Int J Parallel Prog (2008) 36:289–311
DOI 10.1007/s10766-008-0072-7

Supporting OpenMP on Cell

Kevin O'Brien · Kathryn O'Brien · Zehra Sura · Tong Chen · Tao Zhang

Received: 31 October 2007 / Accepted: 27 December 2007 / Published online: 25 April 2008

Abstract The Cell processor is a heterogeneous multi-core processor with one power processing engine (PPE) core and eight synergistic processing engine (SPE) cores. There is a significant amount of ongoing research in programming models and tools that attempts to make it easy to exploit the computation power of the Cell architecture. In our work, we explore supporting OpenMP on the Cell processor. It is attractive to support OpenMP because programmers can continue using their familiar programming model, and existing code can be re-used. We base our work on IBM's XL compiler, and developed new components in the XL compiler and a new runtime library. Three major issues are addressed: (1) synchronization support on heterogeneous cores; (2) code generation targeting the different instruction sets; (3) data transfers and implement the OpenMP memory model. We present experimental results for some SPEC OMP 2001 and NAS benchmarks to demonstrate the effectiveness of this approach. A visualization tool based on Paraver is also used to provide some insights into actual thread and synchronization behaviors.

Keywords OpenMP · Heterogeneous architecture · Thread synchronization · Data transfer

1 Introduction

The cell broadband engine^{TM} (Cell BE) processor [1] is now commercially available in both the Sony PS3 game console and the IBM Cell Blade which represents the first product on the IBM Cell Blade roadmap. The anticipated high volumes for this non-traditional "commodity" hardware continue to make it interesting in a

K. O'Brien · K. O'Brien · Z. Sura · T. Chen (✉) · T. Zhang
IBM T. J. Watson Research, Yorktown Heights, New York, USA
e-mail: chentong@us.ibm.com

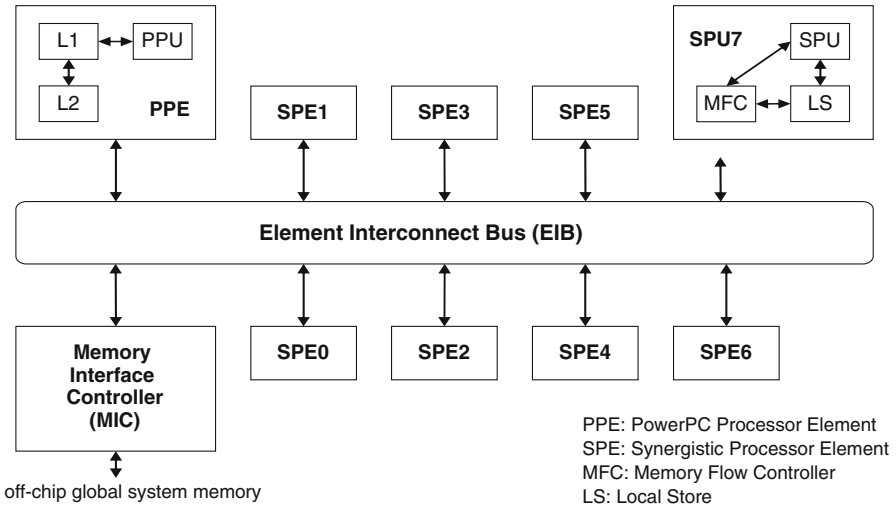


Fig. 1 Cell architecture

variety of different application spaces, ranging from the obvious multi-media and gaming domain, through the HPC space (both traditional and commercial), and to the potential use of Cell as a building block for very high end "supercomputing" systems [2].

This first generation Cell processor (Fig. 1) provides flexibility and performance through the inclusion of a 64-bit multi-threaded power processor^{TM} element (PPE) with two levels of globally-coherent cache and support for multiple operating systems including Linux. For additional performance, a Cell processor includes eight synergistic processor elements (SPEs), each consisting of a synergistic processing unit (SPU), a local memory, and a globally-coherent DMA engine. Computations are performed by 128-bit wide single instruction multiple data (SIMD) functional units. An integrated high bandwidth bus, the element interconnect bus (EIB), glues together the nine processors and their ports to external memory and IO, and allows the SPUs to be used for streaming applications [3].

Data is transferred between the local memory and the DMA engine [4] in chunks of 128 bytes. The DMA engine can support up to 16 concurrent requests of up to 16 K bytes originating either locally or remotely. The DMA engine is part of the globally coherent memory address space; addresses of local DMA requests are translated by a memory management unit (MMU) before being sent on the bus. Bandwidth between the DMA and the EIB bus is 8 bytes per cycle in each direction. Programs interface with the DMA unit via a channel interface and may initiate blocking as well as non-blocking requests.

Programming the SPE processor is significantly enhanced by the availability of an optimizing compiler which supports SIMD intrinsic functions and automatic simdization [5]. However, programming the Cell processor, the coupled PPE and eight SPE processors, is a much more complex task, requiring partitioning of an application to accommodate the limited local memory constraints of the SPE, parallelization

across the multiple SPEs, orchestration of the data transfer through insertion of DMA commands, and compiling for two distinct ISAs. Users can directly develop the code for PPE and SPE, or introduce new the language extension [6].

In this paper, we describe how our compiler manages this complexity while still enabling the significant performance potential of the machine. Our parallel implementation currently uses OpenMP APIs to guide parallelization decisions.

The remainder of the paper is laid out as follows: section two gives an overview of the compiler infrastructure upon which our work is based and presents the particular challenges of retargeting this to the novel features of the Cell platform. The next three sections of the paper look in more depth at each of these challenges and how we have addressed them, and Sect. 6 presents some experimental results to demonstrate the benefit of our approach. We draw our conclusions in the last section.

2 System Overview

In our system, we use compiler transformations in collaboration with a runtime library to support OpenMP. The compiler translates OpenMP pragmas in the source code to intermediate code that implements the corresponding OpenMP construct. This translated code includes calls to functions in the runtime library. The runtime library functions provide basic utilities for OpenMP on the Cell processor, including thread management, work distribution, and synchronization. For each parallel construct, the compiler outlines the code segment enclosed in the parallel construct into a separate function. The compiler inserts OpenMP runtime library calls into the parent function of the outlined function. These runtime library calls will invoke the outlined functions at runtime and manage their execution.

The compiler is built upon the IBM XL compiler [7,8]. This compiler has front-ends for C/C++ and Fortran, and shares the same optimization framework across multiple source languages. The optimization framework has two components: TPO and TOBEY. Roughly, TPO is responsible for high-level and machine-independent optimizations while TOBEY is responsible for low-level and machine-specific optimizations. The overview of the IBM XL compiler is shown in Fig. 2. The XL compiler has a pre-existing OpenMP runtime library and support for OpenMP 2.0 on AIX multiprocessor systems built with Power processors. In our work targeting the Cell platform, we re-use, modify, or re-write existing code that supports OpenMP as appropriate.

We encountered several issues in our OpenMP implementation that are specific to features of the Cell processor:

- Threads and synchronization: threads running on the PPE differ in capability and processing power from threads running on the SPEs. We design our system to use these heterogeneous threads, and to efficiently synchronize all threads using specialized hardware support provided in the Cell processor.
- Code generation: the instruction set of the PPE differs from that of the SPE. Therefore, we perform code generation and optimization for PPE code separate from SPE code. Furthermore, due to the limited size of SPE local stores, SPE code may need to be partitioned into multiple overlaid binary sections instead of generating it as a large monolithic section.

