Adaptive Line Size Cache for Irregular References on Cell Multicore Processor

Qian Cao, Chongchong Zhao*, Junxiu Chen,
Yunxing Zhang and Yi Chen

University of Science and Technology Beijing,
100083 Beijing, China
caoqian125@126.com

Abstract. Software cache promises to achieve programmability on Cell processor. However, irregular references couldn't achieve a considerable performance improvement since the cache line is always set to a specific size. In this paper, we propose an adaptive cache line prefetching strategy which continuously adjusts cache line size during application execution. Therefore, the transferred data is decreased significantly. Moreover, a corresponding software cache - adaptive line size cache is designed. It introduces a hybrid *Tag Entry Arrays*, with each mapping to a different line size. It's a hierarchical design in that the *misshandler* is not invoked immediately when an address is a miss in the short line *Tag Entry Array*. Instead, the long line *Tag Entry Array* is checked first, which significantly increases the hit rate. Evaluations indicate that improvement due to the adaptive cache line strategy translates into 3.29 to 5.73 speedups compared to the traditional software cache approach.

Keywords: Adaptive, Software cache, Irregular reference, Cell processor

1 Introduction

Irregular application is widely used in scientific computing, which exposes unclear aliasing and data dependence information. Such applications are frequently seen in reservoir numerical simulation, molecular dynamics, etc.

Heterogeneous multicore is an area and energy efficient architecture to improve performance for domain-specific applications. The Cell processor is a representative heterogeneous multicore, which comprises a conventional Power Processor Element (PPE) that controls eight simple Synergistic Processing Elements (SPEs), as illustrated in Figure 1. PPE has two levels of cache that are coherent with the globally memory, while SPEs don't have cache but each has 256KB of local store. PPE can access main memory directly while SPE only operates directly on its local store and works as an accelerator. Software cache is a common approach to automatically handle data transfers for irregular reference, providing the user with a transparent view of the memory architecture.

* Corresponding author.

There has been substantial research [1-6] on software cache specifically for Cell processor. Eichenberger et al. [1] propose a compiler-controlled software cache. It's a traditional 4-way set-associative cache implemented in software. It adopts the LRU policy and SIMD mode to look up for a match among the four tags in a set.

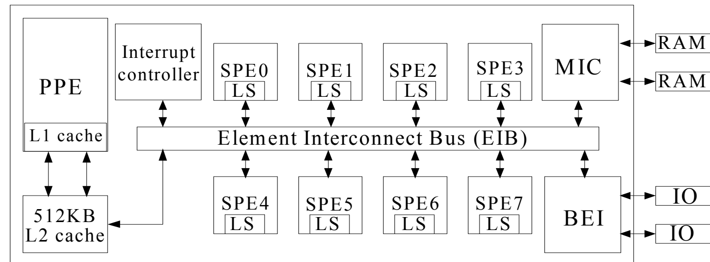

Fig. 1. Cell Architecture

Balart et al. [2] demonstrate a software cache for Cell which adopts hashed list for *lookup* and allows for full associative. This design enables a user to determine code regions guaranteed not to have any cache conflicts. In such a region, the user can reorder *lookup* and *misshandler* operations, so communication and computation can be efficiently overlapped. The strategy usually performs well for specific loops containing few cache accesses with high temporal locality, but it introduces a large implementation overhead for the general applications.

The COMIC runtime system proposed by Lee et al. [3] provides the application with an illusion of a shared memory, in which the PPE and the SPEs can access the shared data. The management of synchronization and coherence is centralized in the PPE and the release consistency is achieved by software cache.

A hybrid access-specific software cache is presented by Marc Gonzalez et al. [4, 5]. It classifies memory accesses into high locality and irregular, and the corresponding high locality cache and the transactional cache are designed. The former applies the write-back mechanism while the latter supports the write-through policy. Its motivation is similar to the direct buffer plus software cache approach.

Chen et al. [6] propose an integrated software cache and direct buffer approach so as to efficiently execute the loops that include both references. Their solution provides compile time analysis and runtime support to minimize the coherency operations.

The software caches usually suffer from poor performance, especially when the irregular reference is encountered. The solutions above always set the cache line to a specific size, which introduces the reduction of data transfers and increases the memory bandwidth overhead. The cache design with adaptive line size could obviously improve the irregular application performance. There are some proposals [7, 8] for the hardware adaptive cache line solutions. But the SPE on Cell has no hardware cache, so we focus on adaptive cache line designs implemented in software.

To the best of our knowledge, the adaptive cache line scheme proposed by Sangmin Seo et al. [9] is the only strategy which continuously adjusts the cache line on Cell processor. Their design is called extend set-index cache (ESC), which is based on 4-way set-associative cache. Nevertheless, the number of TEs could be greater than the number of cache lines. The adaptive strategy in ESC utilizes the runtime to

adapt to characteristics specific to the loop considering that the loops are invoked many times. But their strategy is applied to parallel loops only, and it isn't sensitive to variation across iteration of the loop. Additionally, its storage overhead is large.

