Optimizing the Use of Static Buffers for DMA on a CELL Chip

Tong Chen, Zehra Sura, Kathryn O'Brien, and John K. O'Brien

IBM T.J. Watson Research Center, Yorktown Heights, NY 10598
{chentong,zsura,kmob,caomhin}@us.ibm.com

Abstract. The CELL architecture has one Power Processor Element (PPE) core, and eight Synergistic Processor Element (SPE) cores that have a distinct instruction set architecture of their own. The PPE core accesses memory via a traditional caching mechanism, but each SPE core can only access memory via a small 256K software-controlled local store. The PPE cache and SPE local stores are connected to each other and main memory via a high bandwidth bus. Software is responsible for all data transfers to and from the SPE local stores. To hide the high latency of DMA transfers, data may be prefetched into SPE local stores using loop blocking transformations and static buffers. We find that the performance of an application can vary depending on the size of the buffers used, and whether a single-, double-, or triple-buffer scheme is used. Constrained by the limited space available for data buffers in the SPE local store, we want to choose the optimal buffering scheme for a given space budget. Also, we want to be able to determine the optimal buffer size for a given scheme, such that using a larger buffer size results in negligible performance improvement. We develop a model to automatically infer these parameters for static buffering, taking into account the DMA latency and transfer rates, and the amount of computation in the application loop being targeted. We test the accuracy of our prediction model using a research prototype compiler developed on top of the IBM XL compiler infrastructure.

1   Introduction

The design of computing systems is trending towards the use of multiple processing units working collaboratively to execute a given application, with communication interfaces that enable high bandwidth data transfers between the processor and memory elements of the system. The CELL architecture[4] is one example of such a system, primarily designed to accelerate the execution of media and streaming applications. It includes two kinds of processing cores on the same chip: a general-purpose Power Processor Element (PPE) core that supports the Power instruction set architecture, and eight Synergistic Processor Element (SPE) cores that are based on a new SIMD processor design[3].
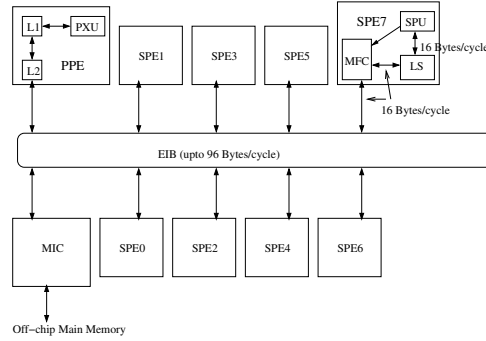
Fig. 1. The CELL Architecture

1.1 CELL Architecture

Figure 1 shows the elements of the CELL architecture and the on-chip data paths that are relevant to the discussion in this paper. The PPE includes the Power Execution Unit (PXU) that accesses main memory via its L1 and L2 caches. Each SPE includes a Synergistic Processor Unit (SPU), a Local Store (LS), and a Memory Flow Controller (MFC). Load/store instructions executed on an SPU can only load from and store to locations in the LS of that SPE. If an SPU needs to access main memory or the LS of another SPE, it must execute code that will issue a DMA command to its MFC explicitly instructing the MFC to transfer data to or from its LS. All the SPE local stores, the PPE's L2 cache, and the Memory Interface Controller (MIC) that provides access to the off-chip main memory, are inter-connected via a high bandwidth (16 Bytes/cycle on each link) Element Interconnect Bus (EIB). It is possible for a DMA transaction on the EIB that involves the LS of one SPE to be initiated by another SPU or by the PXU. However, the code transformations discussed in this paper only involve DMA transactions between main memory and an SPE LS that have been initiated by the corresponding SPU.

The hardware architecture maintains coherence between copies of data in the main memory, data in the PPE caches, and data being transferred on the EIB. However, the hardware does not keep track of copies of data residing in an LS, and software is responsible for coherence of this data. Each LS is a small 256KB memory that is completely managed in software. It contains both the code and data used in SPU execution. The latency of DMA operations between an LS and main memory is quite high, approximately in the order of 100-200 SPU cycles[11]. However, for consecutive DMA operations, it is possible to overlap the latency for the second operation with the DMA transfer of the first, as the MFC can process and queue multiple DMA requests before they are issued to the EIB.

