Applied Mathematics & Information Sciences
An International Journal

http://dx.doi.org/10.12785/amis/090137

BCDR: Data Reuse Framework for Multi-core Systems with Local Memories

Jue Wang[*] *and YanGang Wang*

Supercomputing Center, Computer Network Information Center, Chinese Academy of Science, Beijing, China

Received: 6 Apr. 2014, Revised: 7 Jul. 2014, Accepted: 8 Jul. 2014
Published online: 1 Jan. 2015

Abstract: Emerging heterogeneous multi-core systems, such as the IBM Cell BE, are deployed with multiple hardware accelerators to enhance the performance of the systems. In these systems, each accelerator includes its own local memory where software controlled DMA transfers are provided to utilize the memory bandwidth. Two important software controlled management methods (direct buffering and software controlled cache) are applied in regular and irregular references, respectively. The run-time coherence maintenance is performed when the same global memory location is referenced by both software controlled cache and buffer.This paper proposes a BCDR framework to exploit data reuse for buffers and software cache. The framework includes buffer2buffer data reuse optimizations, buffer2cache/cache2buffer data reuse optimizations and buffered array identification. For buffer2buffer data reuse optimizations, the Retaining Buffered Data technique and pipelining optimization are given to optimize critical region after a basic data reuse optimization. To make use of the opportunity induced by buffer2buffer optimizations, the buffer2cache/cache2buffer data reuse optimizations are presented to improve the performance of applications with irregular accesses. Furthermore, a buffered data identification algorithm is presented to increase the precise of global data-flow analysis for the coherence maintenance between SCC and buffers. The experimental results show that our optimizations expose many opportunities for both buffer and cache. The transferred data amount between the local store and global memory is reduced by 16.35% on average for all cases. Our optimizations further reduce 19.7% of the average execution time. In addition, the run-time coherence maintenance overhead is reduced significantly.

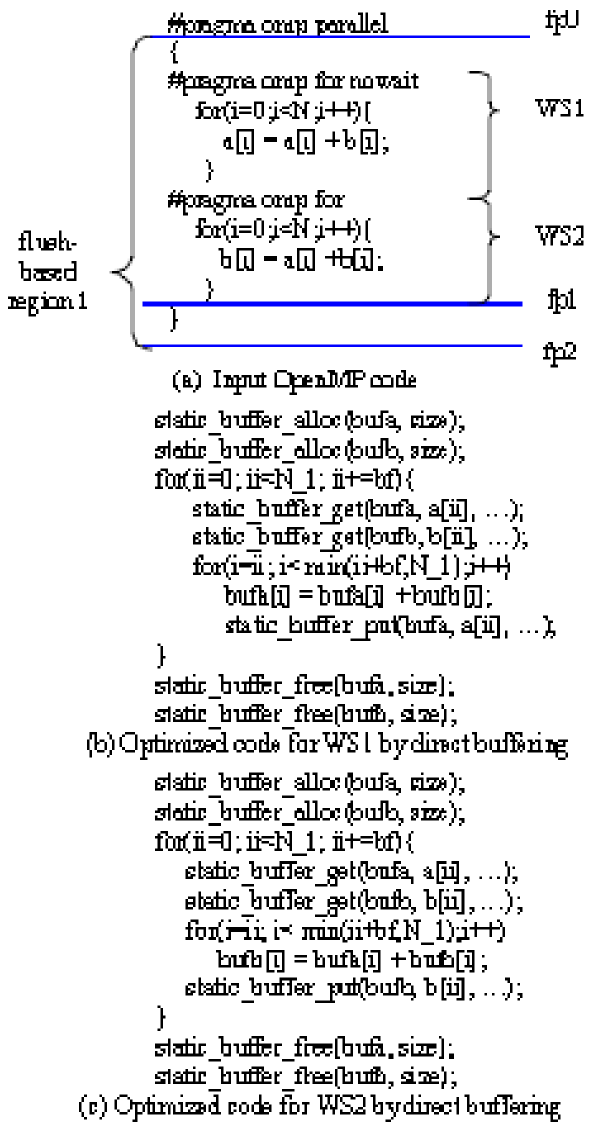
Keywords: Compiling Technique, Software Controlled Cache and Buffer, Run-time Techniques, Multi-core Systems

1 Introduction

Multi-core systems with multiple hardware accelerators, such as the Cell Broadband Engine (Cell BE) architecture, are promising platform for parallel computing. The Cell BE[1,2] is comprised of 8 SPEs and explicitly managed memory hierarchies. Each SPE has its own 256KB fast local memory called a ”local store”. There is not hardware coherence between the local stores or between the local stores and global system memory. This memory design requires programmers not only to judiciously insert DMA operations to use fast local memory effectively, but also to explicitly orchestrate both data and codes to fit in the SPE’s local store. How to effectively manage data of the memory hierarchy has been growing research and industry interest.

The IBM Single Source Compiler [3] for Cell BE takes an OpenMP [4] program as input due to OpenMP’s ease of use and widely acceptance in the industry and research area. The compiler automatically generates binaries for the PPE and SPE on a Cell BE chip. Two important methods proposed in the IBM Single Source Compiler manage the communication between these local stores and the global system memory. These methods are direct buffering and Software Controlled Cache (SCC). The direct buffering technique is used in regular applications whose access patterns are predictable by precise compile-time analysis. For irregular applications in which access patterns are not analysable at compile-time phase, the SCC, which simulates the operations of hardware cache, is an effective method to transfer data between the local store and global memory. The SCC is also able to be used in regular accesses, but the overhead of cache look-up and individual data transfer for cache misses is unavoidable. Especially, SCC is inefficient for regular applications whose access patterns have poor locality. Thus the optimal method suggested by Ref.[3,5] is to combine buffer and SCC to support regular accesses and irregular accesses, respectively. When both

[*] Corresponding author e-mail: wangjue@sccas.cn



```
#pragma omp parallel
{
#pragma omp for nowait
   for(i=0;i<N;i++){
      a[i] = a[i] + b[i];
   }
#pragma omp for
   for(i=0;i<N;i++){
      b[i] = a[i] +b[i];
   }
}
```

(a) Input OpenMP code

```
static_buffer_alloc(bufa, size);
static_buffer_alloc(bufb, size);
for(ii=0; ii<N_1; ii+=bf){
     static_buffer_get(bufa, a[ii], ...);
     static_buffer_get(bufb,b[ii], ...);
     for(i=ii; i< min(ii+bf,N_1);i++)
         bufa[i] = bufa[i] +bufb[i];
         static_buffer_put(bufa, a[ii], ...);
}
static_buffer_free(bufa,size);
static_buffer_free(bufb, size);
```

(b) Optimized code for WS1 by direct buffering

```
static_buffer_alloc(bufa, size);
static_buffer_alloc(bufb, size);
for(ii=0; ii<N_1; ii+=bf){
    static_buffer_get(bufa, a[ii], ...);
    static_buffer_get(bufb, b[ii], ...);
    for(i=ii; i< min(ii+bf,N_1);i++)
        bufb[i] = bufa[i] + bufb[i];
    static_buffer_put(bufb, b[ii], ...);
}
static_buffer_free(bufa,size);
static_buffer_free(bufb, size);
```

(c) Optimized code for WS2 by direct buffering

Fig. 1: An example to optimize the code by direct buffering

SCC and buffer reference the same global memory location, the compile-time/run-time coherence maintenance [5] is performed to check and update data of SCC at the entry or exit of the direct buffer.

