Prefetching Irregular References for Software Cache on Cell

Tong Chen, Tao Zhang, Zehra Sura, Marc Gonzalez Tallada, Kathryn O'Brien, Kevin O'Brien
IBM T.J. Watson Research Center, Yorktown Heights, New York, USA
{chentong, taozhang, zsura, mgonzal, kmob, caomhin}@us.ibm.com

ABSTRACT

The IBM Single Source Research Compiler for the Cell processor (the SSC Research Compiler) was developed to manage the complexity of programming the heterogeneous multi-core Cell processor. The compiler accepts conventional source programs as input, and automatically generates binaries that execute on both the PPU and SPU cores available on a Cell chip. The compiler uses a software cache and direct buffers to manage data in the small local memory of SPUs. However, irregular references, such as a[ind[i]], often become performance bottle-necks. These references are accessed through software cache, usually with high miss rates. To solve this problem, we propose a method to prefetch irregular references accessed through a software cache that is built upon hardware such as Cell. This method includes code transformation in the compiler and a runtime library component for the software cache. Our design simplifies the synchronization required when prefetching into software cache, overlaps DMA operations for misses, and avoids frequent context switching to the miss handler. It also minimizes the cache pollution caused by prefetching, by looking both forwards and backwards through the sequence of addresses to be prefetched. We evaluated our prefetching method using the NAS benchmarks. We found that when applicable, our prefetching can improve the performance of some benchmarks by 2 times on average, and by close to 4 times in the best case. We also present data to show the impact of different configurations and optimizations when prefetching in a software cache.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*code generation, compilers, memory management, optimization*

General Terms

Performance, Design

CGO'08, April 5–10, 2008, Boston, Massachusetts, USA.

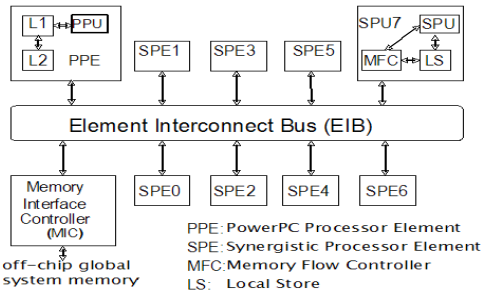

Figure 1: Cell BE Architecture

Keywords

DMA, prefetch, software cache

1. INTRODUCTION

In heterogeneous multi-core systems, reducing hardware complexity and minimizing power consumption are important design considerations. Providing each of the accelerator cores in such systems with its own fast local memory is one means of accomplishing this goal. Typically, such systems will not provide hardware supported coherence between these local memories and the global system memory. When an application (both code and data) fit within the local memory, good performance can be guaranteed. Such a feature is critical for real time applications. The Cell Broadband Engine Architecture (CBEA) [10, 9] is one example of such a heterogeneous multi-core system. It includes on a chip a PPE core, and 8 SPE cores each with 256KB fast local memory, as well as a globally coherent DMA engine for transferring data between local memories and the shared system memory (Figure 1). This novel memory design, suited for generating high performance for a variety of applications, including games, graphics, etc., nonetheless requires careful programming to obtain top performance. Developing techniques to enhance the programmability of these types of architectures is currently an area of active research.

To ease the programming for the Cell architecture, a single source compiler (the SSC Research Compiler) has been proposed by IBM Research [20, 8]. This compiler abstracts the complexity of the underlying memory hierarchy, and presents the programmer with a single shared memory image. It takes conventional source code with OpenMP pragmas as input and generates binaries to be executed on both the PPU and SPUs. In this approach, the programmer can still program using a traditional shared memory programming model, yet still exploit the local memories for efficiency

and performance. The compiler is responsible for managing data transfers transparently, while still ensuring correctness and performance. In the compiler, two complementary strategies for data management are employed: a compiler controlled, software managed cache, and direct buffering of data transfers.

In the compiler managed software cache, a portion of the local memory is allocated for the cache lines. Every load/store to system memory is instrumented with cache related instructions to go through software cache lookup operations, and cache miss handling when needed, at runtime. This approach is able to handle all data references uniformly through the cache and capture data reuse that occurs. However, this is typically an expensive approach since the data cache is completely implemented in software. For each data reference going through software cache, there will be a lookup overhead of approximately twenty cycles. If the lookup misses, miss handling has to be invoked, the overhead of which is an order of magnitude higher due to DMA operations. In practice, additional optimizations have to complement the basic software cache scheme in order to provide reasonable performance.

For regular data accesses, direct buffering optimizations are implemented. In direct buffering, temporary buffers are provided in the SPE local store and DMA is used to transfer chunks of data to be accessed by the SPE to these local buffers. Each reference to a load/store to system memory is then replaced with the direct load/store to the buffer at compile time. In this way, both the software cache lookup cost and the miss handling cost are eliminated. The DMA transfer chunk can adapt to the application instead of being determined by the fixed cache line length. Direct buffering is usually combined with loop blocking to limit the size of the local buffer required. Loop blocking is a combination of loop strip-mining and loop interchanging. As a result, the footprint of data accesses in the innermost loop nest is bounded by the loop blocking factors, which are constants. Since the mapping between references and buffers is done at compile time, direct buffering only optimizes references with regular data accesses and known alias and data dependence information.

In the Cell SSC Research Compiler, direct buffering and software cache are integrated such that direct buffering optimizes regular data accesses and the software cache is used as a fall-back solution. However, irregular references, such as a[ind[i]], frequently become the performance bottleneck in the SSC Research Compiler. These references cannot be optimized by direct buffering and have to be accessed through software cache. Moreover, the cache miss rate for these references is usually high, resulting in a large number of cache misses. The function call overhead for the cache miss handler procedure and the blocking DMA operations issued by the miss handler impose a huge overhead in execution.

To solve this problem, we propose a method to prefetch irregular references for a software cache such as that provided on the Cell architecture. The prefetching we are addressing is quite different from the traditional prefetching for hardware cache [25]. Traditionally, hardware based prefetching has good control of the hardware cache but lacks understanding of the program behavior. On the other hand, software (compiler) based prefetching for hardware cache has a better understanding of the software behavior, but it is constrained by a very limited interface to the hardware cache. In our case, both the data cache and the prefetching scheme are implemented in software and are completely under our control. This is a critical point to differentiate our work, and this property brings us both unique challenges and opportunities. For example, in our case, we can easily implement prefetching strategies involving a lot of interaction between the prefetching scheme and the data cache itself, which has been previously difficult to do. In addition, certain optimization opportunities only exist in our all-software set up. On the other hand, we have to be very careful with the overhead we introduce. The overlapping or efficient synchronization implemented in a hardware cache does not exist in the software cache. Also, we have to consider the characteristics of DMA operations to reduce prefetch overhead and improve performance [15]. We will detail the unique opportunities and challenges in later sections.

