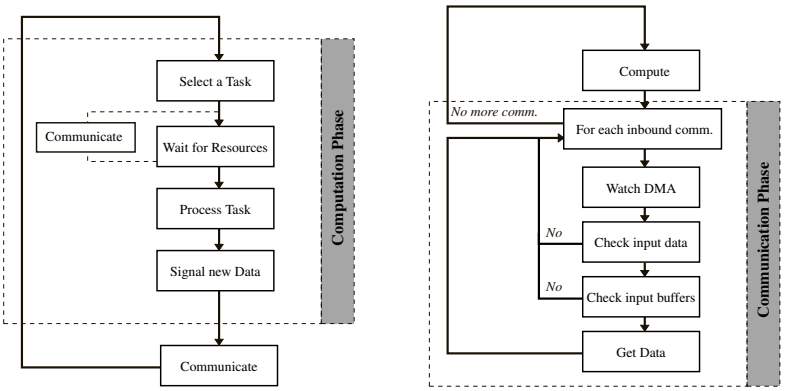
This section presents the experiments conducted to validate the scheduling framework introduced above. We first present the scheduling software developed to run steady-state schedules on the QS 22, then the application graphs in use, and finally report and comment the results.

5.1 Scheduling software

In order to run steady-state schedules on the QS 22, a complex software framework is needed: it has to map tasks on different types of processing elements and to handle all communications. Although there already exists some frameworks dedicated to streaming applications [15, 16], none of them is able to deal with complex task graphs while allowing to statically select the mapping. Thus, we have decided to develop our own framework[1]. Our scheduler requires the description of the task graph, its mapping on the platform, and the code of each task. Even if it was designed to use the mapping returned by the linear program, it can also use any other mapping, such as the ones dictated by the previously described heuristic strategies.

For now, our scheduling framework is able to handle only one PPE. For the following, we thus consider $n_P = 1$ PPE and $n_P = 16$ SPE.

Once all tasks are mapped onto the right processing elements and the process is properly initialized, the scheduling procedure is divided into two main phases: the *computation phase*, when the scheduler selects a task and processes it, and the *communication phase*, when the scheduler performs asynchronous communications. These steps, depicted on Figure 8, are executed by every processing elements. Moreover, since communications have to be overlapped with computations, our scheduler cyclically alternates between those two phases.



(a) Computation Phase. (b) Communication Phase.

Figure 8: Scheduler state machine.

The computation phase, which is shown on Figure 8(a), begins with the selection of a runnable task according to the provided schedule, and waits for the required resources (input data and output buffers) to be available. If all required resources are available, the selected task is processed, otherwise, it moves to the communication phase. Whenever a new data is produced, the scheduler signals it to every dependent processing elements. Note that in our framework, computation tasks are not implemented

[1] An experimental version of our scheduling framework is available online, at http://graal.ens-lyon.fr/~mjacquel/cell_ss.html

as OS tasks but rather as function calls. One computing thread is launched on each processing element (PPE or SPEs). Each thread then internally chooses the next function (or node of the task graph) to run according to the provided schedule. This allows us to avoid fine task scheduling at the OS level.

The communication phase, depicted in Figure 8(b), aims at performing every incoming communication, most often by issuing DMA calls. Therefore, the scheduler starts watching every previously issued DMA calls in order to unlock the output buffer of the sender as soon as data had been received. Then, the scheduler checks whether there is new incoming data. In that case, and if enough input buffers are available, it issues the proper "Get" command.

Libnuma [21] is used for both thread and memory affinity. Since a PPE can simultaneously handle two threads, the affinity of every management threads is set to the first multi-threading unit, while PPE computing thread's affinity is set to the second multi-threading unit. Therefore, management threads and computing thread runs on different multi-threading units, and do not interfere. Moreover, the data used by a given thread running on a PPE is always allocated on the memory bank associated to that PPE.

To obtain a valid and efficient implementation of this scheduler, we had to overcome several issues due to the very particular nature of the Cell processor. First, the main issue is heterogeneity: the Cell processor is made of two different types of cores, which induces additional challenges for the programmer:

- SPE are 32-bit processors whereas the PPE is a 64-bit architecture;
- Different communication mechanisms have to be used depending on which types of processing elements are implied in the communication. To properly issue our "Get" operations, we made use of three different intrinsics: `mfc_get` for SPE to SPE communications, `spe_mfcio_put` for SPE to PPE communication, and `memcpy` for communication between PPE and main memory.

Another difficulty lies in the large number of variables that we need to statically initialize in each local store before starting the processing of the stream: the information on the mapping, the buffer for data transfer, and some control variables such as addresses of all memory blocks used for communications. This initialization phase is again complicated by the different data sizes between 32-bit and 64-bit architectures, and the run-time memory allocation.

All these issues show that the Cell processor is not designed for such a complex and decentralized usage. However, our success in designing a complex scheduling framework proves that it is possible to use such a heterogeneous processor for something else than pure data-parallelism.