In this paper, we propose an adaptive cache line size strategy, which adaptively adjusts cache line size according to the characteristics specific to the irregular reference. The solution gathers the addresses accessed by the irregular reference and divides them into long line addresses (long addresses) and short line addresses (short addresses). The algorithm adaptively chooses the optimal cache line size, regardless of how many times the loop is invoked.

Moreover, a corresponding software cache design - adaptive line size cache (ALSC) is presented. It is based on the 4-way set-associative cache (4WC) and adopts a hybrid *Tag Entry Arrays*, a long line *Tag Entry Array* and a short line *Tag Entry Array*, with each mapping to a different line size. The operations to the long line *Tag Entry Array* is the same as the traditional 4WC, but when a miss occurs in the short line *Tag Entry Array*, the *misshandler* is not invoked at once. Instead, the long line *Tag Entry Array* is checked. So the miss rate is significantly decreased.

In order to implement cache replacement policy for cache design with multiple line sizes, we present a novel LRU policy - IndAlign_LRU. It adopts a link array, with each link mapping to one set in the long line *Tag Entry Array* and two successive sets in the short line *Tag Entry Array*. The data field of the link node stores the cache line index. IndAlign_LRU policy is implemented by moving nodes to the link head or tail.

The experimental results show that our approach obtains speedup factors from 3.29 to 5.73 compared to the traditional software cache scheme with specific line size. Moreover, it significantly reduces the miss rate and the total transferred data size. Additionally, the adaptive approach we proposed shows good scalability.

The rest of the paper is organized as follows. The adaptive software cache line algorithm is presented in Section 2. Section 3 describes the ALSC design. The ALSC operational model is presented in Section 4. Section 5 evaluates our adaptive approach. The last section concludes the paper.

2 The Adaptive Software Cache Line Algorithm

The adaptive cache line prefetching scheme, which is based on our previous work [15], consists of four steps. A loop with normalized boundaries is extracted from CG in the NAS benchmark suite for a clear explanation, as illustrated in Figure 2a. And the adaptive strategy is shown in Figure 2b.

The first step is to initialize the cache lines. For the sake of simplicity, two cache lines, 128B and 256B, are introduced.

The second step is to divide the addresses into long and short addresses. Our adaptive algorithm is applied to each iteration range. We propose a dynamic address collecting solution, which means the address collecting is stopped when the first set conflict is encountered.

```
for(k=0; k<ub; k++)
  sum += a[k]*p[colidx[k]];
```

a. Simplified original code

```
/*step1: initialization*/
lb_tmp=0;
Longln=256;//size of long line
Shortln=128;//size of short line
do{
  /*step2: the addresses dividing process*/
  ub_tmp = collect_dynamic(lb_tmp, ub_tmp);
  for(k=lb_tmp; k<ub_tmp; k++)
    ea[k] = &p[colidx[k]];
  /*align the addresses collected to a 256B boundary*/
  for(k=lb_tmp; k<ub_tmp; k++)
    work_ea[k] = ea[k]&(~(Longln-1));
  /*search the same elements in array work_ea, e.g.,
work_ea[k1] = work_ea[k2]*/
  search_same_data(work_ea, k1, k2);
  if((ea[k1]&(~(Shortln-1)))!=(ea[k2]&(~(Shortln-1))))
    Take ea[k1], ea[k2] as long addresses;
  Take the rest elements in array ea as short ddresses;
  /*step3: prefetch the long and short lines*/
  prefetch(lb_tmp, ub_tmp);
  /*step4: computation loop within the iteration range*/
  for(k=lb_tmp; k<ub_tmp; k++)
    sum = sum + a[k]*cache_buffer[k-lb_tmp]
  /*The return value is used as lb_tmp in next loop*/
  lb_tmp = ub_tmp;
}while(lb_tmp < ub)
```

b. After the adaptive software cache line strategy

Fig. 2. Example for adaptive cache line size algorithm

To explain it clearly, an example is shown in Figure 3. We assume the following:

- There are seven memory requests with addresses ranging from a_1 to a_7 in an iteration range and they are all mapped to the addresses in the range of 256N - (256(N+2)-1)).
- The data located in addresses from 256N to (256(N+1)-1) is represented L_i, and the former 128B is represented by L_{i-1} while the latter 128B is expressed as L_{i-2}. L_{i-1} and L_{i-2} are "adjacent" lines.
- The data located in addresses from 256(N+1) to (256(N+2)-1) is represented L_j, and the former 128B is represented by L_{j-1} while the latter 128B is expressed as L_{j-2}. L_{j-1} and L_{j-2} are "adjacent" lines.

All the seven addresses are taken as short addresses initially, as illustrated in Figure 3a. The addresses a_1 and a_3 are mapped to the "adjacent" lines. If a short line is fetched from the memory, two DMA operations which respectively fetch the L_{i-1} and the L_{i-2} are required. So our adaptive strategy makes the two short lines L_{i-1} and the L_{i-2} merged into a long line L_i. And the addresses a_1 and a_3 are taken as long addresses. Obviously, the data required can be obtained in one DMA. Analogically, addresses a_2, a_4, a_5 are regarded as long addresses.

The data located in addresses a_6 and a_7 is in L_{j-2}, and there's no memory requirement in its "adjacent" line L_{j-1}, so the address a_6 and a_7 are both taken as short line addresses.

