Hybrid Access-Specific Software Cache Techniques for the Cell BE Architecture

Marc Gonzàlez[*+], Nikola Vujic[*],
Xavier Martorell[*+], and Eduard Ayguadé[*+]
[*] Universitat Politècnica de Catalunya (UPC),
[+] Barcelona Supercomputing Center,
Barcelona, Spain.
{ marc.gonzalez nikola.vujic xavier.martorell
eduard.ayguade }@bsc.es

Alexandre E. Eichenberger, Tong Chen,
Zehra Sura, Tao Zhang,
Kevin O'Brien, and Kathryn O'Brien
IBM T.J. Watson Research Center,
Yorktown Heights, New York, USA.
{ alexe chentong zsura taozhang
caomhin kmob } @us.ibm.com

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 classifies at compile time memory accesses 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. Performance evaluation indicates that improvements due to the optimized software-cache structures combined with the proposed code-optimizations translate into 3.5 to 8.4 speedup factors, compared to a traditional software cache approach. As a result, we demonstrate that the Cell BE processor can be a competitive alternative to a modern server-class multi-core such as the IBM Power5 processor for a set of parallel NAS applications.

Categories and Subject Descriptors

D.3.4 Software, PROGRAMMING LANGUAGES, Processors, Compilers; C.3 Computer Systems Organization, SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS, Microprocessor/microcomputer applications

General Terms: Algorithms, Languages, Performance

Keywords: OpenMP, compiler optimizations, local memories, memory classification, software cache

PACT'08, October 25–29, 2008, Toronto, Ontario, Canada.

1. INTRODUCTION.

Emerging multi-core architectures are deploying application-specific processors to address computational acceleration. The proposed architectures, such as the Cell BE processor [1,2,3,4], show a trend towards novel memory hierarchies, providing the computational cores with local memories where the data transfers to/from main memory are explicitly performed under software control. In terms of programmability, the Cell BE and systems alike may require some efforts for hand coded optimizations. General compiler-based solutions [1] are often difficult to deploy due to the lack of sufficient information at compile time to generate correct and efficient code.

One common approach is to hide explicit data transfer via emulating a hardware cache through software techniques. Due to the lack of hardware support to supply the data referenced by load/store operations, every memory reference to global memory is typically guarded by control code to ensure correctness. This code is responsible for all of the actions typically implemented by a hardware cache, namely look-up, placement, data transfers, synchronization, address translation, and consistency. Optimizing the code introduced by the software cache is a difficult task for the compiler, mainly because traditional software cache interfaces [1] mimic typical hardware cache implementations. Such interfaces provide too few guarantees to enable compiler optimizations that could significantly lower software cache overheads.

To illustrate the performance implications of traditional software cache architectures, consider the code example in Figure 1. The original code in Figure 1a shows two references, *v1[i]* and *v2[tmp]*. Assuming the arrays in global memory, Figure 1b depicts the same code with all of the required calls to a traditional software cache handler. Before each reference *r1* and *r2*, we need to check if the data is resident in the software cache (using the HIT macro). When not present, we call the miss handler (using the MAP miss handler). As shown in Figure 1c, the miss handler MAP first locates a suitable cache line to evict, possibly writing the evicted line back to global memory,

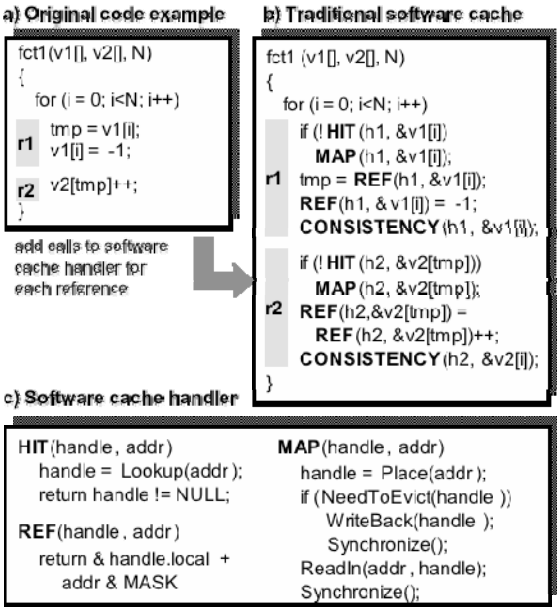


Figure 1: Overhead of traditional cache approaches.

and then loads the requested line. Once the data has arrived (i.e. after a synchronization or blocking DMA), the data can be accessed, using the REF macro, in read or write mode. Following the last reference, a consistency operation is performed to maintain relaxed consistency across all processors in the system. Consistency operations typically consist of updating dirty bits on write-back caches, or data communication on write-through caches.