In our work, we devise an efficient prefetching method, which integrates code transformation in the compiler and runtime library implementation for a software cache. The compiler applies loop blocking and loop distribution to the loop that contains the irregular reference pattern targeted by our optimization. Loop distribution is intended to separate the original loop into a loop to collect the addresses of the irregular references, and a loop to execute the original computation. Between them, a runtime prefetching call is inserted. In that prefetching call, the irregular references for a chunk of iterations are prefetched and will be used by the following computation loop. This design reduces frequent jumps to the miss handler, avoids complicated synchronization checks to determine if the prefetching data transfer is complete, and overlaps the DMA operations for different misses. Let us call the number of iterations to be prefetched by one prefetching call the *prefetching range*, which is controlled by the loop blocking factor. Since the addresses to be prefetched in the prefetching range are available, we are able to optimize the cache replacement. We try to reduce the cache pollution caused by prefetching by looking ahead to see what cache blocks will be needed in the following iterations, and by looking back to see what cache blocks have just been brought into cache by previous iterations. When traditional loop blocking with a constant block factor is used, the prefetching range is statically determined at compile time. We also propose a special code transformation that enables the prefetching range to be dynamically determined by the prefetching runtime library. The static prefetching range is simple while the dynamic prefetching range can adapt to different cache miss rates, neither underutilizing nor overflowing the cache. Furthermore, we can even remove the cache lookups in the computation loop, if each prefetching call for the dynamic prefetching range stops when the first associativity conflict occurs, and it locks the prefetched blocks as needed.

We evaluated our scheme using the NAS benchmarks. We found that our prefetching can improve the performance of the most time critical loops (those that take up more than 95% of the total execution time) in IS and CG, and one trivial loop in FT, by 2–4 times. We also present the impact of the prefetching range and two pollution reducing optimizations, revealing to readers some insight about prefetching in a software cache.

This paper makes the following contributions:

- A method to support prefetching for software controlled cache, using compiler transformations integrated

with runtime library services. It accounts for high latency DMA operations, the overhead of a software cache implementation, and no hardware support for prefetching. The runtime library services are designed to optimize the use of available DMA operations, and overlap some DMA overhead time with data transfer time.

- A facility to dynamically adjust the amount of data prefetched based on runtime cache usage. This enables executing a loop using alternating phases of prefetch and compute, where each prefetch phase brings in just the amount of data that will maximize cache utilization for that execution instance.

- New cache placement/replacement policies for prefetching in software cache. Since the address collection loop used for prefetching provides knowledge about a window of future accesses, the cache placement/replacement policy can be aggressively optimized. However, experiments show that the policy optimization has to be balanced with the runtime overhead due to implementing these policies in software. Thus, multiple new policies have been developed and evaluated.

While we demonstrate our approach on the Cell processor, the techniques described in this paper are generally applicable to systems with software-controlled memory caches, as discussed in section 5.

Our paper is organized as follows. Section 2 describes the problem within the SSC Research Compiler framework; our solution for prefetching, which integrates both compiler transformations and runtime libraries, is presented in section 3; experimental results are shown in section 4 to demonstrate the efficiency of our solution and to provide some insight about how the prefetching works; a description of related work can be found in section 5; and we conclude this paper with section 6.

2. PROBLEM STATEMENT

In the SSC Research Compiler, global data resides in the main memory. The thread running on SPU can access its local memory directly, or transfer data from main memory to its local memory using DMA operations. The SSC Research Compiler provides mechanisms to automatically manage the DMA and local memory.

Software data cache is the basic mechanism for data management in the SSC Research Compiler. It works in a way similar to a hardware data cache, but it is implemented in software. In the current SSC Research Compiler implementation, the software data cache is a four-way associative cache and has a block size of 128B and 512 blocks, giving a total size of 64KB. There is a cache directory recording meta-data for the cache. It contains three major pieces of information for a cache block: cache tag, data pointer and dirty bits. The cache tag records the system memory address for the data in the cache block, as in a hardware cache. The data pointer contains a pointer to a 128B local store space, recording where the cache block is in the local store. Dirty bits for a cache block record which bytes in the block have been modified by this thread. Since there is no hardware cache coherence, dirty bits are mandatory for proper coherence maintenance in a multi-threading environment. The cache directory can be further extended to include other information such as special flags to lock a cache block.

The compiler replaces loads and stores to system memory in the SPE code with instructions that explicitly look up the system memory address in the cache directory of the software cache. If a cache block for the system memory address is found in the directory (which means a cache hit), the value in the cache is used. Otherwise, it is a cache miss. For a miss, the miss handler function is invoked. The miss handler needs to allocate space for the incoming cache block. If there is an unused block in the cache set, the miss handler simply uses it. If not, the miss handler selects and evicts a block. If every bit has been modified in the block to be evicted, then a DMA put operation is used to perform the eviction. Otherwise, an atomic update operation supported by the SPE DMA engine is used. After cache block eviction, DMA is issued to bring in data for the miss. The miss handler simulates a FIFO replacement policy by rotating the blocks in a cache set.

It is expensive to use the software data cache, and it incurs significant runtime overhead due to the cost of cache lookups and miss handling. Some data references are regular references from the point-of-view of compiler optimizations. These references occur within a loop, and the memory addresses that they refer to can be expressed using affine expressions of loop induction variables. For such regular accesses to shared data, we avoid using the software cache, and apply the direct buffering optimization instead. Direct buffering allocates temporary buffers for regularly accessed data in the SPE local store. For read references, it initializes the buffer with a DMA get operation before the loop executes. For write references, it writes out the data from the buffer using a DMA put operation after the loop finishes execution. The compiler statically generates these DMA get and put operations. The compiler also transforms the loop body so that the SPE computation code directly accesses the local buffer without incurring any software cache overhead. Furthermore, DMA operations can be overlapped with computation by using multiple buffers. The compiler can choose the proper buffering scheme and buffer size to optimize execution time and space [3].