When the data of an innermost loop can not fit in the available space in the local store, the buffer optimization techniques [6] block the loop and insert corresponding data transfer operations for regular references. Furthermore, T. Liu et al. [7] focuses on an independent loop to give reference grouping, buffer dependence checking and buffer compression to reduce the data transferred between the local stores and global system memory. the DMATiler [8] applies compressed data transfers and DMA commands to achieve the best DMA performance for a given loop nest. The above optimizations can be complementary to our work. For a parallel region involving multiple loops or work-sharing constructs, the key insight of this paper is how to optimize inter-loop (or inter-work-sharing constructs) to reuse data between the buffers or the buffers and SCC. The WS1 and WS2 shown in Fig. 1 (a) represent two work-sharing constructs in OpenMP. After the loop normalization of the WS1 and WS2, iterations are divided into chunks, and the chunks are assigned to the threads according to the default scheduling (static scheduling) of OpenMP. Fig. 1(b) and Fig. 1(c) present the optimized intermediate codes of W1 and W2 by using the single buffer technique, respectively. The loops are blocked using a block factor of bf due to the limit of available space in the local store. When the data is first brought into buffer, the buffer data become live. When the data is freed or written into global memory, the buffer data is killed. Merging the outer loop can eliminate the DMA get operation of array a and reuse the data residing in the buffer bufa.

In this paper, we propose a Buffer-Cache Data Reuse (BCDR) framework, which includes buffer2buffer data reuse optimizations, and buffer2cache/cache2buffer data reuse optimizations, and buffered array identification. The overview of the BCDR framework is given in Fig. 2. A basic buffer2buffer data reuse optimization, which is based on OpenMP relaxed consistency model, is first given to restructure loops to reduce buffer data. After the basic buffer2buffer optimization, a Retaining Buffered Data (RBD) technique is used to improve the performance of critical and ordered regions. And a pipelining model utilizes the communication between the local stores to optimize the critical regions in OpenMP program instead of the communication between the local store and global system memory. Additionally, the buffer2buffer data reuse optimizations may extend the live range of buffers, which provides an opportunity to reuse data between buffers and SCC. To utilize this opportunity, the buffer2cache (or cache2buffer) optimizations are proposed to improve the performance of applications by checking buffers (or SCC). The above optimizations may make the run-time coherence check operations relatively centralized. The buffered array identification is implemented to improve the precision of data-flow analysis [5] which is used to trace the access sequence of buffer and SCC. Our experiments show that the raise of the precision can reduce the number of coherence checks effectively.

The OpenMP version of NAS benchmark is used to evaluate our optimizations. The transferred data amount is reduced by 16.35%, on average, for all cases. Furthermore, our optimizations reduce 19.7% of the average execution time.

Our compiler-time analysis procedure of BCDR framework has been implemented in Cetus compiler

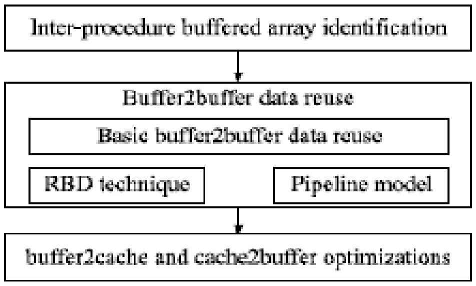

Inter-procedure buffered array identification
Buffer2buffer data reuse
Basic buffer2buffer data reuse
RBD technique
Pipeline model
buffer2cache and cache2buffer optimizations

Fig. 2: Buffer-cache data reuse framework

framework [9] which is a source-to-source translator. The major contributions of this paper are the following:

(1) We propose a basic buffer2buffer data reuse optimization technique to effectively reuse data residing in buffers. For critical regions in OpenMP, the RBD techniques and pipelining model are proposed to reduce the communication amount between the local store and global system memory.

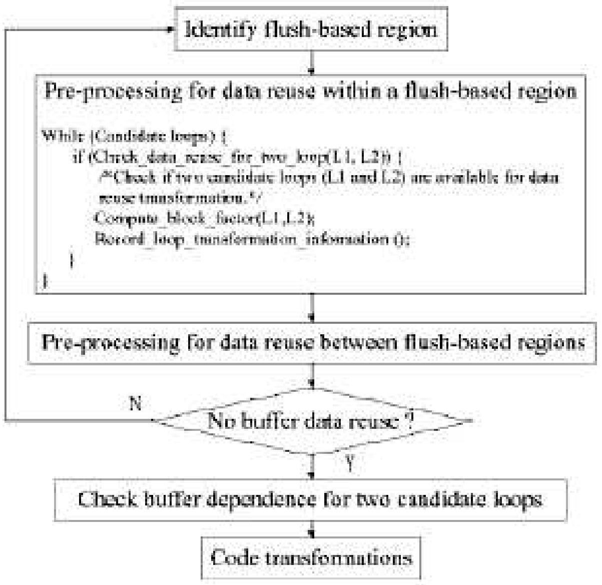
(2) We present cache2buffer and buffer2cache data reuse optimization techniques to reuse data between buffer and SCC. Especially for applications with irregular array accesses, the buffer2cache optimization not only reduces the communication amount but also increases the hit ratio to fetch data in the local store.

(3) We give a buffered array identification algorithm to enhance the precise of data-flow analysis for buffer. Using this algorithm, the coherence maintenance overhead can be reduced significantly.

(4) We give detail experiments and communication analysis for the performance of our optimizations. The rest of the paper is organized as follows: Section 2 gives our buffer2buffer data reuse optimization. Section 3 presents buffer2cache and cache2buffer data reuse optimizations to improve the performance of irregular applications. Section 4 describes the algorithm to identify buffered array for precise data-flow analysis. Section 5 gives the experimental results and analysis to demonstrate the effectiveness of our optimizations. Section 6 reviews related work. Finally, Section 7 concludes the paper.

2 Buffer2buffer Data Reuse Optimization

OpenMP provides a relaxed consistency shared-memory model. Each thread's temporary view of memory is not required to be consistent with memory at all times. The OpenMP flush operation enforces the consistency between the temporary view and memory. It is used in flush directive and implicit operations, such as barrier directive, exit from work-sharing region, entry to and exit from parallel, etc. In an OpenMP program, the global flush splits the parallel region into two parts. We call the sub-region between two global flushes "flush-based"


Identify flush-based region
Pre-processing for data reuse within a flush-based region
Pre-processing for data reuse between flush-based regions
N
No buffer data reuse ?
Y
Check buffer dependence for two candidate loops
Code transformations

```
While (Candidate loops) {
    if (Check_data_reuse_for_two_loop(L1, L2)) {
        /*Check if two candidate loops (L1 and L2) are available for data
        reuse transformation.*/
       Compute_block_factor(L1,L2);
       Record_loop_transformation_information ();
    }
}
```

Fig. 3: Overview of buffer2buffer data reuse optimization

region. Back to the example shown in Fig. 1(a), a parallel region surround two work-sharing constructs WS1 and WS2. fp0 and fp2 present global flush operations implied at the entry and exit of parallel region. fp1 is a flush point implied at the exit of WS2. These flush operations split the parallel region of OpenMP program into two flush-based regions. In our cases, if a global flush is encountered, all data will be killed and enforced to write to memory from SCC or buffers in the local stores. OpenMP's memory model provides buffer reuse opportunities between multiple loops or work-sharing constructs.

In this section, we first give a basic buffer2buffer data reuse optimization. For some special cases, we then present a RBD technique and give a pipelining model to utilize the bandwidth between the local stores.

2.1 Basic Buffer2buffer Data Reuse Optimization

Fig. 3 shows the overview of the basic buffer2buffer data reuse optimization. We first identify flush-based region in the control-flow graph. We then perform the pre-processing for data reuse within each flush-based region and between the flush-based regions. If there is not data reuse opportunity, the true code transformation phase will be issued.