1.2 DMA Buffering

The example code in Figure 2(a) shows a loop that iterates N times, and in each iteration it loads the ith element of array A, multiplies this value by a scalar
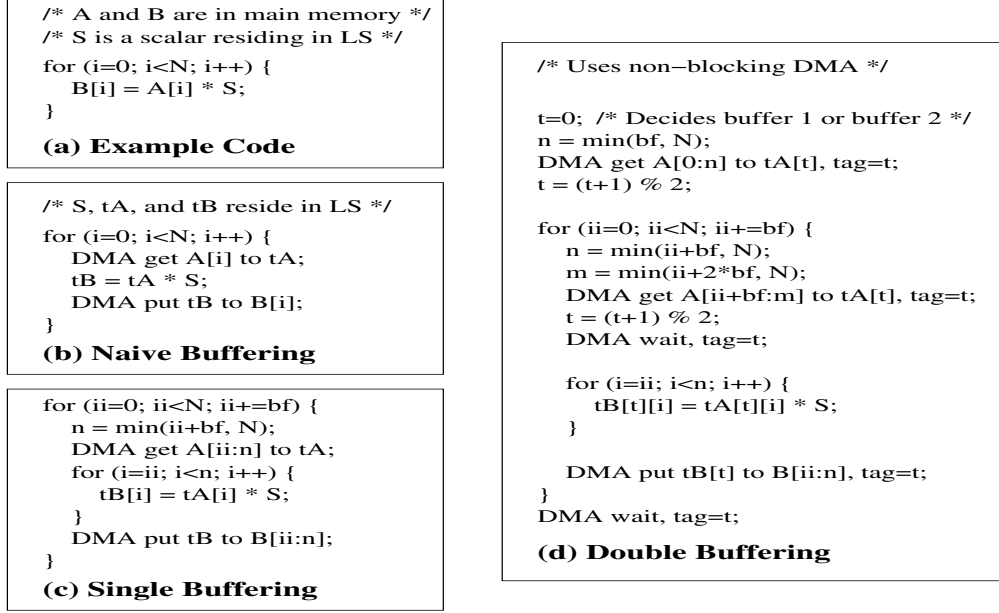
```
/* A and B are in main memory */
/* S is a scalar residing in LS */

for (i=0; i<N; i++) {
    B[i] = A[i] * S;
}
```
(a) Example Code

```
/* S, tA, and tB reside in LS */

for (i=0; i<N; i++) {
    DMA get A[i] to tA;
    tB = tA * S;
    DMA put tB to B[i];
}
```
(b) Naive Buffering

```
for (ii=0; ii<N; ii+=bf) {
    n = min(ii+bf, N);
    DMA get A[ii:n] to tA;
    for (i=ii; i<n; i++) {
        tB[i] = tA[i] * S;
    }
    DMA put tB to B[ii:n];
}
```
(c) Single Buffering

```
/* Uses non−blocking DMA */

t=0;  /* Decides buffer 1 or buffer 2 */
n = min(bf, N);
DMA get A[0:n] to tA[t], tag=t;
t = (t+1) % 2;

for (ii=0; ii<N; ii+=bf) {
    n = min(ii+bf, N);
    m = min(ii+2*bf, N);
    DMA get A[ii+bf:m] to tA[t], tag=t;
    t = (t+1) % 2;
    DMA wait, tag=t;

    for (i=ii; i<n; i++) {
        tB[t][i] = tA[t][i] * S;
    }

    DMA put tB[t] to B[ii:n], tag=t;
}
DMA wait, tag=t;
```
(d) Double Buffering

Fig. 2. Example to Illustrate DMA Buffering

S, and stores the result in the ith element of array B. If this code is targeted to execute on an SPE, the elements of A and B must be located in the LS for the SPU to be able to operate on them. However, A and B maybe allocated in main memory, perhaps because they are also being accessed by other cores, or because they are too large to fit in the limited LS space. In this case, the code is modified to include DMA operations to *get* elements of A into a temporary buffer tA in the LS, and *put* elements of B from a temporary buffer tB in the LS to main memory, as illustrated in Figure 2(b). Since the latency of DMA transfers is high, it is more efficient to transfer multiple array elements in a single DMA operation, effectively pre-fetching data for computation. Figure 2(c) shows how the example code is transformed to do this by blocking the loop using a blocking factor of *bf*, buffers tA and tB of size equivalent to *bf* array elements instead of a single array element, and one DMA get and one DMA put operation per iteration of the outer blocked loop.

The problem with the code in Figure 2(c) is that it allows no overlap between DMA transfer time and SPU computation time. Each instance of the inner blocked loop must wait for the preceding DMA get operation to complete before the inner loop can execute. This restriction can be overcome using a double-buffer scheme, as illustrated in Figure 2(d). Instead of using one *bf*-element buffer for each array data stream, the code uses two such buffers for each data stream. Before the SPU starts computing with the data fetched in one buffer, it initiates a DMA transfer using the other buffer to get data that will be used in the next iteration of the outer blocked loop. This transformation requires that there be no loop-carried flow dependencies among the iterations within one instance of the inner blocked loop. The DMA operations used are non-blocking

versions, and they are tagged with an integer identifying the LS buffer being used in the DMA transfer. The SPU can continue execution of the inner blocked loop while a DMA transfer is in progress. To wait for specific DMA operation(s) to complete, the code calls a DMA wait function with the tag corresponding to a previously issued DMA operation passed as a parameter.

The double-buffer scheme can be extended to use k buffers for each data stream to increase the amount of DMA that is overlapped with the computation in one instance of the inner blocked loop.

1.3   Problem Description

Execution time of a loop blocked for DMA buffering varies with the amount of DMA overlapped with computation. For a k-buffer scheme, the amount of DMA overlap increases both with the value of k, and with the size of the buffers used. In the SPE, all the buffers occupy space in the LS, which is only 256KB in size. This limited LS space is a prime resource, since it is being used for both code and data, and the available space limits the applicability of optimizations that increase code size or require more space to buffer data. Due to the LS size constraint, a restricted amount of space is available for DMA buffering.

The problem we address in this paper is that given a budget for the amount of space to be used for DMA buffering, determine the buffering scheme that will result in the best execution time performance. Since the total buffer size is fixed, performance of a k-buffer scheme needs to be compared with the performance of a $(k + 1)$-buffer scheme that uses individual buffers of a size smaller than the buffers used in the k-buffer scheme. Once the optimal buffering scheme is known, it may be the case that all possible DMA overlap is attained using a buffer size smaller than the maximum buffer size allowed by the total buffer space budget. Note that there is a limit to how much performance can be improved using DMA overlap before the application becomes computation-bound. Thus, we want to determine both the optimal buffering scheme and the smallest buffer size that maximize performance, when constrained by the total buffer space available.

