Automatic Pre-Fetch and Modulo Scheduling Transformations for the Cell BE Architecture

Nikola Vujić, Marc Gonzàlez, Xavier Martorell, Eduard Ayguadé
Barcelona Supercomputing Center
Department of Computer Architecture, Technical University of Catalonia
nvujic@bsc.es, marc@ac.upc.edu, xavim@ac.upc.edu, eduard@ac.upc.edu

Abstract. Ease of programming is one of the main impediments for the broad acceptance of multi-core systems with no hardware support for transparent data transfer between local and global memories. Software cache is a robust approach to provide the user with a transparent view of the memory architecture; but this software approach can suffer from poor performance. In this paper, we propose a hierarchical, hybrid software-cache architecture that targets enabling pre-fetch techniques. Memory accesses are classified at compile time in two classes, high-locality and irregular. Our approach then steers the memory references toward one of two specific cache structures optimized for their respective access pattern. The specific cache structures are optimized to enable high-level compiler optimizations to aggressively unroll loops, reorder cache references, and/or transform surrounding loops so as to practically eliminate the software cache overhead in the innermost loop. The cache design enables automatic pre-fetch and modulo scheduling transformations. Performance evaluation indicates that the optimized software-cache structures combined with the proposed pre-fetch techniques translate into speed-up between 10% and 20%. Evaluation is done on a set of parallel NAS applications.

Keywords: Cell BE Architecture, Software Cache, Pre-fetching, Modulo Scheduling.

1 Introduction

Heterogeneity has become one particular trend in recently proposed computer systems. For instance, the IBM Cell BE processor [1-5] is a multi-core design that mixes two architectures: a traditional superscalar core based on the PowerPC architecture surrounded by eight cores based on the Synergistic Processor Element (SPE)[4]. In the IBM Cell architecture, the SPEs are provided with local memories and data transfers from/to main memory are explicitly performed under software control. In terms of programmability, this adds another level of complexity and programmers have to deal with the burden of explicitly program the necessary data transfers within applications. General compiler-based solutions [5] are difficult to deploy due to the lack of sufficient information at compile time to generate correct and efficient code.

One global solution is that of emulating a hardware cache by software techniques. In software cache based environments, every memory reference is wrapped by control handlers to ensure correctness. Control handlers are responsible for all cache operations: look-up, placement/replacement, data transfers, synchronization, address

translation, and consistency. Figure 1 shows an example of the kind of code emitted by the compiler targeting a software emulated cache.

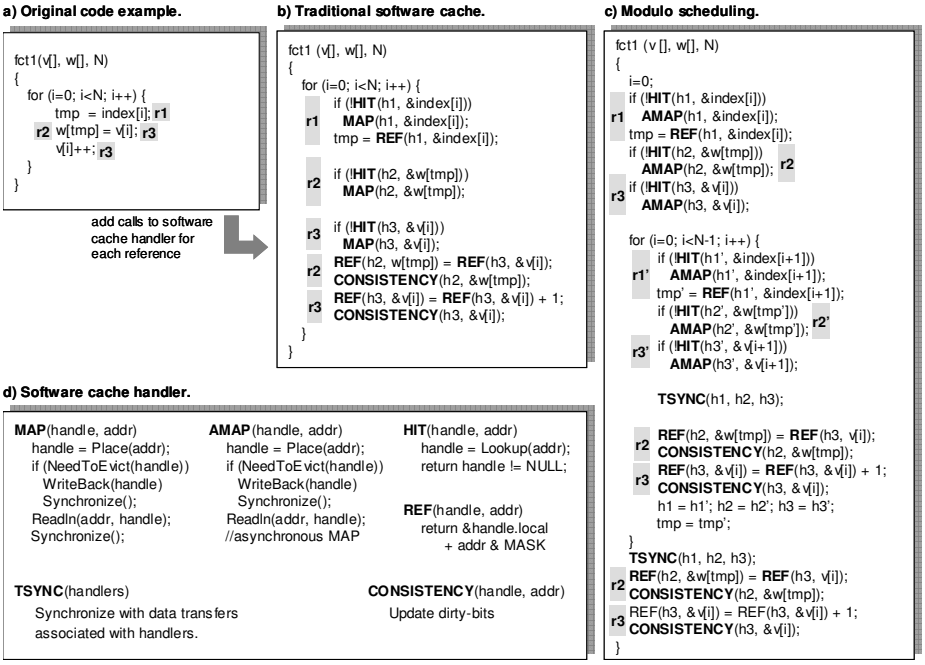

Fig. 1. Overhead of traditional software cache approaches.

The memory references *r1*, *r2* and *r3* have been transformed and the correspondent code is showed in Figure 1b. Before the actual use of data, it is necessary to check if the data is resident in the software cache. This checking is done by invoking the HIT runtime call. If data is not resident then miss handler MAP is invoked to serve a miss. The MAP miss handler is responsible for selecting a cache line to be evicted (if necessary, and then perform the write-back operation), and finally loads the requested line in a synchronous manner. When data is resident in the software cache, the actual access can be allowed, but this operation requires an address translation: the REF handler is responsible for that. For memory reference *r3*, it is necessary to update memory consistency structures, in the example this is associated to the CONSISTENCY handler.