For the pre-processing algorithm to reuse data within a single flush-based region, the conditions to check if two candidate loops are available for data reuse transformation are similar to the conditions of loop fusions [10]. Their main differences include the following items:

(1) The basic buffer2buffer data reuse optimization only focus on the loops involving the regular references.

(2) When block factor is equal to 1, the data size of transformed codes is required to fit in the available local store space.

(3) The mapping of iterations on threads is identical for both candidate loops.

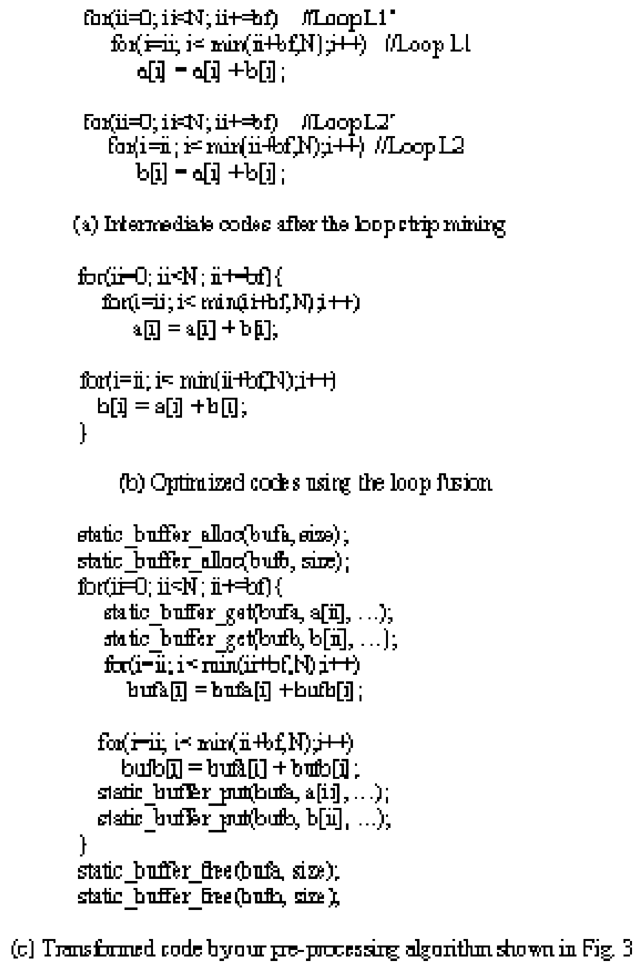
(4) The pre-processing algorithm is unavailable for the codes surrounded by the synchronization constructs (such as the critical construct and ordered construct, etc.) and lock routines. A RBD technique and a pipelining model (the detail are given in the sub-section 2.2 and 2.3) are proposed to improve the performance for this kind of codes.

If two candidate loops meet the conditions of data reuse transformation, corresponding block factor will be re-calculated according to the available space in local store, the access pattern of each references and the buffering scheme(such as single- double- or triple-buffers)[6]. The following step is to record the code transformation information including the locations (to insert the buffer allocation operations, DMA get/put operations, etc) and the data structure to store the transformation from global memory address to buffer address. The advanced loop fusion judgment and global loop fusion [10] can be applied to improve our pre-processing algorithm and code transformation for the buffer2buffer data reuse.

The pre-processing algorithm for inter-region is similar to the algorithm for data reuse within a flush-based region. The first step of the algorithm is to check the data reuse for two adjacent candidate loops in different flush-based regions. In the code transformation phase, the data write-back operation for buffer (if needed) is required to be inserted before the global flush.

Compiler-time/run-time buffer dependence check is required to ensure the correctness of program execution. This dependence check proposed by Liu et al. [7] is applied in the single loop. It is used to determine whether two references have different access patterns but access the same (or overlapped) memory space. If the buffer dependence for two candidate references can not be resolved at compile-time phase, run-time dependence check will be issued. In our work, this method is extended to determine whether two references in different loops access the same (or overlapped) memory space. If so, overlapped buffers are merged into one.

In the code transformation phase, the traditional loop fusion transformation is unavailable due to the OpenMP semantic and loop strip-mining. For the program presented in Fig. 1(a), the intermediate codes after the loop strip mining is presented in Fig. 4(a). The innermost loops L1 and L2 shown in Fig.4(a) can not be merged, but their outer loops L1’ and L2’ may be merged using the loop fusion techniques. Our code transformation phase generates the merged code shown in Fig. 4(b). The direct



```
for(ii=0; ii<N; ii+=bf)    //Loop L1'
    for(i=ii; i< min(ii+bf,N);i++)   //Loop L1
        a[i] = a[i] +b[i];

for(ii=0; ii<N; ii+=bf)    //Loop L2'
    for(i=ii; i< min(ii+bf,N);i++)  //Loop L2
        b[i] = a[i] +b[i];
```

(a) Intermediate codes after the loop strip mining

```
for(ii=0; ii<N; ii+=bf){
   for(i=ii; i< min(ii+bf,N);i++)
       a[i] = a[i] + b[i];

for(i=ii; i< min(ii+bf,N);i++)
  b[i] = a[i] +b[i];
}
```

(b) Optimized codes using the loop fusion

```
static_buffer_alloc(bufa, size);
static_buffer_alloc(bufb, size);
for(ii=0; ii<N; ii+=bf){
    static_buffer_get(bufa, a[ii], ...);
    static_buffer_get(bufb, b[ii], ...);
    for(i=ii; i< min(ii+bf,N);i++)
        bufa[i] = bufa[i] +bufb[i];

   for(i=ii; i< min(ii+bf,N);i++)
       bufb[i] = bufa[i] + bufb[i];
   static_buffer_put(bufa, a[ii], ...);
   static_buffer_put(bufb, b[ii], ...);
}
static_buffer_free(bufa, size);
static_buffer_free(bufb, size);
```

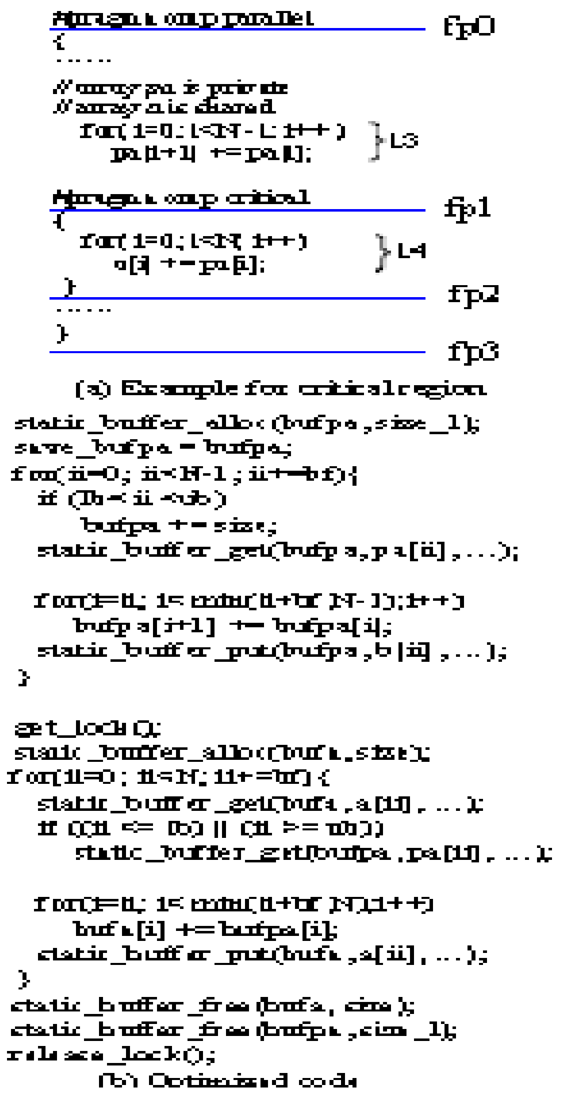
(c) Transformed code by our pre-processing algorithm shown in Fig. 3

Fig. 4: Example of code transformation using our pre-processing algorithm

buffer size is computed according to the block factor and references. The final step of code transformation is to insert the buffer allocate/free and data transfer operations, and to translate the accesses to global memory into local buffers. The optimized codes using direct buffers are presented in Fig. 4(c). To merge the DMA operations effectively, Regular Section Descriptors (RSD) [12] are used to summarize array references.