We will illustrate irregular data accesses using an example extracted from CG in the NAS benchmark suite. This loop, at line 435 in procedure conj_grad, is the most time-consuming loop in this benchmark. It is the innermost loop in a two level loop nest. The loop, with normalized and simplified loop boundaries, is shown in Figure 2(a). The references a[k] and colidx[k] are regular references and can be optimized with direct buffering. To limit the size of the buffer required by direct buffering, this loop is blocked into two loops. Buffers are allocated and freed, DMA operations are added, and the corresponding references are replaced. The buffer management and DMA operations are supported by runtime libraries. The high-level output code with block factor BF is illustrated in Figure 2(b).

Direct buffering is an efficient way to make use of the local memory. Large chunks of data are transferred through DMA operations. The data in local memory is directly accessed without extra overhead. Direct buffering can optimize references with continuous addresses (stride-1) and regular discontinuous addresses (stride-n). However, references with complicated control flow or data dependence cannot be handled by direct buffering. Instead, the software cache has to

```
//LOOP#1
  for (k = 0; k < k_ub; k++) {
    sum = sum + a[k]*p[colidx[k]];
  }
```
(a) simplified original source

```
allocate direct buffer, db1 for a;
allocate direct buffer, db2 for colidx;
//LOOP#1.1
for (kk = 0; kk < k_ub; kk+=BF) {
  new_ub = min(k_ub, (kk+1)*BF)
  DMA read a[kk*BF:new_ub] to db1;
  DMA read colidx[kk*BF:new_ub] to db2;
//LOOP#1.2
  for (k = kk*BF; k < new_ub; k++) {
    sum = sum +
      db1[k-kk*BF]*p[db2[k-kk*BF]];
   }
}
```
(b) after direct buffering optimization

Figure 2: Example extracted from CG

be used. The software cache is able to exploit data reuse at runtime, with extra cache lookup overhead for every reference, and cache maintenance overhead for every miss. As long as data locality is good, the performance of the software cache is acceptable.

However, irregular reference patterns may be a problem for both direct buffering and the software cache. The subscripted subscript array is a common example. In the previous example, p[db2[k-kk*BF]] is such a reference. If the values of the index array, namely the subscript, scatter without much locality, it will result in a high miss rate. Such cache misses with irregular reference patterns are a problem for traditional hardware caches. This problem is even worse for a software cache due to:

- Overhead of blocked DMA operations. The program has to wait for the DMA transfer to finish before proceeding. This is similar to stalls for cache misses in an in-order machine.
- Overhead of frequent jumps to the miss handler function. Each cache miss is served by a miss handler function, which incurs an overhead for context switching.

Prefetching is a technique that can be used to reduce the overhead of cache misses. It is applied to data accesses that are likely to exhibit a high miss rate, and it works by issuing ahead-of-time cache requests for data that is expected to be accessed some distance further along in the computation. Prefetching helps because it enables overlapping cache miss latency with ongoing computation, thus reducing the amount of execution time wasted stalling for data transfers in the memory hierarchy. Prefetching can be done in hardware or software, but hardware prefetching generally targets only regular or patterned accesses, while we target irregular accesses. Our solution uses software prefetching, but unlike traditional software prefetching techniques, it aggregates a number of prefetch requests across multiple loop iterations, and issues all of them together. This design is important considering that we target a system with no hardware support for caching or speculation, instead using a software cache and software-directed DMA commands

```
//LOOP#1.2.1 address collection loop
for (k = kk*BF; k < new_ub; k++) {
      addr_buf[k-kk*BF] = &p[db2[k-kk*BF]]
}
prefetch with addr_buf;        //runtime calls
//LOOP#1.2.2 computation loop
for (k = kk*BF; k < new_ub; k++) {
      sum = sum + db1[k-kk*BF]*p[db2[k-kk*BF]];
}
```

Figure 3: CG loop after transformation

for all data transfers. Traditional software prefetching uses special prefetch commands provided by the hardware cache and may use a separate thread to issue prefetch requests. In the Cell SPE, the context switching overhead for multiple threads can be very high, so we transform code to introduce DMA transfers for prefetching earlier on within the same instruction sequence. Since DMA transfers are software-directed, all synchronization for completion of data transfers and data availability has to be handled in software, and the overhead for doing this can be high relative to using hardware cache prefetch mechanisms. Furthermore, lack of support for speculative execution requires that we ensure the prefetch requests are issued for valid memory addresses. This precludes prefetch optimizations that speculate on the addresses to prefetch data from.

While the use of DMA and a software cache makes our prefetching problem more challenging, it also uncovers more opportunities for optimization. Since DMA transfers use a very high bandwidth bus, we can overlap the transfer of a large amount of data when prefetching. Also, since the cache is completely under software control, we can use compiler and runtime analysis to adapt the cache placement and replacement policies used when prefetching data.

3. PREFETCHING SCHEME

Our solution contains two parts: compiler transformation and runtime library support. The compiler transformation splits the loop containing an irregular reference into two loops. The first loop, called the address collecting loop, collects the addresses of all data accessed by the irregular reference. This loop must gather exactly those addresses used in the original loop, since we do not use speculation when issuing DMA commands for prefetching. The second loop, called the computation loop, performs the computation in the original loop. Between these two loops, the compiler inserts a call to the runtime library to try and prefetch all the addresses collected. As a result, most cache misses that would have occurred in the original loop will be handled in the runtime prefetching call, and the irregular reference in the computation loop will incur less overhead from jumping to the miss handler routine. Using a single runtime call to handle prefetching for multiple irregular references allows us to overlap DMA operations. It also simplifies synchronization for these multiple DMA operations since the prefetched values will not be needed until the second computation loop begins execution. We use Loop 1.2 as an example to demonstrate our prefetching method in Figure 3.

It is possible for a loop to contain more than one irregular reference. If the addresses accessed by these references do not depend on each other, we will be able to collect addresses

for multiple irregular references in a single address collecting loop. If they do depend on each other, we will have to generate multiple levels of prefetching. For example, if we have expressions such as a[b[ind[i]]], or *(*(p+i)+k) where p is a two-level pointer, we need multiple levels of prefetching. In this case, the computation loop of the n^{th} level and the address collection loop of the $(n+1)^{th}$ level can be merged.

