RC24750 (W0902-083) February 19, 2009
Computer Science

IBM Research Report

Runtime Address Disambiguation for Local Memory Management

Tong Chen[1], Tao Zhang[1], Haibo Lin[2], Tao Liu[2], Kevin O'Brien[1], Marc Gonzalez Tallada[3]

[1]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

[2]IBM China Research Lab

[3]Department of Computer Architecture
Barcelona Tech

Runtime Address Disambiguation for Local Memory Management

Tong Chen[1], Tao Zhang[1], Haibo Lin[2], Tao Liu[2], Kevin O'Brien[1], Marc Gonzalez Tallada[3]

[1]IBM Waston Research
{chentong, taozhang, caomhin@us.ibm.com}

[2]IBM China Research Lab
{linhb, liuttao@cn.ibm.com}

[3]Department of Computer Architecture, Barcelona Tech
{marc@ac.upc.edu}

Abstract.

In heterogeneous multi-core systems, such as the Cell/B.E. and certain embedded systems, the accelerator core has its own fast local memory without hardware supported coherence between the local and global memories. It is the software's responsibility to dynamically transfer in and out the working set to the local memory when the total data set is too large to fit in the local memory. Similar to the hardware cache approach, a software-controlled cache can be set up to automatically manage the local memory for data transfer and reuse. However, software cache may bring about extra overheads, such as cache lookup and cache directory maintenance. Usually, the regular references in the program can be optimized with direct buffering, which replaces the references with local buffers directly at compile time to minimize the runtime overhead. However, such optimization relies on the precise alias or dependence information generated by the compiler, or directives provided by users. In applications where such information is absent, the compiler may lose opportunities for direct buffering. In this work, we explore the runtime address disambiguation in the local memory management. We propose a framework in which the references from software cache and direct buffers are checked at runtime with an overlapping detection method optimized for our purpose and hardware. If the addresses overlap, one local copy is kept to solve the coherence problem. Two directories, one for direct buffers and one for software cache are used together with their interaction carefully devised. As a result, our solution keeps the advantages of both software cache and direct buffer, and is able to disambiguate memory accesses efficiently at runtime. We have implemented this method in the XL compiler for acceleration on Cell, and have conducted experiments with small kernels and the NAS OpenMP benchmarks. The results show that our method maintains correctness while increasing the opportunities for direct buffering optimization. The performance of some benchmarks can be improved up to a factor of 3 while no slowdown has been found on any benchmark.

1 Introduction

In heterogeneous multi-core systems, reducing hardware complexity and minimizing power consumption are important design considerations [1, 2]. Providing each of the accelerator cores in such systems with its own fast local memory is one means of accomplishing this goal. Typically, such systems will not provide hardware supported coherence between these local memories and the global system memory. When an application (both code and data) fits in the local memory, performance can be predicted with high precision. Such a feature is critical for real time applications.

The Cell Broadband Engine Architecture (Cell/B.E.) [3] is an example of such a system. It comprises 8 SPEs, each with a 256KB fast local memory, and a globally coherent DMA engine for transferring data between these local memories and the shared global memory. Scratchpad memory [4] in embedded systems is another example of this type of memory hierarchy. This memory design requires careful programming to use the fast local memory efficiently and reduce the long latency accesses to the global memory so as to obtain top performance.

The local memory can be managed manually by programmers, or automatically by compilers, or some place between the two extremes. In the manual management approach, programmers develop or modify their applications customized for the fast local memory. They decide how to use the local memory and insert the necessary data transfer explicitly in the program. This approach gives the users full control of the program and is preferred by some experienced users who are seeking the peak performance. For example, many users in the Cell community pursue performance with the basic utilities provided in Cell software development toolkit (SDK) [5]. However, this approach requires lots of expertise and effort to actually get the optimal performance. To enhance the productivity of programmers, and the quality and portability of the programs, new programming models have been proposed. One example is the acceleration language framework (ALF) [6], which wraps the computation performed by accelerators with functions and specifies the input and output set of the functions, hiding the details of data transferring from users. StreamIt [7] is another example, which adopts streaming program model and relies on compiler to manage the local memory. SPMD programming model is used by CUDA [8] and RapidMind [9].These approaches can provide help for programmers. However, code modification or rewriting is still needed.