5.2 Application scenarios

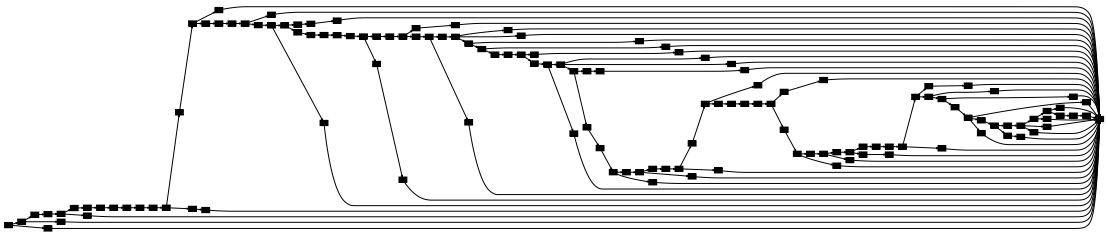
We test our scheduling framework on 25 random task graphs, obtained with the DagGen generator [26]. This allows us to test our strategy against task graphs with different depths, widths, and branching factors.

The smallest graph has 20 tasks while the largest has 135 tasks. The smallest graph is a simple chain of tasks, and the largest graph is depicted on Figure 9(a). On the communication side, the number of edges goes from 19 to 204, the graph with 204 edges is depicted on Figure 9(b).

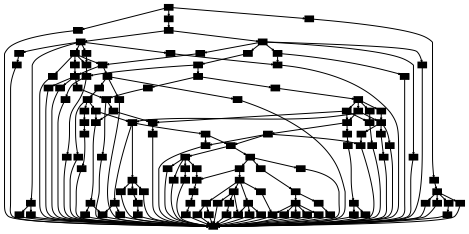
We classified the generated graphs into two sets, one for the smaller graphs, having up to 59 tasks, and one for the larger graphs (87 - 135 tasks).

For all graphs, we generated 10 variants with different Communication-to-Computation Ratio (CCR), resulting in 250 different random applications and 230 hours of computation. We define the CCR of a scenario as the total number of transferred elements divided by the total number of

operations on these elements. In the experiments, the CCR ranges from 0.001 (computation-intensive scenario) to 0.1 (communication-intensive scenario).



(a) Graph with largest number of tasks



(b) Graph with largest number of edges

Figure 9: Largest task graphs used in the experiments

In the experiments, we denote by MIP the scheduling strategy using Mixed Integer Programming to compute an optimal mapping, and described in Section 3.3. ILOG CPLEX [9] is used to solve the linear program with rational and integer variables. To reduce the computation time for solving the linear program, we used the ability of CPLEX to stop its computation as soon as its solution is within 5% of the optimal solution. While this significantly reduces the average resolution time, it still offers a very good solution.

5.3 Experimental results

In this section, we present the performance obtained by the scheduling framework presented above. We first focus on the initialization phase of streaming applications using our framework, showing that steady-state operation is reached within a reasonable amount of instances. Then, we compare the throughput obtained by the heuristics introduced in Section 4 and the algorithm MIP, both using the model of the QS 22 and using experiments on a real platform. Finally, we discuss the impact of the communication-to-computation ration, and estimate the time required to compute mappings using each strategy. Note that except when we study the influence of the number of SPEs, all the results presented below are computed using all 16 SPEs in the QS 22.

5.3.1 Entering steady-state

First, we show that our scheduling framework succeeds in reaching steady-state, and that the throughput is then similar to the one predicted by the linear program. Figure 10 shows the experiments done with the task graph described in Figure 9(b), with a CCR of 0.004, on the QS 22 using all 16 SPEs.

We notice that a steady-state operation is obtained after processing approximately 1000 instances. Note that the input data of one instance consists only of a few bytes, so the duration of the transient phase before reaching the steady-state throughput is small compared to the total duration of the

stream. In steady state, the experimental throughput achieves 95% of the theoretical throughput predicted by the linear program. The small gap is explained by the overhead of our framework, and the synchronizations induced when communications are performed. The schedules given by other mappings, computed using heuristics, also reaches steady state after a comparable transient phase. We discuss their steady-state throughput below.

