Optimizing Explicit Data Transfers for Data Parallel Applications on the Cell Architecture

Selma Saidi, VERIMAG, STMicroelectronics Grenoble, France
Pranav Tendulkar, VERIMAG, Grenoble, France
Thierry Lepley, STMicroelectronics Grenoble, France
Oded Maler, VERIMAG Grenoble, FRANCE

In this paper we investigate a general approach to automate some deployment decisions for a certain class of applications on multi-core computers. We consider data-parallelizable programs that use the well-known double buffering technique to bring the data from the off-chip slow memory to the local memory of the cores via a DMA (direct memory access) mechanism. Based on the computation time and size of elementary data items as well as DMA characteristics, we derive optimal and near optimal values for the number of blocks that should be clustered in a single DMA command. We then extend the results to the case where a computation for one data item needs some data in its neighborhood. In this setting we characterize the performance of several alternative mechanisms for data sharing. Our models are validated experimentally using a cycle-accurate simulator of the Cell Broadband Engine architecture.

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Data parallelization, Double Buffering, Direct Memory Access (DMA), Cell B.E.

ACM Reference Format:
Saidi S., Tendulkar P., Lepley T, Maler O, Optimizing Explicit Data Transfers for Data Parallel Applications on the Cell Architecture ACM Trans. Embedd. Comput. Syst. V, N, Article A (July 2011), 20 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

The semi-conductor industry has chosen to use multi-core computing as the path for sustaining the ever-increasing demand for computational power, hitherto satisfied by essentially making processors faster. While the major reasons for this turn are based on physical arguments, much of the burden in actually realizing this increased computational potential is delegated to *software* developers who must somehow find ways to exploit multi-processor platforms without compromising their own productivity. The success of multi-core computers thus depends on the ability to combine the flexibility of software with the performance advantages of hardware. Since the performance of an application running on a multi-processor system depends significantly on the way computational and data transfer resources are shared among tasks, the application programmer is, in some sense, taken back to the dark times where low-level details had to be considered, thus reducing productivity.

This paper is part of an effort to alleviate this problem by *automating*, as much as possible, various deployment decisions associated with the execution of applications on multi-core platforms. Such decisions include the *mapping* of tasks onto processors, *scheduling* the execution of computational tasks on bounded-resource platforms and various data partitioning and sizing issues. In this paper we treat restricted but important problem of choosing the granularity of data transfers between local and remote memory.

Emerging multiprocessor system on chip (MPSoC) , such as the Cell Broadband Engine [Pham et al. 2006] and Platform 2012 [STMicroelectronics and CEA 2010] are heterogeneous multi-core architectures, with a powerful host processor and a number of smaller cores that serve as a computation fabric to accelerate computations previously performed by specialized hardware. Computation intensive tasks such as video processing are delegated to the multi-core fabric to be accelerated via parallel execution. Such applications work on large data arrays initially stored in a remote off-chip memory whose access time is about 100 times larger than that of the core's local memory. To reduce processor stall time a combination of DMA and double-buffering is used. A DMA is a device that can transfer large amounts of data between memory locations without processor intervention. Using double buffering, the processor(s) can work on data residing in their local memory while the DMA is fetching the next batch of data. Therefore, besides single instruction multiple data (SIMD) low granularity level of parallelism, multithreading...etc, heterogeneous multicore architectures offer another level of parallelism by overlapping computations and data prefetching [Gschwind 2007; Sancho et al. 2006].

We focus on algorithms that apply the same computation to each element of a large array. Such algorithms are used, for instance, in image quality improvement, signal processing and many classes of scientific computations. They exhibit regular memory access patterns and hence the explicit control of data transfers can be more efficient than implicit unpredictable low granularity caching mechanisms [Sancho and Kerbyson 2008].

Double buffering is used to enhance performance by executing in parallel the computation of current buffer and the data transfer of the next one. However, it complicates the software developer's work. Besides writing more complex programs, the programmer has to make decisions about data granularity, data partitioning and scheduling of data transfers to achieve optimal performance. We try to provide tools to help the developer make such decisions, and ultimately automate the whole process of data parallelization for such MPSoC platforms.

The problem is of course not new and has been addressed enormously many times in the context of SIMD (single instruction multiple-data) computers, systolic arrays, FPGA, etc. The difference with the present work is the attempt to apply it in a higher level of granularity that corresponds to tasks operating on large elementary data blocks rather at the instruction level, with the hope to find relatively *general* solutions that can eventually be integrated in the software development cycle and in design space exploration.

The rest of the paper is organized as follows. In Section 2 we formulate the problem from the application side and introduce a generic simplified performance model of DMA mechanisms. In Section 3 we derive optimal granularity for data transfers, first in the case of a single processor and then for several ones. In Section 4 we extend the results to overlapped computations where some data are shared among neighbors and explore optimal granularity for different data sharing mechanisms. In Section 5 we validate the models using a cycle-accurate simulator of the Cell architecture [IBM 2009] while in Section 6 we mention some related work. We conclude by a discussion of the present work and its potential extensions.

2. PRELIMINARIES

In this section, we introduce the generic features of the problem from the application software side and the realizing hardware side.

2.1. Double Buffering

We assume an algorithm computing $y = f(x)$ over a basic data unit that we call a *block*.

PROGRAM 1 (SEQUENTIAL).

```
for $i := 0$ to $n-1$ do
    $Y[i] := f(X[i])$
od
```

Consider the sequential algorithm above, which uniformly applies f to a large input array X to produce an output array Y. To simplify notations we assume that both input and output arrays have the same number of blocks n of the same size each. To execute the program we need first to bring the data from the external memory via *dma_get* command, perform the computation and then write back the output via a *dma_put* command, see Program 2.

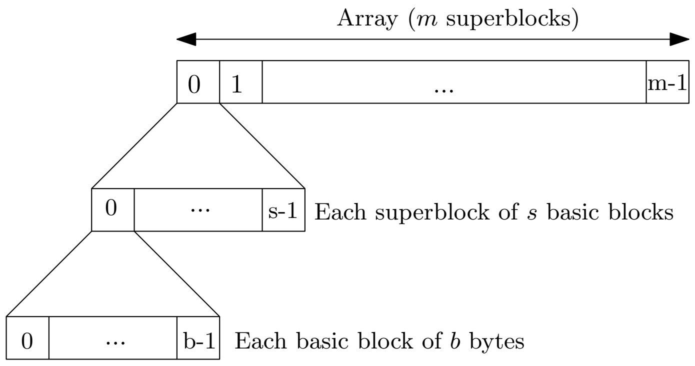

Array (m superblocks)
0
1
...
m-1
0
...
s-1
Each superblock of s basic blocks
0
...
b-1
Each basic block of b bytes

Fig. 1: Decomposition of Input (resp. Output) Array

To amortize the fixed overhead associated with each DMA call, it is typically invoked in a coarse granularity by clustering together several blocks. In other words, the input and output arrays are partitioned into $m = n/s$ *super-blocks* each consisting of s basic blocks stored in the input and output buffers used in Program 2, see Figure 1. A basic block of size b bytes is the minimal granularity the algorithm requires to compute one iteration in Program 1 (in image processing it can be a pixel, a block, a macroblock...) or is chosen as the minimal granularity of transfer.

The event e is associated with each *dma_get* command to ensure the synchronization between data transfer and its corresponding computation.

PROGRAM 2 (BUFFERING).

```
for $i := 0$ to $m-1$ do
    $dma\_get\ (in\_buf,\ X[(i)s\ ..\ (i+1)s-1],\ e);$          fetch in-buffer
    $dma\_wait\ (e);$
    for $j := 0$ to $s-1$ do                                    compute
        $out\_buf[j] = f(in\_buf[j]);$
    od
    $dma\_put\ (out\_buf,\ Y[(i)s\ ..\ (i+1)s-1]);$             write out-buffer
od
```

In program 2, data movements and computations are performed *sequentially* and the processor is idle during data transfers. Using the double buffers, $in\_buf[0]$, $in\_buf[1]$, $out\_buf[0]$ and $out\_buf[1]$, the processor can work on one super-block residing in current buffer $(i\%2)$ while the DMA brings in parallel the next super-block to next buffer buffer $(i+1)\%2$. In the following program, events $e\_in(d)$ and $e\_out(d)$ synchronize data transfers for the corresponding buffers. Program 3 defines a software pipeline with 3 stages: input transfer of super-block $(i+1)$, computation of super-block i and output transfer of super-block $(i-1)$. Fetching the first block and writing back the last block are respectively the beginning and the end of the pipeline.