To support the widely used conventional shared memory programming model, IBM developed XL compiler for acceleration [10]. This compiler manages the local memory automatically for the popular OpenMP programs[11]. The weak consistency memory model used by OpenMP can be efficiently supported on the Cell/B.E. To run a program with a large data set on accelerators with small local memory, a software controlled cache is the basic method used for data transfer and data reuse. The software controlled cache (or software cache in short) is a software implementation of a cache in the local memory. Every load/store instruction to global memory (also called EA space) is instrumented with cache related instructions by the compiler. At runtime, the added instructions perform software cache lookup with the EA address value, and invoke cache miss handling when the accessed location is not in the cache. The cache method is able to handle all kinds of data references uniformly and capture data reuse that occurs on the run. However, it is typically an expensive approach, and in practice we need to complement it with additional techniques in order to provide better performance. In direct buffering, compiler allocates temporary buffers in the SPE local memory and inserts data transfer operation (DMA operation in Cell/B.E.) to move data between local buffer and global memory. Each load/store to global memory is then replaced with the direct load/store to the buffers at compile time. As a result, both the software cache lookup and miss handling cost can be completely eliminated. The size of DMA transfer chunk can be adapted to the application instead of being determined by the fixed cache line length so that the DMA operation can be optimized. However, since the mapping of references between global memory and local buffers is done at compile time, the application of direct buffering is limited by the accuracy of the alias and/or dependence analysis of the compiler. Applications often contain references that can not be analyzed at compile time. However such references do have opportunities for direct buffering.

It is desirable to apply both software cache and direct buffering, especially on large complicated applications so that different types of references can be optimized appropriately. The integration of software cache and direct buffering may create data **coherence problems** among the local buffers created by direct buffering, or between the cache line and the local buffers, because one data item in the global memory may have more than one copy in the local memory. If one copy is modified, the other will have to be updated properly. Otherwise, incorrect result may be generated. This is the coherence problem tackled in the paper. We should note that this coherence problem is caused by the multiple copies of data in the local memory of a single thread, and is orthogonal to the coherence issues among different threads.

One of the solutions to tackle this data coherence problem is, obviously, to improve the alias and dependence analysis in compilers. Another solution presented in [12] is to reduce the requirement for alias and dependence information. With some data synchronization, the alias query is limited to a local range, increasing the chances for direct buffering. These two approaches work for some cases but can not solve the problem when the alias information is actually unknown statically or is changing from time to time. Another approach [13] is to give up direct buffering optimization. Instead, varying length of cache line and inspector-and-executor model is used to get some benefits which usually are easily gained from direct buffering.

In this paper, we explore the approach of runtime address disambiguation for local memory management. When the compiler can not guarantee that there is no overlap between references for different buffers in direct buffering, or between cache and a direct buffer, the direct buffering optimization is applied speculatively at compiler time and then at runtime, memory addresses are checked. If there are overlaps among these addresses, the coherence problem should be addressed at runtime.

There are two major challenges in the runtime address disambiguation approach: how to check the addresses quickly and how to ensure the coherence at runtime without scarifying the performance of original software cache and direct buffering. For example, the direct buffers are accessed directly without any lookup for efficiency. It is undesirable to add any operation for the accesses to direct buffers.

In order to find a good solution for this problem, we need to look into the difficulties in space management. The length of local buffers in direct buffer varies according to the data type and subscript expressions, while the software cache uses fixed length cache lines. When the space for software cache and direct buffer are mixed, we have to guarantee to allocate the contiguous space for direct buffer and avoid space segmentation problem. Since there is no further check when the direct buffers are references associativity conflict has to be avoided. The lookup for cache is instrumented in the code and should be kept intact without introducing extra overhead when runtime address disambiguation is not needed.

