A Novel Asynchronous Software Cache Implementation for the Cell-BE Processor

Jairo Balart [1,], Marc Gonzalez[1], Xavier Martorell[1], Eduard Ayguade[1],
Zehra Sura[2], Tong Chen[2], Tao Zhang[2], Kevin O'brien[2], Kathryn O'brien[2]
[1] Barcelona Supercomputing Center (BSC), Technical University of Catalunya (UPC)
[2] IBM TJ Watson Reserach Center
{jairo.balart, marc.gonzalez, xavier.martorell, eduard.ayguade}@bsc.es
{zsura, chentong, taozhang, caomhin, kmob}@us.ibm.com

Abstract. This paper describes the implementation of a runtime library for asynchronous communication in the Cell BE processor. The runtime library implementation provides with several services that allow the compiler to generate code, maximizing the chances for overlapping communication and computation. The library implementation is organized as a Software Cache and the main services correspond to mechanisms for data look up, data placement and replacement, data write back, memory synchronization and address translation. The implementation guarantees that all those services can be totally uncoupled when dealing with memory references. Therefore this provides opportunities to the compiler to organize the generated code in order to overlap as much as possible computation with communication. The paper also describes the necessary mechanism to overlap the communication related to write back operations with actual computation. The paper includes the description of the compiler basic algorithms and optimizations for code generation. The system is evaluated measuring bandwidth and global updates ratios, with two benchmarks from the HPCC benchmark suite: Stream and Random Access.

1 Introduction

In a system where software is responsible for data transfers between certain memory regions, it is desirable to assist the programmer by automatically managing some or all of these transfers in system software. For asynchronous data transfers, it is possible to overlap the memory access time with computation time by initiating the data transfer request in advance, i.e. before computation reaches the point when it needs to use the data requested. The placement of such memory access calls in the code is important since it can change the amount of overlap between data communication and computation, and thus affect the overall performance of the application. In this work, we target a Cell BE system to explore our approach to automatically managing asynchronous data transfers. Our technique implements a software caching mechanism that works differently from traditional hardware caching mechanisms, with the goal being to facilitate the decoupling of the multiple steps involved in a memory access (including address calculation, cache placement, and data transfer) as well as the actual use of the data. Software caching is not a novel proposal since it has been extensively used in specific domains, like embedded processors [4][5][6].

Our target platform, the Cell BE architecture [2], has nine processing cores on a single chip: one 64-bit Power Processing Element (PPE core) and eight Synergistic Processing Elements (SPE cores) that use 18-bit addresses to access a 256K Local Store. The PPE core accesses system memory using a traditional cache-coherent memory hierarchy. The SPE cores access system memory via a DMA engine connected to a high bandwidth bus, relying on software to explicitly initiate DMA requests for data transfer. The DMA engine can support up to 16 concurrent requests of up to 16K, and bandwidth between the DMA engine and the bus is 8 bytes per cycle in each direction. Each SPE uses its Local Store to buffer data transferred to and from system memory. The bus interface allows issuing asynchronous DMA transfer requests, and provides synchronization calls to check or wait for previously issued DMA requests to complete.

The rest of this paper is organized as follows. In Section 2, we motivate the use of a novel software cache organization for automatically managing asynchronous data transfers. In Section 3, we detail the structure and implementation of this software cache mechanism. In Section 4, we describe the compiler support needed to enable effective use of the runtime software cache services. In Section 5, we evaluate basic performance of our software caching technique using the Stream and Random Access benchmarks from the HPCC benchmark suite. In Section 6 we present some concluding remarks.

2 Motivation

The particular memory model in the Cell BE processor poses several difficulties for generating efficient code for the SPEs. The fact that each SPE owns a proper address space within the Local Storage, plus the limitation on its size, 256Kb shared by data and code, causes the performance being very sensible on how the communications are scheduled along the computation. Overlapping computation with communication becomes a crucial optimization.

When the access patterns in the computation can be easily predicted, static buffers can be introduced by the compiler, double-buffering techniques can be exploited at runtime, usually involving loop tiling techniques [1][7]. In the presence of pointer–based accesses, the compiler is no longer able to transform the code in order to overlap communication and computation. Usually, this kind of access is treated by a runtime library implementing a software cache [1]. The resulting code is difficult to be efficient as every memory reference in the code has to be monitored in order to ensure that the data is present in the Local Store, before any access to it takes place. This is usually implemented through the instrumentation of every memory reference with a runtime call responsible for the monitoring, where many checks have to occur. A general protocol to treat a single memory reference could include the following steps:

1. Check if the data is already present in local storage
2. In case not present, decide where to place it and ...
3. If out of space, decide what to send out from Local Storage
4. If necessary, perform DMA operations
5. If necessary synchronize with DMA

6. Translate from virtual address space to Local Storage address space
7. Perform memory access

Under that execution model, the chances for overlapping computation with communication are quite limited. Besides, the memory references instrumentation incurs in unacceptable overheads. The motivation of this paper is to describe what should be the main features within a software cache implementation that maximizes the chances for overlapping computation and communication, and minimizes overhead related to the memory references instrumentation.