Clearly, the transformed code in Figure 1b is far from optimal, especially for a high spatial-locality reference such as *v1[i]* with *i=0…N*. For *high-locality* references, it is trivial to compute the number of useful data present in the current cache line. In other words, we can easily compute the number of loop iterations for which the current cache line can provide data for such references. Given such a number of iterations without a miss, we would like to iterate over these computations without any further software cache overhead. However, this code transformation is not possible with a traditional software cache interface. First, we must be able to pin a cache line in the software-cache storage, releasing it only when all high-locality references are done with it. Second, the cache must have at least one cache line per distinct high-locality reference in the loop, if we want to remove all checking code from the innermost loop. Thus we need to have some additional control the geometry of the cache.

The code in Figure 1b is also suboptimal with respect to the second reference, *v2[tmp]*, where *tmp* is equal to the original value of *v1[i]*. This access pattern corresponds to an indirect access, which typically exhibit very poor data locality. Because of this *irregular* access pattern, the data is unlikely to be found in the cache. Performance can only be achieved by exercising as many irregular accesses as possible in parallel (i.e. without synchronization or blocking DMA) for maximum communication overlap. Again, traditional caches interfaces are not suitable, as they typically provide only for a small set-associativity, which directly limits the number of concurrent irregular accesses. Also, since no spatial reuse is expected, typical (long) cache lines are likely to waste bandwidth and thus slow down the execution.

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.

Performance evaluation indicates that improvements due to the optimized software-cache structures combined with the proposed code-optimizations translate into 3.5 to 8.4 speedup factors, compared to a traditional software cache approach. As a result, we demonstrate that the Cell BE processor can be a competitive alternative to a modern server-class multi-core such as the IBM Power5 processor for a set of parallel NAS applications.

The rest of this 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 NAS benchmarks. We present related work in Section 5 and conclude in Section 6.

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 according to the OpenMP memory model [15].

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.

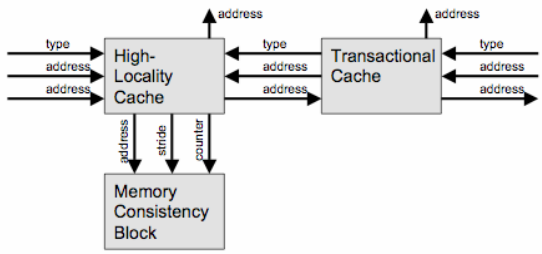


Figure 2: Block diagram of our software cache.

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 configuration, we can store between 16 to 128 cache lines of sizes from 512 to 4K bytes, 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.

The *Cache Translation Record* preserves information generated by the lookup process and to be later used when data is accessed the by the actual reference. It contains 3

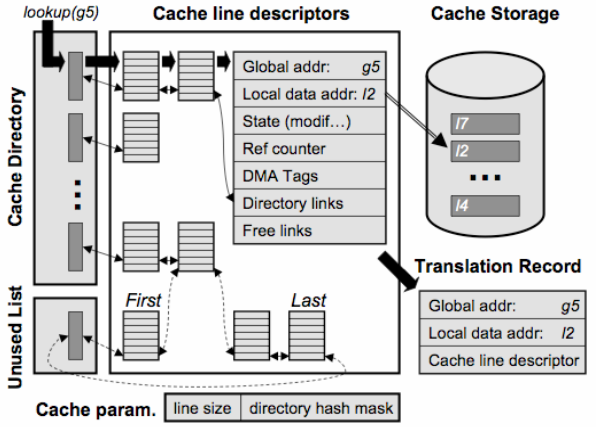


Figure 3. Structures of the High Locality Cache.

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 1c, 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 occurred 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 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 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

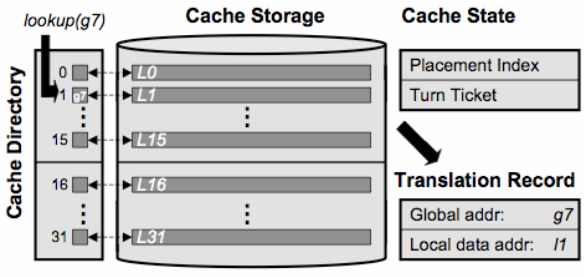


Figure 4. Structures of the Transactional Cache.

very simple structures that allow supporting 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 Transactional Cache Structures

The *Transactional Cache* is composed of the following four data structures, shown in Figure 4: (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 two 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 to bring in the cache lines required by the current transaction, while the other partition is used to buffer the cache lines of the prior transaction while their modified data is being written back to global memory.

2.2.2 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 flipped to point to the other partition. The *Cache Placement Index* is set to the first cache line of the new active partition. In our configuration, its value is either 0 or 16 when the ticket is, respectively, pointing to partition 0 or 1. In addition, all the 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 lookup 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 replicating 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 lookup can operate in one of two ways. If the hit occurred within the active partition, we simply update the transaction record *h*. If, however, the hit occurred within the other partition, we need (for simplicity) to migrate the line to the active partition; a placement operation is used for this operation as well.

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 (not the entire line, unless the entire line was modified) 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...15], and wait for tags [16...31] otherwise. 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...15*], or [*16...31*].

2.3 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 hierarchical, hybrid software cache. It uses here the distinction between memory references with high-locality and irregular access pattern. Accesses with high-locality patterns are mapped to the *High Locality Cache* and all irregular accesses are mapped to the *Transactional Cache*. With no loss of generality, the code transformation targets the execution of loops.