Considering these unique requirements and the special features on Cell/B.E. processor, we propose a novel runtime address disambiguation method in this paper to tackle the ambiguous dependence problem. The proposed method maintains only one copy of global data in the local memory to solve the coherence problem. In order to overcome the difficulties in space management described above, a new buffer directory component is introduced to work along with software cache directory, and acts as the master copy when the same data is accessed both from software cache and direct buffer. With this, the software cache and the direct buffers are integrated together with efficiency and simplicity. There is not data fragmentation issue in our new design. Furthermore, we designed an efficient method for address disambiguation. We check the range of the direct buffers with the SIMD instructions on Cell/B.E. This method is much faster than using cache lookup hash function, which requires the varying length of direct buffers to be broken into multiple cache lines for check.

Our major contributions in this paper include:

1. Explore the runtime address disambiguation approach for local memory management. At compile time, compiler can apply direct buffering optimizations speculatively and is never forced to give up direct buffer optimization due to the lack of precise alias and/or dependence information.
2. Propose a new scheme that integrates direct buffering and software cache. The new scheme keeps the advantages from both software cache and direct buffering methods, and guarantees the coherence for them.
3. Design an efficient method for address overlapping detection. The ranges of direct buffers are tested with SIMD instructions on Cell/B.E. It exploits the instruction level parallelism and reduces the execution time by factor of 4.

We have implemented dynamic address disambiguation scheme within the XL compiler for acceleration framework [8]. Our experiments show that the proposed runtime coherence maintenance method accelerates execution up to 3-4 times on a set of representative benchmarks for high performance computing. The study in this paper also provided some insights on the dependence analysis in our compiler. In fact, the some improvements for static dependence analysis have been introduced.

The rest of this paper is organized as follows: section 2 presents details of our runtime coherence maintenance scheme; the experimental results can be found in section 3; and we conclude this paper with section 4.

2 Runtime Address disambiguation Method

In this section, we present our runtime address disambiguation method for local memory management. The main ideas and overview of our method are introduced first. Then we go into detail about our fast memory overlapping detection algorithm. Finally, we describe in details how to use these ideas to handle the two coherence problems in local memory management: the coherence between cache and a direct buffer, and the coherence between two direct buffers.

2.1 Overview

When a program containing references to global memory is run on the SPE of Cell/B.E., either software cache or direct buffering is used to transfer the data. Fig 1 shows an example with three references to EA space through pointer p1, p2, and p3 in a loop. From the reference pattern, it is desirable to put the regular references, through p1 and p2, into direct buffers. As shown in fig 1(b), the direct buffer lp1 and lp2 are used and corresponding DMA operations are inserted. Usually, loop blocking (not shown in this example) has to be applied to control the direct buffer size. The reference through pointer p3 has a function call in its subscript expression and software cache is used for it. The illustrative code cache lookup and invocation to miss handler is inserted.

Now we will illustrate the coherence problem with this example. Suppose that the compiler can not assert that pointer p1, p2 and p3 are not aliased, such optimization may introduce potential coherence problems:

1. between software cache and direct buffer: the value read from software cache for temp may be obsolete because it has been modified though lp1.
2. between the direct buffers. The value read from lp2 is obsolete because it has been modified through lp1.

```
    /* p1, p2 and p3 are pointers
pointing to the EA space */
    for(j=0; j<n;j++) {
    *p1 = *p2+*(p3+f(j));
    }
```

(a) original code

```
    /* allocate buffer, lp1 and lp2 in local
space */
    DMA p2[0:n] to lp2[0:n];
    for (j=0; j<n; j++) {
      look up*(p3+f(j)) in software cache;
      if miss, call miss handler;
      temp = value of *(p3+f(j)) in software
cache;
      *(lp1+j) = lp2[j]+temp;
    }
    DMA lp1[0:n] to p1[0:n];
```

(b) Optimized code

Fig. 1. Example of coherence problem from direct buffering optimization

Our basic idea to solve this coherence problem is to check whether references from software cache and different direct buffers are overlapped in their EA addresses at runtime. If there is no overlap, they can be handled as they are in previous methods. If there is overlap, we have to modify the previous methods to allow only **one copy** of local data in the system.

