Vector Class on Limited Local Memory (LLM) Multi-core Processors*

Ke Bai, Di Lu, Aviral Shrivastava
Compiler Microarchitecture Lab
Arizona State University
Tempe, AZ 85281, USA
{Ke.Bai, dilu3, Aviral.Shrivastava}@asu.edu

ABSTRACT

Limited Local Memory (LLM) multi-core architecture is a promising solution for scalable memory hierarchy. LLM architecture, e.g., IBM Cell/B.E. is a purely distributed memory architecture in which each core can directly access only its small local memory, and that is why it is extremely power-efficient. Vector is a popular container class in the C++ Standard Template Library (STL), which provides the functionality similar to a dynamic array. Due to the small non-virtualized memory in the LLM architecture, vector library implementation cannot be used as it is. In this paper, we propose and implement a scheme to manage vector *class in the local memory present in each core of LLM multi-core architecture. Our scalable solution can transparently maintain vector data between the shared global memory and the local memories. In addition, different data transfer granularities are provided by our vector class to achieve better performance. We also propose a mechanism to ensure the validity of pointers-to-elements when the vector elements are moved into the global memory. Experimental result shows that our vector class can improve the programmability of vector class significantly while the overhead can be contained within* 7%.

Categories and Subject Descriptors

D.3.m [**Software**]: Miscellaneous; D.1.5 [**Software**]: Object-oriented Programming

General Terms

Algorithms, Design, Experimentation, Performance.

Keywords

Vector, local memory, scratch pad memory, embedded system, multi-core processor, IBM Cell, PS3, MPI

*This research was partially funded by grants from National Science Foundation CCF-0916652, IIP-0856090, NSF I/U-CRC for Embedded Systems, Microsoft Research, SFAz, Raytheon and Stardust Foundation.

CASES'11, October 9–14, 2011, Taipei, Taiwan.

1. INTRODUCTION

As we transition from single core to many cores, maintaining the illusion of a single unified memory in multi-core architecture becomes challenging. There are two main reasons. First is that cache coherency protocols do not scale well to hundreds of cores [7], and second is, that even if possible, the overhead of automatically managing memory like caches is becoming prohibitive in terms of power consumption. Even in single-core processors, caches can consume more than half of the processor power [7], and are expected to consume much larger fraction in many-core systems.

Limited Local Memory (LLM) architecture is a scalable memory architecture, in which each core has a small local memory. Each core can access only its limited local memory. Shown in Figure 1, the IBM Cell B.E. is a popular example of the LLM architecture. It contains one main core, Power Processing Element (PPE) and 8 Synergistic Processing Elements (SPEs). Each SPE has 256 KB of memory [12]. If all the code and data of that task that is mapped to the SPE fit in the local memory of the SPE, then very power-efficient execution is achieved. In fact, the peak power-efficiency of the IBM Cell processor is 5.1 Giga operations per second per watt [17]. Contrast this with the power-efficiency of traditional shared memory multi-cores, e.g., the Intel Core2 Quad is only 0.35 Giga operations per second per watt [17]. However, if the code and data of the application do not fit into the local memory, then the global memory must be leveraged to contain them through explicit DMA calls. This explicit data management is a challenge in LLM architectures.

Standard Template Library (STL) is a popular and generic programming tool and is included in C++ standard library. It provides a set of container classes, which are data structures whose instances are collections of other objects. Vector is a container class which holds data as a dynamic array. As it is dynamic, vector uses a variable size of memory which is proportional to the amount of data it contains. Unfortunately, using STL in LLM architectures is difficult. This is because STL library is not aware of the size of the local memory. When using vector on Cell SPE, vector class works fine with a small amount of data. However, when more data is pushed in the vector, it will throw out an error "*terminate called after throwing an instance of 'std::bad_alloc'* ". This happens when the STL wants to allocate space for more data, but there is no more space in the local memory. To support vector class in a LLM architecture, the vector data must be managed between the local and global memory.

Many works in parallel and multi-thread programming have investigated supporting parallel STL for homogeneous

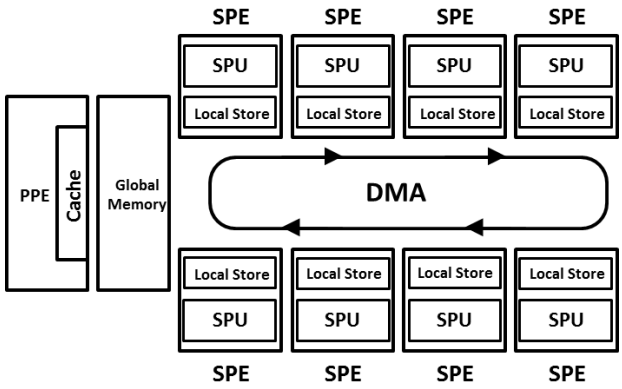

Figure 1: The IBM Cell B.E. is an example of LLM architecture. It has 8 Synergistic Processing Elements (SPE) which can only access its local memory while the Power Processing Element can access the global memory through coherent cache. The data transfers between local memories and the global memory are achieved by explicit DMA calls.

multi-core systems [4, 8, 11, 14, 15, 20]. However, their objective is to design STL containers in which the elements can be accessed by different cores simultaneously. None of the works has considered limited local memory. This is because these solutions have been developed for shared memory multi-core architectures, in which through virtualization, each core can access a large amount of memory, but in an LLM multi-core system, the size of local memory is limited (because it is not virtualized).