The code transformations are performed in three ordered phases: (1) we classify memory references into high-locality and irregular accesses; (2) we transform the code to optimize high-locality memory references, and (3) we transform the code to optimize irregular memory references. We illustrate this process in Figure 5 using the same introductory example presented in Figure 1a.

3.1 Classification of memory accesses.

In Phase 1, memory accesses are classified as high-locality or irregular accesses. Figure 5a shows the classification of the references for our exemplary code, where memory accesses *v1[i]* with *i=0…N* and *v2[tmp]* with *tmp=v1[i]* are labeled as, respectively, high-locality and irregular memory accesses.

3.2 High-Locality Access Transformations.

In Phase 2, we transform the original *for*-loop into two nested loops that basically perform a dynamic sub-chunking

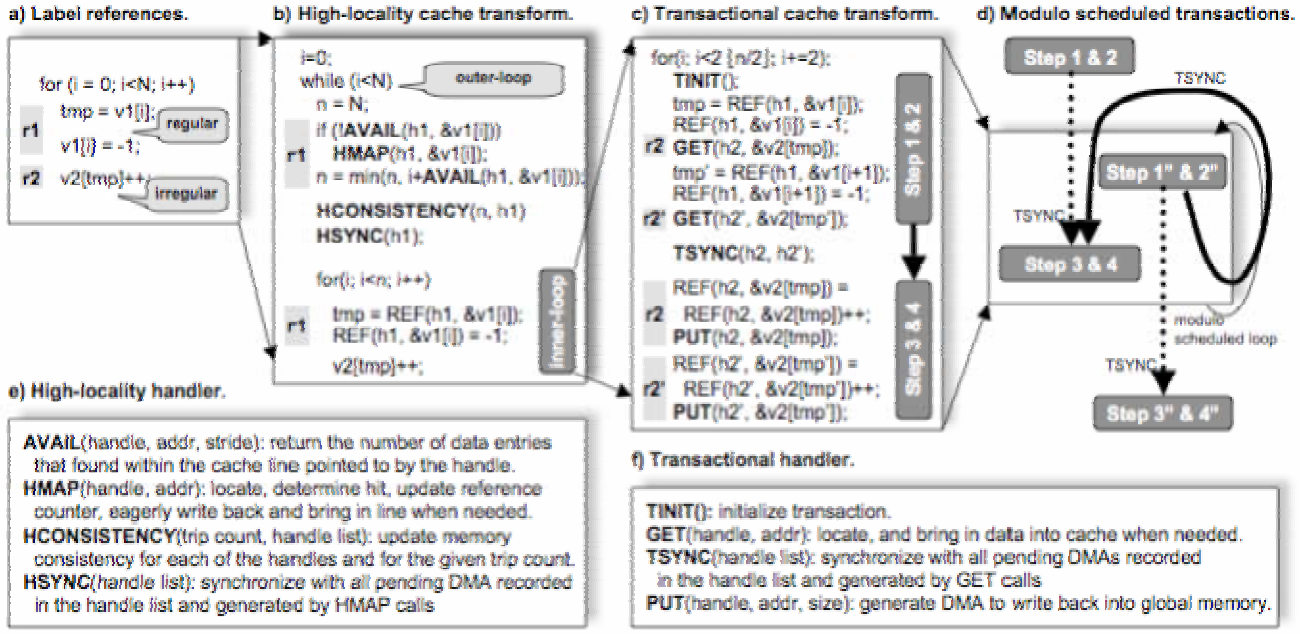


Figure 5: Example of C code and its code transformation.

of the iteration space. As shown in Figure 5b, the outer *while*-loop iterates as long as we have not visited all of the original *N* iteration points. The inner *for*-loop iterates over a dynamic subset of iterations, *n*, where *n* is computed as the largest number of iterations for which none of the high-locality references will experience a miss. As shown in the innermost loop in Figure 5b, there is no cache overhead (REF overheads detailed in Figure 1c are essentially free).

We detail now the work introduced by each high-locality reference in the outer *while*-loop body. Consider in Figure 5b the work associated with reference *r1* with translation record *h1* and global memory address *g1=v1[i]*. First, we compute with the AVAIL macro the number of iterations for which address *g1* will be present in the cache line currently pointed to by its translation record *h1*. If this number is zero, we have a miss, and we invoke the HMAP miss handler. This miss handler performs as indicated in Section 2.1.2. Note that a miss does not imply communication, as the cache line may already be present in the cache due to other references. Once the new line is installed in the translation record *h1*, we recompute AVAIL and update the current dynamic sub-chunking factor *n* so as to take into account the number of iterations for which reference *r1* will not experience a miss.

After processing all high-locality references, the dynamic sub-chunking factor *n* is definitive, and can be used to inform the memory consistency block of all of the memory locations that will be touched by high-locality references in the inner *for*-loop. Note that all this work is performed in parallel with the asynchronous DMA requests possibly initiated by the HMAP miss handler.