We find that the performance of DMA buffering depends on several factors, including the set-up time for each DMA operation, the DMA transfer time, the amount of computation in the loop, the number of buffers being used, and the size of each individual buffer. We develop a model to relate each of these factors to the execution time, and use this model to predict the relative merit of using different buffering schemes and different buffer sizes. Also, we obtain performance numbers for a small set of applications running on a CELL chip using single-, double-, and triple-buffer, and various buffer sizes. We correlate the experimental data with our model. Our experiments are restricted to consider only the innermost loop in a loop nest, where this loop operates on a number of array data streams, has a large iteration count, has no loop-carried dependences, and has no conditional branches within the loop body.

In Section 2, we develop the model used to predict performance for a given buffering scheme. In Section 3, we describe how the model can be used to determine the optimal buffering scheme and buffer size in a compiler transformation

for static buffering. In Section 4, we describe the experiments we performed to validate our model against actual performance data. We discuss related work in Section 5, and conclude in Section 6.

2   Modeling Buffering Schemes

2.1   Assumptions

We assume that a loop is a candidate for DMA buffering optimization if it satisfies the following conditions:

- The loop operates on array data streams. The buffering optimizations considered are not interesting for scalar data, or data accessed through unpredictable indirect references.
- There are no loop-carried dependences between accesses to elements of the array data streams. This enables the loop to be transformed for any k-buffer scheme since the DMA get and put operations can be freely moved out of the inner blocked loop.
- There is no conditional branch statement within the loop body. This is important to be able to accurately gauge the amount of computation in the loop body, which is one of the factors that determines relative performance of different buffering schemes.
- Elements in the array data streams that are accessed in consecutive loop iterations are contiguous in memory, or not spaced too far apart. This is to ensure that when buffers are used to DMA contiguous memory locations in a single operation, the majority of the data being transferred is in fact useful.
- DMA buffers are assigned such that each buffer is only used for a single array data stream, and no buffer is used in both DMA get and put operations. This is a conservative assumption to ensure that a DMA operation on one buffer does not have to wait for a DMA operation on another buffer to complete.
- The loop iteration count is large enough that any prologue or epilogue generated when the loop is blocked has a negligible impact on performance.
- The array data streams are aligned on 128-byte boundaries, and this alignment is known at compile-time. If this is not the case, then code has to be generated to explicitly check alignment at runtime, and to issue DMA operations such that misaligned data is correctly handled. This changes the DMA set-up time, which is one of the factors used to determine the relative performance of different buffering schemes.

2.2   Latency of DMA Operations

We approximate the latency of one DMA operation with the formula $S + D * b$, where S is the set-up time for one DMA operation, D is the transfer time for one byte, and b is the number of bytes transferred by this DMA operation.

When two non-blocking DMA operations for b_1 and b_2 bytes are issued in sequence, the set-up of the second DMA operation can be overlapped with the
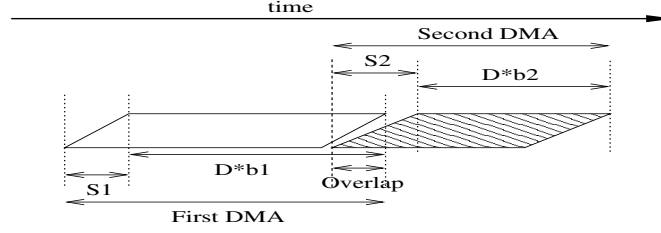
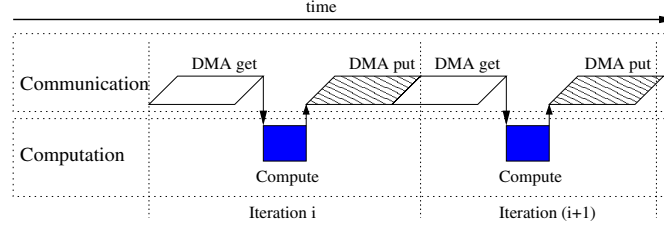**Fig. 3.** Latency of DMA Operations

Fig. 4. Execution Sequence for Single-Buffer

data transfer of the first, as illustrated in Figure 3. When the set-up of the second DMA operation (S_2) is less than or equal to the set-up of the first DMA operation (S_1), it can be completely overlapped. In this case, the combined latency of the two DMA operations will be $S_1 + D * (b_1 + b_2)$. For different values of S_1 and S_2, the amount of overlap of set-up time with transfer time will be different.

In the CELL architecture, the value of S is different for DMA get and put operations[11]. The DMA get operation has a higher value of S because it includes the main memory access time to retrieve data, whereas a DMA put can complete before data is actually written to its main memory location.

In general, a sequence of n DMA transfers will have latency $S+D*(b_1+...+b_n)$, where S is a function of $S_1, ..., S_n$.