It is important to notice that in our method, one runtime call for prefetching will prefetch data for a number of loop iterations, not just a single reference. The prefetching range is the number of iterations to prefetch data for in one runtime prefetching call. The prefetching range for a loop does not have to be the same as the block factor for direct buffering. Loops can be further blocked to allow for a smaller prefetching range. However, the prefetching range cannot be larger than the block factor for direct buffering because the size of the index array used for collecting prefetch addresses is limited by the block factor used in direct buffering. When the prefetching range is determined at compile time, it is called static prefetching range. We will focus on this simple case first. The loop can also be transformed in a way such that the prefetching range is controlled dynamically at runtime. We will discuss the use of a dynamic prefetching range and its advantages in sections 3.5 and 3.6.

3.1 Code Transformation

To apply our software prefetching technique, we first need to identify memory accesses in the code that are suitable targets, i.e. memory accesses that are irregular references and that suffer from high miss rates. Currently, we use pattern matching to identify irregular references. Also, though not incorporated in our system yet, profiling can help detect references with high miss rates.

To determine which irregular references to prefetch, we consider all memory references within the innermost normalized loops, i.e. loops that are suitable targets for direct buffering optimization. If the address accessed by a memory reference is not computed as an affine function of the loop index variable, and there are no loop-carried dependencies between statements used in the address computation, then that memory reference is a candidate for our optimization. For simplicity, we apply our prefetching optimization only for loops where all memory accesses are covered either by direct buffering or by the prefetching optimization. However, our technique is also applicable to loops containing residual memory references that are accessed through the default software cache mechanism.

Once we have identified loops containing irregular memory references to target, we transform the code for these loops by distributing them into an address collection loop and a computation loop. The address collection loop comprises of all statements that contribute to computing the address of the irregular memory references. A store statement is introduced into the address collection loop. This statement records addresses accessed by the irregular memory reference across all loop iterations, writing them into a temporary array, addr_buf. The loop iteration variable is used to index addr_buf, and it determines the element of addr_buf that contains the address of the irregular access in the corresponding loop iteration. The second computation loop is a copy of the original loop. It is possible to optimize the computation loop by applying a transformation analogous to common subexpression elimination, and removing redundant computation statements common to both loops. However, we do not consider such optimizations in our initial implementation. We insert runtime library calls for prefetching between the address collection loop and the computation loop.

```
pf_register(char *addr_buf, int data_size);
  addr_buf: the address of the array containing reference addresses
  data_size: number of bytes to be prefetched in each iteration

pf_do(int iter_num);
  iter_num: prefetching range, i.e. the number of iterations to prefetch for
```

Figure 4: Interface of prefetching runtime library

3.2 Runtime Library Interface

To prefetch data for accesses due to one irregular reference in the loop, the runtime must know the address array and the size of data to prefetch for addresses recorded in this array. To prefetch data for more than one irregular reference in the loop, we define two library functions, pf_register and pf_do. pf_register is used to record information needed to prefetch data for a single irregular reference. pf_do is used to actually perform prefetching. Compiler code transformation will insert a call to pf_register for each irregular reference in the loop, followed by a single call to pf_do. The interface definition for pf_register and pf_do is in Figure 4.

Each pf_register call simply saves all its parameters into an array of structures used to record this information for the subsequent pf_do call. When pf_do is invoked, it executes a loop that iterates over the prefetching range. In each iteration, all registered reference addresses are checked to see if data corresponding to the address already exists in the software cache. If it is not in cache yet, prefetching is performed. We will discuss details about prefetching in the following subsections. In pf_do, requests to prefetch for multiple references are interleaved without bias for any reference. Also, for addresses corresponding to a single reference, the order of prefetching follows the order in which data is used in iterations of the computation loop.

3.3 Optimizing DMA Operations

Each processing element on the Cell chip is connected to a high bandwidth (up to 96 Bytes/cycle) Element Interconnect Bus (EIB) via links that can support up to 16 Bytes/cycle [15]. Each DMA request can be issued with a 5-bit tag without blocking, and then a DMA wait operation for that tag can be executed to guarantee that DMA requests with that tag have been completed. The DMA engine can buffer up to 16 DMA transfer commands, thus allowing concurrent data transfer on the EIB bus for up to 16 DMA requests issued in the SPE code. Since the DMA engine can process and queue multiple DMA requests, it is possible to overlap the latency of subsequently issued DMA requests with the data transfer latency of previously issued DMA requests.

To make prefetching fast, we use two tags and non-blocking DMA. We issue 8 DMA requests with tag 1, then wait for completion of DMA requests previously issued with tag 2, then issue another 8 DMA requests with tag 2, and then wait for DMA requests previously issued with tag 1. This is done repeatedly in a cycle for successive DMA operations. The

```
look up the current reference in the cache;
if (it is a hit at block[hit_index]) {
   if (hit_index >pf_top) {
      swap block[pf_top] and block[hit_index];
       pf_top++;
   } else if (hit_index == pf_top) {
      pf_top++;
   }
} else { //it is miss
   if (pf_top == 4) {// associativity conflict
      skip this reference;
   } else {
      replace block[pf_top] and prefetch this
reference;
      pf_top++;
   }
}
```

Figure 5: Algorithm for look-back replacement

performance advantage of issuing multiple ongoing DMA transfers has to be balanced with the overhead of issuing DMA wait operations. In our experiments, we find that using 8 ongoing DMA transfers with 2 tags for DMA wait synchronization performs better than using the maximum of 16 ongoing DMA transfers with 16 tags for synchronization.

3.4 Reducing Cache Pollution

Prefetching may pollute the cache if the prefetched blocks cause some useful data to be evicted. In our prefetching method, there is no speculation and all prefetched data is useful. No unnecessary blocks are prefetched into the cache. However, we prefetch a bunch of references at once and consume them later. Such burst prefetching increases the likelihood that pollution will occur. For ordinary cache accesses, the software cache simulates the FIFO policy by rotating the blocks in a set whenever a cache miss is serviced. In this section, we describe new policies that can reduce pollution when used in conjunction with our prefetching mechanism.

In general, the optimal replacement policy is to replace the block that will be used farthest in the future. Implementing this policy needs knowledge of future references. Fortunately, we know the sequence of addresses to be prefetched within the prefetching range when pf_do is called. This provides at least partial knowledge of future references for optimization. In many cases, most other references in the loop are optimized by direct buffering, and as a result the major impact of pollution is primarily on prefetched data itself. Thus, we can potentially minimize pollution if we use smart cache placement when prefetching. Since we prefetch a bunch of references together without immediately using any of them, we do not want to replace a block that just has been brought in by prefetching. If that block is replaced, it will cause a miss in the computation loop, which may replace another prefetched block that has not been used as yet. It is definitely a bad decision to replace a prefetched block. In summary, in order to reduce pollution due to prefetching, we have to look ahead to determine which blocks will be used, and look back to determine which blocks have just been prefetched.