Following the previous scheme, the overlap of communication with computation it can only be implemented by uncoupling the DMA synchronization (step 5) from the previous runtime checks (steps 1 to 4). If the runtime were to support such uncoupling, then it could be possible to reorganize the code, placing some amount of computation between step 4 and step 5 of every reference. Notice that this optimization is conditioned by the computation, in the sense that it might happen that data dependences do not allow the code reorganization. Although that, decoupling steps 4 and 5 still can offer some important benefits. It is also possible to mix the 1, 2, 3, and 4 steps of two or more memory references and group all the DMA synchronization in one single step. That would translate on some overlapping between cache management code and data communication, reducing the overhead impact. But such overlapping needs of some specific features within the implementation of steps 1, 2 and 3. It is necessary that no conflict appears between steps 1, 2 and 3 of every memory reference treated before the synchronization step. That is, the placement and replacement mechanisms must not assign the same cache line for two different memory references. This is one point of motivation of the work in this paper: the implementation of a software cache that enhances the chances for the overlapping of computation (whether it is cache control code or application code) and data communication, by uncoupling steps 4 and 5 and reducing the cache conflicts to capacity conflicts.

Because of the limited size of the Local Storage, it is necessary to provide the cache implementation with a write back mechanism to send out data to main memory. The write back mechanism involves a DMA operation moving data from the Local Storage to main memory, and requires the SPE to synchronize with the DMA engine before the flushed cache line is being reused. Deciding the moment to initiate the DMA operation becomes an important issue to increase performance. If the write back mechanism is invoked just when a modified cache line has to be replaced, then the SPE is going to be blocked until the associated DMA operation ends. The implementation described in this paper introduces two mechanisms to minimize as much as possible the number of lost cycles waiting for a DMA operation to complete (related to a flush operation). First, a mechanism to foresee future flush operations, based on information about what cache lines are referenced by the code, and detecting the precise moment where a cache line becomes unreferenced. Second, a specific replacement policy that delays as much as possible the next assignment for a flushed cache line, thus giving time to the flush operation to complete, and avoid lost cycles dedicated to synchronization at the moment of reuse.

3 Software Cache Implementation

The software cache is described according to the cache parameters, cache structures and the main services: look up, placement/replacement policies, write back, communication/synchronization and address translation.

3.1 Cache Parameters

The main cache parameters are the following: capacity, size of cache line and associativity level. For the rest of this document C stands for capacity, L stands for the cache line size, S stands for the level of associativity and N=C/L stands for the number of cache lines.

3.2 Cache Structures

The cache is composed mainly by three structures. Two list-based structures, where the cache lines can be placed depending on their state and attributes value. These are the *Directory* and the *Unused Cache Lines* lists. A third structure under a table shape, basically used for look up and translation operations: the *Look Up and Translating* table.

- The *Directory* list holds all the cache lines that are resident in the cache.
- The *Unused Cache Lines* list holds all cache lines that are no longer in use by the computation. The notion of being under use is defined by the existence of any memory reference in the computation that references the in-use cache line. The cache implementation is able to keep track of what cache lines are being referenced, and what are not.
- The *Look Up and Translating* table holds information for optimizing the look up mechanism and for implementing the translation from the virtual address space to the Local Storage address space.

3.2.1 Directory

The *Directory* is composed of S lists. Cache lines in the *Directory* are stored in a double –linked list form. There is no limitation on the number of cache lines that can be placed in any of the S lists. That makes the cache implementation a full-associative cache. Basically the S lists are used as a hash structure to speed up the look up process.

3.2.2 Unused Cache Lines List

This list holds the cache lines that were previously used by the computation, but that at a given moment they were no longer in use. The main role for this structure is related to the placement/replacement policy. Cache lines placed in this list become the immediate candidates for replacement, thus placement for other incoming cache lines required by the computation. The cache lines are stored in a double-linked list form.

3.2.3 Look Up and Translating Table

This structure is organized as a table, where each row is assigned to a particular memory reference in the computation. A row contains three values used for the look up and translation mechanisms: the base address of the cache line in the Local Storage address space, the base address of the correspondent cache line in the virtual address space and a pointer to the structure representing the cache being used by the memory reference.