Clearly, the transformed code in Figure 1b is far from optimal, especially because of how memory references *r3* and *r1* are treated. Those references expose a high degree of spatial locality, but every of their instances are going to be checked at runtime introducing unnecessary overhead. For references which expose a high degree of spatial-locality, it is trivial to compute the number of useful data present in the current cache line along the execution of the innermost loop. For such type of memory references we can easily compute the number of loop iterations (within the iteration space of i-loop) for which the current cache line can provide data for such references. This would allow iterating without a miss and without any software cache intervention. But this optimization requires some control over the cache geometry.

First, we must be able to pin a cache line in the cache storage, releasing it only when all high-locality references are done with it. Second, the cache must have at least one unoccupied cache line per distinct high locality references in the loops, if we want to remove all checking code from the innermost loop. Third, it would be desirable to have a "big" cache line size in order to maximize the number of iterations that could be executed with no need of any cache intervention. On the other side, reference *r2* should be treated with very different mechanisms: it exposes very poor spatial locality, so a small cache line would be desirable. This suggests a hybrid design where memory references are mapped to specific storages according to the locality they expose.

Another source of significant overhead is the synchronous communication in the MAP handler. Whatever the implementation of the MAP handler, it is necessary to introduce a synchronization between the data transfer and the actual load/store operation the MAP is associated to. This hinders the possibilities of overlapping communication with computation. Pre-fetch techniques can be introduced to hide the memory latencies, but in the context of software cache systems, pre-fetch does not come for free. Pre-fetching requires execution of control code related to the lookup, placement/replacement and data transfer operations. Besides, it is necessary to ensure that the pre-fetch data is in the range of the valid address space. One well known pre-fetch technique is the modulo scheduling execution [7-9]. In Figure 1c this technique has been applied to the original source code. Basically, data used in iteration *i+1* is pre-fetched in iteration *i*. Now the communications in AMAP are asynchronous, which makes possible to overlap some computation in iteration *i* with some communication related to data used in iteration *i+1*. Notice the TSYNCH call which is responsible that the data required for load/store operations is already in the cache storage. But the problem is not yet solved, since there are two undesirable situations that make the transformation in Figure 1c inapplicable. First, it is necessary to ensure that no conflict appear between the set of consecutive AMAP operations. This is related to the associative level of the cache design and suggests a full associative scheme, always limited by the look-up overheads. Second, it is possible that some write-back operation is triggered along an AMAP operation: this implies some communication that has to be performed synchronously, making useless the modulo scheduling transformation.

Our main contribution is to design a *hierarchical, hybrid software-cache* architecture that is designed from the ground up to enable compiler optimizations that reduce software cache overheads. We identify two main data access patterns, one for high-locality and one for irregular accesses. Because the compiler optimizations targeting these two patterns have different objectives and requirements, we have designed two distinct cache structures that best respond to these distinct access patterns and optimization requirements. In particular, our design includes: (1) a high-locality cache with a variable configuration, lines that can be pinned, and a sophisticated eager write-back mechanism; and (2) a transaction cache with fast, fully associative lookup, short lines, and an efficient write-through policy. The cache design includes specific support for automatic pre-fetch and modulo scheduling code transformations.

The rest of the paper is organized as follows. Section 2 presents our software cache design. Section 3 describes the code transformations enabled by our approach. Section 4 evaluates our approach using some applications of the NAS benchmarks. Related work is presented in Section 5 and Section 6 concludes the paper with some conclusions.

2 Software Cache Design

We describe in this section the design of our hierarchical, hybrid software cache. Figure 2 shows the high level architecture of our software cache. Memory references exposing a high degree of locality are mapped by the compiler to the *High-Locality Cache*, and the others, irregular accesses are mapped into the *Transactional Cache*. The *Memory Consistency Block* implements the necessary data structures to maintain a relaxed consistency model. The *Pre-fetching Block* implements necessary data structures to maintain pre-fetching for regular memory references.

The cache is accessed through one block only, either the *High-Locality Cache* or the *Transactional Cache*. Both caches are consistent with each other. The hybrid approach is hierarchical in that the *Transactional Cache* is forced to check for the data in the *High-Locality Cache* storage during the lookup process.

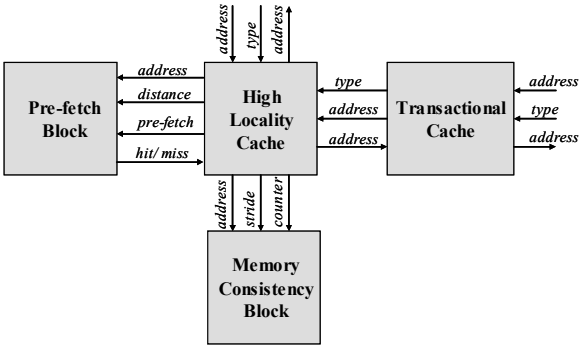

Fig. 2. Block diagram of our software cache design.

2.1 The High Locality Cache.

The *High-Locality Cache* enables compiler optimizations for memory references that expose a high degree of spatial locality. It is designed to pin cache lines using explicit reference counters, deliver good hit ratios, and maximize the overlap between computation and communication.

2.1.1 High-Locality Cache Structures.

The *High Locality Cache* is composed of the following six data structures, depicted in Figure 3: (1) the *Cache Storage* to store application data, (2) the *Cache Line Descriptors* to describe each line in the cache, (3) the *Cache Directory* to retrieve the lines, (4) the *Cache Unused List* to indicate the lines that may be reused, (5) the *Cache Translation Record* to preserve for each reference the address resolved by the

cache lookup, and (6) the *Cache Parameters* to record global configuration parameters.