2.2 RBD Technique

The above analysis and transformation can not be applied in the loop within the critical and ordered region, since the basic buffer2buffer data reuse optimization may result in the expansion of the synchronization region. For example, the loop L3 and L4 in Fig. 5(a) can not be merged. Using the conservative method [5], the data of



```
#pragma omp parallel                fp0
{
......
//array pa is private
//array a is shared
   for( i=0; i<N-1; i++ )   } L3
     pa[i+1] += pa[i];

#pragma omp critical                fp1
{
   for( i=0; i<N; i++)      } L4
     a[i] += pa[i];
 }                                  fp2
......
}                                   fp3
        (a) Example for critical region
static_buffer_alloc(bufpa,size_1);
save_bufpa = bufpa;
for(ii=0; ii<N-1; ii+=bf){
   if (lb<ii<ub)
       bufpa += size;
   static_buffer_get(bufpa,pa[ii],...);

   for(i=ii; i<min(ii+bf,N-1);i++)
      bufpa[i+1] += bufpa[i];
   static_buffer_put(bufpa,b[ii],...);
 }

 get_lock();
 static_buffer_alloc(bufa,size);
for(ii=0; ii<N; ii+=bf){
   static_buffer_get(bufa,a[ii],...);
   if ((ii <= lb) || (ii >= ub))
      static_buffer_get(bufpa,pa[ii],...);

   for(i=ii; i<min(ii+bf,N);i++)
      bufa[i] += bufpa[i];
   static_buffer_put(bufa,a[ii],...);
 }
static_buffer_free(bufa, size);
static_buffer_free(bufpa,size_1);
release_lock();
          (b) Optimized code
```

Fig. 5: Example of data reuse optimization for critical region

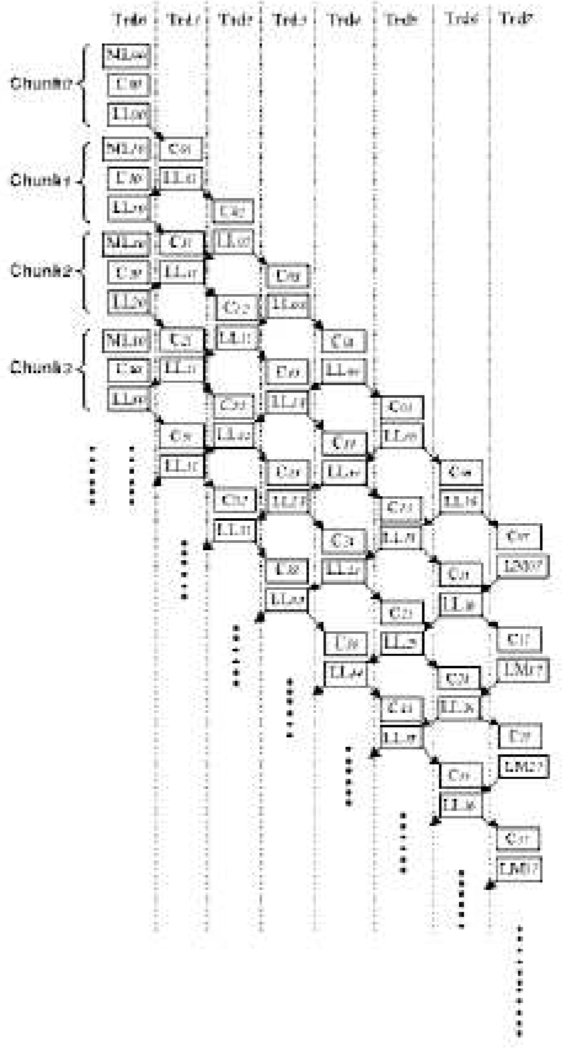


Fig. 6: Pipelining model for critical region

array pa in the local store is not killed until the critical region is encountered. The implicit flush fp1 emits the data of pa to the global memory, which results in the loss of data reuse between the loop L3 and L4. To reuse the data of pa as many as possible, the optimized intermediate codes are given in Fig. 5(b). The variable lb and ub, which are computed at compile-time phase, are used to indicate the range of retained data in bufpa. These variables are determined by the available space in the local memory.

2.3 Pipeline Model for Critical Regions

From OpenMP specification [4], a thread waits at the beginning of a critical region until no thread is executing a critical region. As a result, each thread executes the codes in the critical region. If there is a reference for a shared array in the critical region, the buffered data for the shared array can be reused for the current thread team. Thomas Chen [13] indicated that the bandwidth of SPE-to-Memory DMA transfers is lower than that of SPE-to-SPE DMA transfers. The integrated memory controller (MIC) provides a peak bandwidth of 25.6GB/s to the global memory, while the Element Interconnect Bus (EIB) provides a peak bandwidth of 204.8GB/s for intra-chip data transfers among the local stores.

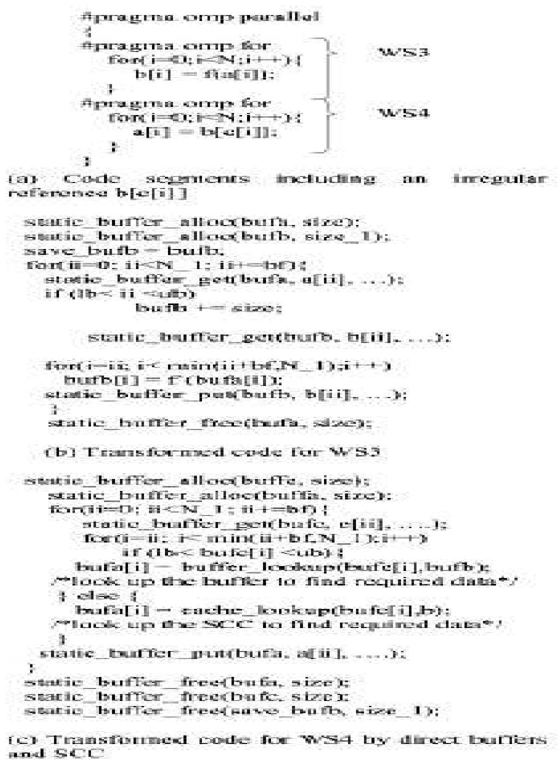
In this sub-section, we give a pipelining model which not only hides the computation in the critical region but also utilizes the DMA operations between the local stores on chip instead of the DMA operations between the local stores and global memory. For example, in Fig. 5, the critical region has a loop L4 where a represents a shared array and pa represents a private array. The loop strip-mining is applied in the loop L4 to reduce the memory footprint. Iterations of loop L4 are divided into chunks to satisfy the available space in local store. The execution diagram of transformed codes is shown in Fig. 6. We use two signal registers to ensure the correctness of pipelined program. A link arrow represents the sending direction of signal. Note that the thread Trd0 (or Trd7) reads (or writes) the shared arrays from (or to) the global memory only once.