3.2.4 Cache Line State and Attributes

For every cache line, the implementation records information about the cache line state and other attributes, necessary to control the placement/replacement, write back, look up, and translation mechanisms.

The state of a cache line is determined by the fact any memory reference in the computation referencing the cache line. The implementation keeps track of what cache lines are under use, by maintaining a reference counter associated to each cache line. The reference counter is incremented/decremented appropriately during the *Look Up* mechanism. Therefore, the state of a cache line can take two different values: USED or UNUSED. Besides the cache line state, there are other attributes:

- CLEAN: the cache line has been only used for READ memory operations. The data stored in the cache line has not been modified.
- DIRTY: the cache line has been used for WRITE and/or READ memory operations. The data stored in the cache line has been modified.
- FLUSHED: the cache line has already been flushed to main memory.
- LOCKED: the cache line is excluded from the replacement policy, which means that a cache line holding this attribute can not be replaced.
- PARTITIONED: the data transfer from/to main memory involves a different amount of data than the actual cache line size. The total number of bytes to be transferred is obtained by dividing the cache line size by a factor of 32.

The implementation also records the mapping between the cache line in the Local Storage, and its associated cache line in virtual memory.

3.3 Look Up

The *Look Up* mechanism is divided in two different phases. First phase takes place within the computation code, second phase occurs inside the cache runtime system. For the first phase of look up, it is necessary some coordination with the compiler support. For each memory reference the implementation keeps track about the base address for the cache line being accessed. This information is stored in the *Look Up and Translating* table. Each time a new instance of a memory reference occurs, the implementation checks if the referenced cache line has changed. If this happens, then the second phase for the look up is invoked. Detecting if the cache line has changed is as simple as performing an AND operation between the memory address generated in the memory reference, and a particular mask value (in C syntax: ~(L-1)), plus a comparison with the value in the *Look Up and Translating* table. It is under the compiler responsibility to assign an entry in the *Look Up and Translating* table for each memory reference in the code. Section 4.2 is giving the detailed description on how this is implemented.

The second phase of the *Look Up* mechanism accesses the cache *Directory* looking for the new required cache line. Only one of the S lists has to be selected to perform the search. This is done through a hash function applied to the base address of the cache line. The implementation ignores the offset bits, and takes all other most significant bits. Then applies an S-modulo operation and determines one of the S lists. The *Look Up* continues with the list traversal, and if the cache line is found, a hit is reported. In case not, the placement/replacement mechanisms are invoked, and the necessary DMA operations are programmed.

During the *Look Up* process, the reference counters for the two cache lines that are going to be involved are incremented/decremented. For the cache line that is no longer referenced by the memory reference, the counter is decremented. For the new referenced cache line, the counter is incremented, no matter the *Look Up* ended with a hit or miss.

At the end of the *Look Up* process the *Look Up and Translating* table is updated. The row assigned to the memory reference the *Look Up* operation was treating is appropriately filled: base address of the cache line in the Local Storage, base address of the cache line in virtual memory and a pointer to the structure representing the cache line.

3.4 Write Back

The Write Back mechanism only applies for modified cache lines, that is, those lines that hold the DIRTY attribute. The write back is activated when the reference counter of a modified cache line reaches the zero value. This event is interpreted by the implementation as a hint of future possible uses of the cache line. Particularly, the event is interpreted as if the cache line is not going to be referenced by the computation up to its completion. Therefore, this point becomes a good opportunity to go in advance to the needs of the computation and program the flush of the cache line, under an asynchronous scheme. Notice that this is giving, but not ensuring, time to the implementation to overlap communication and computation. Of course, it is necessary at some point to synchronize with the DMA operation. In order to do so, the implementation records the TAG used in the DMA operation, and delays the synchronization until the next use of the cache line, when ever the replacement policy determines the next reuse to happen.

3.5 Placement / Replacement

The Placement/Replacement mechanisms are executed during the second phase of *Look Up*. The replacement policy relies on the reference counter and the *Unused Cache Lines* list. When the cache line reference counter equals zero, the cache line is placed on the *Unused Cache Lines* list as the LAST of the list, and as stated in previous section, if the line was modified, a flush operation is immediately programmed. Notice that the cache line is not extracted from the *Directory*.

The *Unused Cache Lines* list contains the cache lines candidates for replacement actions. When new data has to be brought in the cache, a cache line has to be selected. If the *Unused Cache Lines* list is not empty, the implementation selects the FIRST in the list. If the line is holding the FLUSHING attribute, the tag that was recorded during write back execution is used to synchronize with the DMA engine. After that, a DMA operation is programmed under an asynchronous scheme to bring in the data, relying on the compiler for placing in the computation code the necessary synchronization statement. Notice that selecting the FIRST element in the list, while unused cache lines are placed as LAST, is what separates as much as possible the DMA operation associated to a flushed cache line, and its next reuse. Hence, delaying as much as possible the execution of the necessary synchronization with the DMA engine and avoiding unnecessary stalls in the SPE.