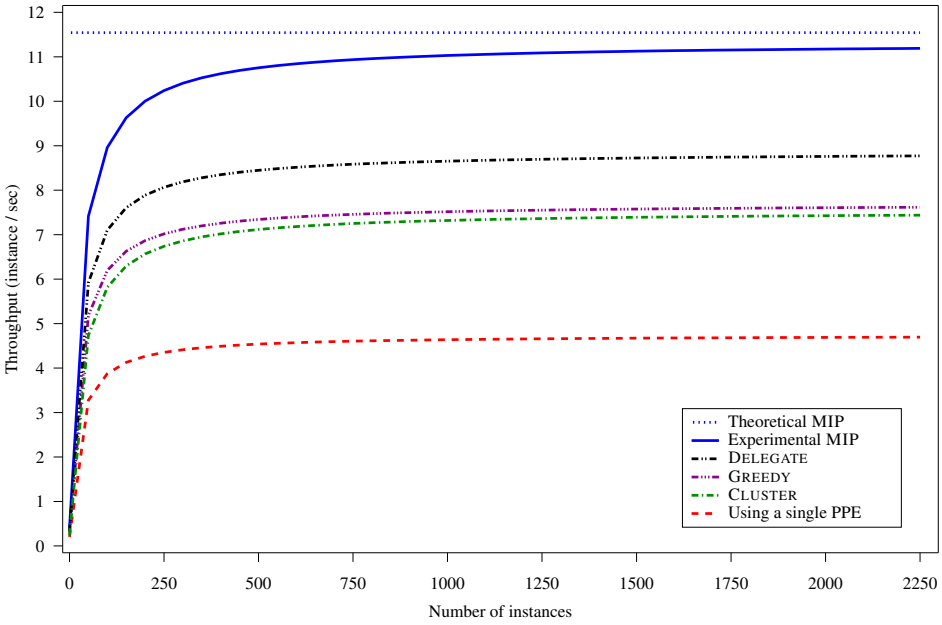


Figure 10: Throughput achieved depending on the number of instances.

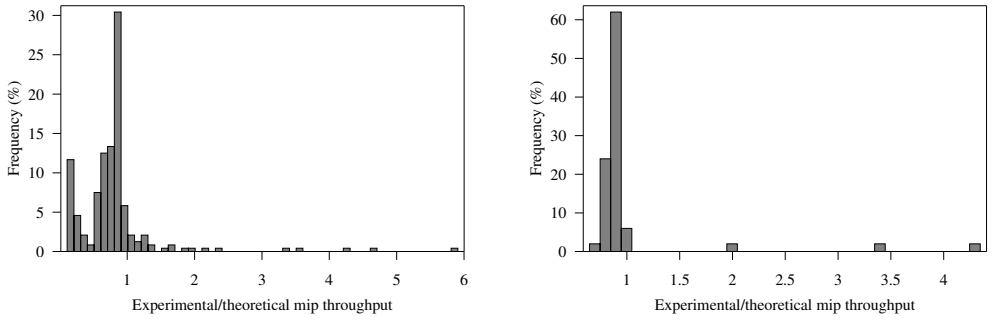
In Table 1, we present the number of instances that are required to reach steady state, that is, to reach 99% of the maximum (experimental) throughput, for all possible scenarios. The average value is around 2000 instances, but the mapping computed using MIP reaches steady state faster than the other solutions. We have observed the same behavior on small and large task graphs. This shows that our scheduling framework is able to reach steady-state operation for any mapping within a reasonable amount of instances compared to the usual length of streaming applications.

Algorithm	Min.	Max.	Average	Std. dev.
GREEDY	300	18,500	2,234	2,589
DELEGATE	250	16,500	2,249	2,381
CLUSTER	300	14,600	2,214	2,586
MIP	300	14,500	2,050	2,182

Table 1: Number of instances required to reach steady state for all scenarios.

Figure 11(a) presents the distribution of the ratio between the experimental throughput of MIP, and the theoretical throughput (the one predicted by the linear program). The average ratio is 0.91, but we can observe that the experimental throughput is sometimes very different from the theoretical one (as less as 20% or as much as 597%). Cases when the experimental throughput is 5 times smaller than the predicted one mainly happen for small task graphs, when the very few tasks are scheduled on each processing elements, which leads to a high synchronization cost. Figure 11(b) shows the results for large task graphs, when the accuracy of the predicted throughput is larger (the average

ratio is 1.10). There still remain some cases with a high ratio (the experimental throughput is larger than the predicted throughput). This corresponds to communication-intensive scenarios: in a few cases, our communication model is very pessimistic for communications and overestimates bandwidth contention.



(a) All task graphs (b) Large task graphs

Figure 11: Distribution of the ratio between experimental and theoretical throughputs for the MIP strategy.

5.3.2 Theoretical comparison of heuristics with MIP

We first start by comparing the expected throughput of the heuristics described in Section 4 with the theoretical throughput of MIP, detailed in Section 3.3. This allows to measure the quality of the mapping produced by heuristics without the particularities of the scheduling framework, and the inaccuracies of the QS 22 model.

For this purpose, we first compute the theoretical throughput of all mappings produced by the heuristics. A mapping is first described using the α and β variables from the linear program presented in Section 3.3. The expected period of the mapping can then be computed using Constraints (14) and (15) (for computations) and Constraints (1), (2), (3), (4), (5), (6), (7), (8), and (9) (for communications).