The Cache Storage is a block of data storage organized as N cache lines, where N is total cache storage divided by the line size. The line size is described by the Cache Line Size parameter, and must be a power of 2. In our configuration, we can store between 16 to 128 cache lines of sizes from 512 to 4KB, within its 64KB cache storage.

Each cache line is associated with a unique *Cache Line Descriptor* that describes all there is to know about that line. Its *Global Base Address* is a global memory address that corresponds to the base address associated with this line in global memory. Its *Local Base Address* corresponds to the base address of the cache line in the local-memory cache storage. Its *Cache Line State* records state such as whether the line holds modified data or not. Its *Reference Counter* keeps track of the number of memory references that are currently referencing this cache line. Its *Directory Links* is a pair of pointers used by the cache directory to list all of the line descriptors that map to the same cache directory entry. Its *Free Links* is a pair of pointers used to list all the lines that are currently unused (i.e. with reference counter of zero). Its *Communication Tags* are a pair of integer values used to synchronize data transfers to/from the software cache. In our configuration, we synchronize using DMA fences, using each of the 32 distinct hardware fences. Our communication tags thus range from 0 to 31.

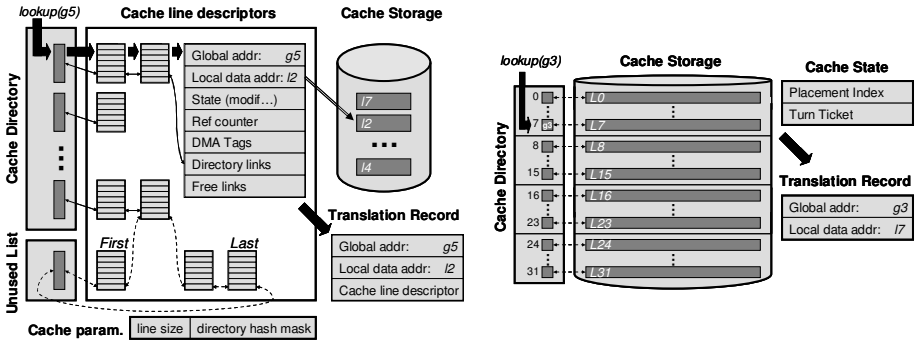

Fig. 3. Structures of the High Locality Cache and Transactional Cache

The *Cache Translation Record* preserves information generated by the lookup process and to be later used when data is accessed by the actual reference. It contains 3 elements; the global base address of the original reference, the local base address in the cache storage, and a pointer to the cache line descriptor.

We implement an efficient, fully associative cache structure using the *Cache Directory* structure. It contains a sufficiently large number of double–linked lists (128 in our implementation), where each list can contain an arbitrary number of cache line descriptors. A hash function is applied to the global base address to locate its corresponding list, which is then traversed to find a possible match. The use of a hash function enables us to efficiently implement cache configurations with up to 128-way fully associative caches.

The *Cache Unused List* is a double-linked list which contains all the cache line descriptor no longer in use. Other cache parameters include parameters such as the *Cache Directory Hash Mask,* a mask used by the cache directory to associate a global base address with its specific linked list.

2.1.2 High-Locality Cache Operational Model.

The operational model for the *High Locality Cache* is composed of all the operations that execute upon the cache structures and implement the primitive operations shown in Figure 1, namely *lookup*, *placement*, *communication*, *synchronization* and *consistency* mechanisms. The following paragraphs describe each type of operation.

The *lookup* operation for a given reference *r,* translation record *h,* and global address *g* is divided in two different phases. The first phase checks if the global address *g* is found in the cache line currently pointed to by the translation record *h*. When this is the case, we have a hit and we are done. Otherwise, we have a situation where the translation record will need to point to a new cache line in the local storage. The lookup process then enters its second phase. The second phase accesses the cache directory to determine if the referenced cache line is already resident in the cache storage. When we have a hit, we update the translation record *h* and we are done. Otherwise, a miss occurs and we continue with placement and communication operations.

The reference counter is often updated during the lookup process. Whenever a translation record stops pointing to a specific cache line descriptor, the reference counter of this descriptor is decremented by one. Similarly, whenever a translation record starts pointing to a new cache line descriptor, the reference counter of this new descriptor is incremented by one.

The placement code is invoked when a new line is required. Free lines are discovered when their descriptor's reference counter reaches zero. Free lines are immediately inserted at the end of the unused cache line list. Modified lines are then eagerly written back to global memory. When a new line is required, we grab the line at the head of the unused cache line list after ensuring that the communication performing the write-back is completed, if the line was modified.

We support a relaxed consistency model. While it is the *Memory Consistency Block* responsibility to maintain consistency, the *High-Locality Cache* is responsible for informing the consistency block of which subsets of any given cache line have been modified and how to trigger the write-back mechanism. Every time a cache line miss occurs, cache thus informs the *Memory Consistency Block* about which elements in the cache line are going to be modified.

The communication code performs all data transfer operations asynchronously. For a system such as the Cell BE processor with a full-featured DMA engine, we reserve the DMA tags *0* to *15* for data transfers from main memory to the local memory, and tags 16 to 31 for data transfers in the reverse direction. In both cases, tags are assigned in a circular manner. Tags used in the communication operations are recorded in the *Communication Tags* field of the *Cache Line Descriptor*. All data transfers tagged with the same DMA tag are forced by the DMA hardware to strictly perform in the order they were programmed.

The synchronization operation is supported by the data in the *Cache Line Descriptor*, in the *Communication Tags* field. The DMA tags stored in this field are used to check that any pending data transfer is completed. The *Communication Tags* record every tag that invokes the corresponding cache line.