If the *Unused cache Lines* list is empty, then the replacement policy traverses the *Directory* from set 0 to S-1, and selects the line that first entered in the cache. This is implemented through the assignment of a number that is incremented each time a cache line is brought in. The minimum number within all resident cache lines determines the cache line to be replaced. If the replaced line was modified, the line is flushed to main memory under a synchronous scheme. After that, the data is brought in through an asynchronous DMA operation, and relying on the compiler to introduce the necessary synchronization statement. Notice that an appropriate relation between the number of cache lines and the number of memory references might perfectly avoid this kind of replacement, since it can be ensured that the list of unused cache lines is never going to be empty (see section 4.6).

Initially, all cache lines are stored in both the *Directory* and the *Unused Cache Lines* list, with the counter reference equaling zero.

3.6 Communications and Synchronization

The implementation distinguishes between DMA transfers related to write back operations and DMA transfers responsible for bringing data into the cache. For the former case, a set of 15 tags are reserved, for the latter another different 15 tags. For both cases tags are assigned in a round robin fashion, which means after 15 DMA operations tags start being reused.

All DMA operations assigned to the same tag, are executed always one after the other. This is achieved through the use of fenced DMA operations that forbids the memory flow controller to reorder any DMA operations associated to the same tag. This becomes necessary for treating the following situation: suppose a modified cache line is no longer in use, so it is flushed to main memory, and placed in the *Unused Cache Lines* list. Then the code being executed references again the data in the cache line, and since it was not extracted from the *Directory*, no miss is produced, but it is necessary to extract the cache line from the *Unused Cache Lines* list. The cache line might or might not be modified, but at some point the cache line will be no longer in use. In the case the cache line was modified it will be flushed again. It is mandatory for memory consistency that the two flush operations get never reordered. To ensure that, the implementation reuses the same tag for both flush operations, and introduces a "memory fence" between them.

All DMA operations are always programmed under an asynchronous scheme, unless those associated to a replacement that found empty the *Unused Cache Lines* list. Those related to flush operations, synchronize at the next reuse of the flushed cache line. Those related to bring data into the cache get synchronized by specific statements introduced by the compiler. It is important to mention that this is what allows the compiler to try to maximize the overlap between communication and communication. Section 4 describes the necessary compiler support to achieve the communication/computation overlapping.

3.7 Address Translation

To perform the translation from the virtual address space to the Local Storage address space the data in the *Look Up and Translating* table is enough. Each memory reference has been assigned with a row in the *Look Up and Translating* table. In that row, the base address for the cache line in the Local Storage can be obtained. Translation is as simple as computing the offset of the access and add the offset to the base address of the cache line in the Local Storage. The offset computation can be done through an AND operation between the generated address in the memory reference and a bit mask according to the size of the cache line (e.g: ~(L-1)).

4 Compiler Code Generation

This section describes the compiler support and code generation for transforming programs to SPE executables relying on the software cache described in the previous section. In this paper we describe the compiler support that is required to target the execution of loops.

4.1 Basic runtime services

This section describes the main runtime available services which the compiler should target while generating code.

- _LOOKUP: runtime service performing the phase 1 in the *Look Up* mechanism.
- _MMAP: runtime service executing phase 2 in the *Look Up* mechanism. In case a miss is produced, then the placement/replacement mechanisms are executed, the reference counters are incremented/decremented, and the all necessary DMA operations are performed asynchronously. In case the replacement algorithm indicates the use of a previously flushed cache line, synchronization with the DMA engine occurs.
- _MEM_BARRIER: runtime service that forces the synchronization with the DMA engine. It is a blocking runtime service.
- _LD, _ST: runtime services responsible for the address translation between the virtual address space and the Local Storage address space. Include arithmetic pointer operations such as the computation of the offset in the access to the cache line base

address in virtual memory, and the computation of the actual Local Storage address by adding the offset to the base address of the cache line in the Local Storage.

4.2 Code Generation

This section describes the basic algorithms and optimizations related to code generation.

4.2.1 Assign identifiers to memory references

The first step for the compiler is to assign a numerical identifier to each different memory reference in the code. This identifier is going to be used at runtime to link each memory reference to the runtime structure supporting the *Look Up* (phase one), and the translating mechanisms. The runtime structure corresponds to one entry in the *Look Up and Translating* table.

```
for (i=0;i<NUM_ITERS;i++) {
  v1[i] = v2[i];
  v3[v1[i]]++;
}
```

Figure 1: Example of C code for code generation.