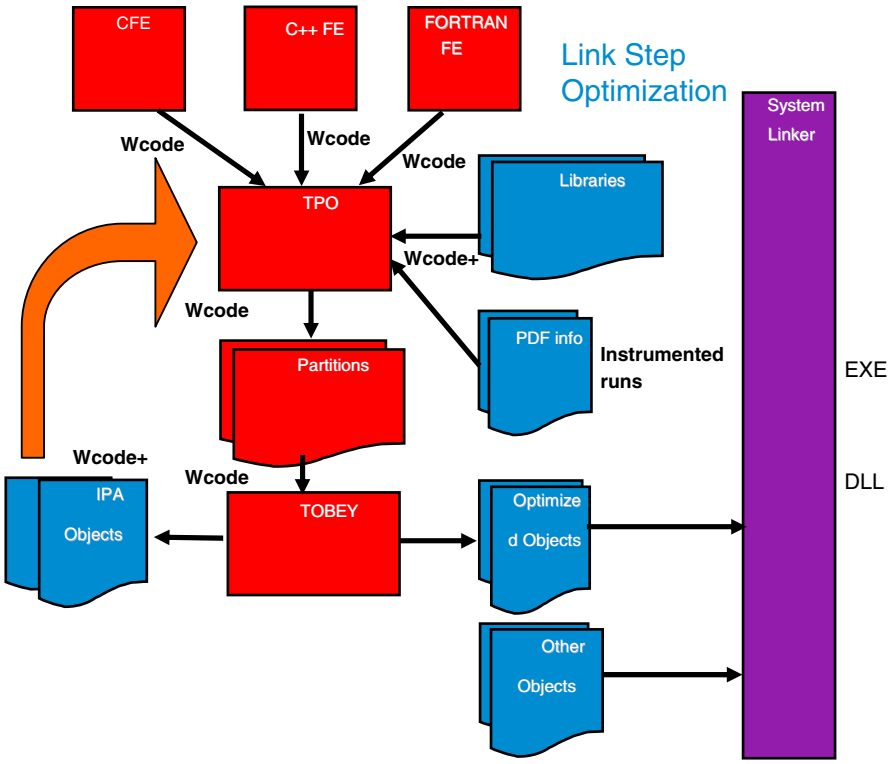


Fig. 2 Overview of the IBM XL compiler

- Memory management: each SPE has a small directly accessible local store, but it needs to use DMA operations to access system memory. Shared data in SPE code needs to be transferred between system memory and SPE local store, and this is done using DMA calls explicitly inserted by the compiler, or using a software caching mechanism that is part of the runtime library. The Cell hardware ensures DMA transactions are coherent, but it does not provide coherence for data residing in the SPE local stores. We implement the OpenMP memory model on top of the novel Cell memory model, and ensure data in system memory is kept coherent as required by the OpenMP specification.

In the following sections, we describe how we solve these issues in our compiler and runtime library implementation.

3 Threads and Synchronization

In our system, OpenMP threads execute on both the PPE and the SPEs. The master thread is always executed on the PPE. The master thread is responsible for creating threads, distributing and scheduling work, and initializing synchronization operations. Since there is no operating system support on the SPEs, this thread also handles all

OS service requests. The functions specific to the master thread align well with the Cell design of developing the PPE as a processor used to manage the activities of multiple SPE processors. Also, placing the master thread on the PPE allows smaller and simpler code for the SPE runtime library, which resides in the space-constrained SPE local store. Different from the OpenMP standard, if required by users, the PPE thread will not participate in the work for parallel loops.

Currently, we always assume a single PPE thread and use the OMP_NUM_THREADS specification to be the number of SPE threads to use. Specifying the number of PPE and SPE threads separately will need an extension of the OpenMP standard. The PPE and SPE cores are heterogeneous, and there may be significant performance mismatch between a PPE thread and an SPE thread that perform the same work. Ideally, the system can be built to automatically estimate the difference in PPE and SPE performance for a given work item, and then have the runtime library appropriately adjust the amount of work assigned to different threads. We do not have such a mechanism yet, so we allow users to tune performance by specifying whether or not the PPE thread should participate in executing work items for OpenMP work-share loops or work-share sections.

We implement thread creation and synchronization using the Cell Software Development Kit (SDK) libraries [7]. The master thread on the PPE creates SPE threads only when a parallel structure is first encountered at runtime. For nested levels of parallelism, each thread in the outer parallel region sequentially executes the inner parallel region. The PPE thread schedules tasks for all threads, using simple block scheduling for work-share loops and sections. The work sections or loop iterations are divided into as many pieces as the number of available threads, and each thread is assigned one piece. More sophisticated scheduling is left for future work.

When an SPE thread is created, it performs some initialization, and then loops waiting for task assignments from the PPE, executing those tasks, and then waiting for more tasks, until the task is to terminate. A task can be the execution of an outlined parallel region, loop or section, or performing a cache flush, or participating in barrier synchronization. Figure 3 depicts the thread creation and job scheduling for PPU and SPE. There is a task queue in system memory corresponding to each thread. When the master thread assigns a task to a thread, it writes information about the task to the corresponding task queue, including details such as the task type, the lower bound and upper bound for a parallel loop, and the function pointer for an outlined code region that is to be executed. Once an SPE thread has picked up a task from the queue, it uses DMA to change the status of the task in the queue, thus informing the master thread that the queue space can be re-used.

The Cell processor provides special hardware mechanisms for efficient communication and synchronization between the multiple cores in a Cell system. The memory flow controller (MFC) for each SPE has two blocking outbound mailbox queues, and one non-blocking inbound mailbox queue. These mailboxes can be used for efficient communication of 32-bit values between cores. When the master thread assigns tasks to an SPE thread, it uses the mailbox to inform the SPE of the number of tasks available for execution. Each SPE MFC also has an atomic unit that implements atomic DMA commands and provides four 128-byte cache lines that are maintained cache coherent

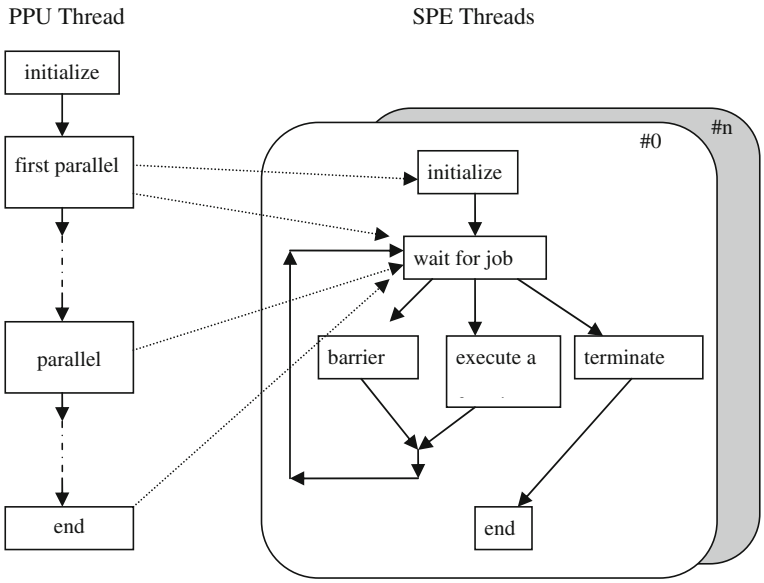


Fig. 3 Thread creation and job scheduling for PPU and SPE

across all processors. We use atomic DMA commands for efficient implementation of OpenMP locks, barriers[1], and cache flush operations.

4 Code Generation

Figure 4 illustrates the code generation process of the Cell OpenMP compiler. The compiler separates out each code region in the source code that corresponds to an OpenMP parallel construct (including OpenMP parallel regions, work-share loops or work-share sections, and single constructs), and outlines it into a separate function. The outlined function may take additional parameters such as the lower and the upper bounds of the loop iteration for parallel loops. In the case of parallel loops, the compiler further transforms the outlined function so that it only computes from the lower bound to the upper bound. The compiler inserts an OpenMP runtime library call into the parent function of the outlined function, and passes a pointer to the outlined function code into this runtime library function. During execution, the runtime function will indirectly invoke the outlined function. The compiler also inserts synchronization operations such as barriers when necessary.

Due to the heterogeneity of the Cell architecture, the outlined functions containing parallel tasks may execute on both the PPE and the SPEs. In our implementation, we clone the outlined functions so that there is one copy of the function for the PPE architecture, and one for the SPE architecture. We perform cloning during TPO link-time optimization when the global call graph is available, so we can clone the whole

[1] We thank Daniel Brokenshire (IBM Austin) for his implementation of barriers on Cell.