When accessing data, the global to local address translation is supported through the translation record. The translation operation is composed of several arithmetic computations required to compute the reference's offset in the cache line and to add the offset to the local base address.

2.2 The Pre-Fetch Block

The *Pre-Fetch Block* enables automatic pre-fetch for regular memory references. The Pre-fetch Block is selective in the sense that not all regular memory references trigger the pre-fetch. It is activated under demand according to the activity in the High Locality Cache. For selected references, the memory addresses are forwarded to the Pre-fetch Block. Then the pre-fetch can be activated and all forwarded addresses determine the next cache lines to be pre-fetched.

2.2.1 The Pre-Fetch Structures.

The *Pre-Fetch Block* is composed of the following four structures: (1) *Pre-Fetch Translation Record* to preserve for each reference the address resolved by the pre-fetch operation, (2) *Pre-fetch Translation Table* to keep track of records being used in pre-fetch operation, and (3) the *Pre-fetch Communication Tags* to preserve DMA tags used for pre-fetching.

The *Pre-Fetch Translation Record* structure consists of four fields: (1) the pre-fetch global address is the base address of the cache line that triggers pre-fetch, (2) the pre-fetch local address is the base address of the cache line allocated to hold the pre-fetched data in the local store, (3) the pre-fetch cache line descriptor is a pointer to the cache line descriptor of the pre-fetched line, and (4) the pre-fetch distance that indicates the next cache line to be pre-fetched as a distance (in a number of cache lines) from the cache line base address that triggered the pre-fetch.

The *Pre-Fetch Translation Table* is a table where each entry holds one *Pre-Fetch Translation Record*. The Pre-fetch Counter keeps track of the number of pending pre-fetch operations.

The *Pre-fetch Communication Tags* consists of all communication tags actively used for pre-fetching purposes. These tags are going to be used to synchronize the data transfers associated to the pre-fetched data.

2.2.2 The Pre-Fetch Block Operational Model.

Memory references that have been selected to trigger the pre-fetch are recorded in the Pre-fetch Translation Table. Pre-fetch is activated from the High Locality Cache and this causes the Pre-fetch Block to traverse the Pre-fetch Translation Table and for every non empty entry performs the look-up, placement and replacement operations as if the cache line being pre-fetched was referenced by the actual computation. Along this process all the communication tags used in the data transfers are recorded in the Pre-fetch Communication Tags register. Under control of the High Locality Cache, it is possible to synchronize with the pre-fetched data using this register.

Introducing pre-fetch support requires reserving some of the available communication tags specifically for that purpose. The range of tags that was used to bring data in to the cache storage is split in two different ranges, one from 0 to 7, the other from 8 to 15. Both ranges are assigned in a circular manner and the High Locality Cache and the Pre-fetch block are coordinated to switch from one range to the other every time the Pre-fetch block is required to perform pre-fetch operations.

2.3 The Transactional Cache

The *Transactional Cache* is aimed at memory references that do not expose any spatial locality. Because miss ratios are expected to be high, this cache is designed to deliver very low hit and miss overhead while enabling overlapped computation and communication. The design introduces very simple structures that allow support for *lookup*, *placement*, *communication*, *consistency*, *synchronization,* and *translation* mechanisms.

In our configuration, the transactional storage is organized as a small 4KB capacity cache, fully associative, and with *32 128*-bytes cache lines. It supports a relaxed consistency model using a write-through policy.

2.2.1 The Transactional Cache Structures

The *Transactional Cache* is composed of the following four data structures, shown in Figure 3: (1) the *Cache Directory* to retrieve the lines, (2) the *Cache Storage* to hold the application data, (3) the *Translation Record* to preserve the outcome of a cache lookup for each reference, and (4) some additional cache state.

The *Cache Directory* is organized as a vector of 32 4-byte entries. Each entry holds the global base address associated with this entry's cache line. The index of the entry in the directory structure is also used as index into the *Cache Storage* to find the data associated with that entry. The directory entries are packed in memory and aligned at a 16-byte boundary so as to enable the use of fast SIMD compares to more quickly locate entries. The *Cache Storage* is organized as 32 cache lines, where each 128-bytes line is aligned at a 128-byte boundary.

To increase concurrency, the cache directory and storage structures are logically divided in four equal-size partitions; the *Cache Turn Ticket* indicates which partition is actively used. Within the active partition, the *Cache Placement Index* points to the cache line that will be used to service the next miss.

At a high level, the active partition is used for buffering cache lines which are going to be used in the current transaction and these cache lines were pre-fetched. The next partition, in circular manner, is used for placing cache lines which we are pre-fetching and which are going to be used in the next transaction in the next iteration of the unrolled loop. Other two partitions are used to buffer data of the two previous transactions while their modified data is being flushed back to the main memory. Based on this explanation, we defined three states in which our partitions can be: *in-use*, *pre-fetching* and *flushing*.

2.2.2 The Transactional Cache Operational Model

In this paper, a transaction is a set of computation and related communication that will happen as a unit (but never rollback). Operations in a transaction happen in four

consecutive steps: (1) initialization, (2) communication into local memory, (3) computation associated with the transaction, and (4) propagation of any modified state back to global memory.

During initialization, in Step 1, the *Cache Turn Ticket* is set to point to the next partition in the circular manner. The *Cache Placement Index* is set to point to the first cache line of the new partition. In our configuration, its value can be 0, 8, 16 or 24 when the ticket is, respectively, pointing to partition 0, 1, 2, or 3. In addition, all cache directory entries in the new active partition are erased.