However, we can not simply merge the direct buffer and software cache into one structure because they have different requirement. For example, the direct buffers are accessed directly without further lookup. We have to guarantee that the space for the direct buffer remains unchanged all the time. On the other hand, the references to software cache have lookup code instructions inserted in binary. The focus for software cache is the efficiency for lookup. Consequently, cache line mapping has limited associativity and cache line eviction may occur at runtime. The local space allocation and de-allocation is also different in the direct buffers and software cache. The software cache requires fixed length cache line space, which is easy to manage. The direct buffers allocate and free continuous varying length space in a stack manner (corresponding to the loop nests structure in the program). Segmentation may be a problem if the space is simply mixed with software cache management.

Another requirement for the design of the new runtime address disambiguation method is to avoid notable slowdown of software cache or direct buffer when the runtime address disambiguation is not needed. Those are the cases when the previous compile time method can handle. For example, the code instrumented into binary for cache lookup has been highly optimized for its execution time and space. It is undesirable to increase its size or execution time for runtime address disambiguation since such change will affect all cache references. For another example, the references of direct buffer are directly to the local buffers without any extra inserted code. We'd better not to insert any code for direct buffer references in the new method so that the performance benefit from direct buffers will not be lost for runtime address disambiguation.

Our solution is to introduce a directory data structure for direct buffer and make it work along with the existing software cache directory. The directory for the direct buffers is a stack structure, which is pushed or popped when direct buffers are allocated or de-allocated respectively. Each entry of the direct buffer records the boundary of EA address and local address. At runtime, both software cache directory and direct buffer directory are checked for possible overlapping in their EA addresses. When the address overlap is detected, the two directories will be synchronized to make sure only one copy of local data is used. In order to manage the one copy, the boundaries of local space for direct buffers are extended to the cache line boundary. Such transformation can usually reduce the DMA time. Since there are more restrictions for direct buffer, we will use the direct buffer directory as the master copy and evict the cache line when needed. The two directories design allows us still to manage the two methods separately in logic. What we merge them is most in the space allocation and de-allocation. Such design philosophy greatly simplifies whole system implementation and preserves the performance of software cache and direct buffers.

Our method focuses on optimizing the performance for the path when there is not overlap because that's the common case in real application. It can still work correctly when the overlap does occur. In some sense, our runtime address disambiguation method is speculative in performance.

We will explain in details how to check the addresses efficiently and how to solve the two coherence problem in the following sections.

2.2 Overlapping detection

When buffers for direct buffer or a cache line for software cache is requested, their global memory addresses have to be checked. If their addresses are overlapped with other buffers or cache line, special care should taken to guarantee only one local copy is generated. We can still use the cache lookup to heck against the cache lines. The new issue is how to check against the buffers.

```
    Assume v_l is the lower bound vector for all the buffers;
    Assume v_u is the upper bound vector for all the buffers;
    Assume the number of buffers is n.
    for (i = 0; i< n; i++) {
            current_u = spu_splats(v_u[i]);
            current_l = spu_splats(v_l[i]);
            for(j=(i+1)/4*4; j < n; j+= 4) {
                        result = spu_or(spu_and(spu_cmpgt(c_u, v_l[j]), spu_cmpgt(u_l[j],
c_l), result);
            }
    }
    if (spu__extract(spu_gather(result, 0)), there is overlap. Otherwise no
```

Fig. 2. Code illustration for overlap check with SIMD intrinsic

In the direct buffer case, the number of buffers is usually quite small, averaging 8 in our test cases. And the length of buffers is usually much longer than the cache line length and varies from loop to loop. The length of software cache line is 128 byte in our implementation. This length is optimized for the space usage and implementation on Cell. When a buffer, with 4k length for example, is checked with cache lookup, it has to be broken down into 32 cache lines and perform 32 cache lookup. It seems not an efficient approach. Considering the characteristics of our problem and the hardware platform, we reinvestigate the naive pair-wise range comparison method. In this method, each pair is comparison with the lower and upper bound. There is no need to break into multiple checks. Furthermore, we found that the comparison can be optimized with SIMD instructions on CELL. If the lower bound and the upper bound of all the buffers are stored properly in vector format, one address can be compared with the upper or lower boundary of four buffers in one cycle. This is because the address is 32-bit and the data length for SIMD instructions is 128-bit in Cell. The computation power provided by Cell can be exploited in this way. The illustrative code can be found in Fig2.