PROGRAM 3 (DOUBLE BUFFERING).

```
$curr = 0;$
$next = 1;$
$dma\_get\ (in\_buf[0],\ X[0 \mathinner{..} s-1],\ e\_in[0]);$                                   first read
for $i := 0$ to $m-2$ step 1 do
    $dma\_get\ (in\_buf[(next],\ X[(i+1)s \mathinner{..} (i+2)s-1],\ e\_in[next]);$                 fetch next
    $dma\_wait\ (e\_in[curr]);$                                                  wait for current fetch
    $dma\_wait\ (e\_out[curr]);$                                             wait for previous write back
    for $j := 0$ to $s-1$ do                                                    process current
        $out\_buf[curr][j] = f(in\_buf[curr][j]);$
    od
    $dma\_put\ (out\_buf[curr],\ Y[(i)s \mathinner{..} (i+1)s-1],\ e\_out[curr]);$                   write current
    $curr = (i+1)\%2;$                                                           toggle buffers
    $next = (i+2)\%2;$
od
$dma\_wait\ (e\_out[next]);$                                                     wait for last block
for $j := 0$ to $s-1$ do                                                        process last
    $out\_buf[next][j] = f(in\_buf[next][j]);$
od
$dma\_put\ (out\_buf[next],\ Y[n-(s+1) \mathinner{..} n-1],\ e\_out[next]);$                           last write
```

Note that local memory, excluding program code and additional working memory, should be large enough to store 4 buffers of s super-blocks each. There are additional constraints that limit the size of a super-block that we summarize by a bound $\bar{s}$ on the number of blocks that can be clustered in one DMA call. In the following, we look for a number of blocks $s^* \leq \bar{s}$ that yields good performance.[1]

2.2. DMA Performance Model

To reason about optimal granularity, it is important to characterize the DMA behavior. It is quite difficult to model this behavior precisely taking into account all low level hardware details that vary from one architecture to another. Nevertheless, all DMAs share some common characteristics.

The DMA copies data from one memory location to another. When a processor needs data, it issues a transfer command to the DMA which takes charge of the transfer. However, there is an initialization cost which is more amortized for large data transfers. This makes it more efficient for coarse data granularity than for low granularity load/store instructions. DMA transfer commands are composed of a source address, a destination address and a block size. The commands are stored in a queue of limited size. When a command is ready, data transfer begins and the block is divided into smaller packets to travel from source address through an interconnect and read/write issued by the memory controller.

We assume a fixed initialization cost I and a transfer cost of α time units per byte. This value depends on several hardware factors such as interconnect bandwidth, memory latency (which is different according to the type of memory: SRAM of another processor or off-chip DRAM), and possible contentions. Assuming the size of a basic block b bytes, the transfer time of a super-block consisting of s blocks is then approximated as

$$T(s) = I + \alpha \cdot b \cdot s.$$

[1]In this paper, we restrict ourselves to one dimensional data arrays. For higher dimensions, the problem becomes more complex as we need to decide both block size and *shape*.

Note that the model defined here is for transfers of contiguous blocks. Applications working on two dimensional arrays require usually rectangular data block transfers. This is possible using *strided* DMA commands, by specifying in addition to the source and destination address the stride, an offset to access next contiguous block in memory. Strided commands are costlier than contiguous commands of the same size, and they have an extended performance model based also on the shape of the block.

Under some circumstance, performance can be further improved by using more than two buffers. For example, in a triple buffering scheme, the DMA may hold two subsequent requests in the queue, and start the initialization of the second *in parallel* with the transmission of data for the first. Initial experiments with more than two buffers did not show a gain, partly because the requests are not necessarily executed in the same order they are issued. Therefore we need some extra synchronization to avoid that and hence we focus on double buffering.

3. OPTIMAL GRANULARITY

In this section we derive the optimal granularity of data transfers in terms of the number of blocks s clustered in a super-block, starting with the case of one processor.

3.1. One Processor

The execution of Program 3 admits repetitive parallel execution of computation (for super-block i) and data-transfer (for super-block $i+1$). We assume the algorithm f to have a fixed (data independent) computation time ω per basic block once the block is in the local memory of a processing core.

The computation time of a super-block is $C(s) = \omega \cdot s$ while the transfer time using DMA is $T(s) = I + \alpha \cdot b \cdot s$ as defined previously. Both functions are plotted in Figure 2 (a) and their point of intersection s^* (we assume $\alpha \cdot b < \omega$) splits $[1..\bar{s}]$ (the domain of s) into two sub-domains, the *computation regime* where computation of a super-block dominates the transfer of the next one $T(s) < C(s)$, and the *transfer regime*, where $T(s) > C(s)$.

The overall execution time is illustrated in Figure 2 (b). It switches from a transfer regime to a computation regime for granularity s^*. Figure 3 illustrates the behavior of the pipeline for the two regimes. Both admit a prologue (first read with no overlapped computation) and epilogue (last write with no overlapped computation), each consuming $I+\alpha\cdot b\cdot s$ time and $n/s-1$ episodes dominated either by transfer or by computation. Therefore, the overall execution time is

$$\tau(s) = \begin{cases} (n/s+1)\,T(s) \backsimeq (n\cdot I)/s + (n\cdot\alpha\cdot b) & \text{for } s < s^* \\ 2\cdot T(s) + n\cdot\omega \backsimeq (\alpha\cdot b\cdot s) + (n\cdot\omega + I) & \text{for } s > s^* \end{cases}$$

Any granularity $s \geq s^*$ guarantees a computation regime in which the processor never stalls. However, as the granularity increases in this interval, the overhead associated with the prologue and epilogue increases and hence optimal granularity is attained at s^*. This overhead may be negligible when n or ω are large.

Note that, if for any granularity s the execution is always in a computation regime, then optimal unit of transfer is the basic block, that is $s^* = 1$ which guarantees minimal prologue and epilogue. Whereas if the execution is always in a transfer regime, then optimal granularity is the upper bound $\bar{s}$ as DMA initialization overhead is more amortized for larger data blocks.

3.2. Multiple Processors

Given p *identical* processors having the same processing speed and the same local store capacity, the input array is partitioned into p contiguous chunks of data distributed among the processors (see Figure 4) to execute in parallel. Typically $p << n$ and the

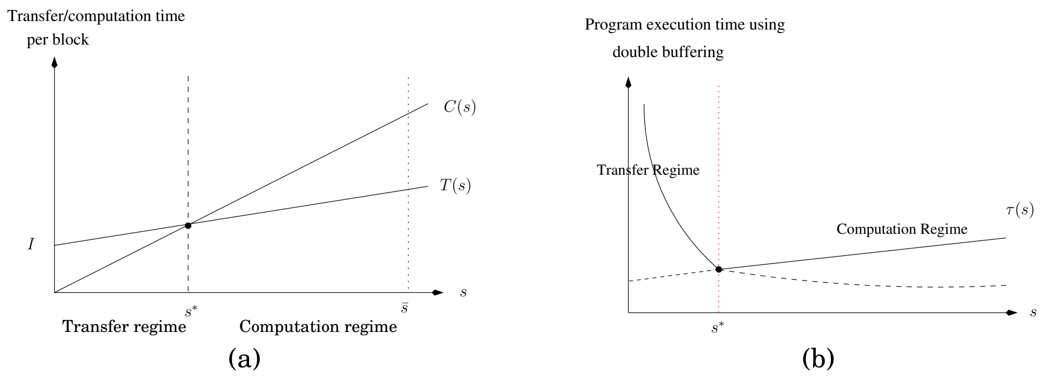


Fig. 2: Computation and transfer regime (a) The dependence of computation C and transfer T on the number of blocks s, (b) Total execution time as a function of s