3 Buffer2cache and Cache2buffer Optimizations

The buffer2buffer data reuse optimization not only makes DMA operations relatively centralized but also gives an opportunity for the case where data of buffers (or SCC) can be reused by SCC (or buffers). After the buffer2buffer data reuse optimizations are performed, the buffer2cache and cache2buffer data reuse optimizations are used to enhance the data reuse between the buffer and SCC.

The run-time SCC technique [3] is used in irregular refererences due to the difficulty of analysis of irregular data access patterns at compile-time phase. The work-sharing construct WS3 and corresponding transformed code are shown in Fig. 7 (a) and Fig. 7 (b). For the further data reuse, the data of array b is retained as many as possible. In Fig. 7 (c), the transformed code of WS4 adopts the buffers and SCC because the construct WS4 includes both regular references (a[i] and c[i]) and irregular reference (b[c[i]]). Due to the implicit flush operation between WS3 and WS4, the conservative method [3,5] can not ensure that the SCC used by WS4 can reuse the array b data buffered in WS3. The run-time function call buffer_lookup is developed to find b[c[i]] in the buffer bufb in local store. The buffer structure, which is designed according to the flat buffer [7], consists of buffer directory and buffer storage. The buffer directory is used in run-time lookup of buffer or cache. The 128-bit SIMD instructions are used to improve the performance of lookup operation.

T. Chen [5] indicated that the buffer-check-cache operation is only used in three cases to ensure the correctness of program execution. In our cases, the buffer-check-cache operation can be done as the cache2buffer optimization to reuse the data in cache. The data in the SCC are retained to be reused by the following buffers. In our compiler implementation, a simple use-use/use-def analysis is developed for adjacent loops to find the data reuse opportunities between the buffers and SCC. To improve the precise of the analysis, the buffered array identification algorithm is given in the following section.


```
#pragma omp parallel
{
#pragma omp for                         } WS3
   for(i=0;i<N;i++){
      b[i] = f(a[i]);
   }
#pragma omp for                         } WS4
   for(i=0;i<N;i++){
      a[i] = b[c[i]];
   }
}
```
(a) Code segments including an irregular reference b[c[i]]

```
 static_buffer_alloc(bufa, size);
 static_buffer_alloc(bufb, size_1);
 save_bufb = bufb;
 for(ii=0; ii<N_1; ii+=bf){
   static_buffer_get(bufa, a[ii], ...);
   if (lb< ii <ub)
            bufb += size;

        static_buffer_get(bufb, b[ii], ...);

   for(i=ii; i< min(ii+bf,N_1);i++)
     bufb[i] = f (bufa[i]);
   static_buffer_put(bufb, b[ii], ...);
   }
   static_buffer_free(bufa, size);
```
(b) Transformed code for WS3

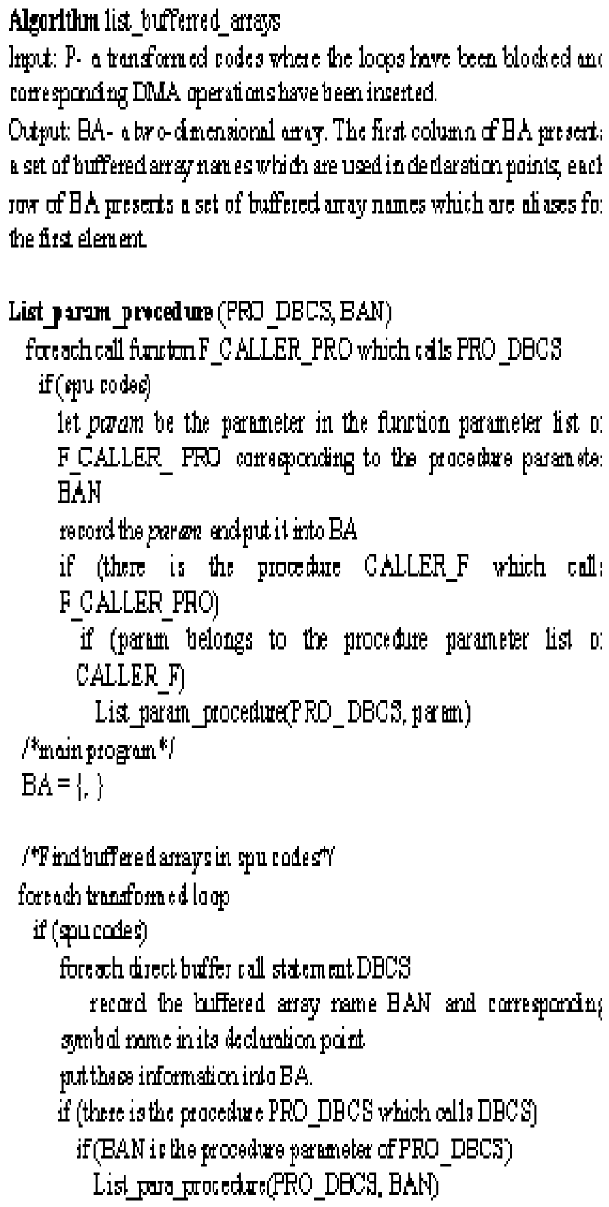
```
 static_buffer_alloc(bufc, size);
   static_buffer_alloc(bufa, size);
   for(ii=0; ii<N_1; ii+=bf) {
      static_buffer_get(bufc, c[ii], ....);
      for(i=ii; i< min(ii+bf,N_1);i++)
          if (lb< bufc[i] <ub) {
     bufa[i] = buffer_lookup(bufc[i],bufb);
   /*look up the buffer to find required data*/
    } else {
     bufa[i] = cache_lookup(bufc[i],b);
   /*look up the SCC to find required data*/
    }
   static_buffer_put(bufa, a[ii], ....);
 }
 static_buffer_free(bufa, size);
 static_buffer_free(bufc, size);
 static_buffer_free(save_bufb, size_1);
```
(c) Transformed code for WS4 by direct buffers and SCC

Fig. 7: Example of buffer2cache optimization

4 Identifying Buffered Arrays

For the applications with SCC and direct buffer, the coherence of cache and buffer has to be maintained to ensure the correctness of program execution according to OpenMP relaxed-consistency model. Our optimization only extends the live range of direct buffer and doesn't change the semantic of relaxed consistency model in OpenMP. Thus, for the coherence problem of buffer and SCC, the compile-time/run-time SCC check technique and corresponding global data flow analysis [5] proposed in T. Chen can also be applied in our cases. T. Chen indicated that the run-time coherence checks are only invoked at the beginning/end of the live range of a direct buffer. In our cases, the extensions to the live range of direct buffer may decrease the total overhead of coherence checks, but it makes the run-time check operations relatively centralized. A precise array data-flow analysis can decrease the overhead of run-time check operations effectively. In this section, an inter-procedural buffered arrays identification algorithm, shown in Fig. 8, is proposed to improve the precision of global data-flow analysis in Ref.[5].