In Step 2, the data corresponding to each global-memory reference is brought into the local memory, using sequences of look-up and possibly calls to the miss-handler. The lookup process for a given reference *r,* translation record *h,* and global address *g* first proceeds with a standard *High-Locality* cache lookup, since we do not want to replicate data in both cache structures. This first lookup can be avoided if address *g* can be guaranteed not to be found in the high-locality cache. When a hit occurs, the *Local Base Address* field in translation record *h* is simply set to point to the appropriate sub-section of the line in the high-locality cache storage. When a miss occurs, however, we proceed by checking the address *g* against the entries in transactional cache directory. This lookup is fast on architectures with SIMD units, such as the SPEs. On platforms where 4 entries fit into one SIMD register, such as the SPEs, we perform a 32-way address match using 8 compare SIMD instructions. When a miss occurs, a placement operation is executed. When a hit occurs, the look-up can operate in one of two ways. If the hit occurred within the active partition (partition where we are going to pre-fetch the data for next iteration), we simply update the translation record *h*. If the hit occurred within the next partition, in circular manner, then we need to do two actions. First, we need to migrate the line to the active partition, a placement operation is used for this operation as well. Second, we need to inform previous partition (partition which is in *in-use* state) about migrated cache line in order to maintain consistency between transactions. If, however hit occurred within the other partitions, we simply update the translation record *h*.

The placement code simply installs a new directory entry and associated cache line data at the line pointed by the *Cache Placement Index*. The placement index is then increased by one (modulo 32). Communications generated by the miss in Step 2 results into an asynchronous 128-byte transfer into local memory.

Step 3 proceeds with the computation, using the same translation record as seen in Section 2.1.

In Step 4, every modified storage location that was modified by a store in Step 3 is directly propagated into global memory. This approach to relaxed consistency eliminates the need for any extra data structures (such as dirty bits) and do not introduce any transfer atomicity issue. These asynchronous communications occur regardless of whether a hit or miss occurred in Step 2. Moreover, only the modified bytes of data are transferred into global memory during Step 4.

In order to ensure consistency within and across transactions, every data transfer is tagged with the index of the cache line being used (from *0* to *31*), and a fence is placed right after the data transfer operation. All data transfers tagged with the same tag are forced by the hardware to perform strictly in the order under which they were programmed. The synchronization code occurs in precisely two places. The first

synchronization is placed between Steps 2 & 3, to ensure that the data arrive before being used. When Partition 0 is active, we wait for data transfer operations with tags [*0..7*], for partition 1 appropriate tags are [*8..15*], for partition 2 tags are [*16..23*] and for partition 3 wait for tags [*24..31*]. For the data transfer initiated in Step 4, the synchronization code is placed at the beginning of the next transaction with the same value for the *Cache Turn Ticket*, synchronizing with the data transfer operations tagged with numbers [*0..7*], [*8..15*], [*16..23*] or [*24..31*].

2.4 The Memory Consistency Block

The *Memory Consistency Block* contains the necessary data structures to maintain a relaxed consistency model. For every cache line in the *High Locality Cache*, information about what data has been modified is maintained using a Dirty Bits data structure. Whenever a cache line has to be evicted, the write-back process is composed by three steps. The cache line in main memory is read, then a merge operation is applied between both versions, the one resident in the cache storage and the one recently transferred, and finally, the output of the merge is sent back to main memory. All data transfers are synchronous and atomic.

3 Code Transformations

We describe in this section the type of code transformation techniques that are now enabled using our pre-fetching and modulo scheduling approach in the software cache. With no loss of generality, the code transformation targets the execution of loops.

The code transformations are performed in three ordered phases: (1) classifying of memory references into regular and irregular accesses; (2) transformation of the code to optimize regular memory accesses, and (3) transformation of the code to optimize irregular memory accesses. We illustrate this process in Figure 4 using the same introductory example presented in Figure 1a.

3.1 Classification of memory accesses

In Phase 1, memory accesses are classified as regular or irregular accesses. Figure 4a shows the classification of the references for our exemplary code. Memory accesses *index[i]* and *v[i]* with *i=0...N* are labeled as regular, while memory access *w[tmp]* with *tmp=index[i]* is labeled as irregular memory access.

3.2 Regular Access Transformations

In phase 2, original *for*-loop is transformed into two nested loops (Figure 4b). Dynamic sub-chunking of the iteration space is done by using those two nested loops. In each dynamic sub-chunk of iterations we are sure that all relevant data are permanent in the cache storage and iterating through them, in the inner *for*-loop of the transformed code, is not going to produce miss. Work done in the inner *for*-loop (related to regular memory accesses) does not have any cache overhead. In the while loop we are introducing necessary code transformations per each high locality memory reference. The lookup, dynamic sub-chunking, consistency maintaining, pre-fetching and synchronization operation are done in the while loop.