In this paper, we propose a scheme to implement vector class on the cores of LLM architectures. Our "completely in software" technique manages arbitrary sized vector data on the limited local memory of the cores. This is done by keeping most of the data in the global memory, and using the local memory as a buffer to the vector data (like a software cache). We preserve the syntax and semantics of the vector API and iterators. A challenge that arises in any data management scheme is that when a data is moved to the global memory, pointers pointing to the data become wrong. We propose a scheme with some functions to resolve these pointers correctly. Some of our observations from the implementation are as follows: i) Performance improves as we increase the buffer size used by the vector on the local memory. ii) For our set of benchmarks, a block size between 16 and 64 elements seems optimal. iii) Higher associativity may result in degradation in performance. This is because more time is spent on searching through the address tag list. iv) Even when more cores are managing vectors in the core, the main core is not overwhelmed by memory requests. Finally this improvement in programmability and automatic management can be provided in less than 7% performance overhead.

2. STL VECTOR ON THE CELL SPE

The STL vector class is a *container* class which can contain a collection of objects. Vector class can perform array-like operations and access elements randomly. The memory allocation is done automatically and transparent to users.

```
main() {
  pthread_create(...spe_context_run(speID)...);
}
```

(a) PPE code

```
main() {
  vector<int> vec1;
  vector<long> vec2;
  for( i = 0 ; i < N ; i++)
    vec1.push_back(i);
}
```

(b) SPE code

Figure 2: Outline of a threaded program on the Cell processor: (1) PPE creates threads on each SPE, and STL vector class is used in the SPE program. (2) In this example, the SPE program works fine if N is small. But, if N is larger or equal to 8192 integers, the SPE program will crash.

Each time when the memory of vector is used up, *vector* will reallocates a double size memory. The same methodology can be applied to other *container* class as well. Rather than container class, Standard Template Library (STL) provides a set of programming interface for generic programming. It mainly includes three parts: *algorithms*, *iterator*, and *container*. *Algorithm* provides programming interfaces for searching and sorting algorithm. *Iterator* is a class that allows users to traverse the elements of a container, which is an interface between containers and algorithms. *Container* classes use template and their collections can be any data types. *vector* class is one of the *container* classes. It is efficient in inserting and removing elements in the back of the vector, and retrieving objects.

The IBM Cell Broadband Engine [12] is a multi-core architecture with limited local memory. As shown in Figure 1, in Cell processor, only Power Processing Element (PPE) memory can directly access the global memory. Each Synergistic Processing Elements (SPE) can only access its local memory. Operating System (OS) run on PPE, but SPEs do not have any OS. Data communication between the global memory and SPE local memories need to be explicitly specified in the program code as DMA calls.

The local memory of SPE has only 256 KB and is shared by the compiled code, global data, stack data, and heap data. The vector data belongs to heap data and grows as the capacity of vector instances grows. Too large size of heap data will crash the program. There are two main reasons. First, since heap data grows towards stack data and therefore they may overwrite each other. Second, if the heap data itself is larger than 256 KB, memory overflow will happen. As shown in Figure 2, the SPE thread in Figure 2 (b) is initiated from PPE thread in Figure 2 (a). For a small N, the program will execute fine, but large values of N will cause failures, i.e. program will end with error "*terminate called after throwing an instance of 'std::bad_alloc'* ".

3. CHALLENGES OF STL VECTOR FOR CELL SPE

The current STL on the Cell processor is from SGI [3], which supports all the functionalities as it is for single-core processor. When the code and all data of the program can

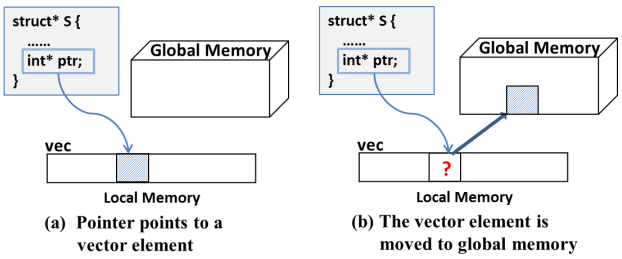

Figure 3: (1) No matter what scheme is implemented, older data need to be evicted to the global memory to make space for new data when the local memory is full. (2) The pointer becomes invalid when the data pointed by it has been evicted.

fit in the local memory of SPE, the current STL can work correctly and efficiently. Since the local memory is small, programmers need to be aware of the memory usage in the local memory every time when more vector data is added. When the total size of code and data is larger than 256 KB, they must be manged between the global memory and the local memories [5,6,16]. The current STL library has no such functionality, and then the burden of managing the data explicitly is transferred to programmers. What is needed is a scheme that can efficiently and intuitively manage the vector data in the local memory of Limited Local Memory (LLM) multi-core architectures. However, two main challenges are found when implementing the new vector class to support unlimited vector data on the local memory of SPE.

- **Management on Global and Local Memory:** The first challenge is where to get the space in the global memory to allocate the evicted vector data from the local memory of SPE. The SPE cannot directly allocates space in the global memory and therefore it requires the thread on the PPE to allocate space for itself. As a result, some way of synchronization must be proposed to achieve this objective. Secondly, although DMA is an efficient way of data transfer, it still has a longer latency compared to the time spent on arithmetic operations. Therefore, when data transfers happens, we must try to minimize the total DMA size. The last but not the least challenge is, when the vector data needs to be evicted to the global memory, what data should be chosen and what is the best granularity.