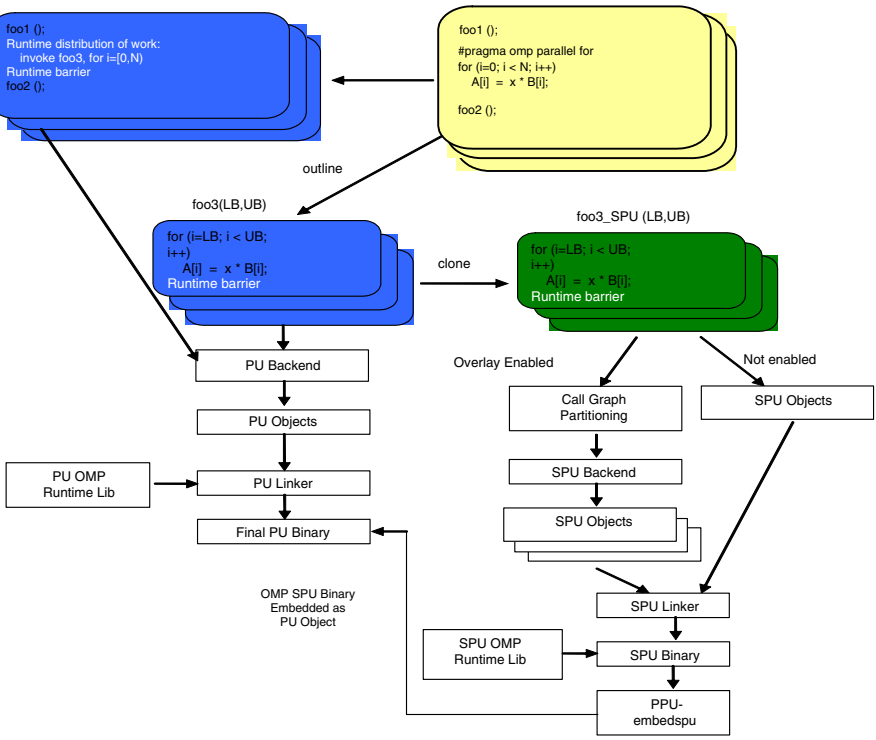


Fig. 4 Code generation process

sub-graph for a call to an outlined function when necessary. We mark the cloned function copies as PPE and SPE procedures, respectively. In later stages of compilation, we can apply machine-dependent optimizations to these procedures based on their target architecture. Auto-simdization is one example. SPE has SIMD units that can execute operation on 128 byte data with one instruction. After cloning, the code for SPE will undergo the auto-simdization to transform scalar code into SIMD code for SPE, while the PPE has totally different SIMD instructions. In other words, we choose to clone the functions to enable more aggressive optimizations.

When a PPE runtime function in the master thread distributes parallel work to an SPE thread, it needs to tell the SPE thread what outlined function to execute. The PPE runtime function knows the function pointer for the PPE code of the outlined function to execute. However, the SPE thread needs to use the function pointer for the SPE code of the same outlined function. To enable the SPE runtime library to determine correct function pointers, the compiler builds a mapping table between corresponding PPE and SPE outlined function pointers, and the runtime looks up this table to determine SPE code pointers for parallel tasks assigned by the master thread.

At the end of TPO, procedures for different architectures are separated into different compilation units, and these compilation units are processed one at a time by the TOBEY backend. The PPE compilation units are processed as for other architectures and need no special consideration. However, if a single large SPE compilation unit is

generated, it may result in SPE binary code that is too large to fit in the small SPE local store all at once. In fact, we observe this to be the case for many benchmark programs. For OpenMP, one way to mitigate this problem is to place all the code corresponding to a given parallel region in one SPE compilation unit, and generate as many SPE compilation units as there are parallel regions in the program. Using this approach, we can generate multiple SPE binaries, one for each SPE compilation unit. We can then modify the runtime library to create SPE threads using a different SPE binary on entry to each parallel region. However, there are two drawbacks to using this approach: first, an individual parallel region may still be too large to fit in SPE local store, and second, we observe through experiments that the overhead for repeatedly creating SPE threads is significantly high.

To solve the SPE code size problem, we rely on the technique of call graph partitioning and code overlay. We first partition the sub-graph of the call graph corresponding to SPE procedures into several partitions. Then we create a code overlay for each of these call graph partitions. Code overlays share address space and do not occupy local storage at the same time. Thus, the pressure on local storage due to SPE code is greatly reduced. To partition the call graph, we weight each call graph edge by the frequency of this edge. The frequency can be obtained by either compiler static analysis or profiling. Then we apply the maximum spanning tree algorithm to the graph. Basically, we process edges in the order of their weight. If merging the two nodes of the edge does not exceed a predefined memory limitation, we merge those two nodes, update the edge weights, and continue. When the algorithm stops, each merged node represents a call graph partition comprising all the procedures whose nodes were merged into that node. Thus, the result is a set of call graph partitions. Our algorithm is a simple greedy algorithm that can be further optimized. After call graph partitions are identified, we utilize SPU code overlay support introduced in Cell SDK 2.0 and place the procedures in each call graph partition into a separate code overlay.

After the TOBEY backend generates an SPE binary (either with or without code overlays), we use a special tool called ppu-embedspu to embed the SPE binary into a PPE data object. This PPE data object is then linked into the final PPE binary together with other PPE objects and libraries. During execution, when code running on the PPE creates SPE threads, it can access the SPE binary image embedded into the PPE data object, and use this SPE image to initialize the newly created SPE threads.

5 Memory Management

OpenMP specifies a relaxed-consistency, shared-memory model. This model allows each thread to have its own temporary view of memory. A value written to a variable, or a value read from a variable, can remain in the thread's temporary view until it is forced to share memory by an OpenMP flush operation. We find that such a memory model can be efficiently implemented on the Cell memory structure.

In the Cell processor, each SPE has just 256 K directly accessible local memory for code and data. We only allocate private variables accessed in SPE code to reside in the SPE local store. Shared variables reside in system memory, and SPE code can access them through DMA operations. We use two mechanisms for DMA transfers: static

buffering and compiler-controlled software cache. In both mechanisms, the global data may have a local copy in the SPE local store. The SPE thread may read and write the local copy. This approach conforms to the OpenMP relaxed memory model and takes advantage of the flexibility afforded by the model to realize memory system performance.

When an SPE thread uses DMA to get/put data from/to the system memory, it needs to know the address of the data to be transferred. However, global data is linked with the PPE binary and is not directly available in SPE code. The Cell SDK [9] provides a link-time mechanism called CESOF, which makes available to the SPE binary the addresses of all PPE global variables once these addresses have been determined. We also use a facility similar to CESOF when generating SPE code.

Besides global data, an SPE thread may need to know the address of data on the PPE stack when, in source code, the procedure executing in the SPE thread is nested within a procedure executing in a PPE thread, and the SPE procedure accesses variables declared in its parent PPE procedure. Though C and Fortran do not support nested procedures (C++ and Pascal do), this case can occur when the compiler performs outlining. For example, in Fig. 1, if the variable "x" were declared in the procedure that contains the parallel loop, after outlining, the declaration of "x" becomes out of the scope of the outlined function. To circumvent this problem, the compiler considers each outlined function to be nested within its parent function. The PPE runtime, assisted by compiler transformations, ensures that SPE tasks that will access PPE stack variables are provided with the system memory address of those stack variables.