For the example shown in Figure 1, three different memory references can be distinguished: *v1[]* , *v2[]* and *v3[]* . The compiler would fro example associate identifiers 0, 1, 2 to memory references to *v1[]* , *v2[]* and *v3[]* respectively.

```
for (i=0;i<NUM_ITERS;i++) {
   if (_LOOKUP(0, ,&v2[i],...)) {
     _MMAP(0,&v2[i],...);
     _MEM_BARRIER(0);
  }
  if (_LOOKUP(1, ,&v1[i],...)) {
    _MMAP(1,&v1[i],...);
    _MEM_BARRIER(1);
  }
  _LD(0,&v2[i],_int_tmp00);
  _ST(1,&v1[i],_int_tmp00);
  if (_LOOKUP(2, ,&v3[_int_tmp00],...)) {
    _MMAP(2,&v3[_int_tmp00],...);
    _MEM_BARRIER(2);
  }
  _LD(2,&v3[_int_tmp00],_int_tmp01);
  _int_tmp01++;
  _ST(2,&v3[_int_tmp00],_int_tmp01);
}
```

Figure 2: Initial code transformation.

4.2.2 Basic Code Generation

For every memory reference, the compiler has to inject code to check if the data needed by the computation is in the Local Storage. The compiler injects a _LOOKUP operation for every memory reference, and a conditional statement depending on the output of the _LOOKUP operation. Figure 2 shows the transformed code for the example in figure 1. All _MMAP operations are controlled by a _LOOKUP operation, relying on the runtime structures pointed out by the assigned identifier according to what has been described in the previous section. Right at the end on the conditional branch, the compiler injects a _MEM_BARRIER operation that enforces the synchronization with the DMA engine.

```
_lb_01 = 0; _ub_01 = NELEM;
_work_01 = (_lb_01 < _ub_01);
while (_work_01) {
  _start_01 = _lb_01;
  _LOOKUP(0, ,&v2[i],...,_lookpu_01);
  if (_lookup_01) _MMAP(0, &v2[_start_01], ..., LOCK);
  _LOOKUP(1, ,&v1[i],...,_lookup_01);
  if (_lookup_01) _MMAP(1, &v1[_start_01], ..., LOCK);
  _next_iters_01 = LS_PAGE_SIZE;
  _NEXT_MISS(0, &v2[_start_01], float, sizeof(float), _next_iters_01);
  _NEXT_MISS(1, &v1[_start_01], float, sizeof(float), _next_iters_01);
  _end_01 = _start_01 + _next_iters_01;
  if (_end_01>_ub_01) _end_01 = _ub_01;
  _lb_01 = _end_01;
  _work_01 = (_lb_01 < _ub_01);
  _MEM_BARRIER();
  for (int i = _start_01; i < _end_01; i=i+1) {
     _LD(0,&v2[i],_int_tmp00);
     _ST(1,&v1[i],_int_tmp00);
     if (_LOOKUP(2, &v3[_int_tmp00],...)) {
        _MMAP(2,&v3[_int_tmp00],...);
        _MEM_BARRIER(2);
     }
     _LD(2,&v3[_int_tmp00],_int_tmp01);
     _int_tmp01++;
     _ST(2,&v3[_int_tmp00],_int_tmp01);
    }
}
```

Figure 3: Code transformation for stride accesses.

This preliminary version of the transformed code does not allow any overlap between computation and communication. It contains unnecessary conditional statements that for sure are not going to be optimal. Besides, it does not take into account the different type of accesses in the code, distinguishing between strided accesses and pointer-based accesses. But before describing any optimization technique, it is necessary to outline what are the limitations that condition the compiler transformations. Since the main target of the compiler is to enhance the overlapping of computation (whether cache control code or original computation in the code) it is reasonable to try to reorganize the preliminary code in order to group _MMAP operations, making them to be executed at runtime right one after the other. Notice that such grouping makes all the communication performed within a _MMAP operation, be overlapped with the execution of the following _MMAP operations. In the example, the if statements corresponding to the accesses to *v1[i]* and *v2[i]* could be joined. One if statement should include the two _MMAP operations, and only one _MEM_BARRIER. Generally, the compiler is only limited by the fact that grouping the _MMAP operations must be done taking to account the possibility of conflicts within the grouped _MMAPs. A conflict may appear along the execution of several _MMAP operations if two of them require the same cache line to bring data in the Local Storage. A conflict is not acceptable to appear before the data of the conflicting _MMAP operations has been accessed. That is, between the execution of a particular _MMAP operation and the _LD/_ST operations with the same identifier, it is not acceptable to place a number of _MMAP operations that can cause a conflict. Since the cache implementation follows a full-associative scheme, conflicts may only appear as capacity conflicts. This determines the limits on the grouping: the compiler can not group _MMAP operations if doing so is causing that between a _LD/_ST operation and the corresponding _MMAP operation (indicated by the identifier associated to _MMAP and _LD/_ST operations) N _MMAP operations are executed, where N stands for the number of cache lines. Formally, we define the distance of a _MMAP operation as *the maximum number of _MMAP operations between the _MMAP and the _LD/_ST operations with the same identifier.* The compiler is now free of reorganizing the code, grouping _MMAP operations, as long as it keeps every _MMAP distance in the range of [0..N].