- **Pointer Problem:** Any scheme for managing the vector data in the local memory will have problems when the corresponding data pointed by a pointer have been moved to the global memory. Therefore, the data resides in the global memory and this pointer becomes invalid. As shown in Figure 3, pointer *ptr* points to an integer of vector in the local memory. After the integer is moved to the global memory, the *ptr* still points to the original address, where the value might be changed to other elements of the vector. Therefore, this pointer becomes invalid. In a unified address space, this kind of pointer problem will not exist. This only happens in the systems where two or more address spaces exist. To enable the use of vector in the LLM multi-core architecture, we need some mechanism to ensure the validity of these pointers.

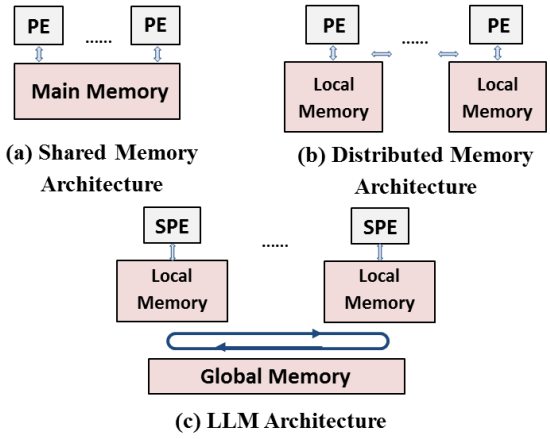

Figure 4: (1) In shared memory architecture, each core can access to a shared memory through coherent cache. (2) In a normal distributed architecture, each Processing Element (PE) has a large local memory to use. (3) In LLM architecture, each SPE only has a small local memory and there is a large global memory on chip. Programmers need to take care of the memory use on each SPE program.

4. RELATED WORK

STL is a software library, which provides data structures like containers and generic algorithm for C++ programming. However, the current STL implementation does not support LLM multi-core architectures well. Many previous works have been done to extend STL for parallel programming. They extended algorithms for parallel processing for different architectures. Work [4] and [20] implemented the containers for parallel programming on shared memory systems. On the other hand, [8], [15], [11] and [19] focused on distributed memory architecture.

MPTL [4] extended the STL algorithms for parallel processing. MCSTL [20] is another project which improves the STL algorithms for multi-core systems. Their works use existing STL container for their algorithms. On the other hand, Intel TBB [14] developed both parallel algorithms and containers. The containers can be concurrently accessed by multiple threads. However, all three of them only concentrated on the shared memory systems as shown in Figure 4(a), without any consideration of distributed memory architecture as in Figure 4 (b) and limited local memory architecture as shown in Figure 4 (c).

POOMA [11], AVTL [19], STAPL [8], and PSTL [15] implemented template container classes on distributed memory architecture as in Figure 4 (b). The classes can be accessed by different threads concurrently. Their works did not consider the case that single Processing Element (PE) may only have a small size of local memory and simply assume the size of local memory is large enough. Consequently, their work are also not applicable for LLM multi-core architecture in Figure 4. Figure 4(b) and (c) shows the main differences between the distributed memory architectures and LLM architectures. In a distributed memory architecture, each core have direct access to a large memory. If a core is assigned a computation task, its local memory is sufficient for the execution of program. However, in LLM architecture, if a

program that run on SPE uses a vector which contains more than 256 KB data in the local memory of SPE, the execution will crash. Programmers need to take care of the size of vector and may have to adjust the vector size on the local memory of SPE. This strongly affects the program development progress. In this work, we address this problem by providing a vector class which hides the programming complexities inside the vector functions. In addition, different data transfer granularity is provided by our vector class to achieve better performance. We also propose a mechanism to ensure the validity of pointers-to-elements when the vector elements are moved into the global memory.

5. OUR APPROACH

5.1 Vector Management on Global Memory

The local memory of SPE is only 256 KB. If we want to increase the capacity of vector class for containing more than 256 KB of objects on the local memory, it is necessary to place some vector data onto the global memory and therefore requires a scheme to manage them. Since SPE cannot directly access the global memory, it cannot perform memory allocation on it. In addition, although DMA is an efficient way of data transfer, it still has a longer latency compared to the time spent on arithmetic operations. If SPE thread simply uses DMA to manage data, the whole process may have a long latency and then cause more penalty. Therefore, our PPE library thread also supports some basic operations, e.g. memory reallocation, on the global memory.

The Cell/B.E. architecture supports two different ways of communication: *DMA* and *mailbox*. DMA provides the functionality of transferring data from one memory to another memory, e.g. from the local memory of SPE to the global memory. Mailbox supports thread to thread communication, such as SPE thread and PPE thread, which can be leveraged for synchronization. These two communication methods have different advantages in different situations. DMA is fast in transferring data and capable of transferring a large amount of data in a time, but mailbox is slower and can only exchange 32 bits short messages. On the other hand, DMA itself can not directly allocate space on the global memory. What is required is a hybrid way of *DMA* and *mailbox*. Figure 5 illustrates the whole process of reallocating vector on the global memory. In the predefined space in the global memory, there has a data structure named *msgStruct*. Different operations have different elements in the *msgStruct*. In this example, we only show the information needed for reallocation operation – *vector_id* specifies which vector should be reallocated, *request_size* indicates the new vector capacity, *data_size* is the number of vector elements that needs to be moved and *new_addr* can be used to return the new global address after reallocation. When SPE thread wants to communicate with PPE thread, it first sends the parameters to *msgStruct* in the global memory by DMA. Then, it sends the type of operation to PPE thread, and starts waiting for the restart signal from the mailbox. *spe_out_intr_mbox_read*() is leveraged in the PPE thread to read the interrupt mailbox for the message from SPE thread. By interpreting the message, the PPE thread can know which basic operation needs to be performed. In this case, memory reallocation is requested and the reallocation is performed with the help of *msgStruct*. After the PPE thread reallocates the vector data, it puts the new global address in *msgStruct*, and sends back a restart signal through mailbox to SPE thread to let it DMA in the *msgStruct*. Then, PPE thread keeps on listening to the next mailbox message. Finally, SPE thread receives the new global address of vector data and continues executing. Since the PPE thread uses the interrupt mailbox, the mailbox read function is a blocking function. When PPE thread is waiting for mailbox message, it consumes very few CPU resources.