In addition, the RSD, which is incorporated into the global data analysis [5], can also reduce the number of coherence maintenance significantly. Fig. 9 shows the total number of run-time coherence check operations after



```
Algorithm list_buffered_arrays
Input: P- a transformed codes where the loops have been blocked anc
corresponding DMA operations have been inserted.
Output: BA- a two-dimensional array. The first column of BA present
a set of buffered array names which are used in declaration points, eacl
row of BA presents a set of buffered array names which are aliases fo
the first element.

List_param_procedure (PRO_DBCS, BAN)
  foreach call function F_CALLER_PRO which calls PRO_DBCS
    if (spu codes)
      let param be the parameter in the function parameter list o
      F_CALLER_ PRO corresponding to the procedure paramete
      BAN
      record the param and put it into BA
      if (there is the procedure CALLER_F which call
      F_CALLER_PRO)
        if (param belongs to the procedure parameter list o
        CALLER_F)
          List_param_procedure(PRO_DBCS, param)
/*main program*/
BA = {, }

/*Find buffered arrays in spu codes*/
foreach transformed loop
  if (spu codes)
    foreach direct buffer call statement DBCS
        record the buffered array name BAN and correspondin
    symbol name in its declaration point
    put these information into BA.
    if (there is the procedure PRO_DBCS which calls DBCS)
      if (BAN is the procedure parameter of PRO_DBCS)
        List_para_procedure(PRO_DBCS, BAN)
```

Fig. 8: Inter-procedural algorithm to identify buffered arrays

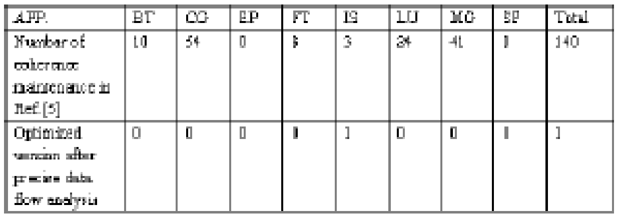
our precise data flow analysis. For NAS OpenMP benchemarks [14], there is only one DMA put operation (from IS) which needs the run-time coherence checks.

5 Performance Evaluation

We have implemented a subset of OpenMP directives and our optimizations using the Cetus compiler [9] and self-developed run-time library. The runtime library which is implemented using the Cell SDK 3.0 [15,16] consists of SPE component and PPE component. Since current Cetus compiler only supports C+OpenMP, all of test cases adopt corresponding C version. The C + OpenMP program is translated into C + run-time library API. For translated codes, some array subscript transformations (such as the translation from the address of the global memory to one of buffer or cache in the local store) and code partitions (for large application to fit in the local store) are implemented manually.

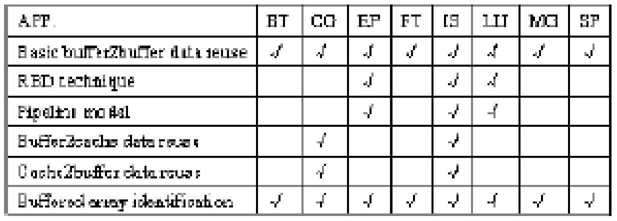
To comprehensively evaluate the performance of our optimizations, we employ the C + OpenMP version [17] of NAS benchmark from Omni OpenMP compiler project. The experiments are conducted on a QS20 Cell Blade which has two Cell BE chips and runs RedHat Linux ES 5.1. We only use one Cell BE chip. A command numactl is used to bind our programs to one Cell chip. The baseline version adopts the double-buffer technique [6] to transform loops (only containing regular references) without the optimizations proposed in section 2, 3 and 4. The total size of buffer is 64k. The SCC, which is used in irregular references, is a 4-way associative cache with 128-byte cache line. The total size of SCC is 64k.

We summarize our optimizations for each application in Fig. 10. The buffered array identification and basic buffer2buffer data reuse technique can be used in all cases. The RBD technique and pipeline model focus on the critical constructs which exist in EP, IS and LU. The buffer2cache and cache2buffer data reuse techniques are



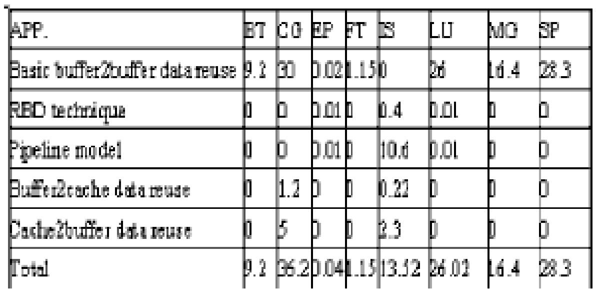
APP.	BT	CG	EP	FT	IS	LU	MG	SP	Total
Number of coherence maintenance in Ref [5]	10	54	0	8	3	24	41	0	140
Optimized version after precise data flow analysis	0	0	0	0	1	0	0	0	1

Fig. 9: The number of coherence maintenance



APP.	BT	CG	EP	FT	IS	LU	MG	SP
Basic buffer2buffer data reuse	√	√	√	√	√	√	√	√
RBD technique			√		√	√		
Pipeline model			√		√	√		
Buffer2cache data reuse		√			√			
Cache2buffer data reuse		√			√			
Buffered array identification	√	√	√	√	√	√	√	√

Fig. 10: Performed Optimizations



APP.	BT	CG	EP	FT	IS	LU	MG	SP
Basic buffer2buffer data reuse	9.2	30	0.02	1.15	0	26	16.4	28.3
RBD technique	0	0	0.01	0	0.4	0.01	0	0
Pipeline model	0	0	0.01	0	10.6	0.01	0	0
Buffer2cache data reuse	0	1.2	0	0	0.22	0	0	0
Cache2buffer data reuse	0	5	0	0	2.3	0	0	0
Total	9.2	36.2	0.04	1.15	13.52	26.02	16.4	28.3

Fig. 11: Communication data reduction (Percentage) between the local memory and global memory

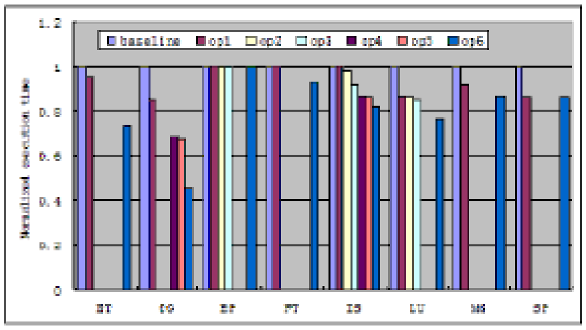


Fig. 12: Normalized execution time on 8 SPEs for different optimization techniques

applied in irregular applications (including CG, FT and IS) where the same array is referenced by both the cache and buffer.

5.1 Communication Data Reduction between Local Memory and Global Memory

The buffered array identification algorithm can enhance the precise of global data flow analysis to reduce the overhead of run-time coherence check. Other optimizations can reduce transferred data between the local memory and global memory. Fig. 11 shows the data reduction (between the local memory and global memory) for each application. The input data of NAS benchmark employ Class A.

In the second row of Fig. 11, since the benchmark EP and IS have only 8 loops which can be tiled by direct buffering and reference very small amount of array data, these benchmarks gain a little of improvement using the basic buffer2buffer data reuse. For the benchmark FT, our optimizations are difficult to find the data reuse due to the complex control-flow of program. The advanced loop fusion technique [18] can be applied to find more data reuse opportunities. In the forth column, the pipelining model replace the communications between the local stores and global memory with the communications between the local stores. The communication data reduction depends on the data amount referenced by the critical region. For the buffer2cache and cache2buffer data reuse optimizations, the size of reused data depends on the ratio of the retained data size to the size of work set of irregular references because the input data of IS and CG is random array.

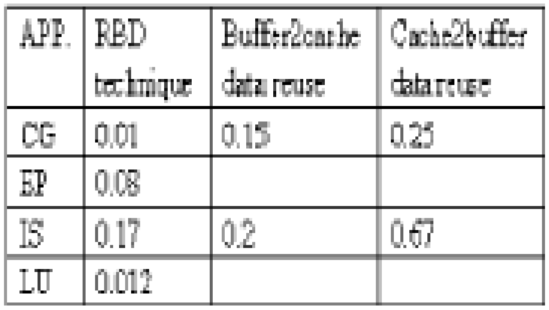
5.2 Execution Time of Optimization Techniques

Fig. 12 shows execution time on 8 SPEs for the baseline version and different optimization techniques. The actual execution time is normalized to the time for the baseline version for each application. The bar marked y baseline denotes the baseline version without the optimizations presented in section 2, 3 and 4. The bar marked by op1 represents the version using the basic buffer2buffer data reuse optimization. The bar marked by op2 represents the version which introduces RBD technique into the version marked by op1. The bar marked by op3 represents the version which introduces pipeline model into the version marked by op2. The rest can be done in the same manner. The last bar marked by opt6 contains the basic buffer2buffer data reuse, RBD technique, pipeline model, buffer2cache data reuse, cach2buffer data reuse and buffered array identification optimizations.