4.3 Optimization for Strided Accesses

Strided accesses offer the possibility of reducing the number of the _MMAP operations that need to be performed during the execution of the loop. The basis for such optimization is that the runtime can be provided with a service that computes how many accesses are going to be produced along a cache line, given the starting address and the stride. This information can be used to partition the iteration space in different chunks, defining the initial iteration of each chunk, a change of cache line (actually a miss) in a strided memory access.

Figure 3 shows the compiler code for the example code in Figure 1. Notice that the original loop, has been embedded in an outer *while* loop. The *while* loop iterates along the chunks of iterations, and the inner loop iterates along the actual iteration space. The use of the runtime service _NEXT_MISS computes the number of iterations that can be performed without having a miss on that access, given an initial address and a stride. For every strided access, the _NEXT_MISS service is invoked, and the minimum of these values defines the number of iterations for the next chunk. In the example, the two stride accesses are treated with two _MMAP operations that are going to overlap the communication of the first one with the cache control code of the second one.

Notice the attribute LOCK provided to the runtime system _MMAP, that ensures that the mapped cache line is going to be excluded from the replacement policies. This causes the runtime to treat the memory references in the inner loop, with a different distance boundary, since now the compiler has to assume 2 less available cache lines in the overall cache capacity. A _MEM_BARRIER is placed right before the inner loop execution ensuring that the data is resent before the next chunk of iterations is executed. This synchronization only involves incoming data to the Local Storage. Write back operations executed along the _MMAP runtime service corresponding to the *v1[i]* access operation, are synchronized whenever the cache lines associated to this access are being reused.

4.4 Optimization for Non-strided Accesses

Non-strided accesses become an important source of overhead, since they do not usually spatial locality. Therefore, overlapping computation and communication for this type of access should be highly desirable. Figure 4 shows the compiler transformation for this kind of access, corresponding to the *v3[v1[i]]* access in the example in Figure 1. Only the innermost loop where the non-stride access is placed is showed. The loop has been unrolled 2 times, offering the possibility of grouping the 2 _MMAP operations associated to that access. The 2 factor has been only used as example, since the limit on the unrolling factor is going to be determined by the number of cache lines, minus 2 (two cache lines have been locked for *v1* and *v2* accesses), as the distance boundary has to be preserved for all _MMAP operations. Notice that the compiler has to assign different identifiers for both accesses to *v3* vector, since they define two different memory references.

```
for (int i = _start_01; i < _end_01; i=i+2) {
  _LD(0,&v2[i],_int_tmp00);
  _ST(1,&v1[i],_int_tmp00);
  _LD(0,&v2[i+1],_int_tmp02);
  _ST(1,&v1[i+1],_int_tmp02);
  _LOOKUP(2, &v3[_int_tmp00],..., _lookup_01)
  _LOOKUP(2, &v3[_int_tmp02],..., _lookup_01)
  if (_look_up_01) {
    _MMAP(2,&v3[_int_tmp00],...);
    _MMAP(3,&v3[_int_tmp02],...);
    _MEM_BARRIER(2);
  }
  _LD(2,&v3[_int_tmp00],_int_tmp01);
  _int_tmp01++;
  _ST(2,&v3[_int_tmp00],_int_tmp01);

  _LD(3,&v3[_int_tmp02],_int_tmp03);
  _int_tmp03++;
  _ST(3,&v3[_int_tmp02],_int_tmp03);
 }
```

Figure 4: Code transformation for non-stride accesses.

4.5 Setting the cache line size

Depending on the number of memory references detected in the code, the cache line size has to be adapted to avoid unnecessary capacity conflicts within the execution of a loop iteration. If the number of references exceeds the number of cache lines, then conflicts are quite probable to appear. Therefore, the compiler has to select a cache line size that ensures that the number of available cache lines is greater or equal that the number of memory references.