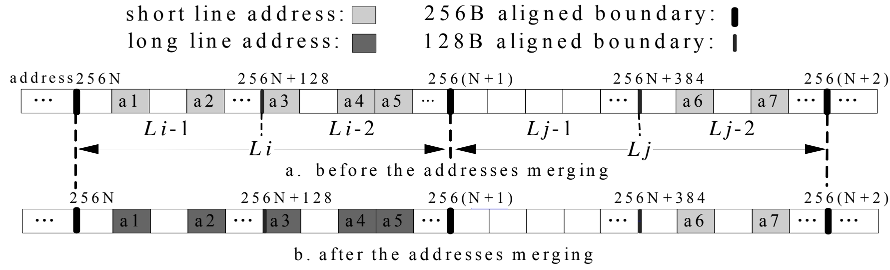

Fig. 3. Example for addresses merging

The third step is to adaptively prefetch the long and short lines from lower bound *lb_tmp* up to upper bound *ub_tmp*.

The last step is the computation loop, which performs the computation.

3 The Adaptive Line Size Cache Structure

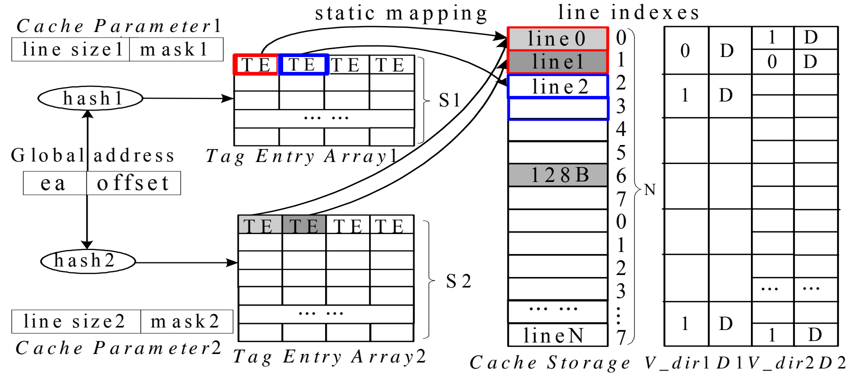

Fig. 4. The ALSC structure

We design a software cache which corresponds to the adaptive cache line algorithm. For the sake of simplicity, we describe the cache including only two kinds of lines, 128B and 256B. It's composed of the following structures, as depicted in Figure 4.

The *Cache Storage* is set to be 64KB in this paper.

The *Cache Parameter*$_1$ includes two components, the *mask*$_1$ and the *line size*$_1$ (L_1). L_1 is set to be 256B, so the number of the long line (n_L) is 256 (64KB/256B).

The *Tag Entry Array*$_1$ is a long address tag lookup table which is composed of S_1 ($S_1 = n_L/4 = 64$) sets. Each TE statically maps to a 256B long line.

The *Cache Parameter*$_2$ is similar to the *Cache Parameter*$_1$. The *line size*$_2$ (L_2) is set to be 128B and the number of the short line (n_S) is 512.

In *Tag Entry Array*$_2$, each tag statically maps to a short line.

Every set in the *Tag Entry Array*$_1$ maps to 8 successive short cache lines. The short lines are orderly numbered 0, 1, 2, ⋯, 7, 0, 1, 2, ⋯, 7, which are the line indexes.

In order to implement LRU replacement policy among multiple line sizes, we extend the traditional LRU replacement policy and present IndAlign_LRU, which adopts an *Index Link Array*. Its initial information and the mapping relationship between the *Index Link Array* and the cache lines are illustrated in Figure 5.