It is expensive to check which block is the farthest reference each time a block is to be replaced. Instead, we can approximate this policy with one scan of the addresses collected within the prefetching range. Before any prefetch requests are issued, all the addresses to prefetch are looked up in the cache. The hit blocks, namely those that are to be used in the computation loop, are sorted in the set as follows: for an N-way cache, the blocks from nearest use to farthest use are placed from way[N-1] to way[0]. This is the look-ahead policy. To track which blocks have been prefetched, we also allocate an integer variable, called pf_top, for each cache set and initialize it to zero. Our policy will maintain that if pf_top equals to k, the ways from 0 to k-1 are blocks that are either a hit block or a block brought into the cache in the last prefetching call. We call this policy the look-back policy. When another block needs to be prefetched into a set with pf_top=N, it results in an associativity conflict. In such cases, we cannot replace any block in the set and have to skip prefetching the reference, because all ways are occupied and needed in previous iterations. The skipped prefetches will result in cache misses in the computation loop. When using the default look-back policy, these cache misses are serviced by always evicting way[0] of the corresponding set. The placement algorithm that uses both look-ahead and look-back can be proven to be optimal when all cache accesses in the computation loop are those that have been subject to our prefetching optimization. Details of the look-back algorithm can be found in Figure 5.

The optimal policy, however, is quite expensive to implement. The look-ahead part is especially costly because it does not overlap with DMA operations. The look-back policy may be overlapped with DMA operations, and it is also necessary for detecting associativity conflicts when using a dynamic prefetching range (as discussed in section 3.5). In our experiments, we also tried to use the look-back policy separate from the look-ahead policy. The drawback of using just the look-back policy is that it may keep evicting and using way[0]. We change the original algorithm for look-back policy so that we always evict way[N-1], and then either rotate the ways from pf_top to N or directly swap way [N-1] and way [pf_top]. We call these policies "look-back+rotate" and "look-back+swap" respectively.

3.5 Dynamic Prefetching Range

Prefetching range is an important parameter in our prefetching scheme. On the one hand, we want the range to be large so that we have more DMA operations to be overlapped, a longer address sequence for replacement optimization, and better amortization of the prefetching call overhead over a large number of iterations. On the other hand, the prefetching range cannot be too large. If the prefetching range and hence the number of prefetches increase, some references may not be prefetched due to associativity conflicts. This will result in expensive cache misses in the computation loop. As we will see in the experimental results, different programs, or even the same program with different inputs or in different phases of execution, may reach peak performance when using different prefetching ranges. The main reason is that different loops have different cache miss rates and data locality patterns. We propose using a dynamic prefetching range to adapt our prefetching method to the runtime behavior of programs.

The key idea is that the prefetching runtime library will dynamically determine what prefetching range to use. The runtime library decides when to stop issuing DMA commands for prefetching, and move on to executing the next

```
//LOOP#1.2.1 address collection loop
for (k = kk*BF; k < new_ub; k++) {
  addr_buf[k-kk*BF] = &p[db2[k-kk*BF]]
}
pf_register(addr_buf, 8);
current_lb = 0;
do {
  stop_point = pf_do_dynamic(current_lb, new_ub);
  //LOOP#1.2.2 computation loop
  for (k = current_lb; k < stop_point; k++) {
    sum = sum + db1[k-kk*BF]*p[db2[k-kk*BF]];
  }
  current_lb = stop_point;
} while (current_lb <new_ub)
```

(a) Dynamic prefetching range

```
//LOOP#1.2.1 address collection loop
for (k = kk*BF; k < new_ub; k++) {
  addr_buf[k-kk*BF] = &p[db2[k-kk*BF]]
}
pf_register(addr_buf, 8, cache_buf, write_flag);
current_lb = 0;
do {
  stop_point = pf_do_dynamic(current_lb, new_ub);
  //LOOP#1.2.2 computation loop
  for (k = current_lb; k < stop_point; k++) {
    sum = sum + db1[k-kk*BF]*(* ((double *) cache_buf[k-kk*BF]));
  }
  current_lb = stop_point;
} while (current_lb <new_ub)
```

(b) Remove lookup

Figure 6: Examples for dynamic prefetching range and lookup removal

set of iterations of the computation loop. The interface for the previous pf_do is accordingly changed to a new interface, called pf_do_dynamic:

int pf_do_dynamic(int pf_lb, int pf_ub)

The call to pf_do_dynamic tries to prefetch from lower bound pf_lb up to upper bound pf_ub, but may stop early depending on certain runtime conditions. The function returns the iteration at which it stopped prefetching. This return value is used as the pf_lb value in the next call to pf_do_dynamic so that prefetching can continue from that iteration. Also, the code for the computation loop is transformed to enclose the computation loop within a new do-while loop that also includes a call to pf_do_dynamic in each iteration. The lower bound and upper bound of the computation loop are now determined by prior calls to pf_do_dynamic. The code for the previous example when transformed to use a dynamic prefetching range is shown in Figure 6(a).

Various rules can be designed for the stopping condition in calls to pf_do_dynamic. For example, we can monitor how many blocks have been prefetched, or how the execution time changes with the change in range in previous invocations. In this paper, we focus on one rule: stop prefetching when the first associativity conflict occurs, i.e. in an n-way associative cache, stop when prefetching requires the $(n+1)^{th}$ block in a set. We call this rule the no-conflict rule. It can be efficiently implemented with our look-back policy. The no-conflict rule minimizes the references skipped by prefetching, and allows further optimization in the program, as discussed in the following section.

3.6 Eliminating Cache Lookup

With the transformation for prefetching, we end up with two cache lookups for each prefetched reference: one lookup in the prefetching function and one in the computation loop. We can avoid the lookup in the computation loop with the no-conflict rule for dynamic range. For each prefetched reference, if the local store address of its cache block with proper offset is recorded in an array of pointers, the references in the computation loop may be replaced with a direct reference from this array of pointers. To apply this transformation for eliminating lookups in the computation loop, we must ensure that the corresponding reference is always prefetched into cache by the runtime, and that the prefetched data is not evicted from cache before its use in the computation loop. The no-conflict rule guarantees that every reference in the following computation loop is prefetched when the maximum number of cache blocks used in one loop iteration is no more than the cache associativity. To prevent premature eviction of prefetched blocks from the cache, we can ensure that there are no other references through software cache in the computation loop, i.e. all references are either through direct buffers or are optimized via our prefetching scheme. Another method to prevent premature eviction of prefetched blocks is to lock the prefetched blocks in cache, taking care to leave at least one block in each set unlocked and available for replacement.