5.1 Static Buffers

Some references are *regular* references from the point-of-view of our compiler optimization. These references occur within a loop, the memory addresses that they refer to can be expressed using affine expressions of loop induction variables, and the loop that contains them has no loop-carried data dependence (true, output or anti) involving these references. For such regular reference accesses to shared data, we use a temporary buffer in the SPE local store. For read references, we initialize this buffer with a DMA *get* operation before the loop executes. For write references, we copy the value from this buffer using a DMA *put* operation after the loop executes. The compiler statically generates these DMA get and put operations. The compiler also transforms the loop structure in the program to generate optimized DMA operations for references that it recognizes to be regular. Furthermore, DMA operations can be overlapped with computations by using multiple buffers [10]. Figure 5 illustrates possible code transformations for a simple loop using static buffers. The compiler can choose the proper buffering scheme and buffer size to optimize execution time and space.

Execution time of a loop blocked for DMA buffering varies with the amount of DMA overlapped with computation. For a k-buffer scheme, the amount of DMA overlapped increases both with the value of k, and with the size of the buffers used. In the SPE, all the buffers occupy space in the local store, which is only 256 KB in size. This limited local store space is a prime resource, since it is being used for both code and data, and the available space limits the applicability of optimizations that increase

```
/* A and B are in main memory */
/* S is a scalar residing in LS */
for (i=0; i<N; i++) {
   B[i] = A[i] * S;
}
```

(a) Example Code

```
/* S, tA, and tB reside in LS */
for (i=0; i<N; i++) {
   DMA get A[i] to tA;
   tB = tA * S;
   DMA put tB to B[i];
}
```

(b) Naive Buffering

```
for (ii=0; ii<N; ii+=bf) {
   n = min(ii+bf, N);
   DMA get A[ii:n] to tA;
   for (i=ii; i<n; i++) {
      tB[i] = tA[i] * S;
   }
   DMA put tB to B[ii:n];
}
```

(c) Single Buffering

```
/* Uses non−blocking DMA */

t=0;  /* Decides buffer 1 or buffer 2 */
n = min(bf, N);
DMA get A[0:n] to tA[t], tag=t;
t = (t+1) % 2;

for (ii=0; ii<N; ii+=bf) {
   n = min(ii+bf, N);
   m = min(ii+2*bf, N);
   DMA get A[ii+bf:m] to tA[t], tag=t;
   t = (t+1) % 2;
   DMA wait, tag=t;

   for (i=ii; i<n; i++) {
      tB[t][i] = tA[t][i] * S;
   }

   DMA put tB[t] to B[ii:n], tag=t;
}
DMA wait, tag=t;
```

(d) Double Buffering

Fig. 5 Example to illustrate static DMA buffering

code size or require more space to buffer data. Due to the local store size constraint, a restricted amount of space is available for DMA buffering.

Given a budget for the amount of space to be used for DMA buffering, we determine the buffering scheme that will result in the best execution time performance. Since the total buffer size is fixed, performance of a k-buffer scheme needs to be compared with the performance of a $(k+1)$-buffer scheme that uses individual buffers of a size smaller than the buffers used in the k-buffer scheme. Once the optimal buffering scheme is known, it may be the case that all possible DMA overlap is attained using a buffer size smaller than the maximum buffer size allowed by the total buffer space budget. Note that there is a limit to how much performance can be improved using DMA overlap before the application becomes computation-bound. Thus, we want to determine both the optimal buffering scheme and the smallest buffer size that maximize performance, when constrained by the total buffer space available.

We find that the performance of DMA buffering depends on several factors, including the set-up time for each DMA operation, the DMA transfer time, the amount of computation in the loop, the number of buffers being used, and the size of each individual buffer. We develop a model to relate each of these factors to the execution time, and use this model to predict the relative merit of using different buffering schemes and different buffer sizes. Our model applies to the innermost loop in a loop nest,

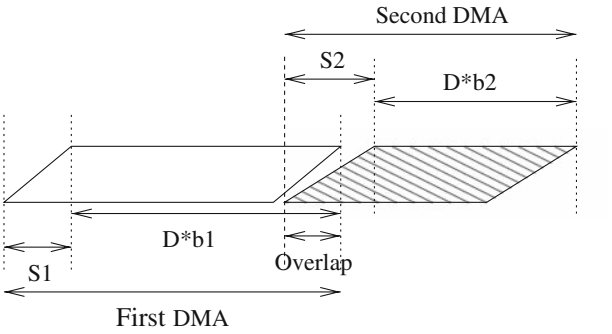


Fig. 6 Latency of DMA operations

where this loop operates on a number of array data streams, has a large iteration count, has no loop-carried dependences, and has no conditional branches within the loop body.

We approximate the latency of one DMA operation with the formula $S+D*b$, where S is the set-up time for one DMA operation, D is the transfer time for one byte, and b is the number of bytes transferred by this DMA operation. When two non-blocking DMA operations for b1 and b2 bytes are issued in sequence, the set-up of the second DMA operation can be overlapped with the data transfer of the first, as illustrated in Fig. 6. When the set-up of the second DMA operation (S2) is less than or equal to the set-up of the first DMA operation (S1), it can be completely overlapped. In this case, the combined latency of the two DMA operations will be $S1+D*(b1+b2)$. For different values of S1 and S2, the amount of overlap of set-up time with transfer time will be different.

In the CELL architecture, the value of S is different for DMA get and put operations [4]. The DMA get operation has a higher value of S because it includes the main memory access time to retrieve data, whereas a DMA put can complete before data is actually written to its main memory location. In general, a sequence of n DMA transfers will have latency $S+D*(b1+\cdots+bn)$, where S is a function of $S1,\ldots,Sn$.

5.1.1 Latency for Single-Buffer

Figure 7 illustrates the execution sequence for the code in Fig. 5c. Ignoring the prologue and epilogue, and clubbing together consecutive DMA operations, each iteration of the outer blocked loop comprises of a DMA put corresponding to the previous iteration, a DMA get to fetch data for the current iteration, and the computation of one instance of the entire inner blocked loop. Note that non-blocking DMA operations can be used, with a DMA wait inserted just before the inner blocked loop. Thus, the latency of one iteration of the outer blocked loop is the latency of both the DMA transfers plus the computation latency of the inner blocked loop. Let N be the iteration count of the original loop, and assume the loop has been blocked using a blocking factor of bf. Let D1 be the DMA transfer time for one byte, b be the number of bytes transferred in all DMA operations corresponding to one iteration of the outer blocked loop, and

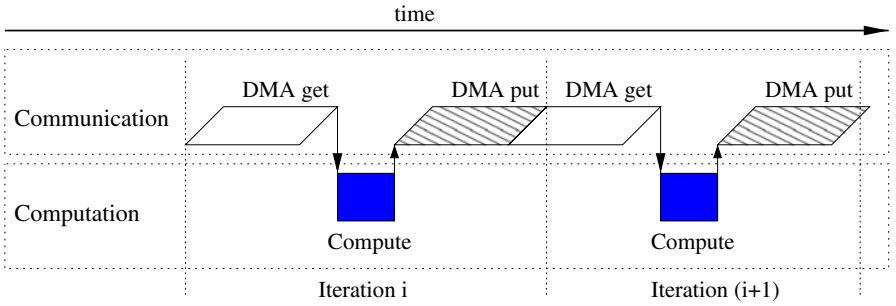


Fig. 7 Execution sequence for single-buffer

S be the composite set-up time for the sequence of non-blocking DMA operations corresponding to one iteration of the outer blocked loop. Also, let C be the computation time for one iteration of the inner blocked loop. The value for C is expressed as ($C_{inner} + C_{outer}/bf$), where C_{inner} is the compute time for each iteration of the inner blocked loop, and C_{outer} is the overhead for issuing DMA requests in an iteration of the outer blocked loop. This overhead primarily includes the function call and runtime checking overhead in compiler-generated code for DMA transfer requests, and it gets amortized over bf iterations of the inner blocked loop. In practice, we expect C_{outer} to be small, and bf to be large, so that C can be approximated by C_{inner}. The total latency of the entire loop is given by: $[(S + D1 * b * bf) + C * bf] * (N/bf) = (S/bf + D1 * b + C) * N$. For simplicity, let $D = D1 * b$ be the DMA transfer time for all data accessed in one iteration of the inner blocked loop. Thus, latency for single-buffer is $(S/bf + D + C) * N$.