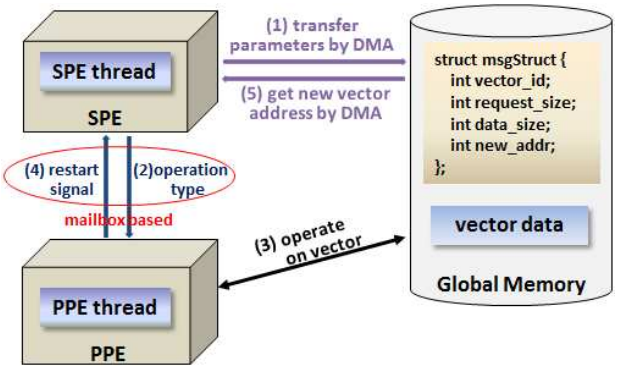

Figure 5: (1) The whole process for memory reallocation is shown and the number means the order of steps. (2) Important information for reallocation is contained in a data structure named *msgStruct*, which is located in the global memory. It is 16 bytes large, but elements can be different depending on the type of operations.

Our PPE library thread supports four operations on the global memory: i) *allocation* ii) *re-allocation* iii) *memory copy* iv) *deallocation*. All operations use the above communication scheme. Allocation is used when a vector is initialized. It allocates space on the global memory, and stores the start address of the allocated memory and its size in *msgStruct*. Reallocation is used when the memory of a vector needs to be expanded or shrunk. The PPE thread first looks up the vector record, then it reallocates the vector. The new size of space is obtained from *msgStruct*. Note that *msgStruct* is used for passing parameters between the PPE and the SPE, as discussed in the last paragraph. Memory copy operation is similar to the function *memcpy*() and it is used for *insert*() member function in vector class. Besides the memory copy functionality, this operation also checks whether the space on the global memory is full and needs reallocation. It will reallocates the memory for vector if necessary. SPE thread will provide the necessary information in the *msgStruct*, such as the source address, the destination address, and the data size. The deallocation operation simply uses the global address to free the allocated space in the global memory. The allocation and reallocation operations are implemented for supporting dynamic memory management. Although pre-allocated buffers for SPE program are very common, the static use of the memory has disadvantages: i) if the number of vector elements is small, the utilization of the allocated memory is quite low; ii) if the number of vector elements is very large, memory overflow will happen. To better utilize the memory in the global memory, we definitely need these two functions. As for memory copy operation, we can also implement only through DMA and SPE thread. There may have two schemes. The first one

is done by transferring the block of vector elements on the global memory into a buffer in the local memory, and then transferring them from this buffer to the new destination address on the global memory. However, the DMA transfers on Cell/B.E. request the transfer size be 1, 2, 4, 8 or 16 bytes, or a multiple of 16 bytes to a maximum of 16 KB. This means this scheme can only be used when a distance between the destination address and the source address is a multiple of 16. In real application, it is a rare case. The other scheme is to transfer a whole block of elements that contains the source address into SPE buffer. After that, SPE thread moves the elements in SPE buffer, and then transfers the block back to the global memory. Compared to simply using PPE thread to do memory copy, this scheme moves the computation overhead of memory copy to SPE thread whose execution environment is slower than PPE, and also introduces extra DMA transfers which will degrade the performance. Therefore, we think that using PPE thread and mailbox is the best choice for memory copy.

5.2 Vector Management on Local Memory

In the local memory of SPE , we use a buffer to cache the vector elements. The purpose of using SPE buffer is to improve the performance of the vector retrieval function, since if vector can find its elements in the local memory, then there is no need to use DMA to get it from the global memory. With a constant-sized buffer in our newly implemented vector library, "not-recently-needed-data" must be evicted from the local memory to the global memory if more elements than SPE buffer can contain are brought into the local memory. Since vector data are always located in the same region of the local memory, memory fragmentation in the current STL vector class will not exist in our scheme. The vector elements are organized in blocks and several blocks can coexist in the SPE buffer. Besides the actual data, each block also contains a block index and the number of elements in the block. The block index is the index of the first element in this block. Inside a block, the order of elements is the same as their original orders in the vector. When vector wants to locate an element, it first calculates the block index this element belongs to. As shown in Figure 6, by simple calculation, vector can know the 133^{th} element is in the block with block index 128. Then, we can check whether this block is in the local memory or not. If the requested element is not in the local memory, the whole block that contains the requested element will be fetched into the SPE buffer. In addition, the global address of the requested block G_{rb} can