This theoretical comparison also helps to assess the limitation of the constraints on DMA calls. In the design of the MIP strategy, the limited number of simultaneous DMA calls for PPEs and SPEs is taken into account with two Constraints (Equations (17) and (18)). This limitation is not taken into account while designing heuristics, because we want to keep their design simple. Our scheduling framework assigns DMA calls dynamically to communication requests, and is able to deal with any number of communications. Of course, when the number of simultaneous communications exceeds the bounds, communications are delayed, thus impacting the throughput. The comparison of theoretical throughput of mappings allows us to measure the limitation on the expected throughput induced by these constraints, which only exist for MIP.

We hence compute the ratio between predicted throughput of heuristic strategies over the predicted throughput of MIP. Detailed results are given in Table 2.

These results show that in some cases, the limitation on the number of DMA calls has an impact on the throughput: there are cases when the CLUSTER strategy finds a mapping with an expected throughput twice the one of MIP. The MIP strategy is able to reach the best throughput among all strategies in only 80% of the scenarios, due to this limitation. However, MIP still gives a better average throughput than all other strategies. Thus, this limitation has a little impact on performance.

All task graphs					
Algorithm	*Min.*	*Max.*	*Average*	*Std. dev.*	*Best cases*
GREEDY	0.55	1.49	0.82	0.12	6%
DELEGATE	0.75	1.05	0.97	0.07	60%
CLUSTER	0.00	2.46	0.32	0.28	1%
MIP	1.00	1.00	1.00	0.00	80%

Small task graphs					
Algorithm	*Min.*	*Max.*	*Average*	*Std. dev.*	*Best cases*
GREEDY	0.55	1.49	0.82	0.13	8%
DELEGATE	0.87	1.05	0.99	0.04	71%
CLUSTER	0.05	2.46	0.32	0.26	1%
MIP	1.00	1.00	1.00	0.00	78%

Large task graphs					
Algorithm	*Min.*	*Max.*	*Average*	*Std. dev.*	*Best cases*
GREEDY	0.63	0.98	0.79	0.10	0%
DELEGATE	0.75	1.04	0.91	0.09	27%
CLUSTER	0.00	0.81	0.33	0.33	0%
MIP	1.00	1.00	1.00	0.00	88%

Table 2: Predicted throughput of heuristics normalized to the predicted throughput of MIP

Among heuristics, the DELEGATE strategy ranks first: it is almost within 25% of the performance of MIP, and on average gives the same results, and gives the best performance on 60% of the scenarios. (Note that several heuristics may give the best throughput for a scenario, hence leading to a sum larger than 100%). Quite surprisingly, the CLUSTER strategy gives very poor results: on average its throughput is only one third of the MIP throughput, whereas the simple GREEDY strategy performs better, with an average throughput around 80% of the MIP throughput. All heuristics perform better on small task graphs, and have more difficulties to tackle the higher complexity of large task graphs.

The bad performance of CLUSTER is surprising because this heuristics takes communication into account, as opposed to GREEDY. However, the communications are considered only when building clusters in the first phase. In the second phase, when clusters are mapped on the resources, communications are totally neglected. However, all communications do not have the same impact, because the communication graph has heterogeneous links. Moreover, since clusters have been made to cancel out the impact of large communications, the load balancing procedure in the second phase has less freedom, and is unable to reach a well balanced mapping. This explains why the mapping produced by CLUSTER are usually worse than the one given by DELEGATE or GREEDY.

5.3.3 Experimental comparison of heuristics with MIP

We move to the experimental comparison of heuristics and MIP, when all mappings are scheduled with our scheduling software on the real QS 22 platform. The experimental throughputs of generated mappings are presented in Table 3. For sake of comparison, all throughputs are normalized by the throughput of MIP.

The comparison between heuristics and MIP is close to the one with expected throughputs. How-

All task graphs					
Algorithm	*Min.*	*Max.*	*Average*	*Std. dev.*	*Best cases*
GREEDY	0.32	3.28	0.91	0.39	8%
DELEGATE	0.75	2.63	1.06	0.27	48%
CLUSTER	0.00	1.80	0.41	0.28	0%
MIP	1.00	1.00	1.00	0.00	54%

Small task graphs					
Algorithm	*Min.*	*Max.*	*Average*	*Std. dev.*	*Best cases*
GREEDY	0.32	3.28	0.93	0.43	11%
DELEGATE	0.78	2.63	1.10	0.28	56%
CLUSTER	0.08	1.80	0.44	0.26	0%
MIP	1.00	1.00	1.00	0.00	46%

Large task graphs					
Algorithm	*Min.*	*Max.*	*Average*	*Std. dev.*	*Best cases*
GREEDY	0.64	1.73	0.85	0.21	0%
DELEGATE	0.75	1.89	0.95	0.20	23%
CLUSTER	0.00	0.82	0.33	0.34	0%
MIP	1.00	1.00	1.00	0.00	80%