5.1.2 Latency for Double-Buffer

In the following discussion, the terms N, bf, C, S, and D have the same meaning as in the single-buffer case discussed earlier. For clarity, the following examples refer to DMA for one pair of buffers. However, the discussion also applies to examples using a set of double-buffers, with S corresponding to the set-up delay for a composite sequence of non-blocking DMA operations issued for each set (Fig. 8).

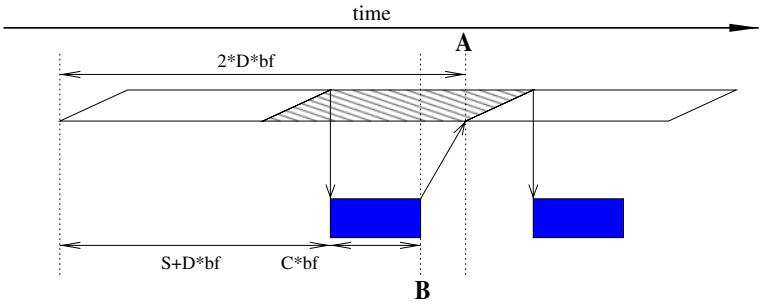


Fig. 8 Execution sequence for DMA-bound double-buffer

Case 1: DMA-Bound: Error! Reference source not found illustrates the case when double-buffer is used and the application is DMA-bound. In this case, there is no delay between any two successive DMA operations. The sequence of DMA operations and computations alternate between using the first buffer and the second buffer. The first and second DMA operations are issued successively before any computation begins. The third DMA operation is issued only after the first computation of the inner blocked loop finishes. If there is to be no delay between the second and third DMA operations, then the time to complete the first computation (point B in the figure) must be less than or equal to the time to complete the first two DMA operations (point A in the figure). This translates to the condition:

$$(S + D * bf + C * bf) \leq (2 * D * bf), \text{ or } D \geq (C + S/bf)$$

When this condition holds, the execution pattern repeats throughout the loop and the application is DMA-bound. The latency for the entire loop is approximated by the time taken by all the consecutive DMA operations, i.e. $S + D * N$. When N is large, this can be simplified to $D * N$.

Case 2: Computation-Bound Figure 9 illustrates the case when double-buffer is used and the application is computation-bound. In this case, there is no delay waiting for DMA to complete between any two successive computations of the inner blocked loop. The sequence of DMA operations and computations alternate between using the first buffer and the second buffer. The first and second DMA operations are issued successively before any computation begins. The third DMA operation is issued only after the first computation of the inner blocked loop finishes. If there is to be no delay between the second and third computations, then the time to complete the third DMA operation (point B in the figure) must be less than or equal to the time to complete the first two computations (point A in the figure). This translates to the condition:

$$(S + D * bf + C * bf + S + D * bf) \leq (S + D * bf + 2 * C * bf), \text{ or } D \leq (C - S/bf)$$

When this condition holds, the execution pattern repeats throughout the loop, and the application is computation-bound. The latency for the entire loop is approximated by the time taken by all the consecutive computations, i.e. $C * N$, when N is large.

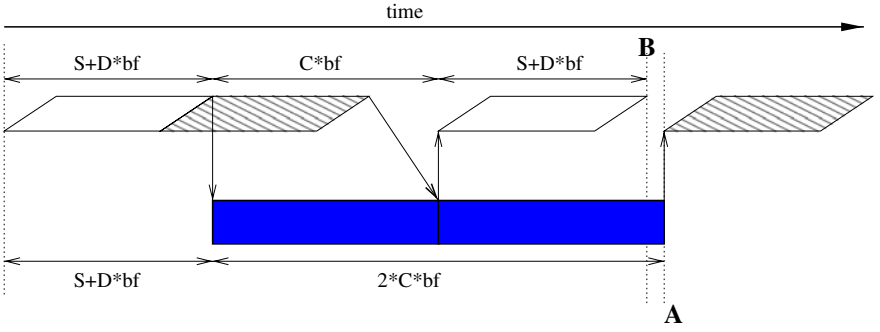


Fig. 9 Execution sequence for computation-bound double-buffer

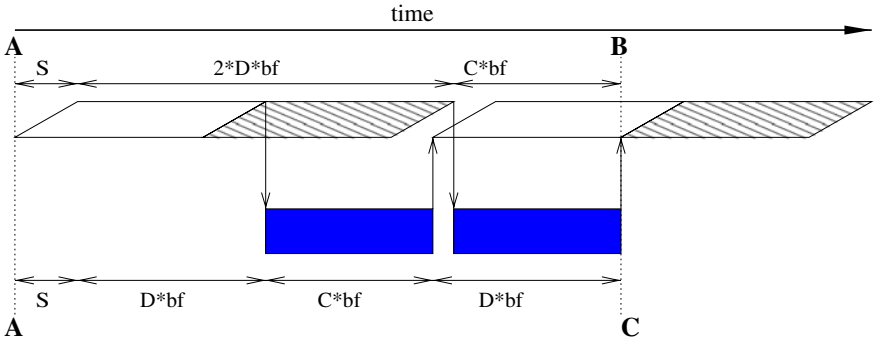


Fig. 10 Execution sequence for double-buffer when $C \leq D < (C + S/bf)$

Case 3: Incomplete Overlap A loop that is neither DMA-bound nor computation-bound has incomplete overlap of DMA operations with computation. We analyze this case by splitting it into two sub-cases: when $C \leq D < (C + S/bf)$, and when $(C - S/bf) < D < C$. The total latency of the loop in both cases is the same: $(S/bf + D + C) * N/2$.

Case A: When $C \leq D < (C + S/bf)$ Figure 10 illustrates this case. Here, the set-up of the third DMA operation is not fully overlapped with the second DMA transfer. Also, there is a delay between the first and second computation, waiting for the second DMA transfer to complete. The second computation finishes at point B in the figure, and it can only start after the second DMA transfer has completed. From the beginning (point A in the figure), the latency for the second computation to finish is $S + 2 * D * bf + C * bf$. From a DMA point of view, the earliest that the fourth DMA operation can start is after the third DMA transfer reaches point C. The third DMA transfer can start only after the first computation finishes. The latency from point A in this case is $S + D * bf + C * bf + D * bf$. The two latencies from A to B and A to C are the same, which means that the fourth DMA starts at the same point that the second computation finishes, and the execution repeats the pattern illustrated in the figure. The delay between the second and third DMA operations is $S + D * bf + C * bf - 2 * D * bf = S + (C - D) * bf$. The total latency of the loop is the latency of all DMA transfers plus the extra delays due to incomplete overlap that occur after every two DMA operations. This latency is given by:

$$N * D + (S + (C - D) * bf) * N/bf/2 = (S/bf + D + C) * N/2.$$

Case B: When $(C - S/bf) < D < C$ Figure 11 illustrates this case. Here, the set-up of the third DMA operation is not overlapped with the second DMA transfer. Also, there is no delay between the first and second computation, but there is a delay between the second and third computation, waiting for the third DMA transfer to complete. The data for the fourth computation will be ready at point B in the figure, made available only after the first DMA transfer, the first two computations, and the fourth DMA transfer have completed. From the beginning (point A in the figure), the latency for the fourth DMA transfer to complete is $S + D * bf + 2 * C * bf + S + D * bf$. From a computation point of view, the third computation will finish at point C in the figure,