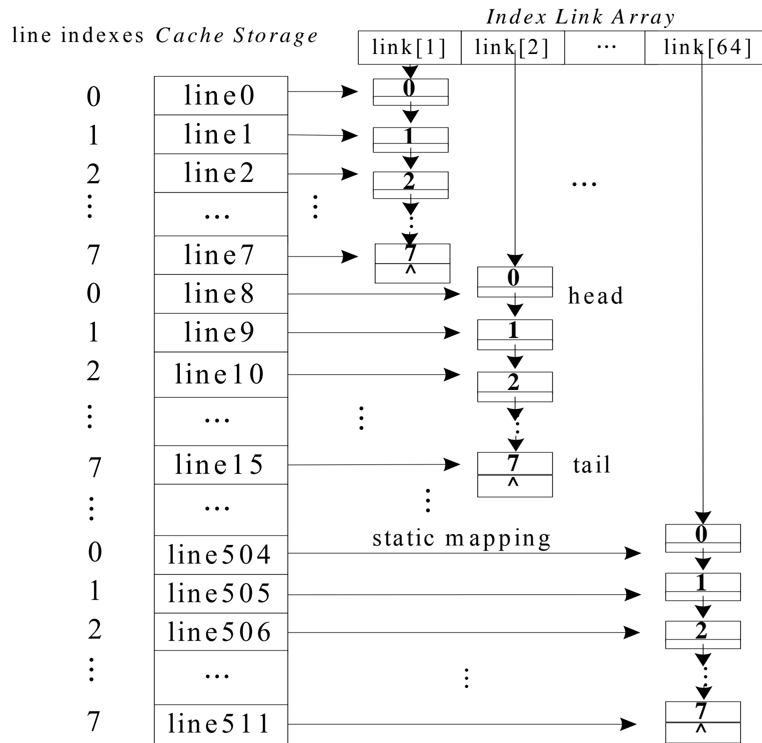

Fig. 5. The mapping relationship between the *Index Link Array* and cache lines

The *Index Link Array* is a link array which has S_1 links. Each link is mapped to one set in the *Tag Entry Array*$_1$ and two successive sets in the *Tag Entry Array*$_2$. Each link contains 8 nodes, with each node having a data field to store the line index. The data filed of link head stores the index of the cache line which is the earliest accessed while the data field of the link tail stores the index of the line which is the latest accessed. The cache line activity is recorded by moving nodes to the head or the tail.

The V_dir_1 and V_dir_2 record the valid bits and the D_1 and D_2 store the dirty bytes.

The cache architecture we propose is a hierarchy design. Figure 6 shows set masks of the long line address and short line address. Both the numbers of set in *Tag Entry*

Array$_1$ and *Tag Entry Array*$_2$ are powers of 2, so a bit-wise AND operation instead of the hash function is used to improve performance, as shown in (1).

$$SetID = (ea \ \& \ SetMask) >> \text{N_bits} \quad (1)$$

Where *SetID* is the number of the set, *SetMask* is the set mask and $2^{\text{N_bits}}$ equals the corresponding cache line size.

Because the number of the set in *Tag Entry Array*$_1$ is 64 and the *line size*$_1$ is 256B, the S*etID* in *Tag Entry Array*$_1$ is decided by the successive 6 bits, ranging from 8th to the 13th bit. Correspondingly, the *SetID* in *Tag Entry Array*$_2$ is decided by the bits from 7th to the 13th. When the cache receives a memory request with a global address *ea*, if it's a miss in the *Tag Entry Array*$_2$, it may be a hit in the *Tag Entry Array*$_1$. Since the number of the bits which decide the *SetID* is one bit less.

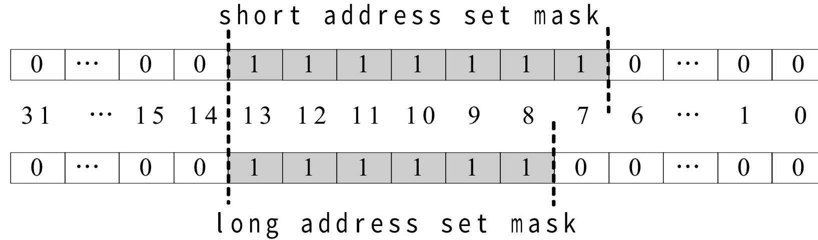

Fig. 6. Set masks of the long line address and short line address

4 The ALSC Operational Model

The simple ALSC operational flowchart is shown in Figure. 7.

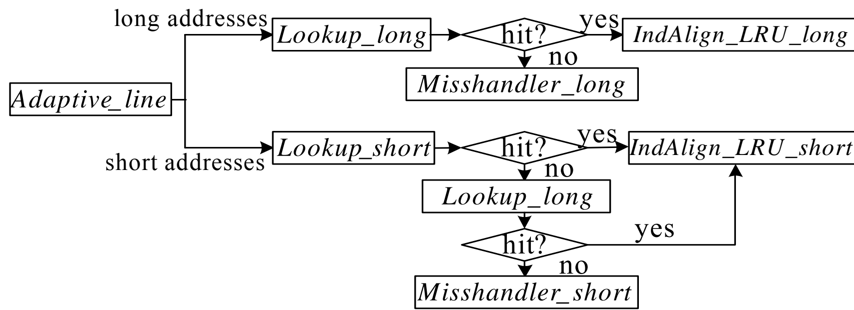

Fig. 7. The ALSC operational flowchart

4.1 *Lookup_long*, *Lookup_short, IndAlign_LRU_long*, *IndAlign_LRU_short*

Both *Lookup_long* and *Lookup_short* are the same as the traditional 4WC. *IndAlign_LRU_long* and *IndAlign_LRU_short* are invoked respectively when the long address hit and short address hit occur.

Suppose a hit of the long address is encountered and the matching set and way is *set_L* and *hit_index_L* respectively, the *set_L*th link nodes whose data fields are (2**hit_index_L*) and (2**hit_index_L+1*) moves to the link tail. This operation is called *IndAlign_LRU_long*. Figure 8a illustrates the operation when *hit_index_L* is 0.

Suppose a short line address comes subsequently and the matching set and way is *set_S* and *hit_index_S* respectively, the *set_L*th set in the *Tag Entry* $Array_1$ and the *set_S*th set in the *Tag Entry* $Array_2$ are both mapped to the *set_L*th link if the equation (*set_L**2=*set_S*) or (*set_L**2 +1=*set_S*) is satisfied.