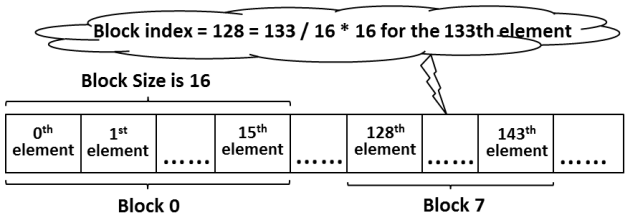

Figure 6: (1) The block size of vector is 16 and the elements are divided into blocks by their index to the first element of the vector. (2) In order to get the block index of the element with element index 133, we can simply calculate it by $133/16 * 16 = 128$.

be calculated by:

$$G_{rb} = G_s + Index_{rb} \times element\_size$$

where $element\_size$ is the number of bytes that an element occupies, $Index_{rb}$ is the block index of the requested block, and G_s is the global address of the first element in vector. G_s is a member of our vector class, which is stored when allocation or reallocation happens. Each time vector allocates or reallocates the space on the global memory, a new G_s will be generated, be stored in $msgStruct$ and finally be transfered to SPE.

Our vector implementation is similar to software cache and two types of SPE buffer, Direct-Map and Associative, are implemented. In the former one, we keep all the blocks in an array and use the block index to check if it is valid. As for associativity buffer, we have a hash table with linked-list data structures. The number of sets is the quotient of total number of cache blocks divided by associativity, and the sets is implemented as an array. Each set has a linked list of empty blocks and a linked list of valid blocks. Each time when we need to add a block, we remove one from the empty list and add it to the valid list. If the program needs one more block but there is no more blocks in the empty list, we use Least Recently Used (LRU) replacement policy to replace the oldest block in the valid list. The data in the oldest block will be written to the global memory. On the other hand, if one block is invalidated, we remove it from the valid list and add it to the empty list.

5.3 Vector Function Implementation

Our aim is to provide a vector with the same semantics as the current STL vector. The implementation complexities are hidden inside each library function. We implemented all the member functions of vector but only 3 commonly-used vector functions (at, $push\_back$ and $insert$) are shown in this section, due to the limited pages of the paper.

$at()$ is the retrieval function. It receives an offset as parameter and returns a reference to the vector element. Generally, in order to return a reference to the vector element, vector keeps a copy of the requested element on the local memory. In our implementation, it first checks if the current vector elements on the local memory is the requested elements. If they are not the requested ones, this function will get the requested elements from the global memory and put the existing data on the local memory to the global memory by non-blocking DMA transfers. Since this function does not require memory allocation, the data communication can be all efficiently handled by DMA operation.

$push\_back()$ is an efficient way of inserting elements. It inserts the elements in the back of the vector data. It first checks whether the block that contains the inserting position is in the local memory. If yes, it directly inserts the element. Otherwise, it brings in the block by DMA first, then insert operation can be made. Besides the operations in the local memory, $push\_back()$ also checks if the allocated region in the global memory is full. When the original space is full, it will reallocate new space in the global memory.

$insert()$ allows the users to insert elements in arbitrary position of the vector. This operation requires moving all the elements after the inserting position on the global memory to give enough space for the inserting elements. Vector class uses the PPE thread to perform the memory copy operation. We illustrate the insertion procedure in Figure 7.

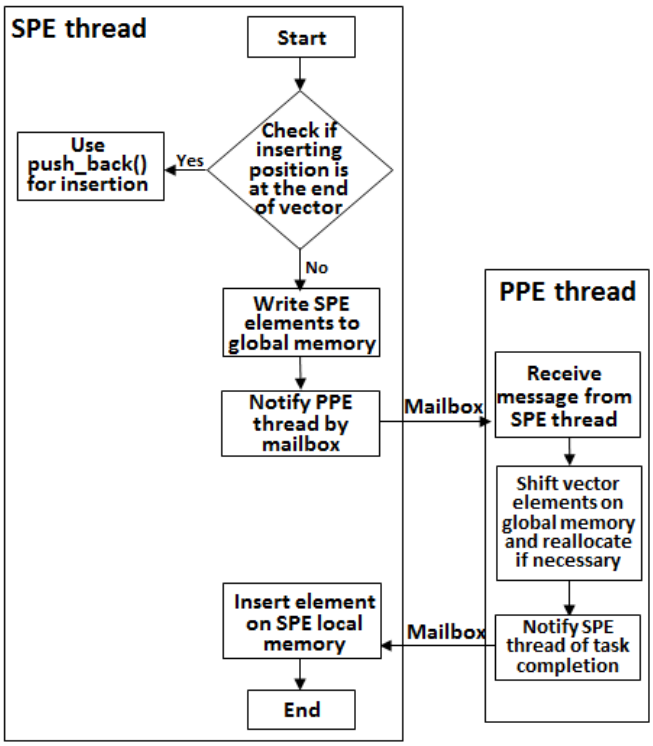

Figure 7: Flow chart of *insert()* function: (1) Write the elements from the local memory to the global memory to avoid data inconsistency that introduced by elements shifting on the global memory. (2) Use mailbox to send message to PPE thread. (3) PPE thread completes the shifting and sends a response message. (4) SPE thread finishes insertion on the local memory of SPE.

The function first checks whether the inserting position is after the last element of vector. If yes, it only need to call *push_back()* function to insert the element. Otherwise, the block containing the inserting position and all blocks behind this position will be moved to the global memory. Then this function shifts the elements after the inserting position on global memory. The memory shift process is handled by the memory copy operation which is supported by PPE thread as we mentioned in Section 5.1. Finally, the block that contains the inserting position is fetched to the local memory, and the element is inserted at the inserting position.