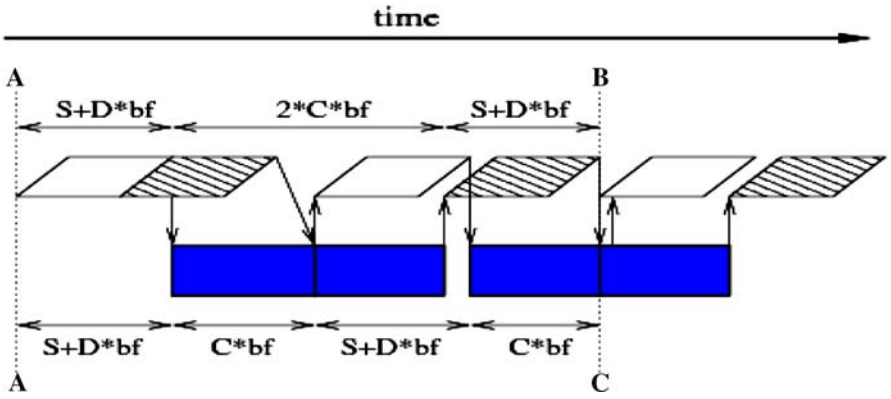


Fig. 11 Execution sequence for double-buffer when (C − S/bf) < D < C

and it can start only after the third DMA transfer completes. The third DMA can start only after the first computation finishes. The latency from point A in this case is $S + D * bf + C * bf + S + D * bf + C * bf$. The two latencies from A to B and A to C are the same, which means that the fourth DMA completes at the same point that the third computation finishes, and the execution repeats the pattern illustrated in the figure. The delay between the second and third computations is $S + D * bf - C * bf = S + (D - C) * bf$. The total latency of the loop is the latency of all computations plus the extra delays due to incomplete overlap that occur after every two computations. This latency is given by:

$$N * C + (S + (D - C) * bf) * N/bf/2 = (S/bf + D + C) * N/2.$$

5.1.3 Latency for k-Buffer

Analogous to the case of double-buffer, we can derive the condition for a k-buffered loop to be DMA-bound of computation-bound. The loop will be DMA-bound when $D \geq \max[C, (C + S/bf)/(k - 1)]$, and the latency of the entire loop can be approximated by $D * N$. The loop will be computation-bound when $D \leq \min(C, (k - 1) * C - S/bf)$, and the latency of the entire loop can be approximated by $C * N$. The latency of a k-buffered loop that is neither DMA-bound nor computation-bound will be $(S/bf + C + D) * N/k$. We do not discuss details of this case here.

5.1.4 Compiler Algorithm

The latency formulae derived in this section can be applied to determine the optimal buffering scheme and buffer size for a loop with limited amount of memory available for buffer space. The algorithm described here can be used in a compiler to automatically transform and optimize code for DMA buffering. In the following discussion, we restrict the choice of buffering schemes to single-, double-, or triple-buffer.

Assume that the amount of memory available for buffering is specified in terms of the largest block factor (say B) that can be used when transforming the loop for a

```
Algorithm: bool Choose_Double_Buffer (C, D, S, B) {
float C: the computation per iteration;
float D: the DMA transfer time per iteration;
int B: buffer space constraint in terms of the maximum
if (D > C) {
return TRUE;
```

Fig. 12 Algorithm to decide whether double-buffer should be used

single-buffer scheme. Then the maximum block factor for double-buffer is B/2, and for triple-buffer is B/3. The performance of a loop will be optimal if it is computation-bound or DMA-bound. Therefore, a DMA-bound double-buffered loop (latency $D*N$) or a computation-bound double-buffered loop (latency $C*N$) should be better than a single-buffered loop (latency $(S/B+D+C)*N$). When the double-buffered loop has incomplete overlap, its latency will be $(S/B/2+D+C)*N/2$. In this case, the difference between the latencies of double-buffer and single-buffer is $(D+C)*N/2>0$. So double-buffer should always outperform single-buffer.

The algorithm in Fig. 12 shows how to choose between double-buffer and triple-buffer. When the same performance can be achieved by different buffering schemes, the scheme with less number of buffers is preferred. When $D>C$, the double-buffer scheme becomes DMA-bound when $D\geq C+S/B/2$, which is the same as $S/(D-C)\leq B/2$. In this case, we choose the double-buffer scheme since it is DMA-bound and optimal. Similarly, when $D<C$, the double-buffer scheme becomes computation-bound for $D\leq C-S/B/2$, which is the same as $S/(C-D)\leq B/2$, and we choose the double-buffer scheme. In all other cases, we choose the triple-buffer scheme since it can provide a greater amount of overlap.

Once the loop becomes DMA-bound or computation-bound, performance will not improve with increasing buffer sizes. In such cases, memory resources can be saved by choosing the smallest buffer size that is optimal. The memory space saved can then be used by other components contending for it, e.g. more local memory can be assigned to the outer blocked loops to increase data re-use, or the size of code buffers can be increased to reduce the frequency of swapping code partitions to and from the SPE local store. Based on the conditions derived in this section for the execution to be DMA-bound or computation-bound, the block factor for double-buffer need not be larger than S/abs(D−C). The block factor for DMA-bound triple-buffer need not be larger than $S/(2*D-C)$, and for computation-bound triple-buffer need not be larger than $S/(2*C-D)$.

5.2 Software Cache

For irregular references to shared memory, we use a compiler-controlled software cache to read/write the data from/to system memory. The compiler replaces loads and

stores using these references with instructions that explicitly look up the effective address in a directory of the software cache. If a cache line for the effective 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, we allocate a line in the cache either by using an empty line or by replacing an existing line. Then, for a load, we issue a DMA get operation to read the data from system memory. For stores, we write the data to the cache, and maintain dirty bits to record which byte is actually modified. Later, we write the data back to system memory using a DMA put operation, either when the cache line is evicted to make space for other data, or when a cache flush is invoked in the code based on OpenMP semantics.

We configure the software cache based on the characteristics of the Cell processor. Since 4-way SIMD operations on 32-bit values are supported in the SPE and we currently use 32-bit memory addresses, we use a 4-way associative cache that performs the cache lookup in parallel. Also, we use 128-byte cache lines since DMA transfers are optimal when performed in multiples of 128 bytes and aligned on a 128-byte boundary. The cache structure and the lookup operation with Cell SIMD instructions are illustrated in Fig. 13. If only some bytes in a cache line are dirty, when the cache line is evicted or flushed, the data contained in it must be merged with system memory such that only the dirty bytes overwrite data contained in system memory. One way to achieve this is to DMA only the dirty bytes and not the entire cache line. However, this may result in small discontinuous DMA transfers, and exacerbated by the alignment requirements for DMA transfers, it can result in poor DMA performance. Instead,

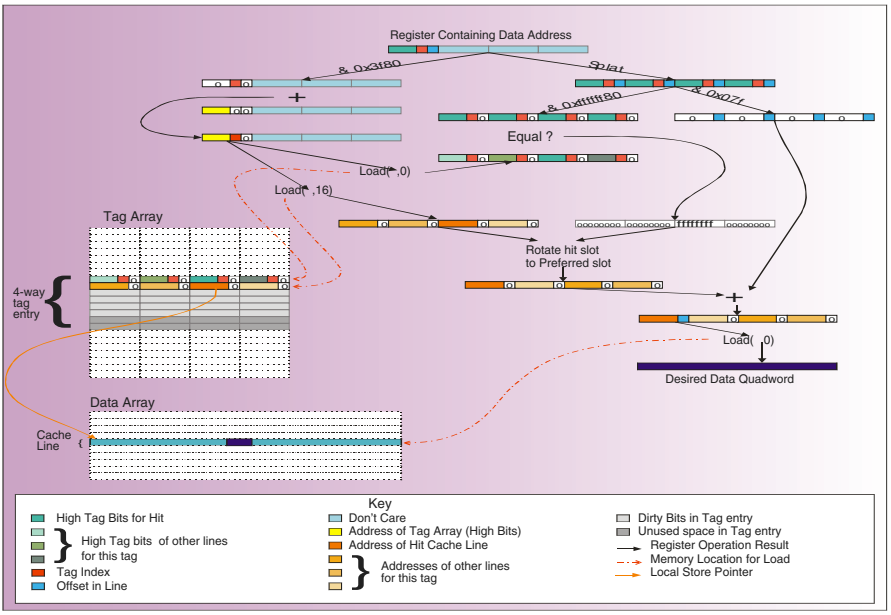


Fig. 13 Software cache structure and lookup operations

```
inputs:
        t_ea is the effective address for this cache line;
        ls_global: a temporary buffer;
        ls_local: the cache line
        dirty_bits: the dirty bits for this cache line
vector unsigned char vec_zero = spu_splats((unsigned char)0);
vector unsigned char vec_allone = spu_splats((unsigned char)0xFF);
do {
   vector unsigned char* global_ptr = (vector unsigned char*) ls_global;
   vector unsigned char* local_ptr =  (vector unsigned char*) ls_local;
   mfc_getllar(ls_global, (uint64_t) t_ea, 0, 0);
   status = mfc_read_atomic_status();
      for(i=0; i<8; i++) {
```

Fig. 14 Code for cache evict

we use the support for atomic updates of 128-byte lines that is provided in the SPE hardware to atomically merge data in the cache line with data in the corresponding system memory, based on recorded dirty bits. The code segment is shown in Fig. 14.