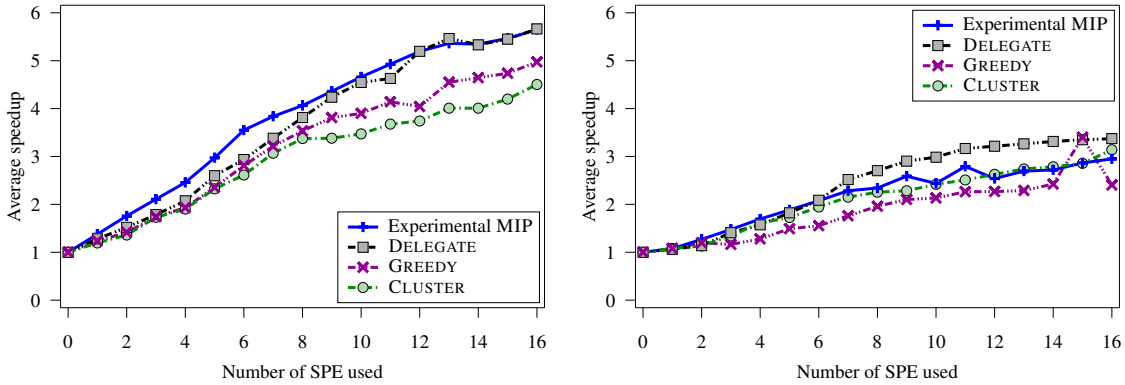
Table 3: Experimental throughput of heuristics normalized to the experimental throughput of MIP

ever, the gap between MIP and each heuristic is smaller: heuristics perform somewhat better on the real platform that on the model. The average throughput of DELEGATE is very close to MIP's throughput (larger for small task graphs, but smaller for large task graphs), and GREEDY gets around 90% of the MIP's throughput. Once again, CLUSTER gives a very limited average throughput, with only 41% of the MIP's throughput. On large task graphs, MIP is above all heuristics, and reaches the best throughput in 80% of the cases. However, on average, DELEGATE is able to achieve 95% of the MIP's throughput.

5.3.4 Scaling and influence of the communication-to-computation ratio

In this section, we present the speed-up obtained by MIP and heuristics. We also study the influence of the Communication-to-Computation Ratio (CCR), both on the duration of the transient phase before reaching steady state, and on the accuracy of the model.

We first present the performance obtained by all strategies, with different number of SPEs. In the MIP solution, the number of SPEs is a parameter, which may vary between 0 and 16. For all heuristics developed above, it is quite straightforward to adapt them to deal with a variable number of SPEs. Figure 12 present the average speed-up of each heuristic obtained for different number of SPEs. We observe that all heuristics are able to take advantage of an increasing number of SPEs. When the CCR is high, the resulting speed-up is smaller than when the CCR is low, because large communications make it more difficult to distribute tasks among processing elements. This is also outlined by Figure 13(b), which presents the variation of the average speed-up of heuristics (for 16 SPEs) with the CCR. For a CCR smaller than 0.02, the average speed-up is almost constant, while it decreases for a larger CCR. We can observe that DELEGATE is a little less sensitive to the increase



(a) With CCR=0.001 (b) With CCR=0.1

Figure 12: Average speed-up vs. number of SPEs in use.

of the CCR than MIP. For some strategies, the speed-up slightly increases when the CCR increases from 0.001 to 0.02. This is because to increase the CCR, we first decrease the computation amount of tasks, which makes the overhead of the scheduling software more visible when all tasks are scheduled on the PPE.

All task graphs, CCR of 0.001				
Algorithm	*Min.*	*Max.*	*Average*	*Std. dev.*
GREEDY	300	400	367	38
DELEGATE	300	400	367	42
CLUSTER	300	400	361	40
MIP	300	400	375	35

All task graphs, CCR of 0.1				
Algorithm	*Min.*	*Max.*	*Average*	*Std. dev.*
GREEDY	200	4750	788	974
DELEGATE	200	4200	750	850
CLUSTER	150	2950	702	681
MIP	150	3200	646	670

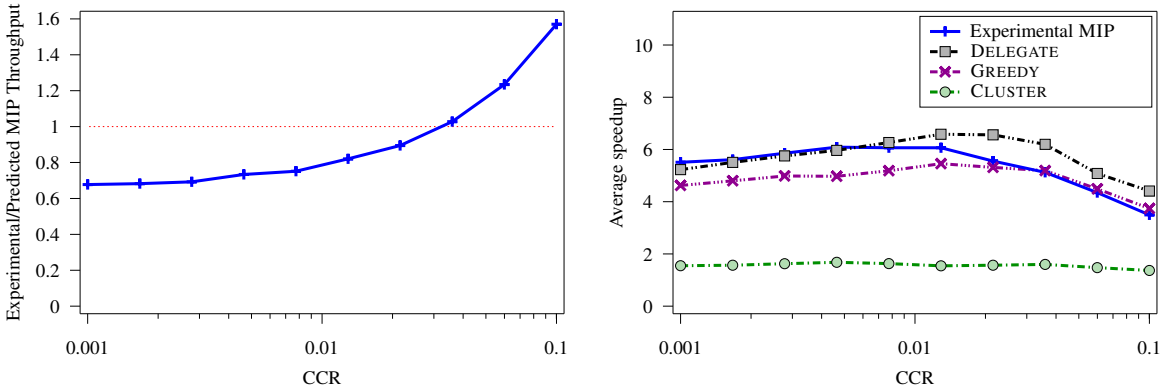
Table 4: Number of instances required to reach steady state