5.4 Iterator Implementation

iterator of vector class is like a cursor that allows user sequentially access the elements in the containers. Similar to the iterator of current STL vector, we implement three types of operators — *random access iterator*, *forward iterator*, and *bidirectional iterator*, which allow the user to do the pointer arithmetic, and support pointer motion in both directions. The iterator of STL vector is actually a pointer to the vector elements. As the pointer problem discussed in Figure 3, *iterator* needs to consider the situation that the elements may be scattered between the local memory and the global memory, instead of assuming all elements are located within one address space. Besides, it must be aware of different copies of data in the global memory and the local memory. Otherwise, it will lead to data incoherence problem. In our implementation of *iterator* in vector class, these two situations are considered, and therefore there will not have any data incoherence problem. Our *iterator* keeps a pointer that points to the vector class instance, and it has a cursor parameter to keep the displacement between the current position and the position of the first element of the vector. When SPE program calls the $*$ operator of iterator, the *iterator* passes the cursor as a parameter to vector function $at()$ and calls it. The $at()$ function will then returns the reference to the element indicated by the cursor. The purpose of doing this is that *iterator* can use the element in the SPE buffer of vector instead of creating another copy of element data in SPE local memory. In addition, since $at()$ has handled the data distribution and communication, the cursor can indicate the vector element without knowing its location. For the pointer arithmetic and pointer motion operators, the arithmetic operation is applied on the cursor, e.g. for ++ operator, *iterator* increments the value of cursor by 1, and then returns the current iterator.

5.5 Pointer Resolution

To ensure that the pointers pointing to the vector elements are valid, we develop a mechanism to bring in the requested vector elements into SPE local memory when necessary. To be able to do it, we first need to identify all *potential pointers*[1] that may point to a vector element. This process needs helps of other pointer analysis tools or can be done manually. Second, pointer management functions must be used to manage them. We provide three functions for handling pointers:

$$\begin{array}{lll} template\langle class\_Tp\rangle & void* & ppu\_addr(vector\langle\_Tp\rangle, int) \\ & void* & ptrChecker(void*) \\ & void* & s2p(void*) \end{array}$$

$ppu\_addr()$ is used to return a global address to the first pointer that is assigned an address pointing to a vector element. The reason why we return global memory address is that the local address is not uniquely mapped to a vector element. It's because the SPE buffer for vector is shared by all vector elements, and different vector elements may have the same local address but different global addresses. By using a local address to identify different vector elements is therefore impossible. $ptrChecker()$ first checks whether the pointer is pointing to a vector element. If it is, the function checks whether the element is in the SPE vector buffer. If the requested element is in SPE buffer, then this function returns its local memory address. Otherwise, it fetches the elements into SPE buffer, and then returns its local memory address. In addition, if the parameter of $ptrChecker()$ was modified to other local pointer and therefore not pointing to any vector element in the global memory, the function just returns the original value. $s2p()$ is used to restore the pointer address back to a global address. This function only processes the pointers that are transformed by $ptrChecker()$.

Figure 8 shows the process of pointer management needs to be done manually. In the Figure 8 (a), all *potential pointers* (pointer **a**, **b** and **c**) are highlighted by red color. At *Line 3*, pointer **a** is initialized with the address of a vector

[1]We call the pointer that may point to a vector element as *potential pointer*. *Potential pointers* are either once assigned to a local memory address of a vector element or assigned by another potential pointer.

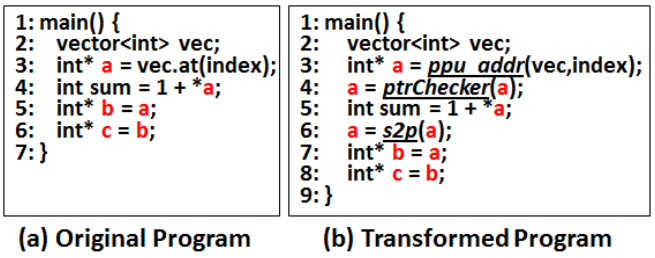

Figure 8: Pointer a is assigned a global memory address by function *ppu_addr*. *ptrChecker* checks whether the pointer is pointing to a vector element and returns its local address. $s2p$ restores the address of the *potential pointer* from a local address to a global address.

element, and it is a *potential pointer* that points to a vector element. Pointer **b** becomes *potential pointer*, since the assignment in *Line 5* shows **b** gets the value of **a**. Similarly, **c** is also the *potential pointer*. In the Figure 8(b), we change the assignment to pointer **a** by function *ppu_addr*() in *Line 3*, and the pointer **a** now contains a global address. Then, at *Line 4*, *ptrChecker*() is added into the original code. It checks its address and fetches the vector element of *vec* to SPE local memory if necessary, and transforms the address of **a** to its local address. After accessing the content of **a**, we use $s2p$() in *Line 6* to restore the address of **a** to the global address of the element that it points to. Note that if the content pointed **a** is not accessed in the program, there is no need to use our interface to ensure the pointer validity. For example, in the Figure 8 (a), *Line 5, 6* are pointer initializations, not data read or data write, then no management function is needed in the Figure 8 (b).