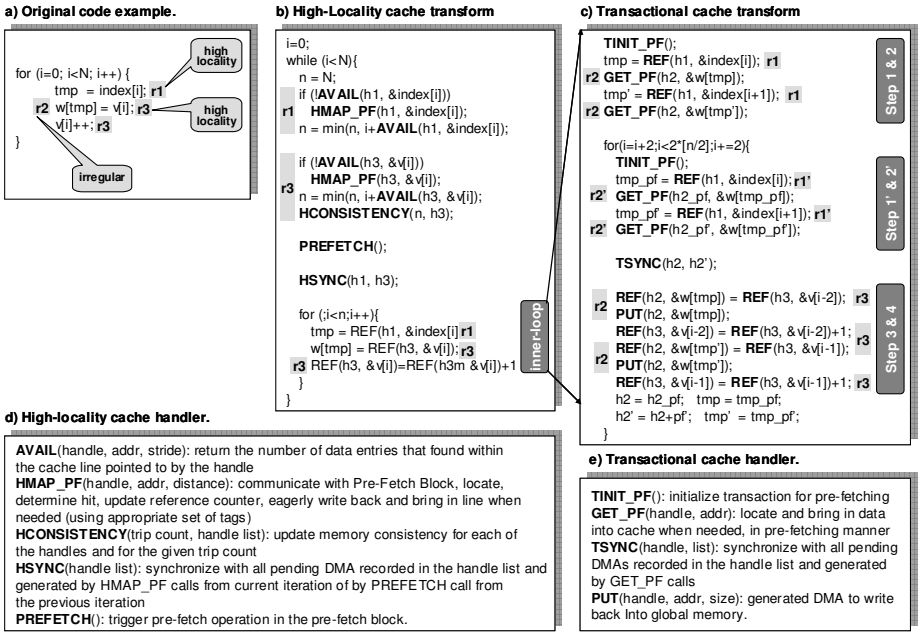

Fig. 4. Example of C code and its code transformation.

The lookup operation checks if the address *&index[i]* of the reference *r1* is in the cache line pointed to by the translation record (handle) *h1*. This checking is done by using AVAIL macro. The AVAIL macro returns for reference *r1* number of iterations for which this reference will be present in the cache line pointed to by handle *h1*. If this number is greater than zero we have hit and then we are proceeding with determining of the upcoming dynamic sub-chunk of the iteration space. If this number is equal to zero then macro HMAP_PF is invoked to serve a miss. Notice the third argument of the HMAP_PF macro, indicating if pre-fetch has to be considered for the given memory reference. This argument corresponds to the pre-fetch distance, indicating the next cache line to be most likely accessed by the memory reference. In case the distance is other than zero, pre-fetch is activated and the address is forwarded to the Pre-fetch Block. Next step is determining of the upcoming dynamic sub-chunk of the iteration space. Once we have sub-chunking factor *n* we can process with consistency and synchronization operations. Since reference *r1* is read only access reference then consistency operation is not processed for *r1* but is processed for *r3* which is read and write access reference. The PREFETCH macro triggers pre-fetch for all forwarded addresses. Notice that the pre-fetch code is executed before the synchronization with the DMA engine takes place, giving the opportunity to overlap the pre-fetch code to actual communication.

3.3 Irregular Access Transformation

In Phase 3, we transform the inner *for*-loop so as to optimize cache overhead for irregular memory accesses. The first task is to determine the transactions. In our

work, the scope of a transaction is a basic block, or a subset of. Large transactions are beneficial as they potentially increase the number of concurrent misses, thus increasing communication overlap. In general, a transaction can contain as many distinct irregular accesses as there are entries in a single partition of the transactional cache, 8 in our configuration. Because of our focus on loops, larger transactions are mainly achieved through loop unrolling. In our example, we unrolled the inner *for*-loop by a factor of 2 (for conciseness) so as to include two *w[tmp]* and *w[tmp']* references within a single transaction.

The code generated for a transaction closely follows the four step process outlined in Section 2.2.2. As shown in Figure 4c, we first initialize the transaction using the macro TINIT (Step 1) and then proceed in asynchronously acquiring the data of each irregular reference *r2* (due to loop unrolling of factor 2 we have two *r2* references) using the GET macro (Step 2). Once all irregular references have been processed, we issue a TSYNC operation to synchronize with pending DMAs issued by appropriate GET macros. We then access the data using the REF macro (Step 3) and write-back the modified data using the PUT macro (Step 4).

Conceptually, the work inside transactions in modulo scheduled loop can be visualized as four tasks. In the loop prologue we pre-fetch data which are going to be used within computation section in the first iteration of the unrolled loop. We assign task *Step1&2* to this prologue. At the beginning of the unrolled loop body we pre-fetch data which is going to be used in the next iteration or in the loop epilogue. In this part of the code we use translation records *h2* and *h2'*. We assign task *Step1'&2'* to this part of the unrolled loop body. After this we have a necessary synchronization point where we synchronize with pending DMAs determined by translation records *h2* and *h2'*. When we are sure that data has arrived in the cache, we execute computation section and at the end, modified data is sent back to main memory (PUT macro). This corresponds to task *Step3&4*. In the steady state of the loop, partitions go changing of state: pre-fetch, in-use, flushing. Note that for conciseness, the loop unrolling has been done assuming that the number of iteration was a multiple of two. Figure 5 shows the evolution of each partition for three iterations of the loop.