If the former or the latter is satisfied, the node whose data field is *hit_index_S* or (*hit_index_S+4*) moves to the link tail, respectively. The two cases are illustrated in Figure 8b and Figure 8c respectively. The operations are called *IndAlign_LRU_short.*

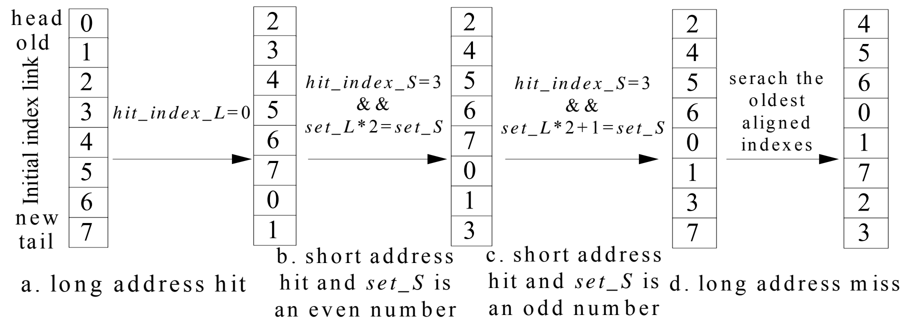

Fig. 8. Examples for the operations in ALSC

4.2 *Misshandler_long*

For a long address, the function *IndAlign*(*index_L*) is defined as follows. If *index_L* is an even number, *IndAlign*(*index_L*) is defined as

$$IndAlign(index_L) = index_L \text{ And } (index_L + 1). \quad (2)$$

If *index_L* is an odd number, *IndAlign*(*index_L*) is defined as

$$IndAlign(index_L) = (index_L - 1) \text{ And } index_L. \quad (3)$$

When a long address which is mapped to the *set_L*th set arrives subsequently and a miss occurs, *Misshandler_long* is invoked, which includes the following phases:

Choosing the oldest lines to be the victim. We first get the oldest indexes, *index_old* from the *set_L*th link head. The two successive short lines whose indexes are *IndAlign*(*index_old*) (as defined in (2)(3)) are chosen to be the victims.

Writing back the dirty bytes and setting the corresponding V_dir bits.

Fetching the long line required from the main memory and filling the corresponding DE. Moreover, the nodes whose data fields are *IndAlign*(*index_old*) moves to the tail of the *set_L*th link, as shown in Figure 8d.

4.3 *Misshandler_short*

For a short address which is mapped to the *set_S*th set, *IndAlign*(*index_S*) is defined as follows. If *set_S* is an even number, *IndAlign*(*index_S*) is defined as

$$IndAlign(index_S) = oldest(0, 1, 2, 3). \quad (4)$$

If *set_S* is an odd number, *IndAlign*(*index_S*) is defined as

$$IndAlign(index_S) = oldest(4, 5, 6, 7). \quad (5)$$

The *oldest*(*ind1*, *ind2*, *ind3*, *ind4*) means the index of the line which is the earliest accessed, with the four parameters in parenthesis denoting the line indexes.

If a short address, which is mapped to the *set_S*th (*set_S* /2 = *set_L*) set in the *Tag Entry Array*$_2$ arrives subsequently and a miss occurs, the *Tag Entry Array*$_1$ is searched first. If there's a valid matching, the operation is similar to *IndAlign_LRU_short*. Otherwise, *Misshandler_short* is called. A simplified ALSC design is shown in Figure 9. We assume the address is mapped to the *set_L*th link.

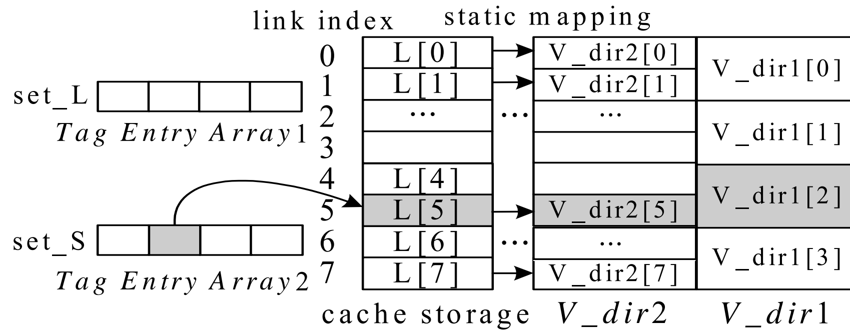

Fig. 9. A simplified ALSC structure

Choosing the oldest index (index_old) from the set_Lth link according to (4) or (5) after judging the parity of *set_S*. We assume the *index_old* is 5.

Checking the V_dir arrays and writing back the dirty bytes. The V_dir_1[2], which is mapped by the *index_old* should be checked first. If it's valid, the dirty bytes in *L*[4] and *L*[5] should be written back. Meanwhile, the V_dir_1[2] is set to be 0. Otherwise, if it's invalid, the V_dir_2[5] is checked. If it's valid, the dirty data in *L*[5] is written back.

Fetching the required short line from the memory, filling the corresponding DE, and setting V_dir_2[5] to valid. Finally, the node whose data field is 5 moves to the link tail.

5 Evaluation

5.1 Evaluation Environments

The experiment is conducted on a Cell BE blade [10] with two Cell processors running at 3.2GHz with 1GB of system memory. In this experiment, the programs are bound to one Cell processor to avoid the NUMA effect.