2.3 Coherence between direct buffers and software cache

In this section, we elaborate details on how we maintain coherence between direct buffers and software cache dynamically. Software cache manages data in the unit of a cache line. So first of all, we extend all direct buffer allocation requests to cache line aligned in terms of both starting address and length. Thus, there will be no partial cache line residing in local buffers created by direct buffer.

We tackle the problems by three steps. First, we need to make sure when direct buffers are allocated and initialized, the coherence requirement is satisfied, i.e., any global data has only one copy in local memory. Second, during the execution of the loop body, we need to properly maintain this property. Finally, some work needs to be done when direct buffers are freed,

In our previous scheme in which compiler guarantees that there is no coherence problem inside the loop body, we only need to maintain coherence between direct buffers and software cache at the boundaries of the live ranges of the direct buffers. In our previous scheme, after a direct buffer is initialized using a DMA read operation we check whether software cache contains a newer copy and update the direct buffer if necessary. Under our new runtime coherence maintenance scheme, when a direct buffer is allocated and initialized, we not only update direct buffer with latest value in software cache as in our previous scheme, we also modify the data pointer of the software cache line to pointing to the proper location in the newly allocated direct buffer so that there is only one copy of the global data in local memory.

During the execution of the loop body, to make sure there is only one copy of global data in local memory, whenever there is a software cache miss, the runtime library needs to check whether the missing global memory address hits in the buffer directory. The code to do that is essentially same as the buffer overlapping detection code detailed previously.

If the missing global data line is not in local direct buffers, the software cache miss handler works normally. If the missing line currently resides in local buffers, the miss handler does not need to do a DMA transfer to get the missing data line since the up-to-date data line is already in local memory. The miss handler just needs to maintain the software cache directory properly. It updates cache tag for the cache line then modifies the data pointer of the cache line to make it point to the location of the data line in the local direct buffer. As a result, software cache and direct buffers will use the same local space for the global data accessed by both methods.

Now that some local space could be shared by direct buffer and software data cache, we need to pay special attention when direct buffer or software cache tries to release the space it uses. As described in section 3.1, in our scheme both direct buffer and software data cache obtain space from a local memory pool. When software data cache has to evict an existing cache line for an incoming cache line, it normally uses the data line previously used by the evicted cache line for the incoming cache line. However, with our runtime coherence maintenance scheme, the miss handler cannot simply reuse the data line. It has to check whether the data line is actually shared with direct buffer. If that is the case, reusing the data line can corrupt the data in direct buffers thus the miss handler has to get a new unused data line. Similarly, special attention is required when direct buffers are released. Direct buffers are released together after the execution of the optimized loop. However, some data lines in direct buffers may be shared by software data cache. To release the local memory safely, the runtime library needs to call cache eviction function of software cache for each of the data lines shared by software cache and direct buffers.

To quickly check whether a data line is shared by direct buffer and software data cache, we add a flag in the tag for a cache line. The flag is maintained by cache miss handler.

2.4 Coherence among direct buffers

After we release the burden of the compiler to guarantee there is no coherence problem between direct buffers for a loop nest, different buffers allocated in direct buffer for a loop nest could have overlapped EA addresses. To eliminate the coherence problem, the basic idea is to make sure for any global memory data there is only one copy of the data in local memory. The fundamental problem in previous approach is that the buffers are statically allocated by the compiler at compile time. But at compile time, we may not know whether those direct buffers allocated overlap in terms of global memory address space. To solve this coherence problem, we need to conduct overlap check for the EA addresses of buffers and allocate buffers properly so that there is only one copy of any global data among all local buffers used by direct buffer. To achieve that, we need to postpone buffer allocation to runtime and allocate local buffers properly. In particular, we do the following in the runtime library:

1) For each direct buffer transformation, we delay the actual allocation for the local buffer from compile time static allocation to runtime dynamic allocation. Instead of creating a static buffer during compilation, compiler generates a buffer allocation request with proper information to the runtime library. At runtime, local buffer allocation is only done after the runtime library collects all buffer allocation requests for this loop nest.
2) After runtime library gathers access ranges of all direct buffers for the loop nest, it does a fast check to see whether any of the access ranges overlap with each other, which is the rare case. The overlapping detection step always incurs whenever compiler cannot guarantee there is no coherence problem between buffers at com-pile time, so it should be implemented very efficiently to reduce runtime overhead.
3) If some of the access ranges do overlap, the runtime library groups the overlapping ranges into an access group. It keeps grouping until none of the access groups overlaps with each other.
4) Finally, the runtime library allocates contiguous local memory space for each access group.

Using this method, whenever two direct buffers overlap a portion of their access ranges, they will share same local buffer space for the overlapped portion, so there will be no coherence problems between different direct buffers. Next, we give more details on our scheme.

3. Performance Evaluation

We implement this method in our Single Source Compiler on Cell/B.E. and conduct experiments with hundreds of small functional test cases, and large OpenMP programs from NAS [14] and SPEC2006 [15]. The small test cases help us in demonstrating the correctness of our algorithm. We will focus on the performance large OpenMP programs.

3.1 Performance of overlapping detection

The proposed overlapping detection method is the corner stone of our system. We first compare our simdized pair-wise method with the cache lookup methods. As stated in our analysis, these two methods perform differently to the number and the length of the buffers to be checked. In the experiment, we choose the parameters in the common range seen in benchmarks. The execution time of cache lookup is normalized to that of the simdized pair-wise. The result is shown in Fig 3.

It can be seen from Fig 2 that our simdized range check method for overlapping detection is much faster than cache lookup when the number of buffer is small. For example, when there are only 4 buffers with size 4k, our overlap diction method is almost 15 times faster. The cache lookup method catches up when the number of buffers increases, and even 2 times faster when there are 64 1k buffers. However, in the common cases found in real benchmarks, our overlap check method is much faster because the number of buffers is usually no more than 16 and the buffer size is more than 2k. Furthermore, the overhead to handle the possible overflow of cache associativity has not even been considered yet in this experiment. When the number of buffers is extremely large and consequently the buffer size is quite small, we can give up direct buffering optimization to avoid possible slowdown. We can define the thresholds based on this experiment.

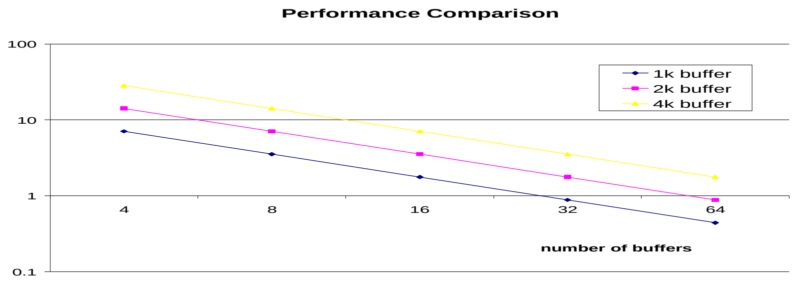

Fig. 3. Performance comparison of simdized range check and cache lookup

3.2 Performance on benchmarks

The runtime address disambiguation method, on one hand, allows the compiler to perform more aggressive direct buffer optimization, but on the other hand, brings about extra maintenance overhead. The overall impact on performance is measured with large benchmarks. The improvement by runtime address disambiguation is reported in Figure 4.