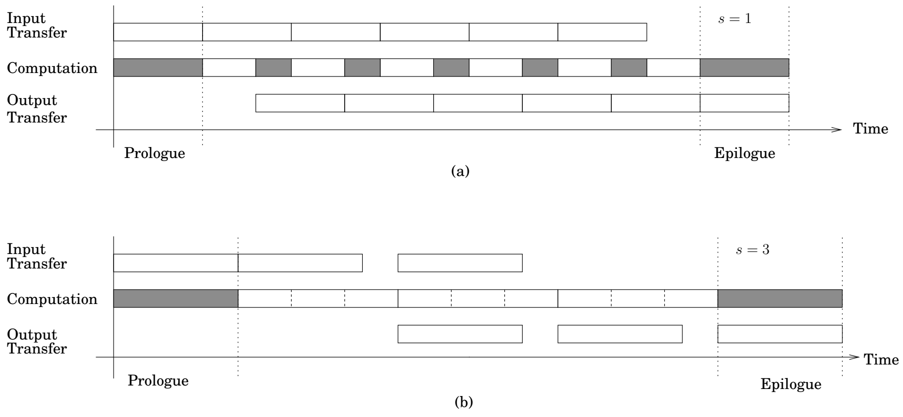


Fig. 3: Pipelined execution on one processor: (a) transfer regime, (b) computation regime. The shaded areas indicate processor stall time.

chunk allocated to a processor is much larger than $\bar{s}$. We extend our efficient double-buffer granularity analysis to this case.

Intuitively, using multiple processors, a conflict arises between computation and data transfers since increasing the number of processors reduces the amount of work per processor but can create contentions on the shared resources and consequently increase transfer time.

DMA mechanisms for multi-core systems can be roughly classified as *centralized* (one device serves all processors) or *distributed* (each processor has its own DMA machine). A distributed system has the advantage of parallelizing the initialization steps of transfer commands, however, all these requests are served by the *same* transfer infra-structure. A distributed mechanism has less control over scheduling such requests while a centralized DMA could avoid contentions, especially for applications that exhibit regular patterns and move large amounts of data. Since our experimental platform, the Cell B.E. (Cell Broadband Engine), as well as other platforms such as Tilera, have a distributed DMA we focus our analysis on the distributed model.

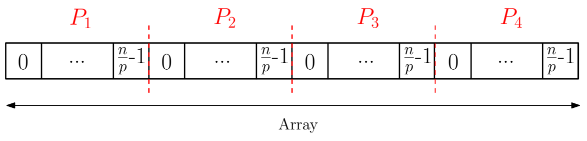


Fig. 4: Contiguous allocation of data blocks per processor.

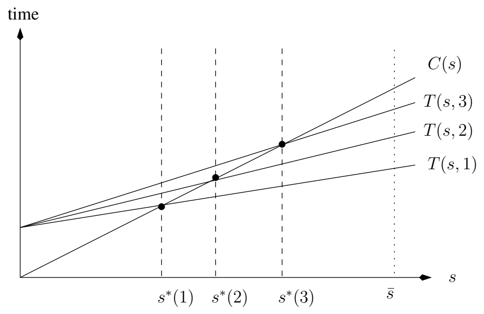


Fig. 5: Optimal granularity for several processors.

To parallelize the initialization phase we synchronize data transfers of all processors at the beginning of the execution.[2] This means that p concurrent transfer requests arrive simultaneously to the interconnect. Arbitration of these requests is left to the hardware which serves the p processors in a low granularity round robin fashion resulting in transfer cost per byte $\alpha(p)$, parameterized by the number of processors. This cost increases monotonically with the number of processors, see Figure 5, as contentions overhead increase with the number of competing processors transfer requests.

The characterization of transfer time of s blocks by a processor becomes

$$T(s,p) = I + \alpha(p) \cdot s \cdot b$$

and optimal granularity for p processors is attained at $s^*(p)$ satisfying $T(s^*(p), p) = C(s^*(p))$ which indicates that the computation time for the amount of data needed to keep a processor busy is exactly the time it takes to fetch its next block as well as the next block of each of the other processors. As one can see in Figure 5, optimal granularity increases with the number of processors due to the increase in transfer time. The overall execution time $\tau(s,p)$ has the same form as for one processor:

$$\tau(s,p) = \begin{cases} (n/sp + 1)\ T(s,p) & \text{for } s < s^* \\ 2 \cdot T(s,p) + n/p \cdot \omega & \text{for } s > s^* \end{cases}$$

Note that beyond some value of p we are always in transfer regime and there is no point in using more processors.

Table I summarizes the notations for the used hardware and software parameters.

4. OVERLAPPED DATA COMPUTATIONS

An overlapped data computation is a data parallel loop in which the computation for each block needs additional data from neighboring blocks. In other words the compu-

[2]The gain from overlapping initialization is less significant for large granularity, see further discussions concerning this choice in Section 5.

Table I: Parameters Notation

n	number of blocks to be processed
ω	computation time per basic block
b	size in bytes of a basic block
s	number of blocks in a super-block
p	number of processors active at a time
I	initialization cost for transfer
α	transfer cost in time units per byte
$\alpha(p)$	transfer cost per byte when p processors are active
β	transfer cost per byte for inter-processor communication
γ	cost per byte of a load/store instruction for copying overhead
R_k	inter-processor communication time to transfer a data block of size k
$T(s)$	transfer time for s blocks
$T(s,p)$	transfer time of s blocks by on processor when p processors are active
$C(s)$	computation Time
$X[i]$	input data vector
$Y[i]$	output data vector
$V[i]$	neighborhood data vector

tation in the inner loop of Program 1 is replaced by

$$Y[i] = f(X[i], V[i])$$

where $V[i]$ is additional data taken from the input left neighbor $X[i-1]$. We assume that the relative location of $V[i]$ in $X[i-1]$ is the same for every i, and that its size is some fixed $k \in [0, b]$. Such patterns are common, for example, in signal processing application with overlapping time windows. As before we partition the workload among p processors and compare several data distribution and sharing strategies for which we construct performance models that can be used to derive optimal granularity . The three strategies that we compare differ in the component of the architecture which carries the burden of transferring the additional data in $V[i]$:

(1) Replication: transfer via the interconnect between local and off-chip memory;
(2) Inter-processor communication: transfer via the network-on-chip between the cores;
(3) Local buffering: transfer by the core processors themselves.

We discuss these strategies briefly below.

4.1. Replication

For each granularity s, the input array X is divided into contiguous super-blocks of size $(s \cdot b + k)$ and allocated to processors. The analysis is therefore similar to section 3.2 considering at each iteration an overhead of k additional bytes for each super-block transfer.

Considering this overhead, optimal granularity is reached for the granularity s^* so that, $T(s^* + k/b,\ p) = C(s^*)$. The overall execution time is similar to section 3.2:

$$\tau(s,p) = \begin{cases} (n/sp + 1)\ T(s + k/b,\ p) & \text{for } s < s^* \\ 2 \cdot T(s + k/b,\ p) + (n/p)\omega & \text{for } s > s^* \end{cases}$$

4.2. Inter-processor Communication

Let R_k be the communication time to transfer a data block of size k from one processor to another. It obviously depends on the transfer mechanism used: load/store instructions via a shared memory location, explicit DMA commands, etc. In this paper, we consider blocking DMA commands that have the same initialization cost I, and a transfer cost per byte $\beta \ll \alpha$ because the network-on-chip connecting the cores as well

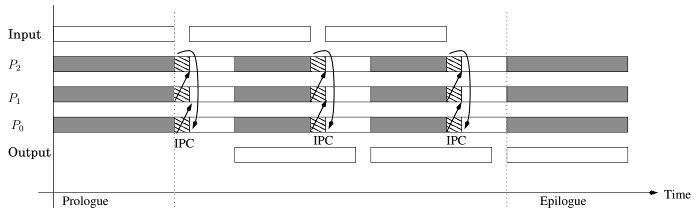


Fig. 6: Inter-processor communication

as their memories is much faster. Thus R_k is approximately $I + \beta \cdot k$ with, perhaps, additional terms that depend on p to represent synchronization overhead between the cores.