2.3   Latency for Single-Buffer

Figure 4 illustrates the execution sequence for the code in Figure 2(c). Ignoring the prologue and epilogue, and clubbing together consecutive DMA operations, each iteration of the outer blocked loop comprises of a DMA put corresponding to the previous iteration, a DMA get to fetch data for the current iteration, and the computation of one instance of the entire inner blocked loop. Note that non-blocking DMA operations can be used, with a DMA wait inserted just before the inner blocked loop. Thus, the latency of one iteration of the outer blocked loop is the latency of both the DMA transfers plus the computation latency of the inner blocked loop. Let N be the iteration count of the original loop, and assume the loop has been blocked using a blocking factor of bf. Let D_1 be the DMA transfer time for one byte, b be the number of bytes transferred in all DMA operations corresponding to one iteration of the outer blocked loop, and S be the
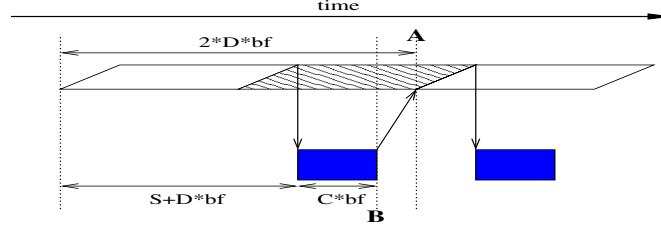
Fig. 5. Execution Sequence for DMA-bound Double-Buffer

composite set-up time for the sequence of non-blocking DMA operations corresponding to one iteration of the outer blocked loop. Also, let C be the computation time for one iteration of the inner blocked loop. The value for C is expressed as $(C_{inner} + C_{outer}/bf)$, where C_{inner} is the compute time for each iteration of the inner blocked loop, and C_{outer} is the overhead for issuing DMA requests in an iteration of the outer blocked loop. This overhead primarily includes the function call and runtime checking overhead in compiler-generated code for DMA transfer requests, and it gets amortized over bf iterations of the inner blocked loop. In practice, we expect C_{outer} to be small, and bf to be large, so that C can be approximated by C_{inner}. The total latency of the entire loop is given by: $((S + D_1 * b * bf) + C * bf) * (N/bf) = (S/bf + D_1 * b + C) * N$ For simplicity, let $D = D_1 * b$ be the DMA transfer time for all data accessed in one iteration of the inner blocked loop. Thus, latency for single-buffer is $(S/bf + D + C) * N$.

2.4   Latency for Double-Buffer

In the following discussion, the terms N, bf, C, S, and D have the same meaning as in the single-buffer case discussed earlier. For clarity, the following examples refer to DMA for one pair of buffers. However, the discussion also applies to examples using a set of double buffers, with S corresponding to the set-up delay for a composite sequence of non-blocking DMA operations issued for each set.

Case 1: DMA-Bound. Figure 5 illustrates the case when double-buffer is used and the application is DMA-bound. In this case, there is no delay between any two successive DMA operations. The sequence of DMA operations and computations alternate between using the first buffer and the second buffer. The first and second DMA operations are issued successively before any computation begins. The third DMA operation is issued only after the first computation of the inner blocked loop finishes. If there is to be no delay between the second and third DMA operations, then the time to complete the first computation (point B in the figure) must be less than or equal to the time to complete the first two DMA operations (point A in the figure). This translates to the condition:

$$(S + D * bf + C * bf) \leq (2 * D * bf), \quad \text{or } D \geq (C + S/bf)$$

When this condition holds, the execution pattern repeats throughout the loop and the application is DMA-bound. The latency for the entire loop is
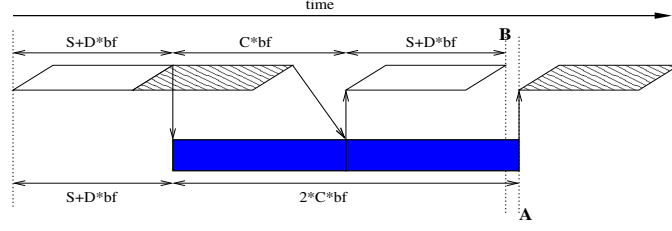
Fig. 6. Execution Sequence for Computation-bound Double-Buffer
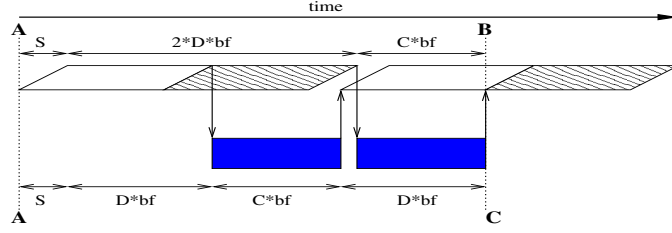

Fig. 7. Execution Sequence for Double-Buffer When $C \leq D < (C + S/bf)$

approximated by the time taken by all the consecutive DMA operations, i.e. $S + D * N$. When N is large, this can be simplified to $D * N$.

Case 2: Computation-Bound. Figure 6 illustrates the case when double-buffer is used and the application is computation-bound. In this case, there is no delay waiting for DMA to complete between any two successive computations of the inner blocked loop. The sequence of DMA operations and computations alternate between using the first buffer and the second buffer. The first and second DMA operations are issued successively before any computation begins. The third DMA operation is issued only after the first computation of the inner blocked loop finishes. If there is to be no delay between the second and third computations, then the time to complete the third DMA operation (point B in the figure) must be less than or equal to the time to complete the first two computations (point A in the figure). This translates to the condition:

$$(S + D * bf + C * bf + S + D * bf) \leq (S + D * bf + 2 * C * bf), \text{ or } D \leq (C - S/bf)$$

When this condition holds, the execution pattern repeats throughout the loop, and the application is computation-bound. The latency for the entire loop is approximated by the time taken by all the consecutive computations, i.e. $C * N$, when N is large.