The last operation is to perform synchronization on all pending DMAs, using the communication tags found in the cache line descriptors associated with each of the high-locality references.

An additional task is to compute the optimal high-locality cache configuration. At the very least, we need one cache line per distinct high-locality memory reference, but a few more enables better latency hiding for the eager write-back process. Note, however, that we may always downgrade one or more high-locality memory accesses to be treated as irregular references. In practice, we select the largest line size that satisfy each of the high-locality memory references present in the loop.

3.3 Irregular Accesses Transformations.

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 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, 16 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 *v2[tmp]* and *v2[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 5c, we first initialize the transaction (Step 1) and

then proceed in asynchronously acquiring the data of each irregular reference *r2* and *r2'* using the GET macro (Step 2). Once all irregular references have been processed, we issue a TSYNC operation to synchronize on all pending DMAs issued by the GET operations. 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 of a transaction can be visualized as two tasks, Steps 1 & 2 followed by Steps 3 & 4, as shown on the right hand side of the code in Figure 5c. The synchronization between the two tasks can be visualized as a dependence edge between the two. Using this representation in Figure 5c, we can see visually that the code is still far from optimal. Though misses do go in parallel inside Task 1, the dependence between the two dependent tasks still serializes most of the work.

To further increase concurrency, we apply a technique known as modulo scheduling or double buffering. Figure 5d illustrates the gist of this technique. Instead of processing one transaction after another, we maintain two consecutive transactions in flight. In a given steady-state iteration (i.e. in the shaded box in Figure 5d), we initiate a new transaction (Steps 1" & 2") and then complete the transaction of the previous loop iteration (Steps 3 & 4). As a result, the synchronization does not occur between the two tasks in the same iterations, like in Figure 5c, but it occurs between the two tasks in two consecutive iterations. Note that special code to fill/drain the modulo-scheduling pipe and handle the unrolling of the innermost loop is omitted here for conciseness.

3.4 Summary of Code Transformations.

To summarize the achievement of the code, we observe that there is no software cache overhead associated with regular memory accesses in the innermost loop (*for*-loop in Figures 5b-d). Second, all communications generated by regular memory-access misses occur concurrently. Third, all communications generated by irregular memory-access misses occur concurrently. Fourth, the synchronization impact of waiting for pending DMA generated by irregular memory-access misses is greatly reduced by maintaining two consecutive transactions in flight (modulo schedule *for*-loop in Figure 5d).

4. EVALUATION

We present in this section an evaluation of our hybrid software-cache approach. We first investigate what are the main components that contribute to the significant speedup achieved by our cache design compared to a traditional software cache approach. We then present a performance comparison between a software-cache-only game-console Cell BE to a hardware-cache-only server-class Power5 multicore processor.

We use here the IS, CG, MG and FT parallel applications from the NAS benchmark suite [7], which are parallelized using OpenMP directives. All measurements are performed on a Cell BE blade [16] 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.

4.1 Cache Overhead Comparison

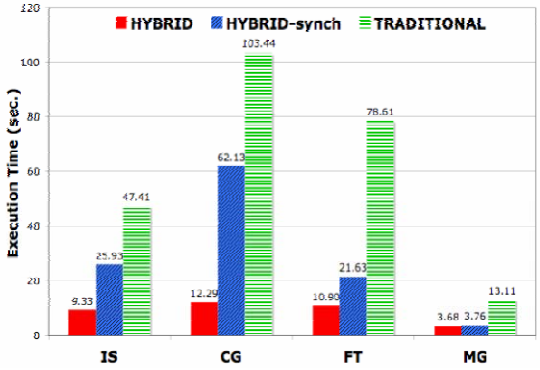
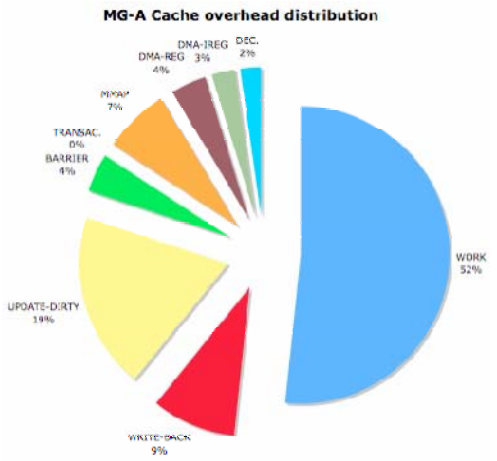


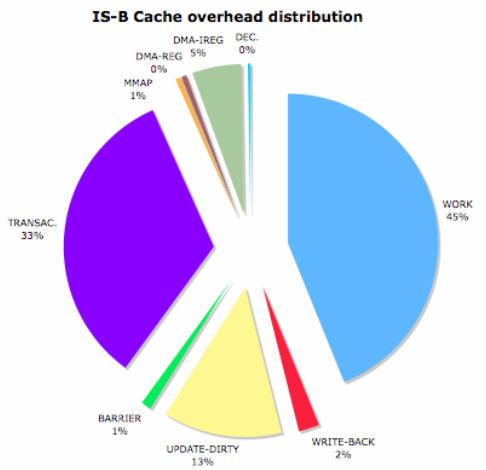
Figure 6. Three software-cache schemes on Cell.