Optimal data partitioning using this mechanism occurs when the communication geometry matches the interconnect geometry which is ring-like in our experimental platform. Processors can then perform in parallel a point to point inter-processor communication. Unlike the other strategies, here we need to allocate memory to processors in a periodic rather than contiguous fashion. As in Section 3.2, transfers from external memory are synchronized and after reception of a super-block each processor p_j sends the overlapping data to its right neighbor p_{j+1}. Processors stall at that time waiting for the end of the communication. Prefetching of next blocks can then be issued to be done concurrently with computations, see Figure 6.

Optimal granularity is therefore reached for s^* so that,

$$T(s^*, p) = C(s^*) + R_k$$

and the overall execution time is,

$$\tau(s,p) = \begin{cases} (n/sp + 1)\ T(s,p) & \text{for } s < s^* \\ 2 \cdot T(s,p) + (n/p)\omega + (n/sp)R_k & \text{for } s > s^* \end{cases}$$

4.3. Local Buffering

The input array is divided into p contiguous large chunks. Neighboring blocks are therefore allocated to the same processor and computed at successive iterations with shared data being stored in the processor local memory for the next iteration.

In this strategy, at each iteration, shared data has to be copied from one local buffer to the other using load/store instructions.[3] The local copying overhead is $k \cdot \gamma$ where γ is the cost per byte of a load/store instruction. It is a fixed computation overhead added at each iteration of computing over a super-block of size $s \cdot b$. Note that this overhead relative to the whole execution time decreases as the number of processors p and the granularity s increases.

Optimal granularity is obtained at s^* so that $T(s^*, p) = C(s^*) + k \cdot \gamma$ and the overall execution time $\tau(s, p)$ is,

$$\tau(s,p) = \begin{cases} (n/sp + 1)\ T(s,p) & \text{for } s < s^* \\ 2 \cdot T(s,p) + (n/p)\omega + (n/sp)(k \cdot \gamma) & \text{for } s > s^* \end{cases}$$

4.4. Comparing Strategies

These approximate models can be used to give some guidelines for choosing granularity. They involve, on one hand, parameters of the application: b, k and ω (assum-

[3]More efficient pointer manipulation is hard to implement because the same buffer is used to store $x[i-1]$ and $x[i+1]$ which is filled in parallel with the computation of $y[i]$.

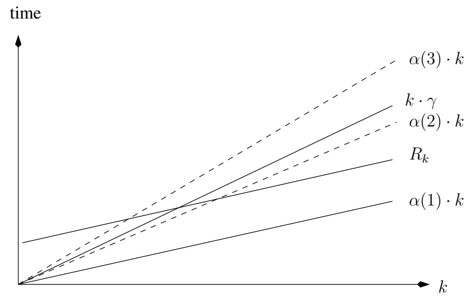


Fig. 7: Comparing parameters for strategies

ing processor speed is fixed) and parameters of the architecture: α, β, γ. According to these models one can optimize the decision variables which are p, s and the sharing strategy. Before moving to experimental validation let us make some observations. In inter-processor communication and in local buffering each iteration involves additional *processor* workload and needs to transfer less data from off-chip memory. Therefore, replication switches to the computation regime at a higher granularity than local buffering and inter-processor communication.

When all three strategies reach their computation regime, replication always performs better than inter-processor communication and local buffering because of the overall overhead that corresponds to the time the processor spends copying shared data locally, or communicating with an other processor: $(n/sp)R_k$ or $(n/sp)(k \cdot \gamma)$. However, the advantage of replication is reduced as s or p increase.

When transfer time is dominant, comparing the strategies boils down to comparing the cost of transferring the *additional* shared data using different hardware mechanisms: $k \cdot \alpha(p)$, $I + k \cdot \beta$ and $k \cdot \gamma$, Figure 7 illustrates the sensitivity of these quantities to k and p. In this example, for one processor, replication cost is lower than local copying and inter-processor communication, but as the number of processors increases, the overhead of contentions while accessing off-chip memory ruins the performance compared to the other strategies, where the transfer of shared data is delegated to the high speed network-on-chip or to the processors and is totally or partly done in parallel.

Note that in the local buffering and IPC schemes, the transfer of shared data is counted as computation time because in local buffering, the processor is busy copying data using load/store instructions and in the IPC the processor is idle waiting for the *synchronous* DMA transfers to finish.

Note that if the optimal granularity for the replication strategy does not fit in the memory limit, the parameters comparison can then give a hint about the strategy that can give best performance given the available memory space budget.

5. EXPERIMENTS

In this section our goal is to validate these simplified models against a real architecture, the Cell Broadband Engine Architecture (Cell BE), represented by a cycle-accurate simulator, and see whether their predictive power is sufficiently good to serve as a basis for optimization decisions.

5.1. The Cell BE

The Cell Broadband Engine Architecture is a 9-core heterogeneous multi-core architecture, consisting of a Power Processor Element (PPE) linked to 8 Synergistic Processing

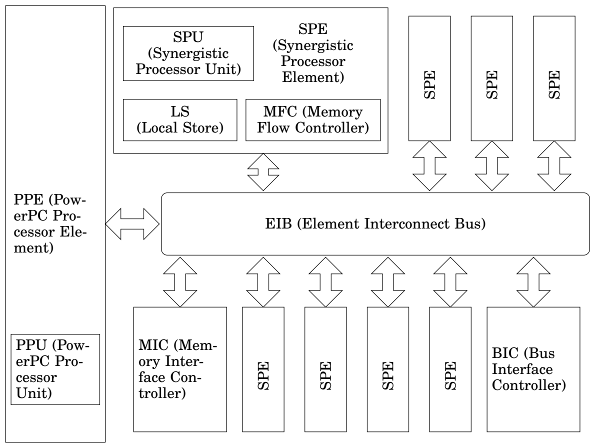

SPU (Synergistic Processor Unit)
SPE (Synergistic Processor Element)
LS (Local Store)
MFC (Memory Flow Controller)
SPE
SPE
SPE
PPE (PowerPC Processor Element)
EIB (Element Interconnect Bus)
PPU (PowerPC Processor Unit)
MIC (Memory Interface Controller)
SPE
SPE
SPE
SPE
BIC (Bus Interface Controller)

Fig. 8: Cell B.E. Architecture

Elements (SPE) acting as co-processors, through internal high speed Element Interconnect Bus (EIB) as shown in Figure 8.

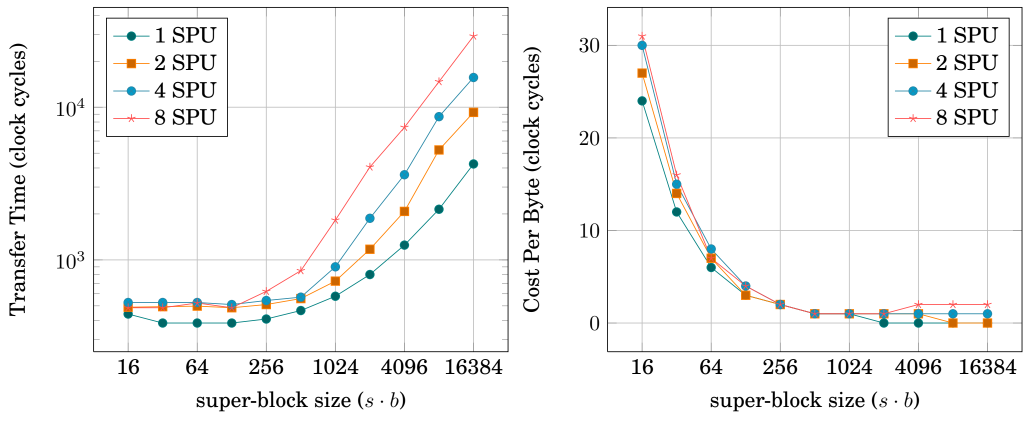
The PPE is composed of a 64-bit PowerPC Processor Unit (PPU) and each SPE is composed of a Synergistic Processing Unit (SPU) which is a vector processing unit, an SRAM local store (LS) of size 256 kbytes shared between instructions and data, and a Memory Flow Controller (MFC) to manage DMA data transfers. The PPU provides a single shared address space across SPEs and the MFC's memory translation unit handles the required address translation. An SPE can access the external DRAM and the local store of other SPEs only by issuing DMA commands. The PPU can also issue DMA commands supported by the MFC. The MFC supports aligned DMA transfers of 1, 2, 4, 8, 16 or a multiple of 16 bytes, the maximum size of one DMA transfer request being 16K. To transfer more than 16K, DMA lists are supported. Further details of the architecture can be obtained at [IBM 2008].