Case 3: Incomplete Overlap. A loop that is neither DMA-bound nor computation-bound has incomplete overlap of DMA operations with computation. We analyze this case by splitting it into two sub-cases: when $C \leq D < (C + S/bf)$, and when $(C - S/bf) < D < C$. The total latency of the loop in both cases is the same: $(S/bf + D + C) * N/2$.
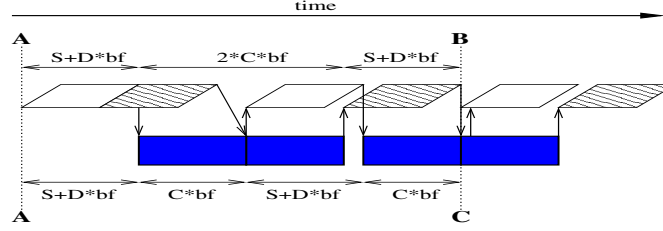
Fig. 8. Execution Sequence for Double-Buffer When $(C - S/bf) < D < C$

Case A: When $C \leq D < (C + S/bf)$**:** Figure 7 illustrates this case. Here, the set-up of the third DMA operation is not fully overlapped with the second DMA transfer. Also, there is a delay between the first and second computation, waiting for the second DMA transfer to complete. The second computation finishes at point B in the figure, and it can only start after the second DMA transfer has completed. From the beginning (point A in the figure), the latency for the second computation to finish is $S + 2 * D * bf + C * bf$. From a DMA point of view, the earliest that the fourth DMA operation can start is after the third DMA transfer reaches point C. The third DMA transfer can start only after the first computation finishes. The latency from point A in this case is $S + D * bf + C * bf + D * bf$. The two latencies from A to B and A to C are the same, which means that the fourth DMA starts at the same point that the second computation finishes, and the execution repeats the pattern illustrated in the figure. The delay between the second and third DMA operations is $S + D * bf + C * bf - 2 * D * bf = S + (C - D) * bf$. The total latency of the loop is the latency of all DMA transfers plus the extra delays due to incomplete overlap that occur after every two DMA operations. This latency is given by:

$$N * D + (S + (C - D) * bf) * N/bf/2 = (S/bf + D + C) * N/2.$$

Case B: When $(C - S/bf) < D < C$**:** Figure 8 illustrates this case. Here, the set-up of the third DMA operation is not overlapped with the second DMA transfer. Also, there is no delay between the first and second computation, but there is a delay between the second and third computation, waiting for the third DMA transfer to complete. The data for the fourth computation will be ready at point B in the figure, made available only after the first DMA transfer, the first two computations, and the fourth DMA transfer have completed. From the beginning (point A in the figure), the latency for the fourth DMA transfer to complete is $S + D * bf + 2 * C * bf + S + D * bf$. From a computation point of view, the third computation will finish at point C in the figure, and it can start only after the third DMA transfer completes. The third DMA can start only after the first computation finishes. The latency from point A in this case is $S + D * bf + C * bf + S + D * bf + C * bf$. The two latencies from A to B and A to C are the same, which means that the fourth DMA completes at the same point that the third computation finishes, and the execution repeats the pattern illustrated in the figure. The delay between the second and third computations

is $S + D * bf - C * bf = S + (D - C) * bf$. The total latency of the loop is the latency of all computations plus the extra delays due to incomplete overlap that occur after every two computations. This latency is given by:

$$N * C + (S + (D - C) * bf) * N/bf/2 = (S/bf + D + C) * N/2.$$

2.5   Latency for k-Buffer

Case 1: DMA-Bound. Analogous to the case of double-buffer, we can derive the condition for a k-buffered loop to be DMA-bound, i.e. if the first computation finishes and starts up the $(k+1)$th DMA operation in time less than or equal to the time it takes to transfer data for k DMA operations. This condition evaluates to $D \geq (C+S/bf)/(k-1)$. The initial pattern repeats throughout the loop when $D > C$. Thus, the loop is DMA-bound when $D \geq max(C, (C + S/bf)/(k - 1))$. The latency of the entire loop is approximated by $D * N$.

Case 2: Computation-Bound. Analogous to the case of double-buffer, we can derive the condition for a k-buffered loop to be computation-bound, i.e. if the DMA transfer corresponding to the $(k + 1)$th computation finishes in time less than or equal to the time it takes to complete the first k computations occuring consecutively one after another, without any intervening delays due to DMA waits. This condition evaluates to $D \leq (k - 1) * C - S/bf$. The initial pattern repeats throughout the loop execution when $D < C$. Thus, the loop is computation-bound when $D \leq min(C, (k - 1) * C - S/bf)$. The latency of the entire loop is approximated by $C * N$.

Case 3: Incomplete Overlap. Analogous to the case of double-buffer, the latency of a k-buffered loop that is neither DMA-bound nor computation-bound is $(S/bf + C + D) * N/k$. We do not discuss details of this case here.

3   Compiler Analysis

In this section, we describe how the latency formulae derived in Section 2 can be applied to determine the optimal buffering scheme and buffer size for a loop with limited amount of memory available for buffer space. We expect that the algorithm described here will be used in a compiler that automatically transforms code for DMA buffering. In the following discussion, we restrict the choice of buffering schemes to single-, double-, or triple-buffer.

Assume that the amount of memory available for buffering is specified in terms of the largest block factor (say B) that can be used when transforming the loop for a single-buffer scheme[1]. Then the maximum block factor for double-buffer is B/2, and for triple-buffer is B/3.