For eliminating cache lookups, in addition to the change in the runtime library, the compiler needs to transform the code in two ways:

- Add two new parameters for pf_register. An array of pointers, called cache_buf, and a write flag are added. The array of pointers (cache_buf) is used to record the local store addresses of the prefetched data. The write flag tells the runtime library to set the dirty bits if the reference is a write. This is necessary because in our implementation the dirty bits modification is ordinarily coupled with the cache lookup, which is now eliminated.

- Remove the cache lookup for the prefetched reference in the computation loop, and replace it with a direct reference using the corresponding cache block pointer from cache_buf.

The previous example with lookup removed is illustrated in Figure 6(b).

4. EXPERIMENTAL RESULTS

We implemented our prefetching method using the IBM SSC Research Compiler for the Cell BE as a base. We inserted our code transformation in the component of this compiler that performs inter-procedural optimizations based on a high level intermediate representation. We also incorporated our prefetching runtime library into the existing software cache library for the SSC Research Compiler. We performed our experiments on a Cell blade with two 3.2GHz Cell processors and 1G memory. We used the NUMA control command, *numactl*, to bind our execution to the Cell processor at slot 0, and used a single SPU in this processor to run our experiments. Although our binary is able to use all 16 SPUs in the two Cell processors, we focus on results with one SPU in this paper. The impact on bus bandwidth,

Table 1: Loop characteristics

	type size (bytes)	no. of reads	no. of writes	miss rate	% of overall time
CG/L-A	8	1	0	41.9%	96%
CG/L-B	8	1	0	89.2%	96%
FT/L-1	8	1	0	26.2%	0.8%
FT/L-2	8	1	0	38.4%	1.0%
FT/L-3	8	1	0	41.0%	1.2%
FT/L-4	8	1	0	47.0%	1.4%
FT/L-5	8	1	0	44.7%	1.4%
FT/L-6	8	1	0	45.8%	1.5%
IS/L-1	4	1	1	94.1%	94%

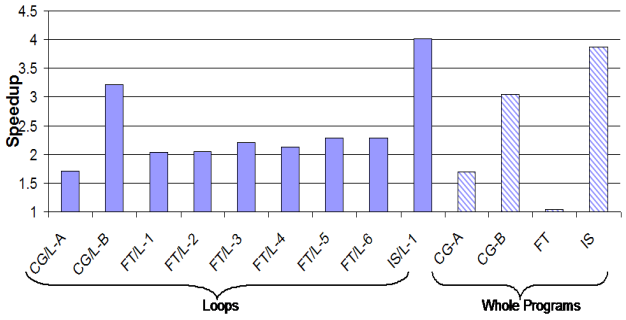

Figure 7: Speedup due to prefetching

memory banks, etc. when prefetching with multiple threads is an interesting topic for future work.

We evaluated our prefetching method using OpenMP versions of the NAS benchmarks. We found that prefetching occurs in three loops, one each in the benchmarks IS, CG and FT. The loops in IS and CG are the most time-consuming loops in the benchmarks, while the loop in FT is not a critical loop. We tried two data sets, CLASS A and CLASS B. We report the result of CLASS A, with the additional result of CLASS B for CG. The cache behavior of CG changes dramatically from CLASS A to CLASS B, and our algorithm adapts to changes in the input set, as illustrated by our experimental results. We also found that the loop in FT has 6 invocations, each of which has a different cache behavior. We report them separately to show how our method works with phase changes. The data access pattern in these loops is simple: only one datum in each iteration is accessed through software cache, and only one-level prefetching is needed. We list the characteristics of these loops in Table 1.

First, in Figure 7, we report the speedup with prefetching using the prefetching configuration that is best for each individual loop (9 loops in all), and each whole program (CG-A, CG-B, FT, and IS). The base line is the program optimized with software cache and direct buffering, but without prefetching. All the loops show significant improvement in performance, with speedups ranging from 1.7 to 4. With the most important loop optimized with prefetching, the performance of the whole program of CG-A, CG-B, and IS are significantly improved. The loops with high miss rate, CG/L-B and IS/L-1, obviously benefit more from prefetching than other loops. We will compare different prefetching methods in the following discussion.

We first consider the effect of using a static value for the prefetching range. Figure 8 shows the effect on execution

Table 2: Performance of dynamic range

	Normalized exec time			
	dynamic range only	with lookup removal	prefetch range	cache block usage
CG/L-A	1.10	0.99	116	21.1%
CG/L-B	1.10	1.01	103	19.7%
FT/L-1	1.04	0.93	155	22.3%
FT/L-2	1.02	0.92	113	19.3%
FT/L-3	1.07	0.97	120	20.7%
FT/L-4	1.04	0.94	68	12.1%
FT/L-5	1.08	0.97	127	20.9%
FT/L-6	1.06	0.96	113	20.9%
IS/L-1	1.11	0.98	133	25.8%

time when using various statically determined values for the prefetching range. The value plotted is the percentage increase in execution time compared with the minimum time across all static prefetching ranges. We observe that different loops reach their minimum execution time at different values of the static range, and they respond differently to the change in prefetching range. Since the iteration count of each invocation of the loop in CG is relatively small, the performance of CG stabilizes once the prefetching range is large enough. Since CG/L-B has a higher miss rate than CG/L-A, CG/L-B requires a smaller range (128) than CG/L-A ($\geq$ 256). Loops with higher miss rates, such as FT/L-4, FT/L-5, FT/L-6 and IS/L-1, are quite sensitive to the prefetching range because cache sets, or even the entire cache may be full.

For dynamic range, we experimented with the no-conflict heuristic, which stops prefetching at the first associativity conflict. We compare its execution time with the best static range, and determine the average number of iterations for the dynamic prefetching range. The result is shown in Table 2. The execution time for dynamic range prefetching (in the first column) shows that the dynamic range can perform very close to the best static range prefetching, with at most 11% degradation in the worst case. The dynamic range determined by this heuristic is around 120 iterations, and the cache block usage is about 20%, with FT/L-4 being an outlier for this statistic. The index value in FT/L-4 is multiplied by 4, and as a result the addresses accessed are more likely to conflict in a set when using the hash function in our cache mapping implementation. As described in section 3.6, it is possible to eliminate cache lookups in the computation loop. The performance improvement after the removal of lookups, compared to the best static range performance, is shown in the second data column. All loops, except CG/L-B, now run faster than the best static range prefetching. Thus, it is profitable to use dynamic prefetching ranges.