The MFC of each SPE is composed of a Direct Memory Access Controller (DMAC) to process DMA commands queued in the MFC and of a Memory Management Unit (MMU) to handle the address translation of DMA selected commands. The MMU has a translation look-aside buffer (TLB) for caching recently translated addresses. TLB misses can affect significantly the performance, therefore we neglect this effect by doing a warm-up run which will load the TLB entries before profiling of the program. Also, we allocate large page tables to have a smaller number of TLB entries for data array. After the address translation, the DMAC splits the DMA command into smaller bus transfers and peak performance is achievable when both the source and destination address are 128-byte aligned and the block size is multiple of 128 bytes [Kistler et al. 2006]. We can observe reduction in performance when this is not the case.

In the Cell Simulator that we use [IBM 2009] the PPE and SPE processors run at 3.2GHz clock frequency and the interconnect is clocked with half the frequency of the processors. The cell-simulator is very close to the actual processor. However, it models DDR2 memory instead of RAMBUS XDR which is used in the real processor. Further it does not have Replacement Management table for cache, however, we measure the performance directly on SPU's and it does not affect our model.

5.2. Basic Parameters Measurement

We measure a DMA transfer time as we increase the super block size in one transfer, and accordingly the cost per byte, see Figure 9. The DMA cost per byte is reduced sig-



(a) DMA transfer time (b) DMA Read Cost per Byte

Fig. 9: DMA performance

nificantly for block size larger than 128 bytes. Below this value, the DMA has a high initialization overhead. As we increase the number of processors and synchronize concurrent DMA transfers, we can observe that the transfer time is not highly affected for a small granularity because the initial phase of the transfer is done in parallel in each processor's MFC, whereas for large granularity the transfer time increases proportionately to the number of processors due to the contentions of concurrent requests on the bus and bottleneck at the Memory Interface Controller (MIC) as explained in [Kistler et al. 2006]. The initialization phase time I is about 400 cycles and the DMA transfer cost per byte to read from main memory $\alpha(1)$ is about 0.22 cycles per byte, it increases proportionately to the number of processors to reach $p \cdot \alpha(1)$ (for large granularity transfers).

5.3. Experimental Results

We run experiments implementing double buffering for independent and overlapped data computations, first on synthetic algorithms of computation where f is an abstract function for which we vary the computation workload per byte ω. We then implement a convolution algorithm that computes an output signal based on an input signal and an impulse response signal. The size of the impulse signal determines the size of the data window required to compute one output item. Moreover, we vary the size of the impulse signal to vary the size of the neighboring shared data.

For synthetic algorithms, we fix the size b of a basic block to 16 bytes and the size n of input data array to 64K, the total size of the array being 1Mbytes. Also we keep the number of threads that are spawn on SPUs as maximum equal to number of SPU's. This will avoid context switching and give more predictable results. Also there will be no question of scheduling which is another part of the problem and not in the scope of this paper. The memory capacity limits then the possible number of blocks clustered in one transfer to $\bar{s} < 4K$, excluding memory space allocated to code size. We vary the number of blocks s and the number of processors. We compare both predicted and measured optimal granularity, and the total execution time for both transfer and computation regimes. Figure 10 shows the predicted and measured values for 2, 4 and 8 processors. We can observe that values are very close to each other. The predicted optimal points are not exactly the measured ones but they give very good performance. Per-

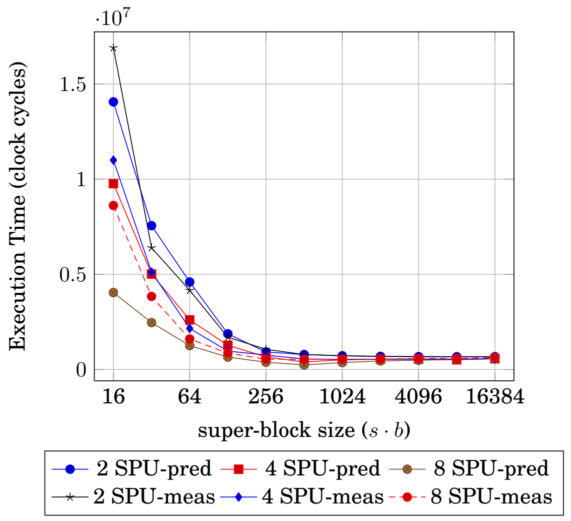


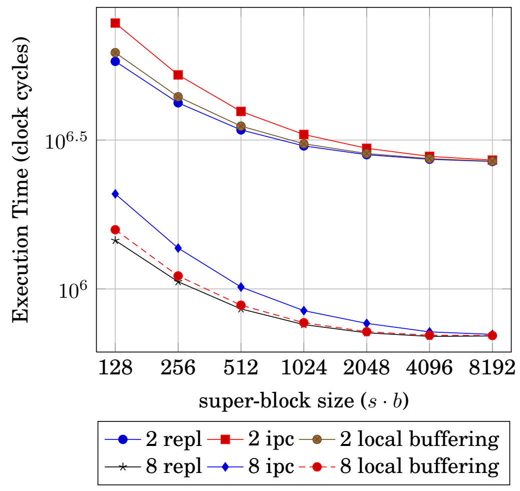
Fig. 10: Independent data computations

formance prediction in the computation regime is better than in the transfer regime, because the computations which constitute the dominant part of the total execution time is performed on processors. Besides, as mentioned in [Sancho et al. 2006] we have sufficient time to hide delays caused due to the network latency and bandwidth.

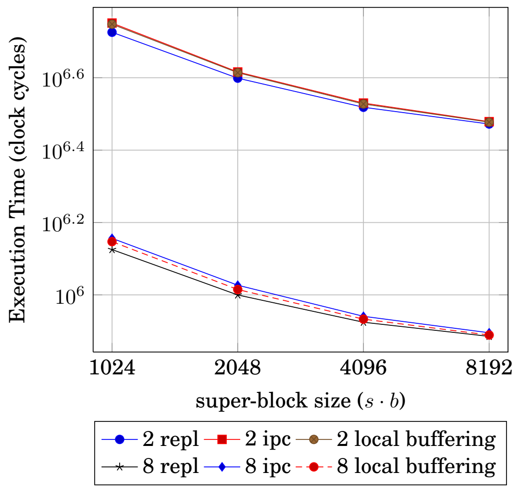
For overlapped data computations, we implement the different strategies for transferring shared data explained in section 4. We run experiments with different values of k, s and p, for both computation and transfer regimes by varying the computation workload ω. We present the results for a small and a large value of k, 128bytes and 1K respectively, and for 2 and 8 processors.