5 Evaluation

The software cache implementation has been tested with two benchmarks from the HPCC benchmark suite [3]: Stream and Random Access. The Stream benchmark measures bandwidth ratios. It is composed by four synthetic kernels that measure sustainable memory bandwidth (in GB/s) and the corresponding computation rate for simple vector codes. The Random Access benchmark is composed by one kernel that operates on a single vector data type. The benchmark computes Global Updates per Second (GUPS). GUPS are calculated by identifying the number of memory locations that can be randomly updated in one second, divided by 1 billion (1e9). The term "randomly" means that there is little relationship between one address to be updated and the next, except that they occur in the space of 1/2 the total system memory.

All the measures were taken in a Cell BE-based blade machine with two Cell Broadband Engine processors at 3.2 GHz (SMT enabled), with 1 GB XDR RAM (512 MB each processor), running Linux Fedora Core 6 (Linux Kernel 2.6.20-CBE).

The software cache implementation was configured with the following cache parameters: 64Kb of capacity, 1024 sets and a varying cache line size ranging from 128 bytes up to 4096 bytes.

5.1 Stream benchmark

Figure 5 shows the comparison between three different implementations, differing in the way the communications are managed. The Synchronous version forces a DMA synchronization after every DMA operation is programmed. This version corresponds to an implementation that would not allow for any overlapping between computation and communication. The Synchronous Flush version, allows for having asynchronous data communication from main memory to the Local Storage, but implements the flush operations (transfers from Local Storage to main memory) under a synchronous scheme. This version is not using the reference counter for cache lines, as a hint for determining the moment where a cache line has to be flushed before any reuse of it is required. Finally, the Asynchronous version, implements the software cache described in this paper, trying to maximize the overlapping of computation and communication.

For every version, the performance of each kernel (Copy, Scale, Add and Triad) is shown varying the size of the cache line (128, 256, 512, 1024, 2048 and 4096 bytes, from left to right). The results correspond to the obtained performance while executing with 8 SPEs. For brevity, the results executing with 1, 2 and 4 SPEs have been omitted as they were showing a very similar behavior. Clearly, and as it could be expected, every version significantly improves as long as the cache line size is increased. The comparison of the three versions allows for measuring the capabilities of the software cache implementation to overlap computation and communication. The results for the Synchronous version are taken as a baseline to be improved by the two other versions. The performance for the 128 bytes executions show how the different kernels behave while being dominated by the DMA operations.

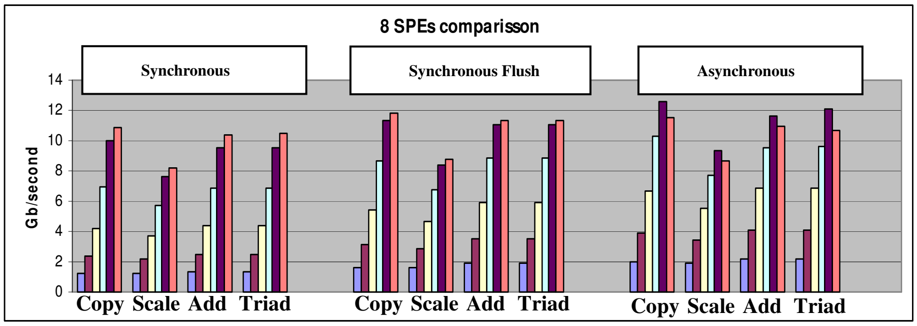

Figure 5: Code transformation for non-stride accesses.

The Synchronous version reaches 1.26 Gb/sec in average for the 4 kernels, the Synchronous Flush version reaches 1.75 Gb/sec, and finally the Asynchronous version reaches 2.10 Gb/sec. This corresponds to a speed up about 1.66. Similar behavior is observed when the cache line is increased from 128 up to 2048, reaching the best performance with a 2048 cache line size: 9.17 Gb/sec for Synchronous, 10.46 for Synchronous Flush and 11.38 for Asynchronous. This corresponds to a speed up about 1.24. Notice that when the cache line size is 4096, the increment of performance is not sustained. For the moment, it is not clear the reason of that behavior, so this needs more study.

5.2 Random Access benchmark

The Random Access benchmark is used for evaluate the overlapping of computation and communication when the parallel code includes pointer-based memory accesses. Four different versions of the benchmark have been evaluated, depending on the unroll factor in the loop computation. Figure 6 shows the core of the computation. The unroll factor determines how many DMA transfers can be overlapped for the memory references to variable *Table*, according to the transformation described in section 4.5. For a 2 unroll factor, 2 _MMAP operations can be executed one immediate after the other. An unroll factor of 4 allows for overlapping 4 _MMAP operations, a factor of 8 allows for overlapping 8 _MMAP operations.

```
for (i=0; i<NUPDATE/128; i++) {
  for (j=0; j<128; j++) {
     ran[j] = (ran[j] << 1) ^ ((s64Int) ran[j] < 0 ? POLY : 0);
     Table[ran[j] & (TableSize-1)] ^= ran[j];
   }
}
```

Figure 6: Source code for Random Access benchmark.