Now we investigate the use of different replacement policies for reducing pollution. We use the optimal policy, with look-ahead and look-back in the access sequence, as the baseline to compare with future-1x, lookback, lookback+rotate and lookback+swap policy. The future-1x policy is a variant of the look-ahead policy that examines not all the addresses collected in the address collection loop, but only those addresses contained within the next n iterations, where n is the value of the last dynamic prefetching range. We chose to experiment with future-1x rather than the look-ahead policy because the latter proves to be very expensive. We report

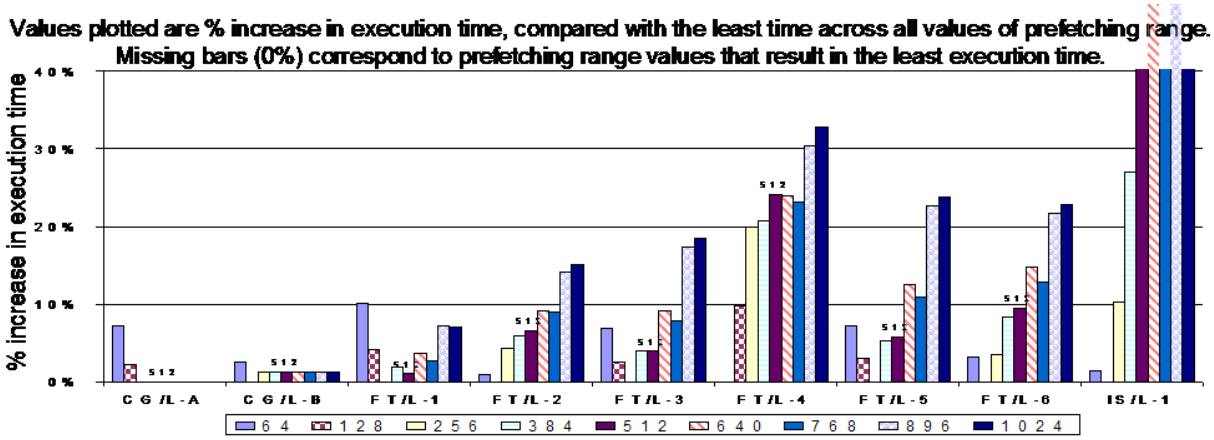

Figure 8: Effect of prefetch range on the performance of basic prefetching

Table 3: Comparison of different replacement policies

	Normalized execution time				Increase in cache misses			
	future-1x	lookback	lookback +rotate	lookback +swap	future-1x	lookback	lookback +rotate	lookback +swap
CG/L-A	0.91	0.64	0.71	0.64	3.1%	8.9%	6.3%	6.6%
CG/L-B	0.89	0.70	0.79	0.69	1.1%	1.9%	1.7%	1.8%
FT/L-1	0.51	0.44	0.40	0.37	19.6%	73.2%	15.2%	14.2%
FT/L-2	0.49	0.40	0.37	0.34	15.2%	39.5%	16.5%	11.0%
FT/L-3	0.52	0.42	0.40	0.37	10.7%	29.1%	9.8%	7.5%
FT/L-4	0.42	0.28	0.28	0.25	19.6%	31.5%	10.5%	9.1%
FT/L-5	0.52	0.45	0.41	0.38	8.3%	24.4%	7.7%	5.9%
FT/L-6	0.51	0.44	0.39	0.36	6.0%	20.0%	12.0%	6.3%
IS/L-1	0.57	0.48	0.55	0.48	4.3%	4.7%	4.6%	4.6%

both normalized execution time and the percent change in the number of cache misses in Table 3. We used dynamic prefetching ranges when collecting this data.

The optimal policy is only optimal in the number of misses, but far from optimal in execution time. Other simpler policies are much faster, up to 4 times faster in FT/L-4 with lookback+swap policy. This is due to the high overhead of implementing the optimal policy in software. From the statistics on the number of misses, we observe that the simple policies have up to 70% more cache misses. As the hit rate increases, and therefore the scope for prefetching optimization decreases, the optimal policy fares better compared to other policies in terms of reduction in the number of cache misses. The lookback+swap policy is effective in reducing both the cache misses and the cost of implementation.

Overall, we find the best prefetching configuration to be one that uses a dynamic prefetching range with lookup removal, and a lookback+swap replacement policy.

5. RELATED WORK

There has been a lot of research done on cache prefetching [25]. Normally different cache prefetching techniques target different memory reference patterns. There are four commonly studied memory reference patterns in previous approaches: stride-1 references [13], stride-n references [19, 21, 2, 12], linked references [26, 12, 17, 22, 16] and irregular references [26, 12, 22, 23, 1, 6, 5, 11]. In our compiler, stride-1 and stride-n references are already handled by our direct buffering optimizations and they will not go through software cache. Linked references are not the focus of the work presented in this paper. Our prefetching technique targets irregular references such as A[B[i]].

We find that previous work uses one of three general approaches to tackle irregular references. First, memory references may be designated to be dependent such that an access to one reference triggers a prefetch for another reference. This approach is exemplified by work done in [26, 22]. Second, the address for the irregular reference may be pre-computed for prefetch using a separate execution thread that runs concurrent with the main execution [12, 1, 6, 5]. Third, an inspector-executor model may be used with coordinated inspector and executor execution such that the inspector computes and prefetches data before the executor uses that data in the main computation [23, 7, 11]. In general, the solution we present in this paper follows this third approach. However, unlike any prior work that we are aware of, our solution integrates with a software cache, and uses novel techniques to optimize cache usage. The combination of an address-gathering "inspector" loop and a software cache gives us unique optimization opportunities, and we exploit these in our solution to dynamically adapt the degree of prefetching based on cache utilization, and to improve the cache placement and replacement policies.

The use of a software cache allows us to better control the cache usage when prefetching, but it also presents additional challenges due to the overhead of software implementation. This overhead, together with lack of hardware support for prefetching in the Cell architecture, results in a different cost model and different design choices for our prefetching. For example, DMA operations in the Cell always expect valid system memory addresses, and the latency of DMA transfers

is high compared to conventional memory systems. Thus, unlike hardware supported prefetching techniques [1, 6, 5], we do not speculatively prefetch data in our case.