When an OpenMP flush is encountered, the compiler guarantees that all data in the static buffers in local store has been written back into memory, and that existing data in the static buffers is not used any further. The flush will also trigger the software cache to write back all data with dirty bits to system memory, and to invalidate all lines in the cache.

6 Experimental Results

We compiled and executed some OpenMP test cases on a Cell blade that has both the PPU and the SPUs running at 3.2 GHz, and has a total of 1 GB main memory. All our experiments used one Cell chip: one PPE and eight SPEs. The test cases include several simple streaming applications, as well as the standard NAS [11] and SPEC OMP 2001 [12] benchmarks. To observe detailed runtime behavior of applications, we instrumented the OpenMP runtime libraries with Paraver [13], a trace generation and visualization tool.

Figure 15 shows the PPE and SPE thread behavior for a small test case comprising of one parallel loop that is repeatedly executed 100 times. The PPE thread is assigned no loop iterations, and is responsible only for scheduling and synchronization. The figure shows a high-level view of one complete execution of the parallel loop. The first

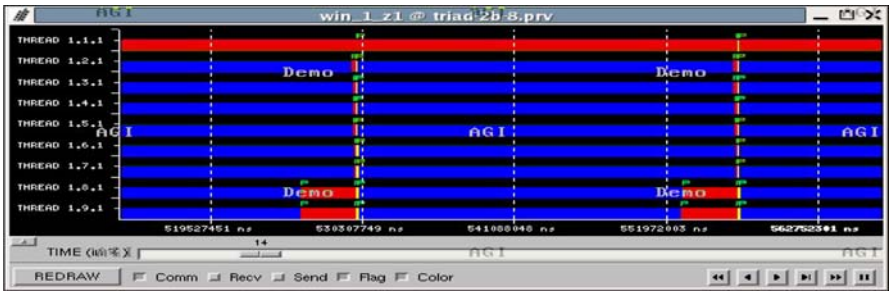


Fig. 15 Execution of PPE and SPE threads

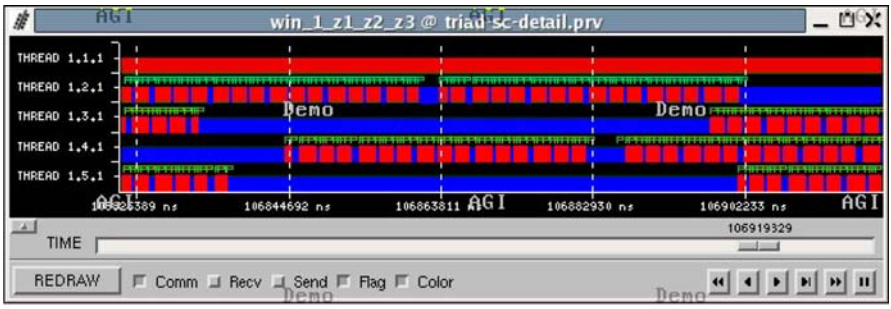
row corresponds to the PPE thread and the remaining rows correspond to SPE threads. Blue areas represent time spent doing useful computation, yellow areas represent time spent in scheduling and communication of work items, and red areas represent time spent in synchronization and waiting for DMA operations to complete. We can clearly identify the beginning and end of one instance of a parallel loop execution from the neatly lined up set of red synchronization areas that indicate the implicit barriers being performed at parallel region boundaries, as specified by the OpenMP standard.

If we zoom into a portion of the SPE useful computation shown in Fig. 15, we can see in greater detail the time actually spent in computing and the time spent in waiting for data transfers. Figure 16a shows this detail for a segment of execution when only the software cache is used for automatic DMA transfers. We see many areas in the SPE execution that are dominated by waiting for DMA operations to complete (red areas), and the need to optimize this application is evident. Figure 16b shows a similar segment of execution for the same application when it has been optimized using static buffering. We observe the improved ratio for time spent doing actual computation (blue) versus time spent waiting for DMA operations (red).

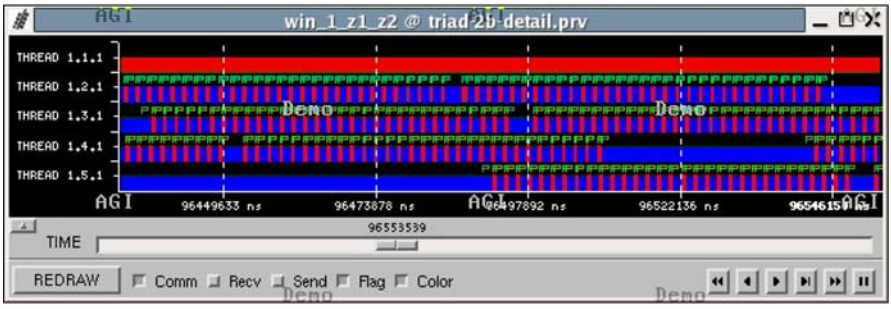
To evaluate our approach, we first tried some simple stream benchmarks, comparing the performance of code generated by our compiler with the performance of manually written code that was optimized using Cell SDK libraries and SIMD instructions. The performance comparison is shown in Fig. 17. The speedups shown in this figure are the ratios of execution time when using eight SPUs and execution time when only using the PPE. Both SIMD units and multiple cores contribute to the speedups observed. We observe that our OpenMP compiler can achieve as good a performance (except for *fft*) as the highly optimized manual code on these stream applications. *fft* performs poorly in comparison because auto-simdization cannot handle different displacements in array subscript expressions for different steps in the FFT computation. Manual code performs slightly better on *dotproduct* and *xor* because the compiler does not unroll the loop an optimal number of times.

We also experimented with some applications from the NAS, equake from Spec OMP 2001 benchmark suites, and lbm from SPEC 2006 benchmark suites. We report speedups of the whole program normalized to one PPU and one SPU respectively in Fig. 18.

Compared to the performance of one PPU, many applications show significant speedup with our compiler on eight SPUs. The performance of some others (such as CG, LU) seem to have room for further improvement. We analyzed the benchmarks



(a) software cache only



(b) optimized with static buffer

Fig. 16 Data transfer with software cache and static buffer

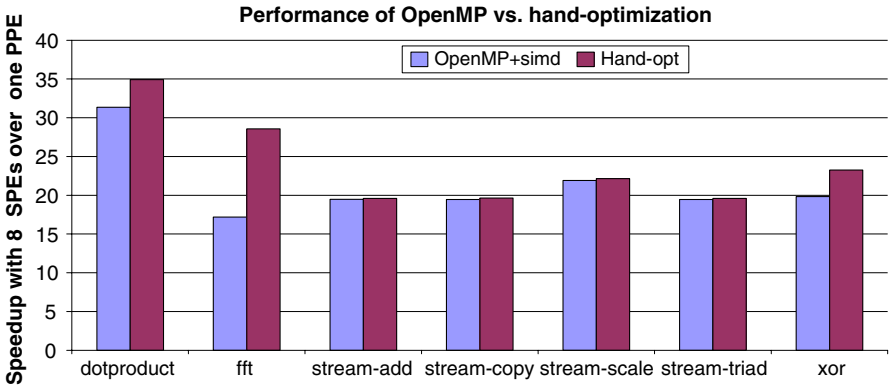


Fig. 17 Performance comparison with manually optimized code

and traced compiler transformations to determine the reasons for the performance shortfall. We identified two main reasons for it:

1. Software cache is not large enough. CG is dominated by irregular data accesses and the miss rate can be greatly reduced if the size of our software cache is doubled. However, larger software cache will cause size problem for some other benchmarks. A smarter resource allocation method, perhaps with profiling feedback, may help.