Table 4 presents the number of instances that are processed in the transient phase before reaching steady state, for the two extreme values of CCR: 0.001 (computation-intensive scenario) and 0.1 (communication-intensive scenario). We observe that the larger the CCR, the longer the duration of the transient phase: when communications are predominant, it is more difficult to reach a steady state phase even if communications are well overlapped by computations.

Figure 13(a) presents the evolution of the ratio between the experimental throughput of MIP over its predicted throughput in function of the CCR. We notice that for all CCR smaller than 0.05, our model is slightly optimistic, and predicts a throughput at most 30% larger than the one obtained in the experiments. For larger CCRs, corresponding to communication-intensive scenarios, the model is pessimistic and larger throughputs are obtained in the experiments. Our model is thus best suited for average values of the CCR. In case of extreme values, it is necessary to modify the model, either by taking into account synchronization costs, when CCR is very small, or by better modeling contention between communications, in case of a very large CCR.

5.3.5 Time required to compute schedules

When considering streaming application, it is usually assumed that streams have very long durations. This is why it is worth spending some time to optimize the schedule before the real data processing. Yet, since we are using mixed linear programming, we report the time needed to compute the



(a) Model accuracy vs. CCR

(b) Average speed-up vs. CCR

Figure 13: Evolution of the model accuracy and the speedup with the CCR.

schedules, to prove that this is done in reasonable time.

Table 5 presents the time required to compute mappings using each algorithm. On average, the MIP strategy requires twice as much time than all heuristics, and runs for less than two minutes. In some cases, corresponding to large graphs, the run time of MIP can reach 12 minutes, whereas DELEGATE need at most 3 minutes. These running, although large, are very reasonable in the context of streaming applications that are supposed to run for several minutes or hours.

Algorithm	*Min.*	*Max.*	*Average*
GREEDY	2.7	189	39
CLUSTER	2.6	190	39
DELEGATE	2.9	205	48
MIP	3.6	706	80

Table 5: Scheduling time in seconds

6 Related Work

Scheduling streaming applications is the subject of a vast literature. Several models and frameworks had been introduced like StreamIt [25], Brook [10] or Data Cutter Lite [7]. However, our motivation for this work was rather to extract a simple and yet relevant model for the Cell processor, so as to be able to provide a way to compute interesting mapping for pipeline applications. To the best of our knowledge, this has never been considered for the Cell processor, nor for another multi-core architecture that exhibits a similar heterogeneity. Note that the small scale of the target architecture makes the problem very particular, and that it was not clear that such complex mapping techniques could result in a feasible and competitive solution for streaming applications. To the best of our knowledge, none of the existing frameworks which target the Cell processor make it is possible to implement our own static scheduling strategies. The main two limitations of existing frameworks are the restriction to simple pipeline applications (chain task graphs), whereas we target any DAG, and the fact that only SPEs are used for computation, the PPE being restricted to control, whereas we take advantage of the heterogeneity.

Hiding memory accesses latencies is a crucial need when considering the Cell processor. In [14], the authors introduce an improvement to the Decoupled Threaded Architecture [13] consisting in a prefetching mechanism. They propose an implementation of this mechanism on the Cell processor through the use of DMA calls and extensions to compiler instructions set, which interest is demonstrated through simulation (no real implementation seems to be available). In our scheduler, we propose a real implementation of a similar prefetching mechanism.

7 Conclusion

In this paper, we have studied the scheduling of streaming applications on a heterogeneous platform: the IBM Bladecenter QS 22, made of two heterogeneous multi-core Cell processors. The first challenge was to come up with a realistic and yet tractable model of the Cell processor. We have designed such a model, and we have used it to express the optimization problem of finding a mapping with maximal throughput. This problem has been proven NP-complete, and we have designed a formulation of the problem as a mixed linear program. By solving this linear program with appropriate tools, we can compute a mapping with optimal throughput. We have also proposed a set of scheduling heuristics to avoid the complexity of mixed linear programing.

In a second step, we have implemented a complete scheduling framework to deploy streaming applications on the QS 22. This framework, available for public use, allows the user to deploy any streaming application, described by a potentially complex task graph, on a QS 22 platform or single Cell processor, given any mapping of the application to the platform. Thanks to this scheduling framework, we have been able to perform a comprehensive experimental study of all our scheduling strategies. We have shown that our MIP strategy usually reaches 90% of the throughput predicted by the linear program, that it has a good and scalable speed-up when using up to 16 SPEs. Some of the proposed heuristics, in particular DELEGATE, are also able to reach very good performance. When the task graph to schedule is large, MIP is the one which offers the best performance, but DELEGATE achieves 95% of its throughput on average. We have shown that considering load-balancing among processing elements, and carefully estimating communications between the different components of this bi-Cell platform, is the key to performance.