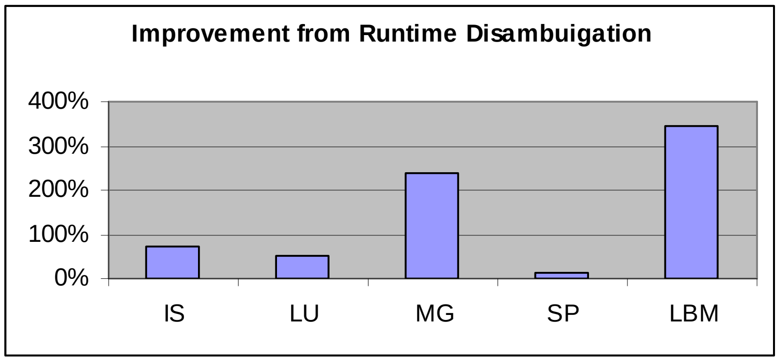

Fig.4. Performance improvement from runtime address disambiguation

Totally 10 benchmarks are tested and the runtime address disambiguation method is invoked in five of them. The improvement is quite significant for IS, MG, LU, MG and LBM because some critical loops in them require runtime address disambiguation so that references can be optimized with direct buffer.

4. Conclusions

In this paper, we have presented our method of runtime address disambiguation for local memory management on accelerator architectures. The precise alias information is no longer a prerequisite for direct buffering optimization. The addresses for software cache and direct buffers are checked at runtime, when compiler can not guarantee that they are not aliased. An efficient overlapping detection method has been optimized with the SIMD instructions on the Cell/B.E. To overcome the coherence problem when the addresses are overlapped, we proposed a new software cache and direct buffering scheme. In this scheme, only one local copy of data is used even if the global memory location is accessed through cache or different direct buffers. In the meanwhile, our integration of software cache and direct buffering preserved the advantages of software cache and direct buffering.

We have implemented this method in the XL compiler for acceleration, and have conducted experiments with the selected NAS OpenMP benchmarks. The results show that our method maintains correctness while keeping most of the opportunities for direct buffer. The execution performance can improve by more than 3 times compared to a static disambiguation method.

Acknowledgement

The researchers at BSC-UPC were supported by the Spanish Ministry of Science and Innovation (contract no. TIN2007-60625), the European Commission in the context of the

HiPEAC Network of Excellence (contract no. IST-004408), and the MareIncognito project was under the BSC-IBM collaboration agreement.

Reference

[1] Rakesh Kumar , Keith I. Farkas , Norman P. Jouppi , Parthasarathy Ranganathan , Dean M. Tullsen, Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction, Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture, p.81, December 03-05, 2003

[2] Canturk Isci , Alper Buyuktosunoglu , Chen-Yong Cher , Pradip Bose , Margaret Martonosi, An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget, Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, p.347-358, December 09-13, 2006

[3] Brian Flachs et al. The Microarchitecture of the Streaming Processor for a CELL Processor. In *Proceedings of the IEEE International Solid-State Circuits Conference*, 2005.

[4] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan and Peter Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In Proceedings of the tenth international symposium on Hardware/software Co-design (CODES), 2002

[5] Cell SDK: http://www.ibm.com/developerworks/power/cell/

[6] ALF: http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/41838EDB5A15CCCD002573530063D465

[7] Ujval Kapasi, Peter Mattson, William J. Dally, John D. Owens and Brian Towles. Stream scheduling. In *Proceedings of the 3rd Workshop on Media and Streaming Processors*, 2001.

[8] CUDA: www.nvidia.com/cuda

[9] RapidMind Multi-core Development Platform: http://www.rapidmind.net/pdfs/WP_RapidMindPlatform.pdf

[10] Alexandre E. Eichenberger et al. Optimizing Compiler for the CELL Processor. In *Proceedings of International Conference on Parallel Architecture and Compilation Techniques*, 2005.

[11] Official OpenMP specifications. http://www.openmp.org/drupal/mp-documents/spec25.pdf.

[12] Tong Chen, Haibo Lin, Tao Zhang, Kathryn O'Brien, Kevin O'Brien. Orchestrating Data Transfer on Cell Processor. International Conference on Supercomputing (ICS) 2008

[13] Marc Gonzalez and et al. Hybrid Access-Specific Software Cache Techniques for the Cell BE Architecture. PACT 2008 .

[14] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. NAS Technical Report NAS99-011, NASA Ames Research Center, 1999.

[15] www.spec.org/cpu2006/