The performance is measured with Sparse matrix-vector (SpMV) multiplication and IS, CG, FT, MG from NAS parallel benchmarks [11]. The sparse matrix epb1.mtx is a 14734*14734 symmetric matrix, which is obtained from the University of Florida Sparse Matrix Collection [12]. The benchmarks IS, CG, FT, MG are tested with CLASS A. The sequential regions are executed on the PPE while the iterations in the parallel loops are distributed among the available SPEs. The system runs Fedora9 (Linux Kernel 2.6.25-14). Our programs are compiled in the Cell SDK3.1.

5.2 Execution Speed

In this section, we evaluate four software cache configurations. The first one is a traditional software cache design implementing a 4-way set-associative cache. This cache design is with 64KB storage and 128-byte cache lines. It's referred to TRADITIONAL. The second cache configuration is ESC [9]. It is an alternative to implement the adaptive software cache line on Cell processor, so we compare it with our scheme. The last configuration is our adaptive software line size cache, which adopts a 64KB cache storage. We refer to this configuration as ADAPTIVE. The last two configurations adaptively choose a cache line size among 128B, 256B, 512B and 1024B during application execution.

Figure 10 illustrates the normalized speedup of different applications, and the baseline is the execution speed of TRADITIONAL. On the whole, our adaptive cache line strategy combined with the optimized cache structure performs better than the other two cache designs.

We first compare our adaptive cache line solution with the ESC. Obviously, our adaptive cache solution achieves noticeable performance improvements. It mainly results from the following factors:

First, the adaptive algorithm in ESC is applied to the parallel loop only, but it isn't sensitive to variation across iterations of the loop. Our adaptive approach could be applied to not only the parallel loop but also the iteration ranges, so it could choose the optimal cache line size more precisely.

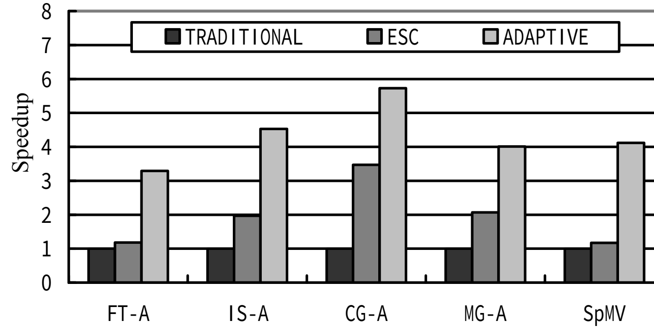

Fig. 10. Normalized speedup

Second, the algorithm in the ESC sometimes might choose a cache line size which is not the optimal. An example is given in Figure 11. There are five different cache line sizes, LS_0 to LS_4, with the size increasing. Suppose that their performance levels are 1, 3, 2, 4, 0, with each higher level corresponding to the better performance. Obviously, LS_3 is the optimal line size. The adaptive execution in ESC is as follows.

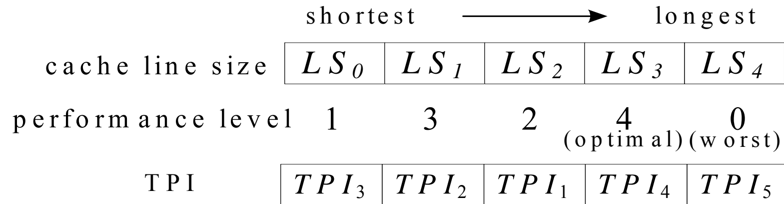

Fig. 11. An example of the adaptive algorithm in ESC

When the loop is invoked the first time, it's executed with the initial line size, LS_2. And the time per iteration (TPI) is TPI_1. When the loop is invoked the next time, it executed with a shorter line, LS_1. The corresponding TPI is TPI_2. TPI_1 must be greater than TPI_2 according to the performance level, so a shorter line LS_0 is chosen and the corresponding TPI_3 is measured in the next loop invocation. Obviously, TPI_3 is greater than TPI_2, so LS_1 is chosen to be the optimal. Nevertheless, LS_3 is the optimal line size according to the performance level. This results from the following two reasons.

- Though TPI_1 is greater than TPI_2 in the above algorithm, it's not necessary the case that a shorter line performs better for the loop, as shown in Figure 11.
- The adaptive strategy in ESC depends on many factors, for example, the initial cache lines size, choosing which line (a longer or a shorter line) in the second invocation of the loop.

Third, the loop has to be invoked many times to compare TPIs in the ESC strategy, which greatly degrades the performance. Additionally, a certain loop may be invoked once in some applications, so the adaptive algorithm in ESC doesn't work well in such cases. The loop in SpMV is invoked only once, and not surprisingly, the ESC has only a slight performance improvement compared with the traditional approach.

Then we compare the ADAPTIVE with the TRADITIONAL configuration. We achieve speedup factors of 3.29 for FT, 4.53 for IS, 5.73 for CG, 4.01 for MG and 4.12 for SpMV. The benchmarks which are sensitive to the cache line size and are dominated by irregular references benefit more from the ADAPTIVE configuration. CG is such an application, so it achieves a significant performance improvement. Though IS isn't sensitive to the cache line size, it is dominated by irregular memory references and thus it exposes a high miss rate. Not surprisingly, it obtains a significant speedup from our adaptive prefetching scheme.