Overall, this demonstrates that scheduling a complex application on a heterogeneous multi-core processor is a challenging task, but that scheduling tools can help to achieve good performance.

This work has several natural extensions, which includes refining the model presented for the QS 22 to more complex platform, for example to consider a cluster of QS 22, or other multi-core platform.

References

[1] M. Ålind, M. Eriksson, and C. Kessler. BlockLib: a skeleton library for Cell broadband engine. In *IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering*, pages 7–14. ACM, 2008.

[2] AMD Fusion. http://fusion.amd.com.

[3] Kevin J. Barker, Kei Davis, Adolfy Hoisie, Darren J. Kerbyson, Mike Lang, Scott Pakin, and Jose C. Sancho. Entering the petaflop era: the architecture and performance of roadrunner. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, 2008.

[4] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert. Steady-state scheduling on heterogeneous clusters. *International Journal of Foundations of Computer Science*, 16(2):163–194, 2005.

[5] Olivier Beaumont and Loris Marchal. Steady-state scheduling. In *Introduction to Scheduling*, pages 159–186. Chapman and Hall/CRC Press, 2010.

[6] Pieter Bellens, Josep M. Pérez, Felipe Cabarcas, Alex Ramírez, Rosa M. Badia, and Jesús Labarta. CellSs: Scheduling techniques to better exploit memory hierarchy. *Scientific Programming*, 17(1-2):77–95, 2009.

[7] Michael D. Beynon, Tahsin M. Kurç, Ümit V. Çatalyürek, Chialin Chang, Alan Sussman, and Joel H. Saltz. Distributed processing of very large datasets with datacutter. *Parallel Computing*, 27(11):1457–1478, 2001.

[8] ClearSpeed technology. http://www.clearspeed.com/technology/index.php.

[9] ILOG CPLEX: High-performance software for mathematical programming and optimization. http://www.ilog.com/products/cplex/.

[10] William J. Dally, Francois Labonte, Abhishek Das, Patrick Hanrahan, Jung-Ho Ahn, Jayanth Gummaraju, Mattan Erez, Nuwan Jayasena, Ian Buck, Timothy J. Knight, and Ujval J. Kapasi. Merrimac: Supercomputing with streams. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 35, Washington, DC, USA, 2003. IEEE Computer Society.

[11] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Memory - sequoia: programming the memory hierarchy. In *SC '06: Proceedings of the ACM/IEEE conference on Supercomputing*, page 83, 2006.

[12] Matthieu Gallet, Loris Marchal, and Frédéric Vivien. Efficient scheduling of task graph collections on heterogeneous resources. In *IEEE International Parallel and Distributed Processing Symposium*, 2009.

[13] Roberto Giorgi, Zdravko Popovic, and Nikola Puzovic. Dta-c: A decoupled multi-threaded architecture for cmp systems. In *SBAC-PAD*, pages 263–270, 2007.

[14] Roberto Giorgi, Zdravko Popovic, and Nikola Puzovic. Exploiting DMA to enable non-blocking execution in decoupled threaded architecture. In *IEEE International Parallel and Distributed Processing Symposium*, 2009.

[15] X. Hang. A streaming computation framework for the Cell processor. Master's thesis, Massachusetts Institute of Technology, 2007.

[16] T. Hartley and U. Catalyurek. A component-based framework for the Cell broadband engine. In *IEEE International Parallel and Distributed Processing Symposium*, 2009.

[17] Stephen L. Hary and Fusun Ozguner. Precedence-constrained task allocation onto point-to-point networks for pipelined execution. *IEEE Trans. Parallel and Distributed Systems*, 10(8):838–851, 1999.

[18] Ibm software kit for multicore acceleration. http://www.ibm.com/chips/techlib/techlib.nsf/products/IBM_SDK_for_Multicore_Acceleration.

[19] James A. Kahle, Michael N. Day, H. Peter Hofstee, Charles R. Johns, Theodore R. Maeurer, and David J. Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4-5):589–604, 2005.

[20] Michael Kistler, Michael Perrone, and Fabrizio Petrini. Cell multiprocessor communication network: Built for speed. *IEEE Micro*, 26(3):10–23, 2006.

[21] Andreas Kleen. A numa api for linux. http://andikleen.de/, 2005.

[22] Mercury technology. http://www.mc.com/technologies/technology.aspx.

[23] Ibm bladecenter qs22. http://www-03.ibm.com/systems/bladecenter/hardware/servers/qs22/.

[24] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, Cambridge, MA, USA, 1989.

[25] Streamit project. http://groups.csail.mit.edu/cag/streamit/index.shtml.