5.6 Comparison With Software Cache

Software Cache [9, 10] is a technique that allows the data sets to span over the local memories of SPEs and the global memory. Software cache can fetch/store data from the global memory by replacing DMA calls with software cache read-/write interfaces. However, it is complicate or even impossible to be used to extend the vector class on Cell SPE, since it is mainly used for accessing data on the global memory. Specifically, the data are originally generated on the global memory, and using software cache can reuse the data on the local memory. Although the IBM Single Source Research Compiler [10] allocates SPE program data in the global memory and has the compiler and runtime automatically manage the movement of the data between the global memory and local memory, programmers initially needs to know the global address. Therefore, programmers need to explicitly program the dynamic allocation code on both PPE program and SPE program. We illustrate the complexities of using software cache to manage vector data by transforming the original code in the Figure 2 into the code in the Figure 9. Figure 9 shows one scheme about how software cache manages vector data. Two points need to be pointed out: i) Programmers must create a new thread to allocate space in the global memory and send the global address to the SPE code through *mailbox*. It's error-prone and counter-intuitive. However, we provide a library thread on the PPE and programmers only need to directly call it and all com-

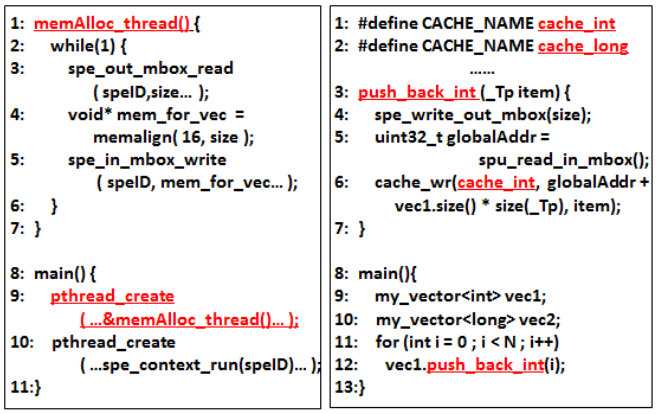

Figure 9: Using software cache to extend the vector class: (1) Programmers need to explicitly write a PPE thread to allocate memory for vector on global memory. (2) In the SPE code, a software cache can only support one data type, therefore different types of data requires different declarations of software caches. (3) If there are more than one template classes used in vectors, several versions of vector functions need to be implemented. In this example, there must be two versions of *push_back* functions if *vec2* is used, one uses *cache_int* for *cache_wr*, and the other one uses *cache_long* for *cache_wr*.

plexities are hidden. ii) If there are plenty of data type in the SPE code, the software cache requires programmers to explicitly declare that number of software cache, since one software cache only supports one data type. Even worse, if you want to warp software cache functions to implement vector class, it is also complex. We need to implement different *push_back* functions for different data type, since *cache_wr* needs to access the correct cache. With our vector class, the only change is a library function call in the PPE code.

6. EXPERIMENTS

6.1 Experiment Setup

Our experimental environment is the Cell Processor in Sony PlayStation 3, installed with Fedora 9 and IBM SDK version 3.1. We conduct experiments on 7 benchmarks — heap sort [2], radix sort [2], dijstra [1], FFT [13], invfft [13], SOR [18], and sparse matrix [18] for measuring the performance and effectiveness of our technique. The basic descriptions of different benchmarks and the number of elements in each benchmark are shown in Table 1. We use $\_mftb()$ and $time()$ to measure the runtime. To minimize the impact of hardware and operating system, we execute each experiment 10 times and take the average of them.

6.2 Programmability Improvement

To demonstrate the effectiveness of our vector class, we apply our vector and the current STL vector class on the benchmark *heap sort*. The code size of this benchmark is 36432 bytes, the size of global data is 12340 bytes and the stack size is less than 1 KB. The remaining space can be used for vector. In this experiment, we use a direct map buffer

Benchmarks	Description	Max Data Size (# of Elements)
heap sort	heap sort algorithm	1,000,000
radix sort	radix sort algorithm	2,000,000
FFT	FFT algorithm	4,196,352
invfft	Inverse FFT algorithm	4,196,352
dijkstra	dijkstra algorithm	1,003,000
SOR	Successive Over-relaxation	1,000,000
sparse matrix	Sparse Matrix Multiply	2,100,001

Table 1: Description of benchmarks

for our vector which can contain 16384 integers and change the input data from 100 integers to 5,000,000 integers.

As shown in Figure 10, the original STL vector class can only support 8,192 integers at most. This is because vector class creates large fragmentations which we will delve into more explanations in the next paragraph. In addition, there is no management of vector data between the global memory and the local memory. When the data exceeds 8,192 integers and vector starts to reallocate the memory space, the SPE program crashes. However, our vector class can handle more than 5,000,000 integers. We also observe that although our vector class has management overhead, it does not have a significant impact on the performance of the application when the input data size is not large. For input data size 8,192, the additional overhead caused by our vector functions is 7% of the runtime of the original STL vector.

Figure 11 shows details of the fragmentation problem caused by the current STL vector classes on the local memory. Let us assume the first memory allocation starts at memory address s_0 and it allocates 4 bytes. Since each time vector will reallocate 2× space of its current allocated memory size from the end address of the current memory allocation, the vector will allocate spaces at address s_0+4 for the 1^{st} reallocation, at address s_0+12 for the 2^{nd} reallocation, $\cdots$, and at address $s_0+2^{n+2}-4$ for the n^{th} reallocation. Even worse, the reallocation will never reuse the memory space before its current space, since it is never large enough to support the required memory space. In processors with OS monitoring, memory fragmentation caused by vector can be reused. However, in a processor like SPE, which does not have OS, several kilobytes of memory fragmentation is wasted.