This work has focused on evaluating our prefetching technique for a software cache in the Cell BE system. However, our method is also applicable to software controlled caches on other systems, such as the MIT RAW microprocessor [18], the Stanford VMP multiprocessor [4], as well as runtime management of scratchpad memory in embedded processors such as the ARMv6, IBM 440 and 405, Motorola's MCORE and 6812, TI TMS-370, and the Intel IXP network processor. Prior work [24] has explored dynamically managing data contained in scratchpad memory based on compiler analysis of the application code, but this work targets temporal re-use patterns, and does not consider prefetching optimizations for accesses to large arrays. The work in [14] does perform array prefetching for scratchpad memories, but it applies only to regular array accesses, and it statically determines memory usage based on compile-time heuristics.

6. CONCLUSIONS

A prefetching method targeting a software cache on the Cell processor is presented in this paper. This method consists of compiler transformations and runtime library support. The compiler will identify the irregular references to target, and distribute each of the loops containing them into an address collection loop and a computation loop. A prefetching call for all the addresses collected is inserted between the two loops. In this way, we can overlap DMA operations for prefetching and reduce the overhead of context switching for the cache miss handler. The prefetching range can be determined statically or dynamically at runtime. We also design various replacement policies that can be used with our prefetching technique.

The experiments with NAS benchmarks show that our prefetching can speed up some of the benchmarks up to 4 times. The no-conflict rule for dynamic range can achieve performance close to the best static range. It can also eliminate some lookup cost, which can reduce execution time by about 10policies, the simple ones may introduce more cache misses, but they have better performance compared to the optimal policy. This indicates that we still have potential for improvement.

7. REFERENCES

[1] M. Annavaram, J. Patel, and E. Davidson. Data Prefetching by Dependence Graph Precomputation. *28th Annual International Symposium on Computer Architecture*, pages 52–61, July 2001.

[2] J. Baer and T. Chen. An Effective On-chip Preloading Scheme to Reduce Data Access Penalty. *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, November 1991.

[3] T. Chen, Z. Sura, K.M. O'Brien, and J.K. O'Brien. Optimizing the Use of Static Buffers for DMA on a CELL Chip. *Workshop on Languages and Compilers for Parallel Computing*, 2006.

[4] D.R. Cheriton, G.A. Slavenburg, and P.D. Boyle. Software-controlled Caches in the VMP Multiprocessor. *SIGARCH Computer Architecture News*, 14(2):366–374, 1986.

[5] J. Collins, D. Tullsen, H. Wang, and J. Shen. Dynamic Speculative Precomputation. *34th International Symposium on Microarchitecture*, pages 306–317, December 2001.

[6] J.D. Collins, H. Wang, D.M. Tullsen, C. Hughes, Y. Lee, D. Lavery, and J.P. Shen. Speculative Precomputation: Long-range Prefetching of Delinquent Loads. *SIGARCH Computer Architecture News*, 29(2):14–25, May 2001.

[7] R. Das, J. Saltz, and R.v. Hanxleden. Slicing Analysis and Indirect Accesses to Distributed Arrays. *Workshop on Languages and Compilers for Parallel Computing*, 1993.

[8] A.E. Eichenberger et. al. Optimizing Compiler for the CELL Processor. *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 161–172, 2005.

[9] B. Flachs et. al. A Streaming Processing Unit for a CELL Processor. *IEEE International Solid-State Circuits Conference (ISSCC)*, February 2005.

[10] D. Pham et. al. The Design and Implementation of a First-Generation CELL Processor. *IEEE International Solid-State Circuits Conference (ISSCC)*, February 2005.

[11] R.v. Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler Analysis for Irregular Problems in Fortran D. *Workshop on Languages and Compilers for Parallel Computing*, 1992.

[12] L. Harrison and S. Mehrotra. A Data Prefetch Mechanism for Accelerating General Computation. *Technical Report 1351, CSRD, University of Illinois at Urbana-Champaign, Dept. of Computer Science*, 1994.

[13] Norman P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, June 1990.

[14] M. Kandemir, J. Ramanujam, M.J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic Management of Scratch-Pad Memory Space. *Design Automation Conference*, 2001.

[15] M. Kistler, M. Perrone, and F. Petrini. CELL Multiprocessor Communication Network: Built for Speed. *IEEE Micro*, 26(3), May/June 2006.

[16] M.H. Lipasti, W.J. Schmidt, S.R. Kunkel, and R.R. Roediger. SPAID: Software Prefetching in Pointer- and Call-intensive Environments. *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture*, 1995.

[17] C.-K. Luk and T.C. Mowry. Compiler-based Prefetching For Recursive Data Structures. *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, October 1996.

[18] C.A. Moritz, M. Frank, W. Lee, and S. Amarasinghe. Hot Pages: Software Caching for Raw Microprocessors. *MIT/LCS Technical Memo LCS-TM-599*, August 1999.

[19] T.C. Mowry, M.S. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm For Prefetching. *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, October 1992.

[20] K. O'Brien, K.M. O'Brien, Z. Sura, T. Chen, and T. Zhang. Supporting OpenMP on Cell. *International Workshop on OpenMP*, June 2007.

[21] S. Palacharla and R.E. Kessler. Evaluating Stream Buffers as a Secondary Cache Replacement. *Proceedings of the 21st Annual international Symposium on Computer Architecture*, April 1994.

[22] A. Roth, A. Moshovos, and G.S. Sohi. Dependence Based Prefetching For Linked Data Structures. *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[23] J. Su and K. Yelick. Array Prefetching for Irregular Array Accesses in Titanium. *Sixth Annual Workshop on Java for Parallel and Distributed Computing*, 2004.

[24] S. Udayakumaran, A. Dominguez, and R. Barua. Dynamic Allocation for Scratch-Pad Memory using Compile-Time Decisions. *ACM Transactions on Embedded Computing Systems*, 5(2):472–511, May 2006.

[25] S.P. Vanderwiel and D.J. Lilja. Data Prefetch Mechanisms. *ACM Computing Surveys*, 32(2):174–199, June 2000.

[26] Z. Zhang and J. Torrellas. Speeding up Irregular Applications in Shared-Memory Multiprocessors: Memory Binding and Group Prefetching. *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 188–199, 2000.