In Inter-Processor Communication (IPC) strategy, we make sure that neighboring processors exchanging data are physically mapped close to each other. Specifying affinity during thread creation in linux allows the logical threads to be mapped physically next to each other. This gives advantage of having higher bandwidth as mentioned in [Sudheer et al. 2009]. The global data partitioning specified in section 4.2 has a ring geometry of communication similar to the EIB topology, so that each processor can send data to its right neighbor. The processors must synchronize with each other to send shared data after the DMA request for fetching a super-block from main memory has completed. We experiment with two variants of IPC synchronization: a point to point signaling mechanism to check the availability of shared data and acknowledge the neighbor, and a global barrier based on signals mentioned in [Bai et al. 2008] to synchronize all processors. Because of the high synchronization overhead compared to point to point signaling, we do not present the results obtained with this variant here. After processors synchronization, the shared data transfer is done by issuing a DMA command.

As discussed in section 4.4, in the computation regime replication performs always better than local buffering and IPC as shown in Figures 11a and 11b. Besides, we can see that IPC performs worse than local buffering because the cost per byte γ via load/store operations which is around 2 cycles per byte, is much lower than the cost R_k to perform IPC synchronization and DMA calls. R_k involves a DMA inter-processor cost per byte β and a synchronization cost. In practice, it is very difficult to estimate R_k precisely, mainly because of the difficulty in prediction of exact arrival time in



(a) shared data size 128 bytes



(b) shared data size 1024 bytes

Fig. 11: Overlapped data in computation regime

presence of continuous read/write data-transfers. In our experiments, the initial time of an inter-processor DMA command was around 200 cycles, the transfer cost per byte β around 0.13 cycles and the synchronization cost between 200 and 500 cycles as we vary the number of processors. The synchronization cost using barrier is much larger, between 800 and 2400 at each iteration.

In the transfer regime, performance varies according to the value of k and number of processors. We can observe in Figure 12a that the costs of local buffering and replication are nearly the same, and that replication performs even better for a transfer of block size between 512bytes and 2K. This demonstrates that using DMA for transferring additional data can perform sometimes better than local buffering even for a small value of k, and that keeping shared data in the local store may have a non-negligible cost. Therefore, even when considering contiguous partitioning of data, redundant fetching of shared data using replication strategy can be as efficient, if not more efficient than keeping shared data in the local store. However, the cost of transferring shared data using replication becomes higher than other strategies when the number of processors increase because of the contentions even for small values of k.

In the following, we detail the results for each strategy in terms of efficiency and conformance with the prediction of the models.

— *Replication*: For computation regime, the measured results are very close to the predicted ones. The only source of error would be the contentions on the network with huge network traffic which causes differences in the arrival time of the data. The error between the measured and predicted values is about 3%. In the transfer regime, replication always performs worse than other strategies for 8 processors due to contentions.
— *IPC*: When synchronization is done using messages between neighboring processors, we observe variabilities in the arrival time of data transfers and exchanged messages due to contentions in the network. This effect increases in the transfer regime which makes the gap between the measured and estimated performance bigger, with an error of about 6%, that goes to 30% when barriers are used with a high number of processors.

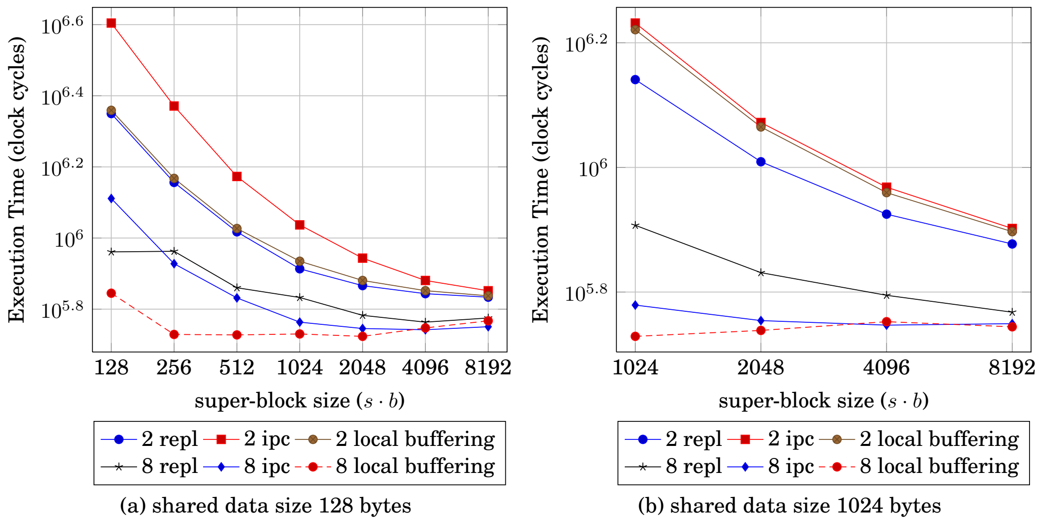


Fig. 12: Overlapped data in transfer regime

— *Local buffering*: In the computation regime, local buffering outperforms IPC for most of the cases, whereas in the transfer regime, the DMA engine can be more efficient for transferring shared data than copying it from one buffer to another in the local store, despite the inter-processor synchronization overhead.

Note that for a given k, as super-block size increases the cost of transferring shared data relative to the overall execution time decreases and all strategies give similar performance results. In the transfer regime, the gap between estimated and measured performance becomes larger as it is more dependent upon contentions in the network traffic for which our modeling is less accurate. There are two major sources of contentions that we currently do not model:

(1) In the 3 stage software pipeline there is an overlap between reading super-block i and writing super-block $i-1$. This is the main reason why the estimated optimal granularity point is in reality still in the transfer regime.
(2) Inter-processor synchronization, in which messages exchanged between processors add contention to the network. This overhead increases with the number of processors even for the more efficient inter-processor signaling variant. The exact arrival time of each message is difficult to model owing to continuous read/write network traffic and the scheduling policy of the DMA controller.

We believe that more detailed models can further improve performance predictions.

5.4. Convolution

Convolution is one of the basic algorithm is signal processing [Nussbaumer 1981]. Assuming an input signal array X of a large size n and an impulse system response signal array B of a smaller size m ($m << n$), the output system signal array Y of same size n is computed as follows,

$$Y[i] = \sum_{j=0}^{m} X[i-j] \cdot B[j]$$

Therefore to compute each sample $Y[i]$ of the output signal, a window of m data samples is required from input array X. When multiple processors are used, input array is partitioned into contiguous chunks, see Figure 13. The area in grey illustrates shared data between processors.

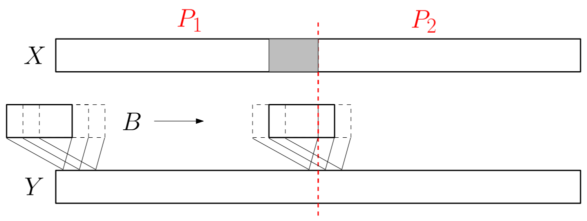


Fig. 13: Convolution Algorithm.

In our experiments, the size of input array X is chosen to be 1Mbytes of data, so it cannot fit in the scratchpad memory, whereas B is small enough to be permanently stored in each SPU's local store. Hence double buffering is implemented to transfer data blocks of array X (resp. Y). As for the synthetic examples, we compare the different strategies of section 4 and we vary the size of B to vary the size of shared data.

Signal samples are encoded as *double* data type. The minimal granularity size b is chosen to be the size of the data window required to compute one output data sample, that is $b = m \cdot 8$ (8 being the size of a *double* data type). In the implementation of the algorithm, we use SIMD operations to optimize the code. The measured cost per byte ω is about 53 cycles.

Note that for this algorithm, despite an optimized implementation using vector operations the computation cost per byte ω is much higher than the transfer cost per byte with maximum contentions resulting from the use of maximum number of available cores, $\alpha(8)$ being 7.22 cycles. Therefore, the overall execution is always in a *computation regime* for all strategies.

The reason why the cost per byte for this algorithm is so high is the use of *double* data type to encode signals samples which does not take fully advantage of the 16 bytes SIMD engine since operations on at most 2 elements of the array can be done in parallel. Floating point data types would take more advantage from the SIMD operations, however at the cost of results accuracy since the Cell B.E. does not support a floating point unit. Besides since SPU's general registers are SIMD registers, this makes operations on the SPU unoptimized for scalar operations and branching instructions, resulting in a high execution latency.