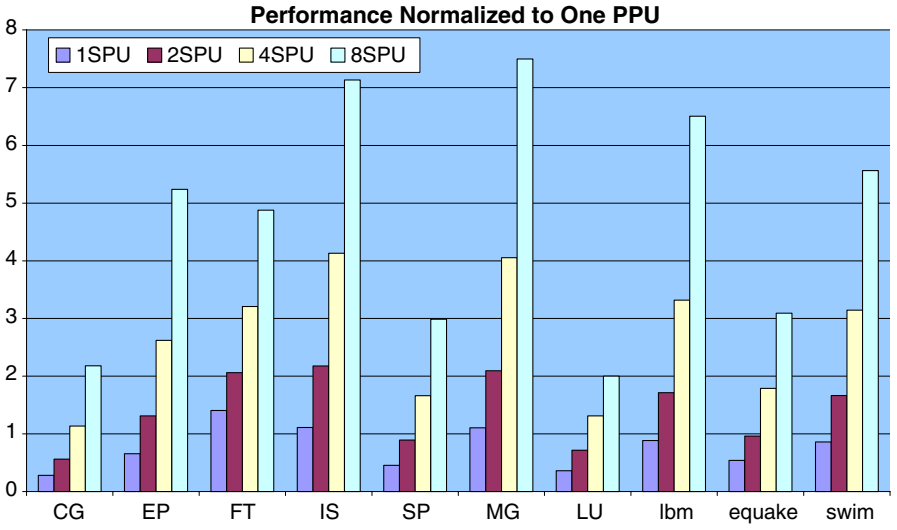


Fig. 18 Performance compared to one PPU

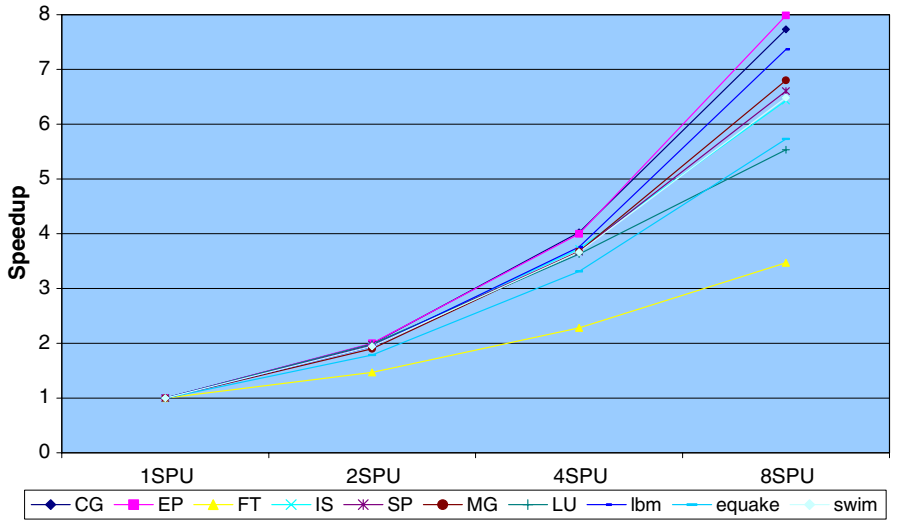


Fig. 19 Performance compared to one SPU

2. Applications have not been optimized for the Cell architecture: LU contains discontinuous data references to main memory. DMA data transfers become the bottleneck in such cases. It seems to us this is a careless mistake resulted from translating LU from Fortran to C. However, this problem could be solved if compiler is able to perform array reshaping automatically.

The performance normalized to one SPU (in Fig. 19) shows the scalability. All of them, except FT, equake, and IS, show good scalability with speedup more than six on eight SPUs. The speedup of FT is below four. The reason is that the data accesses in

FT caused memory bank conflicts. The reason that IS did not scale up well is that IS contains some computations in either master pragma or critical pragma. Those computations are done sequentially. For equake, some loops are not parallelized due to the current implementation limitation of compiler. Therefore, its speedup of eight SPUs is only above five.

7 Conclusions

In this paper, we describe how to support OpenMP on the Cell processor. Our approach allows users to simply reuse their existing OpenMP applications on the powerful Cell Blade, or easily develop new applications with the OpenMP API without worrying about the hardware details of the Cell processor. We support OpenMP by orchestrating compiler transformations with a runtime library that is tailored to the Cell processor. We focus on issues related to three topics: thread and synchronization, code generation, and the memory model. Our compiler is novel in that it generates a single binary that executes across multiple ISAs and multiple memory spaces.

Experiments with simple test cases demonstrate that our approach can achieve performance similar to that of manually written and optimized code. We also experimented with some large, complex benchmark codes. Some of these benchmarks show significant performance gains. Thus, we demonstrate that it is feasible to extract high performance on a Cell processor using the simple and easy-to-use OpenMP programming model. However, we need to further improve our compiler implementation for improved performance on a wider set of application programs.

References

1. Pham, D., Asano, S., Bolliger, M., Day, M.N., Hofstee, H.P., Johns, C., Kahle, J., Kameyama, A., Keaty, J., Masubuchi, Y., Riley, M., Shippy, D., Stasiak, D., Suzuoki, M., Wang, M., Warnock1, J., Weitzel, S., Wendel, D., Yamazaki, T., Yazawa, K.: The design and implementation of a first-generation CELL processor. In: IEEE International Solid-State Circuits Conference (ISSCC) (2005)
2. Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P., Yelick, K.: The potential of the cell processor for scientific computing. In: Proceedings of the 3rd Conference on Computing Frontiers (CF '06) (Ischia, Italy, May 3–5, 2006). ACM, New York, NY, pp. 9–20 (2006)
3. Gordon, M.I., Thies, W., Amarasinghe, S.: Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. SIGARCH Comput. Archit. News **34**(5), 151–162 (2006)
4. Kistler, M., Perrone, M., Petrini, F.: Cell multiprocessor communication network: built for speed. IEEE Micro. **26**(3), 10–23 (2006)
5. Eichenberger, A.E., Wu, P., O'Brien, K., O'Brien, K., O'Brien, K.: Vectorization for SIMD architectures with alignment constraints. In: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI '04) (Washington, DC, USA, June 9–11, 2004). ACM, New York, NY, pp. 82–93 (2004)
6. Bellens, P., et al.: CellSc: a programming Model for the cell BE architecture, SC (2006)
7. IBM xl compiler for Cell: http://www.alphaworks.ibm.com/tech/cellcompiler
8. Eichenberger, A.E., O'Brien, K., O'Brien, K., Wu, P., Chen, T., Oden, P.H., Prener, D.A., Shepherd, J.C., So, B., Sura, Z., Wang, A., Zhang, T., Zhao, P., Gschwind, M.: Optimizing compiler for the CELL processor. In: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT) (September 17–21, 2005). IEEE Computer Society, Washington, DC, pp. 161–172 (2005)
9. SDK for Cell: http://www-128.ibm.com/developerworks/power/cell/

10. Chen, T., Sura, Z., O'Brien, K.M., O'Brien, J.K.: Optimizing the use of static buffers for DMA on a CELL chip. In: Workshop on Language and Compiler for Parallel Computing (LCPC), pp. 314–329 (2006)
11. NAS parallel benchmarks: http://www.nas.nasa.gov/Resources/Software/npb.html
12. Spec OMP benchmarks: http://www.spec.org/
13. Paraver: http://www.cepba.upc.es/paraver/