[26] F. Suter. DAG generation program. http://www.loria.fr/~suter/dags.html.

[27] William Thies. *Language and Compiler Support for Stream Programs*. PhD thesis, Massachusetts Institute of Technology, 2001.

[28] Q. Wu, J. Gao, M. Zhu, N.S.V. Rao, J. Huang, and S.S. Iyengar. On optimal resource utilization for distributed remote visualization. *IEEE Trans. Computers*, 57(1):55–68, 2008.

A Complete benchmark results

The following tables presents the measuder bandwidth and latency when performing a “read” operation from a SPE to a distant local store. Table 6 presents the average latency for such a transfer, while Table 7 shows the obtained bandwidth for a large transfer.

Reader	Distant local store															
SPE	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	120	128	120	128	113	128	113	113	428	443	443	450	450	443	443	450
1	113	128	113	120	113	143	113	128	458	420	450	443	450	413	465	465
2	113	113	128	120	128	120	113	113	450	443	428	420	443	435	443	450
3	120	113	128	120	120	128	113	128	450	458	450	443	435	435	428	435
4	120	135	128	113	113	113	113	128	465	458	458	435	443	443	443	443
5	113	113	113	113	113	113	113	120	450	450	443	435	450	450	458	450
6	113	113	113	113	113	120	113	113	458	435	443	435	405	443	450	450
7	113	113	128	113	113	113	113	120	435	450	450	450	435	458	450	450
8	480	458	488	473	473	473	458	443	113	113	113	113	113	113	113	113
9	450	480	473	495	428	458	480	458	113	113	113	113	113	113	113	113
10	443	443	428	465	443	450	443	443	113	113	113	113	113	113	113	113
11	480	473	458	488	465	473	443	465	113	120	113	113	113	113	113	113
12	458	450	450	458	458	465	443	435	113	113	113	113	113	113	113	128
13	458	420	435	450	480	443	443	450	113	113	120	113	113	113	113	113
14	443	443	428	450	458	443	435	450	128	113	120	113	113	113	113	113
15	450	488	435	435	443	450	458	435	113	113	113	113	113	113	113	113

Table 6: Latency when a SPE reads from a distant local store, in nanoseconds.

Reader	Distant local store															
SPE	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	23.1	23.4	24.2	24.8	24.8	23.9	23.5	23.9	4.9	4.8	4.9	4.9	4.9	5.1	4.9	5.1
1	23.9	23.7	23.4	24.5	23.2	24.5	23.6	24.2	4.9	4.9	4.8	4.9	5.0	4.8	4.9	4.8
2	23.9	24.2	23.1	24.2	23.0	23.4	23.7	23.9	5.0	4.9	4.9	4.8	5.0	4.9	5.0	4.9
3	24.5	23.9	24.2	23.5	22.3	24.0	25.1	24.8	5.1	4.7	4.9	5.0	4.9	4.9	4.9	5.0
4	25.4	25.4	23.9	25.1	25.4	25.4	25.4	25.4	4.9	5.0	5.0	5.0	5.0	5.0	4.8	4.8
5	25.4	25.4	24.0	25.4	25.4	25.4	24.8	25.4	4.9	4.8	4.8	4.8	5.0	4.9	4.9	4.9
6	25.1	25.1	25.1	25.4	24.8	25.4	25.4	24.8	4.9	4.9	4.9	4.9	4.9	5.0	5.0	5.0
7	24.3	25.1	24.5	25.1	23.1	24.6	25.1	24.5	4.9	5.0	4.8	4.8	4.9	4.8	4.9	5.1
8	3.3	3.3	3.5	3.4	3.4	3.4	3.5	3.5	25.4	25.4	24.8	25.1	25.4	25.4	25.4	25.4
9	3.5	3.3	3.5	3.3	3.3	3.4	3.5	3.5	25.4	25.4	24.8	25.1	24.5	25.1	25.4	25.4
10	3.3	3.3	3.4	3.4	3.4	3.4	3.5	3.3	25.4	25.4	24.6	25.1	24.8	25.4	25.4	25.1
11	3.5	3.3	3.4	3.3	3.4	3.4	3.3	3.4	25.4	25.4	25.1	25.4	25.4	25.4	25.4	25.4
12	3.3	3.3	3.4	3.3	3.5	3.4	3.4	3.4	25.4	25.4	25.4	25.1	25.4	25.4	24.6	25.4
13	3.3	3.2	3.4	3.3	3.3	3.3	3.3	3.4	25.4	25.4	25.1	24.8	25.4	25.4	25.4	25.4
14	3.4	3.3	3.4	3.4	3.4	3.3	3.5	3.5	25.4	25.1	24.8	25.4	25.4	25.4	25.4	25.4
15	3.4	3.4	3.5	3.4	3.4	3.3	3.4	3.4	25.1	25.4	24.3	25.4	25.4	25.1	25.1	25.4

Table 7: Bandwidth when a SPE reads from a distant local store, in GB/s.