Figure 14 summarizes performance results for size of $B = 256$ bytes that is 32 samples, using 2 and 8 processors. As explained in section 4.4, in the computation regime replication strategy outperforms local buffering and IPC strategies since it avoids the computational overhead at each iteration of copying shared data locally or exchanging data between neighboring processors using synchronous DMA calls. This overhead is proportional to the number of iterations and therefore decreases with higher granularities to be eventually negligible which leads all strategies to perform with nearly the same efficiency.

Note that in the program execution time estimation in sections 3 and 4, we ignored the overhead at each iteration of setup variables. It is proportional to the number of iterations and therefore is reduced for high granularity.

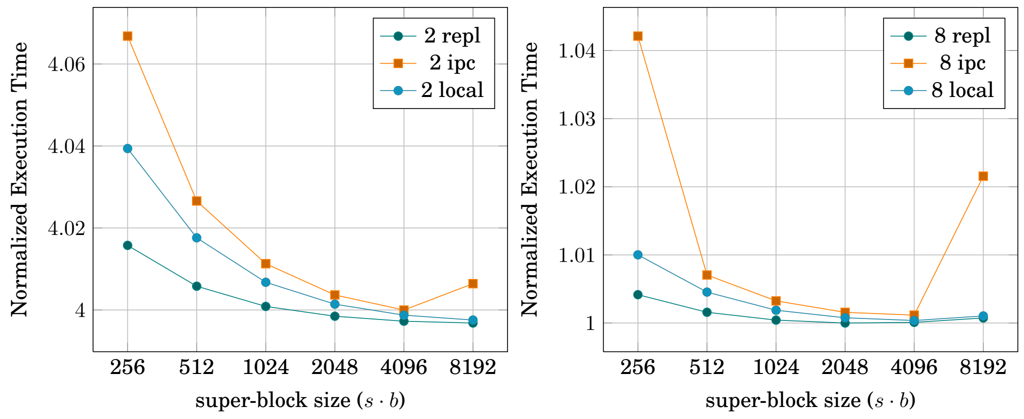


Fig. 14: Convolution using double buffering, shared data size is 256 bytes

6. RELATED WORK

Data prefetching is a well and intensively studied topic for cache based architectures to improve memory performance by hiding memory access latency. Several techniques have been proposed ranging from purely hardware prefetching solutions such as [Dahlgren et al. 1995; fu Chen and loup Baer 1995; Fritts 2002] requiring a hardware unit connected to the cache to handle prefetching at runtime but at the cost of extra circuitry, to software prefetching approaches such as [Mowry and Gupta 1991; Callahan et al. 1991] relying on compiler's analysis to insert additional fetch instructions in the code. Other works such as [Chen and Baer 1994; Gornish and Veidenbaum 1999; Wang et al. 2003] combined both hardware and software prefetching approaches.

Furthermore, splitting large amount of data into smaller partitions to optimize program execution is an old and known problem [Wolfe 1989] for non-DMA architectures. For various cache-based architectures, loop tiling is performed in order to have optimal performance [Coleman and McKinley 1995; Lam et al. 1991; Esseghir 1993; Agarwal et al. 1995] which can be considered analogous to the choice of data granularity on architectures with DMA.

A major characteristic of the Cell B.E. (and other more recent multi-core computers) is software managed local storage rather than a hidden cache mechanisms. This simplifies hardware design but at the cost of program complexity because programmers have to manage explicitly the memory hierarchy. To handle these issues new programming models have been introduced such as *Sequoia* and *Cellgen* [Fatahalian et al. 2006], [Schneider et al. 2009], [Yeom et al. 2009]. There are more abstract theoretical models such as re bulk synchronous parallel (BSP) model [Valiant 1990], LogP [Culler et al. 1993], and HMM (Hierarchical Memory Model) [Aggarwal et al. 1987] that are intended to analyze the concrete complexity of parallel execution of algorithms.

In [Chen et al. 2007] a more general parametric analysis is presented for evaluating multi buffering for data independent parallel loops. Given a space budget, it determines the optimal buffering scheme (simple, double, or triple buffering) and optimal buffer size. However, this analysis is done only for independent data computations and for one processor. It does not address the problem with multiple SPUs competing for the same resources and data exchange between SPUs, which is dealt as a separate topic in [Kistler et al. 2006].

The work of [Carpenter et al. 2010] propose a compiler based approach for buffer size allocation. The target applications are streaming applications modeled as data flow graphs. They propose an algorithm to determine which edge to enlarge and by how many buffers in order to increase performance. They perform experiments on an IBM QS blade which has two Cell BEs. Even though the approach is more general, it is not clear how the use of DMA and contentions are modeled and how they influence performance.

There are several of publications which are trying to enhance the performance of the Cell B.E. on various applications. For example, [Blagojevic et al. 2008] refers to another modeling of parallel applications particularly targeted at the Cell. However, it remains silent about the variation of buffer sizes with respect to double buffering. [Sancho and Kerbyson 2008] explains implementing double buffering on Cell B.E. but without formal analysis.

The work of [Zinner and Kubinger 2006] (done on Texas Instrument TMS320C6000 processor) comes very close to ours. However, it restricts itself to the uni-processor scenario. With the Cell architecture, we have extended the double buffering scenario to more than one SPU simultaneously accessing the resources.

Some other work like [Beltran et al. 2009] and [Ahmed et al. 2008] address the problem of overlapping computations and data transfers by using multithreading so that the context switch occurs at the request of data not available in local store. This technique can be more efficient than a double or multi-buffering technique for applications with less regular computation and data access patterns.

Methodologically speaking, the work of [Petrini et al. 2007] studies the gain on performance with respect to architecture related enhancements. Further it suggests that such metrics should be automatized and integrated in the compiler to achieve peak performance, which is congruent to our proposition.

7. DISCUSSION

This paper was motivated by one of the major problems that will determine the success or failure of multi-core computers: how to facilitate the development of efficient application software while avoiding as much as possible the need to solve challenging combinatorial puzzles for each new algorithm and each new architecture.

To demonstrate a potential automation of such decisions we have targeted a relatively-easy (but important) class of applications with regular patterns of computation and (high volume) data transfers. Under some simplifying assumptions we constructed models, with parameters obtained by profiling, that turned out to approximate reasonably well the behavior of synthetic programs on a real architecture. A crucial point of our methodology is that the hardware characterization and modeling phase can, in principle, be done once for each new platform and then be used by different applications running on it. Our current plans are to repeat the same experience on the Tilera platform.

Modeling always involves a tension between model fidelity and the tractability of analyzing it. We believe that further experience will sharpen our intuitions about the appropriate level of model granularity. Let us mention some limitations and possible extensions of this work. On the hardware side, it should be noted that we (and the Cell simulator that we employ) do not use a detailed model of the external memory that could induce further variations in access time. It will be also interesting to study the performance of centralized DMA mechanisms that are more amenable to scheduling policies that avoid contentions.

In terms of the application models, many extensions are planned for subsequent work including:

— Allowing some limited variations in computation time and data volumes.

— Combining data-parallelism with task-level parallelism. This will change the symmetric nature of the problem and may create new bottlenecks. Consideration of code distribution should also be taken into account;
— Adding streaming aspects with new instances of the array arriving in some periodic or quasi-periodic fashion. In this scenario instance pipelining will play a major role;
— Allowing data dependent executions;
— Experimenting with additional applications exploiting more efficiently computation optimization capabilities of the SPU cores in the Cell B.E.

A particular difficulty in the embedded multi-core domain is the cultural gap between the software and hardware communities that hold different partial views of the same system. Our models, based on a clear separation between parameters characterizing the applications and those which are specific to the hardware platform will hopefully help in bridging these gaps and contribute to progress in this important domain.

REFERENCES

AGARWAL, A., KRANZ, D., AND NATARAJAN, V. 1995. Automatic partitioning of parallel loops and data arrays for distributed shared-memory multiprocessors. *Parallel and Distributed Systems, IEEE Transactions on 6,* 9, 943 –962.