In this section, we evaluate the performance impact of the two major features that significantly contribute to higher performance. The first one is the ability of reducing the amount and complexity of the control code surrounding the global memory references; and the second one is the ability of overlapping communication and computation.

We investigate here three software cache configurations. The first configuration is our cache prototype as described in Section 2, using all of the code transformations proposed in Section 3, and with a 64-KB high-locality cache with lines of 4-KB. We refer to this configuration as HYBRID thereafter. The second configuration is a modified version of HYBRID, where each data transfer is performed synchronously to prevent any overlap between computation and communication. We refer to this configuration as HYBRID–synch thereafter. The third configuration is a traditional software-cache design implementing a 4-way set-associative, 64-KB cache with 128-byte cache lines using a relaxed consistency model We refer to this configuration as TRADITIONAL thereafter. maintained through a dirty bits structure. Data transfers are synchronous, and every global-memory reference is guarded by dedicated control code similar to the one shown in Figure 1b.

Figure 6 shows the execution time in seconds for the IS-B, CG-B, FT-A and MG-A benchmarks. Let us first compare the HYBRID-synch and TRADITIONAL designs. This comparison gives us an indication about which configuration has higher software-cache overhead as both configurations uses blocking DMA requests. Among the 4 benchmarks, FT-A and MG-A, are highly dominated by regular memory accesses. For these two, the large lines (4KB) found in the high-locality cache of the hybrid schemes significantly reduce the number of communications, compared to the 128B lines of the traditional software cache. Also, the hybrid approaches get rid of all lookup code within the innermost loop, whose trip count is





WORK: time spent in actual computation.
UPDATE-DIRTY: time spent updating the dirty-bits info.
DMA-REG: time spent synchronizing the DMA data transfers in the high-locality cache.
TRANSAC.: time spent executing control code of the transactional cache.
MMAP: time spent in executing lookup, placement actions and DMA programming.

WRITE-BACK: time spent in the write back process.
DMA-IREG: time spent synchronizing the DMA transfers in the transactional cache.
DEC.: time spent in the pinning mechanism for the cache lines.
BARRIER: time spent in the synchronization barrier at the end of the parallel computation.

Figure 7. Cache overhead distribution for MG (loop 5) and IS (loop 1).

roughly proportional to the size of 4KB cache line; whereas the traditional approach has one lookup per global memory reference. This results in speedup factors of 3.63 for FT-A and 3.48 for MG-A over the entire applications.

The two other benchmarks, CG-B and IS-B, are highly dominated by irregular memory accesses. These two benchmarks achieve speedup factors of 1.82 for IS-B and 1.66 for CG-B when comparing the execution time of HYBRID-synch versus TRADITIONAL. Interestingly, this speedup factors do not come from reduced miss ratios, as shown in Table 1. Indeed, since both configurations use blocking DMA, the reduced execution time can only be explained by faster software cache primitives found in the hybrid configurations. Specifically, the write-through mechanism used by the irregular cache is far more efficient than a mechanism based on dirty bits and atomicity. Indeed, the dirty-bit mechanisms (such as found in traditional scheme and in the high-locality cache of the hybrid approach) require an atomic operation involving one DMA to read the line, a merge of the modified data, followed by a DMA to write back the modified line. In addition to these two DMA operations, the execution of the control code for the Dirty Bits structure is typically a highly branching code. None of these overheads are found in the transactional cache scheme in the HYBRID design.

Let us now compare, in Figure 6, the TRADITIONAL configuration to the HYBRID configuration, which uses asynchronous DMA commands to enable overlapping communication and computation. We now achieve speedup factors of 5.08 for IS-B, 8.41 for CG-B, 7.21 for FT-A and 3.50 for MG-A. Not surprisingly, the benchmarks dominated by irregular memory references benefit much more from the asynchronous communications, as the transactional cache and associated code transformations attempt to maintain up to 32 DMA miss in flight.

Figure 7 shows the time distribution for two different types of loops: the most time consuming loop of MG-A (a loop that is dominated by the activity in the High Locality Cache) and the most time consuming loop in IS-B, dominated by the activity in the Transactional Cache. For the first case, the actual loop computation takes close to 52% of the total loop execution. Notice how the memory consistency support corresponds to a considerable amount of overhead, close to 29%, divided in to a 19% for updating the dirty-bits information and 9% devoted to the write-back operation. Notice that mapping a cache line to the HLC corresponds to a 7% of total execution time. For the second case, the distribution is very different. Now the actual computation corresponds to a 43%, while close to 33% is devoted to control code on the HLC, plus a 5% of time devoted to synchronize with the DMA engine for transactional data transfers. In this case the memory consistency support corresponds to a lesser percentage, close to 15%.