The performance of a loop will be optimal if it is computation-bound or DMA-bound. Therefore, a DMA-bound double-buffered loop (latency $D * N$) or

---

[1] Therefore, in the case of single-buffer, the actual size of an individual buffer will be B times the size of an array element.

```
Algorithm: bool  Choose_Double_Buffer (C, D, S, B) {

    float C: the computation per iteration;
    float D: the DMA transfer time per iteration;
    float S: the set−up latency for DMA;
    int B: buffer space constraint in terms of the maximum
          block factor used in a single−buffering scheme;


    if (D > C) {
       if (S/(D−C) <= B/2)
          return TRUE;
    } else if (D < C) {
       if (S/(C−D) <= B/2)
          return TRUE;
    }

    return FALSE;
}
```

Fig. 9. Algorithm to Choose Between Double- and Triple-Buffer Schemes

a computation-bound double-buffered loop (latency $C*N$) should be better than a single-buffered loop (latency $(S/B + D + C) * N$. When the double-buffered loop has incomplete overlap, its latency will be $(S/B/2 + D + C) * N/2$. In this case, the difference between the latencies of double-buffer and single-buffer is $(D + C) * N/2 > 0$. So double-buffer should always outperform single-buffer.

The algorithm in Figure 9 shows how to choose between double-buffer and triple-buffer. When the same performance can be achieved by different buffering schemes, the scheme with less number of buffers is preferred. When $D > C$, the double-buffer scheme becomes DMA-bound when $D \geq C + S/B/2$, which is the same as $S/(D - C) \leq B/2$. In this case, we choose the double-buffer scheme since it is DMA-bound and optimal. Similarly, when $D < C$, the double-buffer scheme becomes computation-bound for $D \leq C - S/B/2$, which is the same as $S/(C-D) \leq B/2$, and we choose the double-buffer scheme. In all other cases, we choose the triple-buffer scheme since it can provide a greater amount of overlap.

Once the loop becomes DMA-bound or computation-bound, performance will not improve with increasing buffer sizes. In such cases, memory resources can be saved by choosing the smallest buffer size that is optimal. The memory space saved can then be used by other components contending for it, e.g. more local memory can be assigned to the outer blocked loops to increase data re-use, or the size of code buffers can be increased to reduce the frequency of swapping code partitions to and from the SPE LS. Based on the conditions derived in Section 2 for the execution to be DMA-bound or computation-bound, the block factor for double-buffer need not be larger than $S/abs(D - C)$. The block factor for DMA-bound triple-buffer need not be larger than $S/(2 * D - C)$, and for computation-bound triple-buffer need not be larger than $S/(2 * C - D)$.

4   Experiments

To verify how precise our analysis models are, we performed experiments on a CELL blade. The clock rate for the PPU and SPU in the blade is 3.2GHz. All our experiments were run using a single SPE. We use the IBM XL CELL
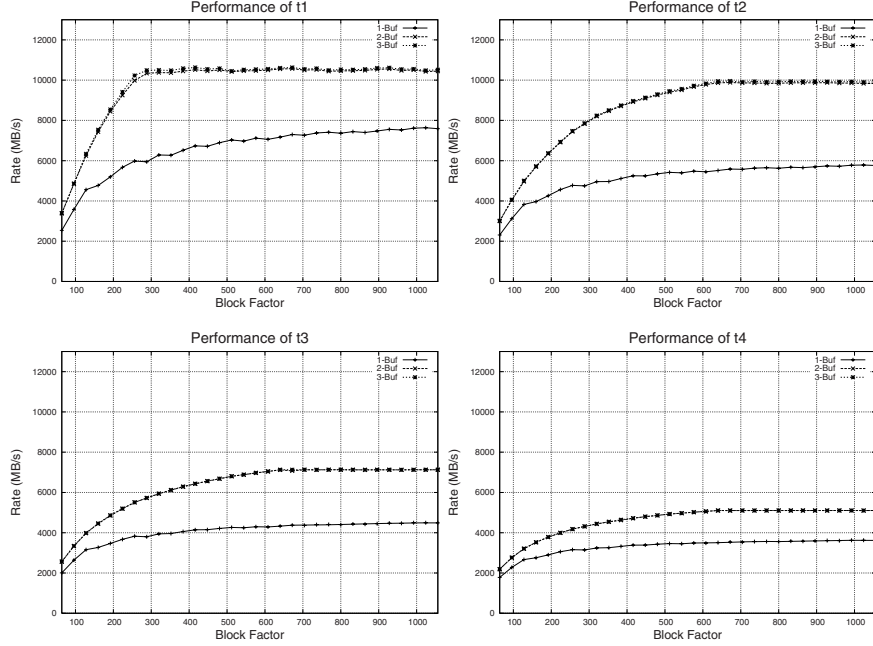
Fig. 10. Performance of Applications with Varying Buffer Schemes and Buffer Sizes

single-source compiler [2] to automatically apply single-, double-, and triple-buffer schemes to a set of test applications. This compiler uses OpenMP directives to decide what parts of the code will execute on the SPE(s), and it automatically handles DMA transfers for all data in an SPE LS.

We adapted a simple streaming benchmark to obtain a set of test kernels with varying amount of computation in the loop. Currently, we report results for 4 test cases: t1, t2, t3, and t4. The amount of computation in the kernel loop increases from t1 to t4. Each test case has only one OpenMP parallel loop that has a very large iteration count (15 million), and is invoked multiple times. The four test cases use the same data types, and have the same data access pattern: two reads and one write. Performance is measured in terms of throughput (MB/s).