AGGARWAL, A., ALPERN, B., CHANDRA, A., AND SNIR, M. 1987. A model for hierarchical memory. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. STOC '87. ACM, New York, NY, USA, 305–314.

AHMED, M., AMMAR, R., AND RAJASEKARAN, S. 2008. Spenk: adding another level of parallelism on the cell broadband engine. In *Proceedings of the 1st international forum on Next-generation multicore/manycore technologies*. ACM, 1–10.

BAI, S., ZHOU, Q., ZHOU, R., AND LI, L. 2008. Barrier synchronization for cell multi-processor architecture. In *Ubi-Media Computing, 2008 First IEEE International Conference on*. 155 –158.

BELTRAN, V., CARRERA, D., TORRES, J., AND AYGUADÉ, E. 2009. CellMT: A cooperative multithreading library for the Cell/BE. In *High Performance Computing (HiPC), 2009 International Conference on*. IEEE, 245–253.

BLAGOJEVIC, F., FENG, X., CAMERON, K. W., AND NIKOLOPOULOS, D. S. 2008. Modeling multigrain parallelism on heterogeneous multi-core processors: a case study of the cell be. In *Proceedings of the 3rd international conference on High performance embedded architectures and compilers*. HiPEAC'08. Springer-Verlag, Berlin, Heidelberg, 38–52.

CALLAHAN, D., KENNEDY, K., AND PORTERFIELD, A. 1991. Software prefetching. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*. ASPLOS-IV. ACM, New York, NY, USA, 40–52.

CARPENTER, P., RAMIREZ, A., AND AYGUAD, E. 2010. Buffer sizing for self-timed stream programs on heterogeneous distributed memory multiprocessors. In *High Performance Embedded Architectures and Compilers*, Y. Patt, P. Foglia, E. Duesterwald, P. Faraboschi, and X. Martorell, Eds. Lecture Notes in Computer Science Series, vol. 5952. Springer Berlin / Heidelberg, 96–110.

CHEN, T., SURA, Z., O'BRIEN, K., AND O'BRIEN, J. K. 2007. Optimizing the use of static buffers for dma on a cell chip. In *Proceedings of the 19th international conference on Languages and compilers for parallel computing*. LCPC'06. Springer-Verlag, Berlin, Heidelberg, 314–329.

CHEN, T.-F. AND BAER, J.-L. 1994. A performance study of software and hardware data prefetching schemes. *SIGARCH Comput. Archit. News 22*, 223–232.

COLEMAN, S. AND MCKINLEY, K. S. 1995. Tile size selection using cache organization and data layout. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*. PLDI '95. ACM, New York, NY, USA, 279–290.

CULLER, D., KARP, R., PATTERSON, D., SAHAY, A., SCHAUSER, K. E., SANTOS, E., SUBRAMONIAN, R., AND VON EICKEN, T. 1993. LogP: towards a realistic model of parallel computation. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*. PPOPP '93. ACM, New York, NY, USA, 1–12.

DAHLGREN, F., DUBOIS, M., AND STENSTRÖM, P. 1995. Sequential hardware prefetching in shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst. 6*, 733–746.

ESSEGHIR, K. 1993. Improving data locality for caches.

FATAHALIAN, K., HORN, D., KNIGHT, T., LEEM, L., HOUSTON, M., PARK, J., EREZ, M., REN, M., AIKEN, A., DALLY, W., ET AL. 2006. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. ACM, 83–es.

FRITTS, J. 2002. Multi-level memory prefetching for media and stream processing. In Multimedia and Expo, 2002. ICME '02. Proceedings. 2002 IEEE International Conference on. *Multimedia and Expo, 2002. ICME '02. Proceedings. 2002 IEEE International Conference on 2*, 101–104 vol.2.

FU CHEN, T. AND LOUP BAER, J. 1995. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers 44*, 609–623.

GORNISH, E. H. AND VEIDENBAUM, A. 1999. An integrated hardware/software data prefetching scheme for shared-memory multiprocessors. *International Journal of Parallel Programming 27*, 35–70. 10.1023/A:1018792002672.

GSCHWIND, M. 2007. The cell broadband engine: exploiting multiple levels of parallelism in a chip multiprocessor. *International Journal of Parallel Programming 35,* 3, 233–262.

IBM. 2008. Cell SDK 3.1. https://www.ibm.com/developerworks/power/cell/.

IBM. 2009. Cell Simulator. http://www.alphaworks.ibm.com/tech/cellsystemsim.

KISTLER, M., PERRONE, M., AND PETRINI, F. 2006. Cell multiprocessor communication network: Built for speed. *Micro, IEEE 26,* 3, 10 –23.

LAM, M. D., ROTHBERG, E. E., AND WOLF, M. E. 1991. The cache performance and optimizations of blocked algorithms. *SIGOPS Oper. Syst. Rev. 25*, 63–74.

MOWRY, T. AND GUPTA, A. 1991. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing 12*, 87–106.

NUSSBAUMER, H. J. 1981. *Fast Fourier transform and convolution algorithms*. Springer-Verlag, Berlin ; New York :.

PETRINI, F., FOSSUM, G., FERNANDEZ, J., VARBANESCU, A., KISTLER, M., AND PERRONE, M. 2007. Multicore surprises: Lessons learned from optimizing sweep3d on the cell broadband engine. In *IPDPS2007*. Vol. CDROM. IEEE Society, 1–10.

PHAM, D., ANDERSON, H.-W., BEHNEN, E., BOLLIGER, M., GUPTA, S., HOFSTEE, H. P., HARVEY, P. E., JOHNS, C. R., KAHLE, J. A., KAMEYAMA, A., KEATY, J. M., LE, B., LEE, S., NGUYEN, T. V., PETROVICK, J. G., PHAM, M., PILLE, J., POSLUSZNY, S. D., RILEY, M. W., VEROCK, J., WARNOCK, J. D., WEITZEL, S., AND WENDEL, D. F. 2006. Key features of the design methodology enabling a multi-core soc implementation of a first-generation cell processor. In *ASP-DAC*, F. Hirose, Ed. IEEE, 871–878.

SANCHO, J. AND KERBYSON, D. 2008. Analysis of double buffering on two different multicore architectures: Quad-core Opteron and the Cell-BE. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 1–12.

SANCHO, J. C., BARKER, K. J., KERBYSON, D. J., AND DAVIS, K. 2006. Quantifying the potential benefit of overlapping communication and computation in large-scale scientific applications. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. SC '06. ACM, New York, NY, USA.

SCHNEIDER, S., YEOM, J., AND NIKOLOPOULOS, D. 2009. Programming multiprocessors with explicitly managed memory hierarchies. *Computer 42,* 12, 28–34.

STMICROELECTRONICS AND CEA. 2010. Platform 2012: a many core programmable accelerator for ultra efficient embedded computing in nanometer technology.

SUDHEER, C., NAGARAJU, T., BARUAH, P., AND SRINIVASAN, A. 2009. Optimizing assignment of threads to spes on the cell be processor. *Parallel and Distributed Processing Symposium, International 0*, 1–8.

VALIANT, L. G. 1990. A bridging model for parallel computation. *Commun. ACM 33*, 103–111.

WANG, Z., BURGER, D., MCKINLEY, K. S., REINHARDT, S. K., AND WEEMS, C. C. 2003. Guided region prefetching: a cooperative hardware/software approach. *SIGARCH Comput. Archit. News 31*, 388–398.

WOLFE, M. 1989. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*. Supercomputing '89. ACM, New York, NY, USA, 655–664.

YEOM, S., ROSE, B., LINFORD, J., SANDU, A., AND NIKOLOPOULOS, D. 2009. A Comparison of Programming Models for Multiprocessors with Explicitly Managed Memory Hierarchies.

ZINNER, C. AND KUBINGER, W. 2006. Ros-dma: A dma double buffering method for embedded image processing with resource optimized slicing. *Real-Time and Embedded Technology and Applications Symposium, IEEE 0*, 361–372.

Received July 2010; revised ; accepted