5.3 Transferred Data Size

If the transferred data size of the cache design with 128B is the baseline, the transferred data size of cache line design with 128B, 256B, 512B and 1024B is illustrated in Figure 12.

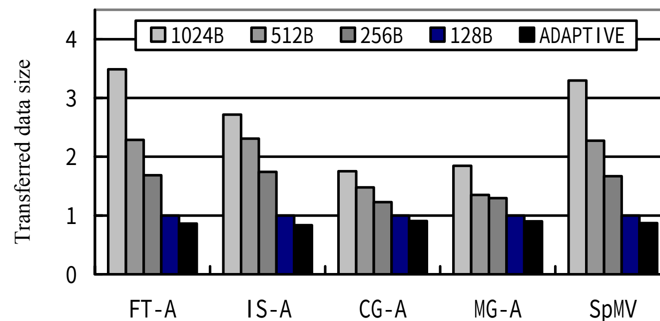

Fig. 12. Normalized transferred data size

Obviously, the transferred data of the ALSC is less than the other cache designs, especially compared with the 1024B line design. Because even though only a few bytes are needed, the whole 1024B line is transferred if the cache line is set to be 1024B. The ALSC outperforms designs with the fixed short cache line because the latter immediately transfers data when a miss occurs. Nevertheless, the ALSC first checks the long line *Tag Entry Array*. If it's a hit, there is no need to transfer data.

5.4 Scalability

Figure 13 presents the scalability of our approach. All the benchmarks, except IS, show good scalability from 1 to 8 threads, with speedup more than 6 on 8 SPEs. The main reason is that every thread executes in an exclusive SPE. IS doesn't scale up well because it contains some computations in either master execution or critical codes. Those computations are executed sequentially. Therefore, its speedup of 8 SPEs is only about 3.8.

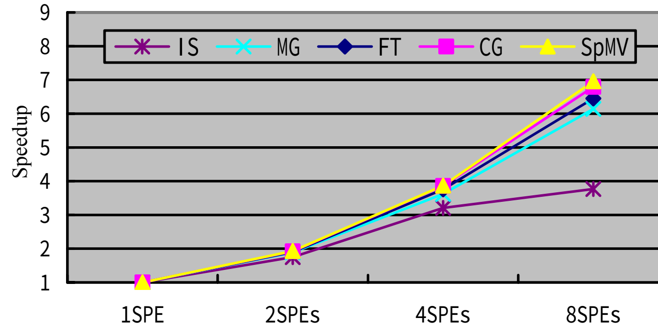

Fig. 13. Scalability of the adaptive approach on Cell processor

5.5 Storage Overhead

Table 1 depicts the extra storage overhead except the storage for *N* cache line. FAC and 4WC denotes the fully associative and 4-way set-associative cache, respectively.

To formally describe the extra storage overhead we assume as followed:

- The number of the short line is *N*, and the number of set in 4WC is *S* (*S*=*N*/4).
- S' is the number of sets in ESC, which is four times the closest power of 2 to *S*.
- The tags are 4-byte integers while dirty and valid bits are one byte.
- All the caches introduce fetch buffers. And every TE needs a line index. The FAC and the ESC both need a field to record the global address.

Table 1. Extra storage overhead (in bytes)

Heading level	Example	4WC	ESC	ALSC
TE	tag+line index =8	tag+line index =8	10N	tag+line index =8
TE Array	TE*N=8N	(4*TE)* S=8N	(4*TE)* S'=32S'	(4*TE)*3S/2=12N
LTE	V+D+tag =6	V+D=2	V+D+tag =6	(V+D)+(V+D)/2 =3
Cache line table	LTE*N= 6N	LTE*N =2N	LTE*N= 6N	LTE*N=3N
Total Size	14N	10N	32S'+6N	15N

Suppose that the whole cache size is 64K, the total storage overhead is listed in Table 2. The ALSC storage overhead is a little more than that of the FAC and 4WC and much less than that of the ESC.

Table 2. Extra storage overhead with a 64KB cache (in bytes)

Line size	Number of lines	FAC	4WC	ESC	ALSC
128B	512	7168	5120	19456	7680
512B	128	1792	1280	4864	1920
1KB	64	896	640	2432	960
4KB	16	224	160	608	240

6 Conclusions and Future Works

We present an algorithm which adaptively adjusts the software cache line for irregular reference on Cell processor. Moreover, a corresponding software cache design is proposed, which significantly improves the hit rate and decreases the reduction of data transfers. The evaluation results indicate that our strategy achieves the speedup factor from 3.29 to 5.73 compared with the traditional software cache approach. Additionally, the adaptive strategy shows good scalability.

Acknowledgments. The research is partially supported by the Hi-Tech Research and Development Program (863) of China under Grant No. 2008AA01Z109, the Key Project of Chinese Ministry of Education under Grant No. 108008, and by the National Key Technology R&D Program under Grant No. 2006BAK11B00.