To apply our formula, we have to determine the values of D, S, and C. For D and S, we ran code that only performs a large number of DMA operations, and inferred the values of D and S using linear regression analysis on the performance results of this code. To determine C for each test case, we used a profiling tool called Paraver [5] to instrument code and measure the amount of computation time for each iteration of the outer blocked loop. This was done for each test case using a single-buffer scheme and a large block factor to amortize the overhead. The constant values that were determined are S=130ns, D_1=0.0877ns per byte, C_{outer}=300ns (for all test cases), C_{inner}=0.51ns(t1), 1.73ns(t2), 2.83ns(t3), and 3.93ns(t4). All of these benchmarks need to transfer 24 bytes of data per iteration of the inner blocked loop, so the D per iteration is 2.112ns.
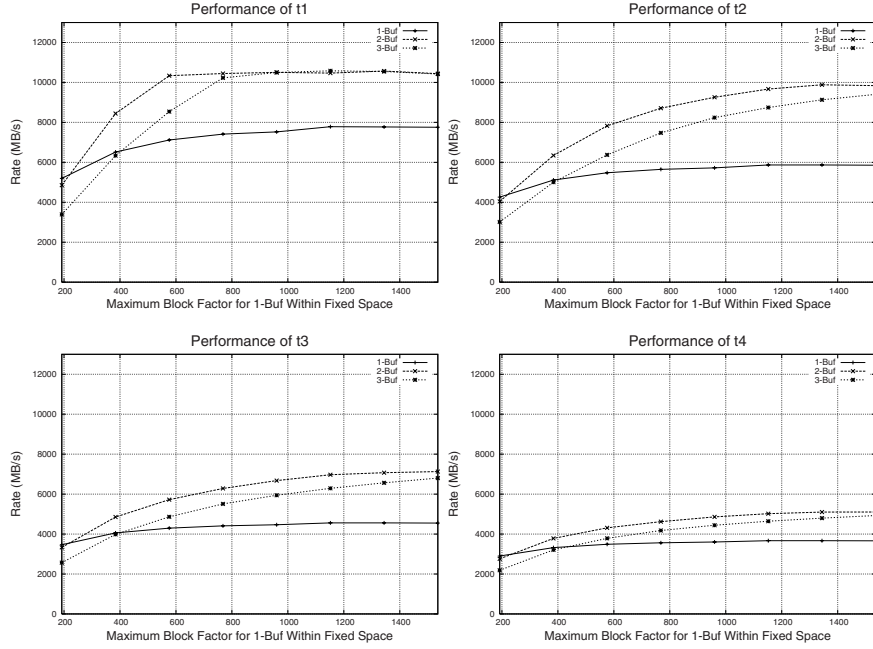
Fig. 11. Comparison of Buffering Schemes Using a Fixed Space Constraint

The performance of single-buffer (1b), double-buffer (2b), and triple-buffer (3b) for the 4 test cases is shown in Figure 10. The x-axis is the block factor, and the y-axis is the performance. Here, different buffer schemes are compared when they use the same block factor. We notice that 2b and 3b have a similar performance curve, while the performance of 1b is much lower. We also observe that the performance of 2b and 3b becomes flat when the block factor becomes large enough. t1 and t2 reach almost the same peak performance. Since their values of C_{inner} are smaller than the value of D per iteration, they should become DMA-bound. On other hand, the values of C_{inner} in t3 and t4 are larger than the value of D, and they become computation-bound. Their peak performance is determined by the amount of computation, C_{inner}. The overall performance trend conforms to our model. In Figure 11, we compare the performance of different buffering schemes with a fixed amount of space available for buffering. The x-axis is the block factor of the 1b scheme. Within the same available space, the corresponding block factor for 2b should be half, and one-third for 3b. For large block factors, the performance trend conforms to our analysis. However, for small block factors, in all 4 test cases, 1b outperforms 2b and 3b. This is contradictory to the analysis presented earlier. In the analysis in Section 2, we assumed C could be approximated by C_{inner}. However, when the block factor is small and the overhead of issuing a DMA request is high, the C_{outer}/bf component of C dominates. In this case, the difference between the latencies of the whole loop for 1b and 2b evaluates to $(D + C_{inner}) * N/2 - C_{outer} * N/B$ (as opposed to $(D + C) * N/2$, derived in Section 3), which may or may not be positive. With fixed space, 1b can use a larger block factor than 2b
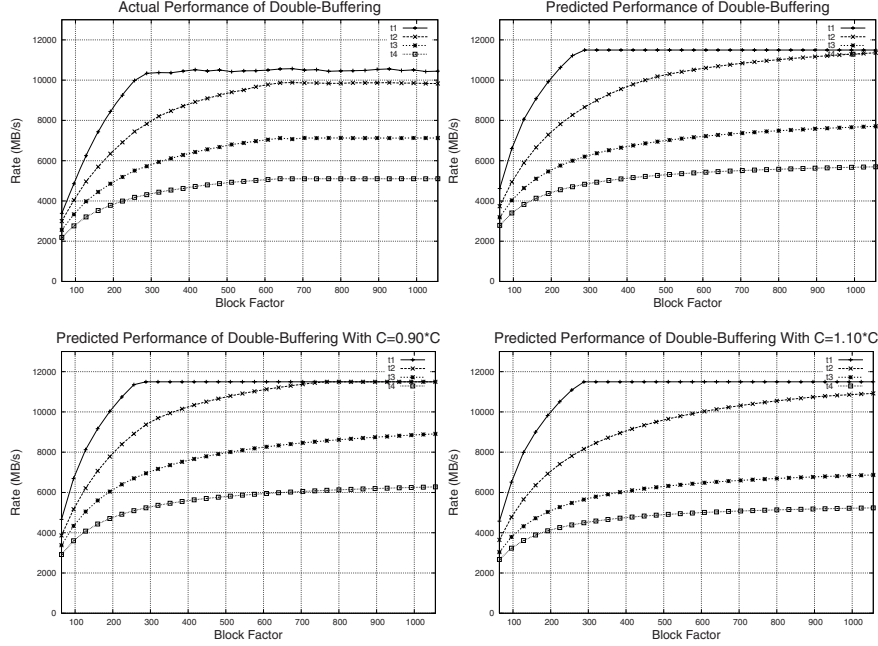
Fig. 12. Actual and Predicted Performance for Double-Buffer