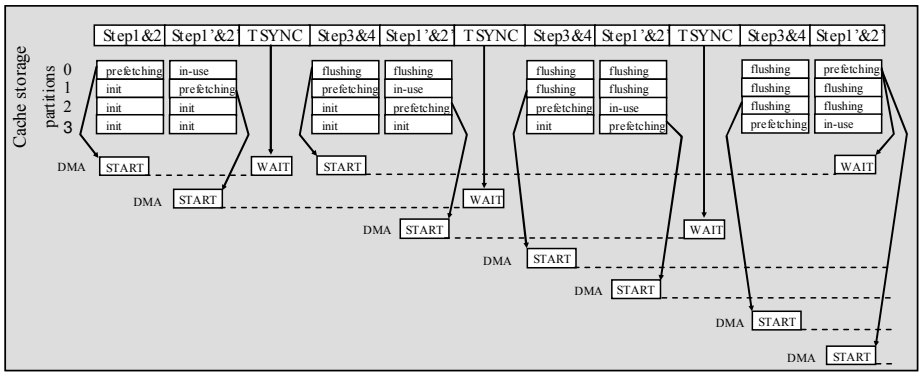

Fig. 5. Sequence of events in a modulo scheduled loop.

4 Evaluation

In the evaluation section we measure the impact in performance of the proposed pre-fetching techniques: automatic pre-fetch for regular references, modulo scheduling for irregular references. In this evaluation we never combine these two techniques in the same loop. We compare two cache configurations, one where pre-fetch is enabled, another where pre-fetch is not active. Improvement is measured in terms of speed-up.

We have evaluated the proposals with the CG, IS and FT parallel applications from the NAS benchmark suite [10] and STREAM parallel application [6], which are parallelized using OpenMP directives. All measurements are performed on a Cell BE blade with two Cell BE processors running at 3.2 GHz with 1 GB of memory (512 MB per processor) under Linux Fedora Core 6 (Kernel 2.6.20-CBE). Only one Cell BE processor is used for the evaluation.

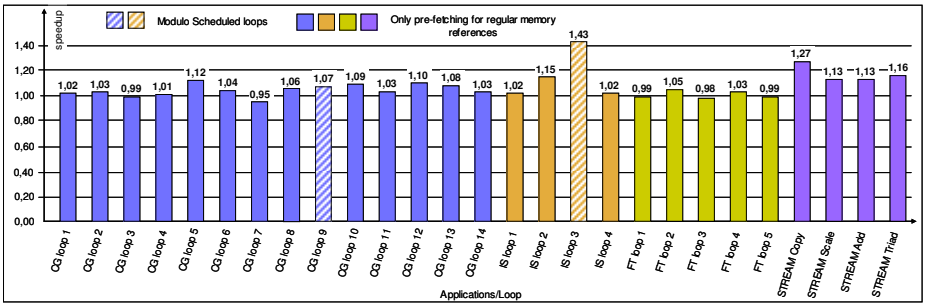

Fig. 6. Speed-up factors for automatic pre-fetch and modulo scheduling. Modulo scheduling is used in CG loop9 and IS loop3, due to CG and IS are totally dominated by irregular memory accesses in the mentioned loops.

Figure 6 shows speedup factors obtained by enabling pre-fetch in a variety of loops from the CG-B, IS-B, FT-B and STREAM benchmarks. Overall execution times for CG-B, IS-B and FT-B are shown in Figure 7. For STREAM, the improvements are noticeable, but very predictable in the sense that the four tested kernels are not computationally bounded. Communications represent an important percentage of overall execution time. This is yet more noticeable in the differences we observe between the four kernels: the *copy* kernel which is not including any floating point operation doubles the performance of the other kernels.

In the case of CG-B, the improvements range from 3% up to 10 % at most. Loops 3 and 7 suffer from slight degradation (not even a 1% and 5% respectively). The reason for that is related to the differences on how deeply the loops are affected by communications. The CG-B loop 9 is dominated with irregular memory references and is the most consuming loop in the CG-B. Improvement achieved in this loop has good influence on overall execution time of the CG-B (Figure 7). The case of the IS-B is different. Here the benefits are quite impressive: loop 2 improves about 15% and loop 3 improves about 40%. Loop 3 is totally dominated by irregular memory references and the introduction of the modulo scheduling transformation is what causes such improvement. Improvement in loop3 has good influence on overall execution time of the IS-B. The case of FT is very different and exposes very poor

improvements, ranging from slight performance degradation (2% at most) up to some improvement close 5%. All loops are dominated by the computation, not by the communication overheads. There is no improvement in overall execution time for FT.

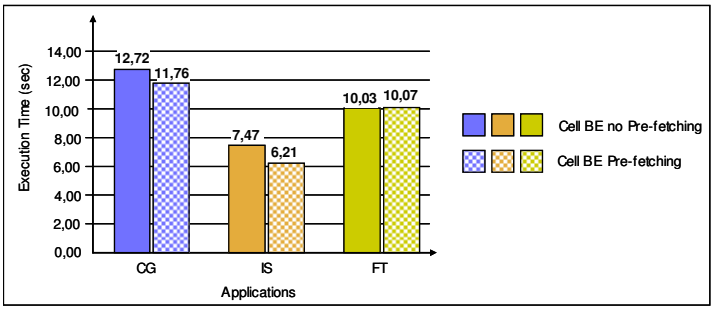

Fig. 7. Execution times of NAS benchmarks. Corresponding speedup factors in overall execution times are: CG - 1,082, IS - 1,203 and FT - 0,996.

5 Related Work

Although a different technique, tiling transformations and static buffers may be used to reach the same level of code optimization [5]. In general, 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. This approach, however, requires precise information about memory aliasing at compile time, which is not always available. In general, the association between static buffers and memory references should be postponed until run time. This is what we do in this paper, since cache lines are treated as buffers that are dynamically allocated, solving all the difficulties related to memory aliasing. Of course, if the performance of a software cache approach is to match that of static buffers, clearly, any efficient implementation should work with a cache line size similar to that of the static buffers (usually 1KB, 2KB, 4KB, depending on the number of memory references to treat) [5]. This is the case of the software cache design presented in this paper.