Figure 7 shows the results executing with 1 and 8 SPEs. The cache line size has been set to 4096, but the access to the *Table* variable it is performed with he _PARTITIONED_ attribute, which makes every DMA transfer just involve 4096/32 = 128 bytes of data. This shows the ability of the software cache implementation to deal with both strided accesses and non strided accesses. The base line measurement corresponds to the version with no loop unrolling. The Y axis measures GUPS (Giga Updates per Second). The 1-SPE version significantly improves while the unrolling factor is increased. Improvements are about 30%, 56% and 73% with 2, 4 and 8 unroll factors respectively. Similar improvements are observed in the 8-SPE version.

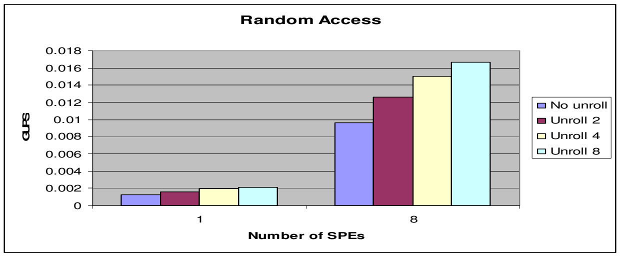

Figure 7: Performance for Random Access.

Although the difference in performance between the non-unrolled and the 8-urolled versions, we have detected a limiting factor due to the relation between the execution time for the _MMAP runtime service, and the DMA time for small data transfers (e.g.: 128 bytes). Small transfers perform very fast in the Cell-BE so they do not offer many chances for overlapping unless the execution time for the _MMAP service is such that can be fitted several times in one DMA transfer. The measurement for the Random

Access show that our implementation is limited to the overlapping of 8 _MMAP operations for small DMA transfers. This suggests further study on how to optimize the _MMAP mechanism.

6 Conclusions

This paper describes the main features that have to be included in the implementation of a software cache implementation for the Cell BE processor, in order to maximize the chances for overlapping computation and communication.

It has been proved that a full-associative scheme offers better chances for overlapping computation and communication. It also has been pointed out the necessity of providing with mechanisms to detect the precise moment to initiate write back operations. This translates to overlapping the data transfer from the cache to main memory with actual computation, since the implementation guarantees that the necessary synchronization associated to the write back operation is going to be produced at next reuse of the flushed cache line. Besides, this is accompanied with a replacement policy that tends to increase the time between a use / reuse of the same cache line. Thus, delaying as much as possible the synchronization point and giving the hardware the necessary time to complete the data transfer.

The implementation has been evaluated with two benchmarks in the HPCC suite: Stream and Random Access. For both benchmarks, improvements are significant, ranging from 1.25 and 1.66 of speed up.

References

1. A. E. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J.C. Shepherd, B. So, Z. Sura, "Optimizing Compiler for a Cell Processor ", 14th Parallel Architectures and Compilation Techniques, Saint Louis (Missouri) September 2005
2. Michael Kistler, Michael Perrone, Fabrizio Petrini, "Cell Multiprocessor Communication Network: Built for Speed", IEEE Micro Volume 26 , Issue 3 (May 2006), Pages: 10 - 23
3. Luszczek, P., Bailey, D., Dongarra, J., Kepner, J., Lucas, R., Rabenseifner, R., Takahashi, D. "The HPC Challenge (HPCC) Benchmark Suite," SC06 Conference Tutorial, IEEE, Tampa, Florida, November 12, 2006.
4. Qin Wang, Weihua Zhang and Binyu Zang, "Optimizing Software Cache Performance of Packet Processing Applications", LCTES07
5. Jinquan Dai, Long Li, Bo Huang, "Pipelined Execution of Critical Sections Using Software-Controlled Caching in Network Processors", Proceedings of the International Symposium on Code Generation and Optimization table of contents, Pages: 312-324, 2007, ISBN:0-7695-2764-7
6. Rajiv Ravindran, Michael Chu and Scott Mahlke, "Compiler Managed Partitioned Data Caches for Low Power", LCTES07
7. Tong Chen, Zehra Sura, Kathryn O'Brien and Kevin O'Brien , "Optimizing the use of static buffers for DMA on a Cell chip", 19th International Workshop on Languages and Compilers for Parallel Computing, November 2-4, 2006, New Orleans, Louisiana