4.2 Cell BE versus Power 5

This section compares the performance achieved by the POWER5 [14] multi-core processor with a Cell BE processor using the software cache described in this paper on the NAS benchmark. The POWER5 is a dual core processor with 2 SMT threads per core, resulting in a maximum of 4 concurrent threads per chip. The measurements have been obtained on a POWER5-based blade with two processors running at 1.5 GHz, 16 GB of memory (8 GB each processor). Note that we compare here the Cell BE with a server-class general purpose processor with large caches and a high throughput memory subsystem. Moreover, the POWER5 has been optimized for numerical applications such as the ones found in the NAS benchmark. Contrast this to the Cell BE, which has been optimized for mass market production and for a different.

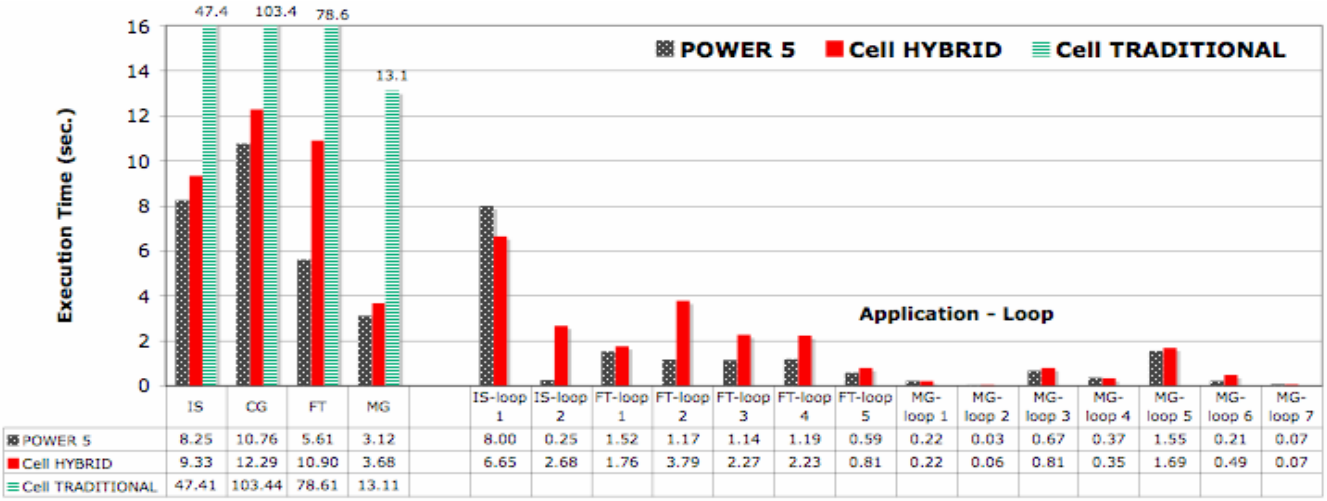


Figure 8. Performance comparison between the Cell BE and the POWER5 processors

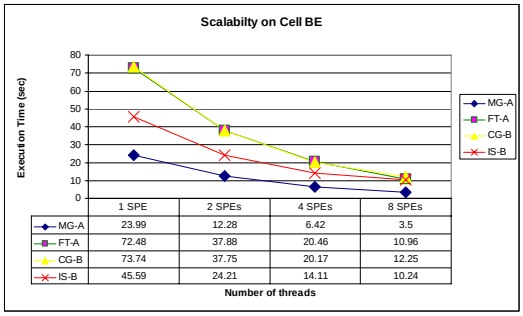
	HYBRID	TRADITIONAL
IS	33.39%	35.46%
CG	17.29%	17.16%

Table 1. MISS ratios for the IS and CG applications.

Leftmost part of Figure 8 shows the execution time in seconds for the evaluated applications. The Cell BE processor is competitive with the POWER5 for three of the four benchmarks, namely it performs within 13, 14, and 18% of the POWER5 execution time for, respectively, IS-B, CG-B, and MG-A respectively. The IS-B and CG-B applications are highly dominated by irregular memory references. For both benchmarks, the Cell BE and the POWER5 suffer from a high degree of communication with main memory. For the POWER5, we know that the working set and access patterns cause a high miss ratio in its cache. The limited the local store in the Cell BE has the same effect in the Cell BE. Though the MG-A application is dominated by regular references, its working set is also too large for the POWER5 cache hierarchy, and thus behave similarly to the two previous benchmarks.

Of all benchmarks, only FT-A performs significantly slower on the Cell BE than on the POWER5. The cause of this discrepancy in performance is due to the following effect. Basically, the FT-A benchmark computes several FFT over a 3-dimensional matrix of complex numbers. The FFT are processed using the following pattern: every plane in a 3-dimensional matrix is evenly cut and every cut is then copied to a temporary variable where the actual FFT computation is done. After that, the output of the FFT is copied back to the 3-dimensional matrix. In the POWER5 version, the temporary variable fits nicely in cache, thus every FFT computation always hits in cache. On the Cell BE, however, FT-A suffers from a much higher degree of communication because the local stores of the Cell BE are much smaller than the cache of the POWER5. Also, the sequential parts of the NAS benchmark applications are executed on the PPE of the Cell BE, which is significantly slower than a POWER5, e.g. the PPE has only a 512KB L2 cache compare to the 1.85MB L2 of the POWER5.