Specifically for the Cell BE, there has been proposal to perform data pre-fetching under an inspector/executor model [12]. For indirect accesses, a slicing compilation technique is introduced to generate a code version that at runtime computes all memory addresses generated in indirect accesses. This makes possible to overlap DMA transfers with the slice execution. This approach has been showed to return considerable improvements for indirect accesses, but the technique is limited to the associative level in the cache design. Cache conflicts cause to switch between the inspector and executor code versions, diminishing the effects of this technique.

The Memory Hierarchical Layer Assignment (MHLA) [13] is a unified technique which addresses the problem of optimizing the data assignments into memory layers and the block transfers between memory layers. This technique starts from the source code specification of the application and by collecting profiling information optimizes memory mapping and execution order of data transfers. Also, memory organization is potentially customized by this technique. The similarity of this technique with our

approach is that pre-fetching is implemented by invoking DMA operations in order to overlap computation and communication. In contrast to our technique, MHLA is aimed for buffering techniques and simple memory organizations due to application-specific pre-fetching approach.

Hare [14] is a pre-fetching scheme consisting of a programmable engine controlled by the user instructions. This technique uses hardware support for pre-fetching. Indeed, pre-fetching is initiated by the hardware at run-time. Programming the proposed engine by user code takes advantages from compile-time analyzes and hardware eliminates additional pre-fetch instruction overhead. In contrast with this proposal, in our work we do not have any hardware support for pre-fetching.

Interrupt Triggered Software Pre-fetching (ITSP) [15] is a pre-fetching technique for real-time embedded systems that adds pre-fetching instructions in interrupt handler software to target cache misses. Pre-fetching optimizations done in ITSP tunes the software to be executed based on observed performance during previous executions. In contrast with our work, ITSP relies on profiling information collected during previous executions of application and hardware pre-fetching instructions are used.

6 Conclusions

This paper presents a novel hybrid software cache architecture designed for pre-fetching purposes. Hybrid software cache architecture maps memory accesses according to the locality they are exposing. According to difference in mapping, pre-fetching is organized in order to enable good overlap of communication and computation for both types of memory accesses. We show performances of pre-fetching for regular and irregular memory accesses. We also show impact of additional instruction overhead introduced due to software implemented pre-fetching. We show that with our approach good speedup can be obtained in some benchmarks (speedup factors from 1.15 to 1.43) and also we show that additional instruction overhead in software implemented pre-fetching sometimes has negative impact on overall performances of some applications and some particular loops in the applications.

Acknowledgments

This research has been supported by the IBM MareIncognito project, in the context of the research projects between BSC and IBM, and by the Ministry of Education of Spain (CICYT) under contract TIN2007-60625 and by the HIPEAC European Network of Excellence under the contract IST-004408.

References

1. H. Peter Hofstee, “Power Efficient Processor Architecture and The Cell Processor”, Proceedings of the 11th Int’l Symposium on High-Performance Computer Architecture, 2005
2. D. Pham et al. “The Design and Implementation of a First-Generation Cell Processor”, Proceedings the IEEE International Solid-State Circuits Conference, 2005
3. M. Kistler et al. “Cell Multiprocessor Communication Network: Built for Speed”, IEEE Micro Volume 26, Issue 3 (May 2006), Pages 10-23

4. M. Gschwind et al. “A Novel SIMD Architecture for the Cell Heterogeneous Chip-Multiprocessor”, In Hot Chips 17, 2005
5. A. E. Eichenberger et al. “Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture”, IBM Systems Journal, Vol 45, No 1, 2006
6. McCalpin, John D., 1995: "Memory Bandwidth and Machine Balance in Current High Performance Computers", IEEE Computer Society Technical Committee on Computer Architecture (TCCA)
7. B. Ramakrishna Rau et al. ”Code Generation Schema for Modulo Scheduling Loops”, Proccedings of the 25th Annual International Symposium on Microarchitecture, 1992
8. B. Ramakrishna Rau et al. ”Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops”, Proceedings of the 27th annual International Symposium on Microarchitecture, 1994
9. Daniel M. Lavery, “Modulo Scheduling of Loops in Control-intensive Non-numeric Programs”, Proceedings of the 29th annual ACM/IEEE International Symposium on Microarchitecture, 1996
10. D. Bailey et al. “The NAS parallel benchmarks” Technical Report TR RNR-91-002, NASA Ames, August 1991
11. B. Sinharoy et al. “POWER 5 system micro-architecture”, IBM Journal of Research and Development, Vol. 49 No. 4/5 July/September 2005
12. Tong Chen et al. “Prefetching irregular references for software cache on cell”, Proceedings of the sixth annual IEEE/ACM international symposium on Code Generation and Optimization, 2008, Pages: 155-164
13. Minas Dasygenis et al. “A Combined DMA and Application-Specific Prefetching Approach for Tackling the Memory Bottleneck”, IEEE Transactions on Very Large Integration (VLSI) Systems, Vol. 14, Issue 3, March 2006, Pages: 279-291
14. Tien-Fu Chen, “An Effective Programmable Prefetch Engine for On-Chip Caches”, Proceedings of the 28th Annual International Symposium on Microarchitecture, 1995
15. Ken W. Batcher et al. ”Interrupt Triggered Software Prefetching for Embedded CPU Instruction Cache”, Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium, 2006