The basic buffer2buffer data reuse technique not only reduces the number of DMA opertions and corresponding data amount but also minimizes the wait time between the computation and communication. In Fig. 12, the numbers for the basic buffer2bufer data reuse technique conform to the result shown in Fig. 11. The buffer2cache data reuse can reduce the communication amount between the local store and global memory as well as the number of cache miss and corresponding overhead. For CG, our buffer2cache data reuse increases the hit ratio of local store (including the hit of SCC and buffer in the local store) from 59% to 96%. For IS, the hit ratio is increased from 3.3% to 6.1%.

5.3 Runtime Overhead of Optimization Techniques

We measured the runtime overhead of proposed optimization techniques. The basic buffer2buffer data reuse technique and buffered array identification does not give rise to the overhead. The main run-time overhead of our optimization techniques include the overhead of branch statements in the RBD technique, the overhead of branch statements and lookup operations for buffer2cache



APP.	RBD technique	Buffer2cache data reuse	Cache2buffer data reuse
CG	0.01	0.15	0.25
EP	0.08		
IS	0.17	0.2	0.67
LU	0.012		

Fig. 13: Run-time overhead of proposed optimization techniques on 8 SPEs (Percentage)

data reuse and cache2buffer data reuse optimization, and the overhead of coherence maintenance. The overhead of loop up operations and coherence maintenance can be reduced using SIMD instructions and be carefully overlapped with the non-blocking DMA operations. Additionally, the coherence maintenance has been reduced to one issue in our cases. It can be ignored compared to the total execution of program. Fig. 13 gives the run-time overhead of our optimizations (including the RBD technique, buffer2cache data reuse and cache2buffer data reuse) for on 8 SPEs.

5.4 Performance Comparison

Fig. 14 shows the comparative result of the speedups of our BCDR framework and that of the single-source OpenMP compiler in the Cell SDK 3.0. The speedups are computed using 8 SPEs over 1 PPE. The bar CBEXLC represents the speedups of applications using single-source OpenMP compiler. The bar BCDR represents the speedups of applications using our BCDR framework. Lee et.al. [19] found that the actual performance of the single-source OpenMP complier is not good as the numbers reported in Ref. [5,7]. The reason [7] may be that some loops in the benchmark applications were manually modified to expose more optimization opportunities. Thus, we compare with the best experimental results reported in Ref. [7].

From Fig. 14, we can find that our BCDR framework performs better than single-source OpenMP compiler in two applications (CG and LU). For EP and IS, our BCDR framework performs as well as single-source OpenMP compiler. For FT, MG, SP, our framework performs worse than single-source OpenMP compiler. However, note that many compiler optimizations (such as DBDB framework [7] and cache prefetching [20]), which enhance the performance of single loop of program, have been incorporated into the single-source OpenMP compiler. These optimizations which boost performance results reported in Ref.[7] are complementary to our BCDR

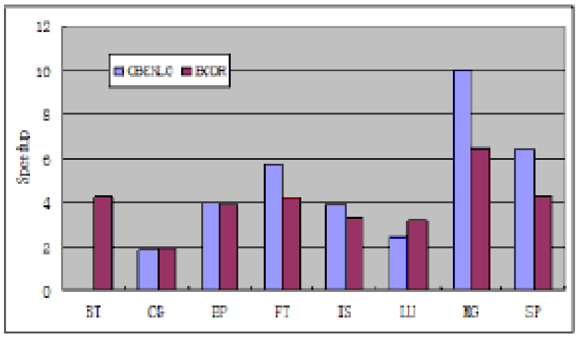


Fig. 14: Performance comparisons of BCDR framework and single-source OpenMP compiler

framework. Additionally, the Ref. [7] indicates that some loops have been modified manually (before OpenMP compiler optimizations) to expose more parallelism. But our BCDR framework adopts original C version of NAS benchmark provided by Omni OpenMP compiler project. We believe that the optimizations of BCDR framework can be used as the optimization of the single-source OpenMP compiler to improve the performance of applications.

6 Related work

Direct buffering, which is only available for regular data accesses in loop nests, is used to store data referenced by loop after strip mining. Chen et al. [6] developed a performance model of direct buffer for single DMA transfer. This model focuses on continuous memory accesses to compute block factor and select the buffer scheme among double- or triple- buffer. Furthermore, Liu et al. [7] designed a Direct Blocking Data Buffer (DBDB) to reuse buffer data for the single innermost loop. They provided two different buffer structures (flat buffer and compressed buffer) for contiguous data accesses and discontinuous accesses, respectively. A performance model was proposed to guide the DMA transfer scheme selection among single-DMA, multi-DMA and DMA-list. Cell SuperScalar [21], CellGen [22] and IBM ALF [23] automate double buffering to overlap DMA and computation. These buffer optimizations can be incorporated into our framework to improve the performance of single loop execution. Our BCDR framework is to reuse data between buffers and SCC.

SCC is an effective and elegant method for irregular data accesses which are unpredictable at compile-time phase. A lot of research efforts [24,25,26] have been focused on SCC. For Cell BE processor, Balart et al. [27] implemented an asynchronous software cache to overlap communication and computation. Furthermore, Gonzalez et al. [28] optimize the performance of applications with high-locality and irregular accesses using a hierarchical, hybrid software-cache architecture. This hybrid software cache architecture is similar to the buffer and cache scheme of IBM single source compiler [3]. COMIC [19] provided a new memory consistency model called centralize release consistency where coherence management is centralized in the PPE. It includes a page buffer implementation which is similar to a 4-way set associative cache. Seo et al. [29] further design an extended set-index cache and adaptive execution strategies that select the optimal cache line size and replacement policy. The software prefetching technique [20] is proposed to improve the performance of SCC. Looking both forwards and backwards schemes are given to minimize the cache pollution caused by prefetching.

To gain the best performance, IBM single source compiler [3] adopts SCC and direct buffering for irregular and regular accesses respectively. Chen et al. [5] presented a compile-time and run-time analysis method to maintain the coherence of SCC and buffers. A global data-flow analysis was proposed to reduce the overhead of run-time coherence check. In this paper, we give an inter-procedure buffered array identification algorithm to increase the precise of the global data-flow analysis as to reduce the number of run-time coherence checks.

Stream reuse techniques [30,31,32] are proposed to optimize stream programs on stream processors[33, 34] which also has limit local memory. Our optimizations are to improve data transfer for OpenMP programs. To the best of our knowledge, we present the first method to exploit the data reuse between SCC and buffers on Cell-like systems.

7 Conclusion

In this paper, we proposed a set of data reuse optimizations for the buffer and software controlled cache. We first give optimization techniques (including basic buffer2buffer data reuse, Retaining Buffered Data and pipelining model) for data reuse between the buffers to reduce the number of DMA operations and to minimize the amount of transferred data. After buffer2buffer data reuse optimizations, the data reuse optimizations between the software controlled cache and buffer are performed to improve the performance of irregular applications. Further, an inter-procedure buffer array identification algorithm is given to improve the precision of data flow analysis as to reduce the overhead of run-time coherence checks. The experimental results show that our optimizations provide many opportunities for buffer and cache. The transferred data amount is reduced by 16.35% on average. Our optimizations further reduce 19.7% of the average execution time. The run-time coherence maintenance overhead is reduced significantly.