Figure 8 also shows the execution time for each of the dominant loops for each of the four applications. CG-B is excluded from this analysis since its computation is totally dominated by the execution of one parallel loop that takes 95% of overall execution time. For IS-B, there are two main loops: loop 1 is parallel, loop 2 is sequential. Notice that the loop 1 is executed faster in the Cell BE, but the sequential part is much slower in the PowerPC core and that causes the Cell BE being behind the POWER5. The sequential loop performs a reduction operation that collects the data of every thread (one integer vector) into a shared vector. For the Cell BE this implies 8 memory copy operations (there are 8 threads) of 8 vectors that do not fit in the 512KB second level cache of the PowerPC core. In the POWER5, only 4 vectors have to be copied (there are 4 threads) with a much higher locality in the 1.85MB second level cache. The FT-A is composed by 5 loops. Loop 1 is sequential; loops 2, 3, 4 and 5 are parallel. Although the comparison for this application is totally conditioned by the fact that the FFT algorithm uses temporary storage that always is resident in the POWER5 cache hierarchy, it is worth to analyze all the loops. Loops 2, 3, and 4 are the actual loops computing FFTs, and the differences between the POWER5 and the Cell BE are significant. Loop 5 is a parallel loop but its working set does not fit into the cache hierarchy of the POWER5. For this loop, the Cell BE is 25% behind the POWER5. The MG-A application is composed by 7 different loops. All of them are parallel except for loops 2 and 6. Notice that for the Cell BE both loops account for 0.55 seconds (15% of its overall execution time), while for the POWER5 they account for 0.24 seconds (7.6% of its overall execution). The loops 1,



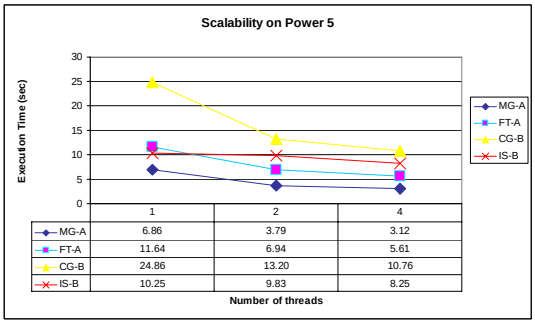


Figure 9: Scalability on the Cell BE and the POWER5

4, and 7 perform the same in the Cell BE and POWER5 unless for loops 3 and 5 with differences of 21% and 8.2%.

We present some scalability numbers in Figure 9. The POWER5 shows good scalability from 1 to 2 threads, but executions with 4 threads do not result in the same level of improvement, because the additional threads are SMT threads sharing a core. The Cell BE shows good scalability from 1 to 8 threads, since every thread executes in an exclusive core.

In conclusion, we observe that Cell BE's performance is very competitive to the POWER5's performance for parallel loops. Both processors perform similarly, as long as the working set does not fit in to the cache memory hierarchy of the POWER5. It is important to remember that both processors have very different costs and constraints. With the technology described in this paper, however, the Cell BE can achieve similar performance as to POWER5.

5. RELATED WORK

Although a different technique, tiling transformations and static buffers may be used to reach the same level of code optimization [1]. 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. When not available at compile time, one could still generate several code versions depending on runtime alignment conditions, at a cost in code size. Or, we can follow the approach in this paper, where we postpone the association between static buffers and memory references until run time. In doing so, we solve all the difficulties related to memory aliasing since the high-locality cache lines are treated as buffers that are dynamically allocated. 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.

Chen et al [17] have further developed an integrated static-buffer and software-cache approach so as to enable the parallelization of loops that include references directed to both structures. Their scheme specifically addresses software-cache references that may potentially use data currently present in static buffer, or vice versa. Their approach attempts to minimize coherency operations using a layered approach including compile time and runtime disambiguation. Further optimizations include prefetching [18].

The *Flexicache* design [11] proposes a software instruction cache. The main issue is how to treat the direct/indirect jumps. The proposed mechanisms, although falling into software caching techniques, are very different from the ones presented in this paper. There is one particular aspect in common to our proposal. The *Flexicache* approach must be able to pin basic blocks in the instruction software cache while there might be jumps that point to them. Our solution is based on counters indicating how many memory references point to a cache line. The *Flexicache* framework uses a different approach.

The work in *HotPages/FlexCache* [9][12] is similar to our work but is not as powerful and simple as the work described in this paper. It relies on mechanisms of considerable complexity and needs of a compiler analysis with some degree of accuracy (e.g: pointer analysis). In addition, it does not provide for a solution where the compiler fully removes the checking code around memory references presumed to reference the hot pages.