References

1. Eichenberger, A.J., O'Brien, J.K., O'Brien, K.M., Wu, P., et al.: Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture. J. IBM Sys. 45, 59--84 (2006)
2. Balart, J., Gonzalez, M., Martorell, X., Ayguade, E., et al.: A Novel Asynchronous Software Cache Implementation for the Cell BE Processor. In: Adve, V.S., Garzarán, M.J., Petersen, P. (eds.) LCPC 2007. LNCS, vol. 5234. pp. 125--140. Springer, Heidelberg (2007)
3. Lee J., Seo, S., Kim, C., Kim, J., et al.: COMIC: A Coherent Shared Memory Interface for Cell BE. In: 17th international conference on Parallel architectures and compilation techniques, pp. 303--314. ACM Press, New York (2008)
4. Marc, G., Nikola, V., Xavier, M., Eduard, A., et al.: Hybrid access-specific software cache techniques for the cell be architecture. In: 17th international conference on Parallel architectures and compilation techniques, pp. 292--302. ACM, New York (2008)
5. Vujić, N., Gonzàlez, M., Martorell, X., Ayguadé, E.: Automatic Pre-Fetch and Modulo Scheduling Transformations for the Cell BE Architecture. In: Amaral, J.N. (ed.) LCPC 2008. LNCS, vol. 5335, pp. 31--46. Springer, Heidelberg (2008)
6. Chen, T., Lin, H.B., Zhang, T.: Orchestrating data transfer for the cell b.e. processor. In: 22nd annual international conference on Supercomputing, pp. 289--298. ACM Press, New York (2008)
7. Dubnicki, C., LeBlanc, T.: Adjustable block size coherent caches. In: 19th annual international symposium on Computer architecture, pp. 170--180. ACM Press, New York (1992)

8. Veidenbaum, A.V., Tang, W., Gupta, R., Nicolau, A., et al.: Adapting Cache Line Size to Application Behavior. In: 13th international conference on Supercomputing, pp. 145--154. ACM Press, New York (1999)
9. Seo, S., Lee, J., Sura, Z.: Design and Implementation of Software-Managed Caches for Multicores with Local Memory. In: 15th International Symposium on High-Performance Computer Architecture, pp. 55--66. IEEE Press, New York (2009)
10. Altevogt, P, Boettiger, H, Kiss, T, Krnjajic, Z: IBM BladeCenter QS21 Hardware Performance. IBM Technical White Paper WP101245 (2008)
11. Bailey, D., Harris, T., Saphir, W., Wijngaart, R.V.D., et al.: The NAS Parallel Benchmarks 2.0. NAS Technical Report NAS-95-020, NASA Ames Research Center, Moffett Field, CA (1995)
12. University of Florida Sparse Matrix Collection, http://www.cise.ufl.edu/research/sparse/matrices/
13. Gelado, I., Kelm, J.H., Ryoo, S., Lumetta, S.S., et al.: CUBA: An Architecture for Efficient CPU/Coprocessor Data Communication. In: 22nd annual international conference on Supercomputing, pp. 299--308. ACM Press, New York (2008)
14. Chen, T., Zhang, T., Sura, Z, Tallada M.G.: Prefetching irregular references for software cache on cell. In: 6th annual international symposium on Code Generation and Optimization, pp. 155--164. ACM Press, New York (2008)
15. Cao, Q, Zhao, C.C, Zhang, Y.X, Chen, J.X., et al. Adaptive Tuning of Sparse Matrix-Vector Multiplication on Cell Architecture. In: 2nd International Conference on Computer Engineering and Technology, pp. 292--296. IEEE Press, New York (2010)
16. Schneider, S., Yeom, J.S., Rose, B., Linford, J.C., et al.: A Comparison of Programming Models for Multiprocessors with Explicitly Managed Memory Hierarchies. In: 14th ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 131--140. ACM, New York (2009)
17. Crawford, C.H., Henning, P., Kistler, M., Wright, C.: Accelerating Computing with the Cell Broadband Engine Processor. In: 5th ACM Conference on Computing frontiers, pp. 3--12. ACM Press, New York (2008)
18. WANG, Z., O'BOYLE, M.: Mapping Parallelism to Multicores: A Machine Learning Based Approach. In: 14th ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 75--84. ACM Press, New York (2009)
19. JIMENEZ, V.J., VILANOVA, L., GELADO, I., GIL, M., et al.: Predictive Runtime Code Scheduling for Heterogeneous Architectures. In: Seznec, A., Emer, J.S., O'Boyle, M.F.P., Martonosi, M., Ungerer, T. (eds.) HiPEAC 2009. LNCS, vol. 5409, pp. 19--33. Springer, Heidelberg (2009)
20. WANG, P., COLLINS, J.D., CHINYA, G., JIANG, H., et al.: Architecture and Programming Environment for a Heterogeneous Multicore Multithreaded System. In: conference on Programming language design and implementation, pp. 156--166. ACM, New York (2007)
21. REN, M., PARK, J., HOUSTON, M., AIKEN, A., et al.: A tuning framework for software-managed memory hierarchies. In: 17th international conference on Parallel architectures and compilation techniques, pp. 280--291. ACM, New York (2008)