Acknowledgement

This paper is supported by Natural Science Fund of China under grant 61303050, the Hi-Tech Research and Development Program (863) of China No.2011AA01A205, and around Five Top Priorities of One-Three-Five Strategic Planning, CNIC(Grant No. CNIC_PY-1404).

References

[1] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic Processing in Cell's Multicore Architecture. IEEE Micro, **26**, 10-24, March/April 2006.
[2] IBM, Sony, and Toshiba. Cell Broadband Engine Architecture. IBM, 2011. http://www.ibm.com/developerworks/power/cell/.
[3] K. O'Brien, K. O'Brien, Z. Sura, T. Chen, and T. Zhang. Supporting OpenMP on Cell. International Journel of Parallel Programming (IJPP), **36**, 289-311, June 2008.
[4] OpenMP, http://openmp.org/wp/.
[5] T. Chen, H. Lin, T. Zhang et al., ”Orchestrating data transfer for the Cell/B.E. processor,” in Proc. of the International Conference on Supercomputing (ICS'08), 2008.
[6] T. Chen, Z. Sura, K. O'Brien, and K. O'Brien, ”Optimizing the use of static buffers for DMA on a CELL chip,” in Proc. of the International Workshop on Languages and Compilers for Parallel Computing (LCPC'06). Springer Berlin, 2006, pp. 314-329.
[7] Tao Liu, Haibo Lin, Tong Chen, John Kevin O'Brien, Ling Shao, ”DBDB: optimizing DMA transfer for the Cell BE Architecture”, in Proc. of the International Conference on Supercomputing (ICS '09),2009.
[8] Haibo Lin, Tao Liu, Lakshminarayanan Renganarayanan, Huoding Li, Tong Chen, Kevin O'Brien, Ling Shao: Automatic Loop Tiling for Direct Memory Access. IPDPS 2011: 479-489
[9] http://cetus.ecn.purdue.edu, Cetus Release 1.3, June 2011.
[10] Bacon, D.; Graham, S.; Sharp, O.; ”Compiler Transformations for High Performance Computing”; ACM Computing Surveys; **26**, pp. 345-420, Dec. 1994.
[11] John Ng,Dattatraya Kulkarni,Wei Li,Robert Cox,and Scott Bobholz, Inter-procedural Loop Fusion, Array Contraction and Rotation,in 12th International Conference on Parallel Architectures and Compilation Techniques (PACT'03),2003.
[12] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. IEEE Transactions on Parallel and Distributed Systems, **2**, 350-360, 1991.
[13] Chen, T. Raghavan, R. Dale, J. N. Iwata, E., Cell Broadband Engine Architecture and its first implementation: a performance view, IBM JOURNAL OF RESEARCH AND DEVELOPMENT, 2007, **51**, pages 559-572.
[14] H. Jin, M. Frumkin, and J. Yan, ”The OpenMP implementation of NAS parallel benchmarks and its performance.” NAS Technical Report NAS-99-011, NASA Ames Research Center, Moffett Field, CA, October, 1999.

[15] IBM DevloperWorks. Cell broadband engine resource center. http://www.ibm.com/developerworks/power/cell/downloads.html.
[16] IBM. Software Development Kit for Multicore Acceleration version 3.0, Programmer's Guide. IBM, 2007. http://www.ibm.com/developerworks/power/cell/.
[17] Parallel and High Performance Applicational Software Exchange Editorial Committee. Omni OpenMP compiler project. http://phase.hpcc.jp/omni.
[18] Manjikian, N.; Abdelrahman, T.; "Fusion of Loops for Parallelism and Locality"; IEEE Transactions on Parallel and Distributed Systems; **8**, pp. 193-209, Feb. 1997.
[19] Lee, S. Seo, C. Kim et al., "Comic: A Coherent Shared Memory Interface for Cell BE," in Proc. the 17th international conference on Parallel Architectures and Compilation Techniques (PACT'08), 2008.
[20] T. Chen, T. Zhang, Z. Sura, and M. G. Tallada, "Prefetching Irregular References for Software Cache on Cell," in Proc. of the sixth annual IEEE/ACM international symposium on Code generation and optimization (CGO'08), 2008, pp. 155-164.
[21] P. Bellens, J. M. P?erez, R. M. Badia, and J. Labarta. CellSs: A Programming Model for the Cell BE Architecture. In Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing (Supercomputing'06), page 86, 2006.
[22] S. Schneider, J.-S. Yeom, B. Rose et al., "A Comparison of Programming Models for Multiprocessors with Explicitly Managed Memory Hierarchies," in PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming. New York, NY, USA: ACM, 2008, pp. 131-140.
[23] C. H. Crawford, P. Henning, M. Kistler, and C. Wright. Accelerating Computing With the Cell Broadband Engine Processor. In Proceedings of the 2008 ACM Conference on Computing Frontiers (CF08), pages 3-12, 2008.
[24] O. S. Unsal, R. Ashok, I. Koren et al., "Cool-cache for hot multimedia," in Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture (MICRO34). Washington, DC, USA: IEEE Computer Society, 2001, pp. 274-283.
[25] Z. Radovi?c and E. Hagersten, "Removing the overhead from software-based shared memory," in Proc. of the 2001 ACM/IEEE conference on Supercomputing (SC'01). New York, NY, USA: ACM, 2001, pp. 56-56.
[26] C. A. Moritz, M. Frank, and S. P. Amarasinghe, "Flexcache: A framework for flexible compiler generated data caching," in the Second International Workshop on Intelligent Memory Systems (IMS'00). London, UK: Springer-Verlag, 2001, pp. 135-146.
[27] J. Balart, M. Gonzalez, X. Martorell et al., "A Novel Asynchronous Software Cache Implementation for the Cell-BE processor," in Proc. of the 20th International Workshop on Languages and Compilers for Parallel Computing (LCPC'07), 2007.
[28] M. Gonz'alez, N. Vujic, X. Martorell et al., "Hybrid access-specific software cache techniques for the Cell BE architecture," in Proc. the 17th international conference on Parallel Architectures and Compilation Techniques (PACT'08), 2008, pp. 292-302.
[29] S. Seo, J. Lee, and Z. Sura, "Design and implementation of software-managed caches for multicores with local memory," in Proc. of the 15th International Symposium on High-Performance Computer Architecture (HPCA'09), 2009.
[30] Abhishek Das, William J. Dally, and Peter Mattson. Compiling for stream processing. In PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques, pages 33-42, New York, NY, USA, 2006.
[31] Xuejun Yang, Ying Zhang, Jingling Xue, Ian Rogers, Gen Li, and Guibin Wang. Exploiting loop-dependent stream reuse for stream processors. In PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques, pages 22- 31, 2008.
[32] Xuejun Yang, Li Wang, Jingling Xue, Yu Deng, Ying Zhang: Comparability graph coloring for optimizing utilization of stream register files in stream processors. in PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming(PPOPP 2009). New York, NY, USA: ACM: 111-120.
[33] Intel many integrated core architecture. http://en.wikipedia.org/wiki/Intel_MIC, Oct. 2012. (from Internet)
[34] C. Intel, Knight Corner Software Developers Guide. Intel, 2012.

Jue Wang is currently working as a associate professor in the supercomputing center of Chinese Academy of Science. The motivation behind his work is to improve soft systems by increasing the productivity of programmers and by increasing software performance on modern architectures including many cores clusters, GPU and Intel MIC.

Yangang Wang is an associate researcher in Supercomputing Center of Chinese Academy of Sciences. His research interests include computational mathematics and high performance computing.