Software caching techniques have been applied to reduce the amount of power consumption associated to cache management. These proposals face similar problems as the ones addressed in this paper. For instance, Direct Addressed Caches [8] propose the elimination of the tag checks by making the hardware to remember the exact location of a cache line, so that hardware can access data directly. This proposal requires the definition of new registers in the architecture to relate load/store operation to specific cache lines, leaving to the compiler the decision of what memory references have to be associated to the additional registers. In addition, this proposal requires new load/store instructions to actually make use of the associated registers. The work in *SoftCache* [10] is based on binary rewriting, which requires a complete control-flow

and data-flow analysis of binary images. Clearly this is not the case in the proposal of our paper, which avoids such complexity.

In Udayakumaran et al. [13] data allocation in scratchpad memories is addressed. Although a different context, the problems solved are similar to the ones addressed in this paper. The proposal is based on code region analysis through code profiling, plus frequency analysis of memory accesses. Derived from this information, a timestamp process is applied to basic blocks and specific points in the code are defined where control code is injected to move data between DRAM and SRAM. The solution is able to efficiently treat memory accesses that could be statically associated to a specific code point. Pointer-based accesses and aliasing are not totally solved, since the presented solution explicitly maintains a side structure (e.g: a height tree) for detecting if data is missing in the SRAM and ensure a correct address translation. The proposal in this paper is generic enough to not rely on any profiling information for the compiler to statically allocate data. On the opposite, the proposal in this paper delays at runtime the allocation and mapping decision, and succeeds in removing most if not all the control code surrounding the memory references.

6. CONCLUSIONS

This paper presents a novel, hybrid software cache architecture that maps memory accesses according to the locality they are exposing. Performance evaluation indicates that the hybrid design is significantly more efficient than traditional software-cache approaches, with speedup factors in the 3.5 to 8.4 range. We have also shown that a Cell BE nearly match the computational power of a POWER5 for a set of NAS parallel benchmarks. Recall that the POWER5 is a server-grade multiprocessor optimized for parallel applications, whereas the Cell BE is a low-cost game processor using the SPEs as accelerators surrounding a PowerPC core. We show that, using our approach, it is possible to achieve competitive performance on machines with simple accelerators and dedicated local memories.

ACKNOWLEGMENTS

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

REFERENCES

[1] A. E. Eichenberger *et al.*, "Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture," IBM Sytems Journal, Vol. 45, No. 1, 2006.

[2] M. Kistler *et al*., "Cell Multiprocessor Communication Network: Built for Speed," IEEE Micro, Vol. 26, Issue 3, 2006.

[3] D. Pham *et al*., "The Design and Implementation of a First-Generation CELL Processor," in the Proceedings of the IEEE International Solid-State Circuits Conference, 2005.

[4] M. Gschwind *et al*., "A Novel SIMD Architecture for the CELL Heterogeneous Chip-Multiprocessor," In Hot Chips 17, 2005.

[5] T. Chen *et al*., "Optimizing the use of static buffers for DMA on a Cell chip," in the Proceedings of the International Workshop on Languages and Compilers for Parallel Computing, 2006.

[6] A. E. Eichenberger *et al*., "Optimizing Compiler for a Cell Processor," in the proceedings of the International Conference on Parallel Architectures and Compilation Techniques, 2005.

[7] D. Bailey et al. "The NAS parallel benchmarks," Technical Report TR RNR-91-002, NASA Ames, 1991.

[8] E. Witchel *et al*. "Direct Addressed Caches for Reduced Power Consumption," in the Proceedings of the Annual International Symposium on Microarchitecture, , 2001.

[9] C. A. Moritz *et al*., "Hot Pages: Software Caching for Raw Microprocessors," MIT-LCS Technical Memo LCS-TM-599, 1999.

[10] J. B. Fryman *et al.*, "SoftCache: A Technique for Power and Area Reduction in Embedded Systems," CERCS; GIT-CERCS-03-06

[11] J. E. Miller and A. Agarwal, "Software-based Instruction Caching for Embedded Processors," in the Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems, 2006.

[12] C. A. Moritz *et al*., "FlexCache: A framework for flexible compiler generated data caching," in the Proceedings of the 2nd Workshop on Intelligent Memory Systems, 2000.

[13] S. Udayakumaran *et al*., "Dynamic Allocation for Scratch-Pad Memory Using Compile-Time Decisions," ACM Transactions on Embedded Computing Systems, Vol. 5, No. 2, 2006.

[14] B. Sinharoy *et al.*, "POWER 5 system micro-architecture," IBM Journal of Research and Development, Vol. 49, No. 4/5, 2005.

[15] J. Hoeflinger and B. de Supinski, "The OpenMP Memory Model," in the Proceedings of the First International Workshop on OpenMP, 2005.

[16] P. Altevogt *et al.*, "IBM BladeCenter QS21 Hardware Performance," IBM Technical White Paper WP101245, 2008.

[17] T. Chen *et al.*, "Orchestrating Data Transfer for the Cell B.E. processor," in the Proceedings of the Annual International Conference on Supercomputing, 2008.

[18] T. Chen *et al*., "Prefetching Irregular References for Software Cache on Cell, Proceedings of the sixth Annual International Symposium on Code Generation and Optimization.