and 3b, thus issuing fewer DMA requests and paying the C_{outer} overhead fewer times. This results in higher performance for 1b, as observed. We performed experiments to manually code the 1b, 2b, and 3b versions of our test cases, and determined that the C_{outer} overhead can be made small enough such that the C_{outer}/bf component of C is always negligible. It is possible to do this in the compiler, but we do not have this implemented yet.

In Section 3, we discuss how to choose the block factor so as to avoid wasting memory resources. Figure 12 shows the actual and the predicted performance of double-buffer when using different block sizes. Overall, the prediction is quite precise in terms of the shape of the performance curve. The relative performance of each test case is correctly predicted. However, the absolute performance of all test cases is over-estimated by about 15%. The values of S and D need to be determined just once, and precise values for these constants can be obtained empirically on a given machine. However, the value of C is application-specific, and may need to be estimated by the compiler. The in-order, statically scheduled SPU architecture enables a high degree of accuracy in compiler estimation. However, compiler optimizations that occur after the DMA buffering transformation can significantly affect the code generated. Therefore, it may be necessary to use an extra compiler pass to feedback the estimated value of C to the DMA buffering optimization. To investigate the sensitivity of our prediction to the value of C, we also plot in Figure 12 the performance predicted using a value of C that is 10% less and 10% more than the value determined by profiling. We observe that the variation in predicted performance is 6% on average, and 15% maximum.

5   Related Work

In [11], the performance of DMA on a CELL chip is studied, and the latencies of DMA operations for different workload characteristics are determined. The use of loop stripmining and unrolling to optimize network communications is studied in [7]. This work focuses on determining the minimum size for stripmining to avoid performance degradation. In [12], the dependencies and communication time between tasks in a parallel execution are modeled with the aim of identifying possible computation-communication overlap. In our work, we optimize both performance and local memory usage.

In [1], remote accesses in UPC programs are optimized in the compiler by coalescing small accesses into one large access, and by using one-way communication supported in the underlying network layer. The work in [8] also discusses optimization of remote accesses in UPC programs, using runtime synchronization and scheduling. In [10], a stream programming model is used to inform the compiler of the high-level program structure. The compiler then uses this information to optimize scheduling and buffering for execution on the Imagine stream processor. A compiler-based loop optimization targeted to improve the communication-computation overlap is described in [9]. The technique of communication pipelining for distributed memory systems is discussed in [6][13].

6   Conclusion

We have developed a model to predict the performance of different buffering schemes and the optimal buffer size for DMA buffering in the CELL SPE local stores. We compare the predicted and actual performance for a set of kernels with varying amounts of computation in the loop, and observe a high degree of correlation between the two. In this work, we have considered the use of a single SPE, but we plan to extend our model to multiple SPEs in future work.

References

1. W. Chen, C. Iancu, and K. Yelick. Communication optimizations for fine-grained UPC applications. *Parallel Architecture and Compilation Techniques*, 2005.
2. A.E. Eichenberger et al. Optimizing compiler for the CELL processor. *Parallel Architecture and Compilation Techniques*, September 2005.
3. B. Flachs et al. A streaming processing unit for a CELL processor. *IEEE International Solid-State Circuits Conference (ISSCC)*, February 2005.
4. D. Pham et al. The design and implementation of a first-generation CELL processor. *IEEE International Solid-State Circuits Conference*, February 2005.
5. European Center for Parallelism of Barcelona (CEPBA). Paraver: Parallel program visualization and analysis tool reference manual. November 2000. URL: http://www.cepba.upc.es/paraver.
6. S. Hiranandani, K. Kennedy, and C-W. Tseng. Evaluation of compiler optimizations for Fortran D on MIMD distributed memory machines. *International Conference on Supercomputing*, 1992.

7. C. Iancu, P. Husbands, and W. Chen. Message strip mining heuristics for high speed networks. *6th International Meeting of VECPAR*, 2004.
8. C. Iancu, P. Husbands, and P. Hargrove. HUNTing the overlap. *Parallel Architecture and Compilation Techniques*, September 2005.
9. K. Ishizaki, H. Komatsu, and T. Nakatani. A loop transformation algorithm for communication overlapping. *International Journal of Parallel Programming*, 2000.
10. U.J. Kapasi, P. Mattson, W.J. Dally, J.D. Owens, and B. Towles. Stream scheduling. *Proceedings of the 3rd Workshop on Media and Streaming Processors*, 2001.
11. M. Kistler, M. Perrone, and F. Petrini. CELL multiprocessor communication network: Built for speed. *IEEE Micro*, 26(3), May/June 2006.
12. J.S. Leu, D.P. Agrawal, and J. Mauney. Modeling of parallel software for efficient computation communication overlap. *Proceedings of the 1987 Fall Joint Computer Conference on Exploring Technology: Today and Tomorrow*, 1987.
13. D.J. Palermo, E. Su, J.A. Chandy, and P. Banerjee. Communication optimizations used in the PARADIGM compiler for distributed memory multicomputers. *International Conference on Parallel Processing*, August 1994.