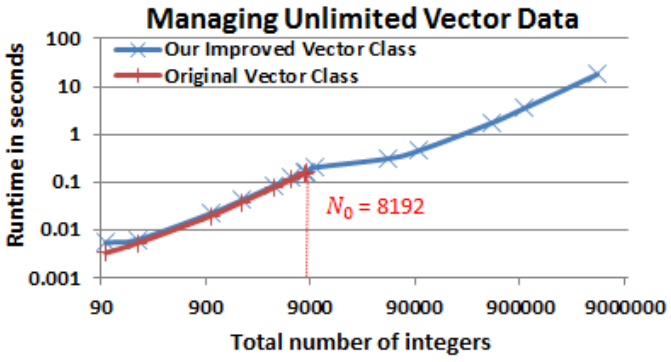

Figure 10: (1) The original vector class can contain at most 8192 integers. (2) Our new implemented vector class can support almost unlimited vector data.

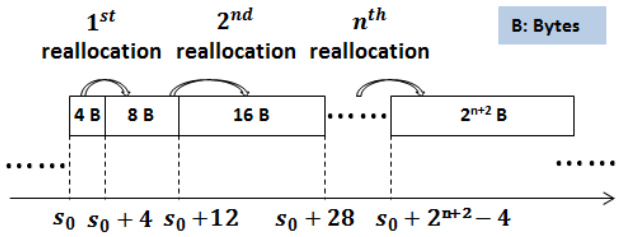

Figure 11: Severe fragmentation will be caused by the current STL library on the local memory.

Another interesting observation is that the runtime of benchmark that using our vector class has a higher runtime than that using the current STL vector when the input data size is very small, e.g. less than 500 integers. This is because of the high overhead of initialization of our vector class. In this experiment, the SPE buffer can contain 16384 elements and this buffer is divided into 1024 blocks. Consequently, at the initialization time, vector needs to allocate memory for all these blocks which creates high overhead. On the contrary, STL vector only reallocates for several times for few hundreds of elements, and the reallocation for a small memory space is much cheaper.

6.3 Impact of Vector Management Parameters

6.3.1 *Impact of Buffer Size*

In this experiment, we evaluate the effectiveness of the SPE buffer by using different buffer sizes on different benchmarks. A larger buffer size means buffer can contain more data in SPE local memory, and the performance should not get worse. We fix the block size[2] to 16, and increments the total buffer size from 512 to 4096 by increasing the number of blocks, since $buffer\_size = number\_of\_block \times block\_size$.

As shown in Figure 12, the runtime of *heap sort*, *FFT*, *SOR*, *sparse matrix* and *invfft* decreases as the total buffer size increases. As for *dijkstra* benchmark, we observe that there is a big drop in runtime when the data size increases from 512 to 1024. This is because there are three vectors in this benchmark and each uses only 1000 elements which can be all contained in SPE buffer. It results in a significant drop in data transfer overhead, and the total runtime of the program. Another interesting thing is that, the runtime of *radix sort* is not affected by the buffer size. Since the algorithm accesses the elements sequentially, when its data size is larger than the buffer size, there are only misses on access to the first element of a block.

6.3.2 *Impact of Block Size*

In this experiment, we evaluate the impact of block size of buffer on different benchmarks. The total buffer size is fixed to 512 elements and the direct map buffer is adopted. The block size is also the granularity of communication between the local memory of SPE and the global memory. A larger block size provides the functionality of prefetching as it brings in a few nearby elements together to the local memory.

[2]SIZE in this paper means the number of elements with one data type, e.g. buffer size 10 means the buffer can contain 10 elements.

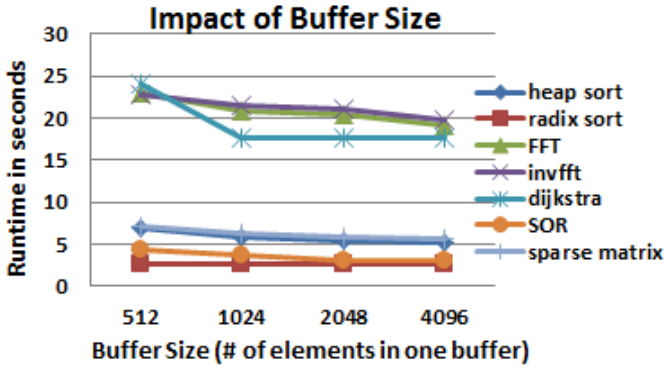

Figure 12: The block size is fixed at 16 and the buffer size is changed from 512 to 4096.

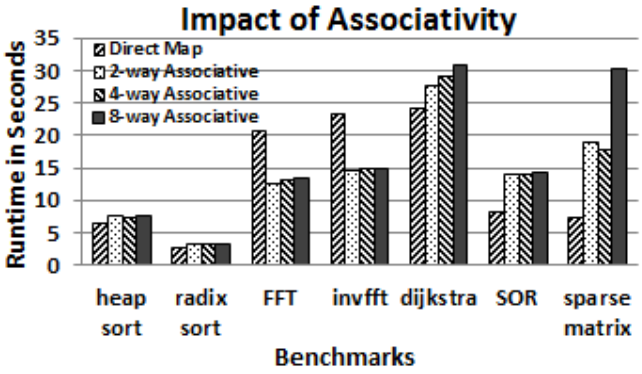

Figure 14: 4 different associativities of vector buffer are implemented and the effect on different benchmarks are shown.

As we observe in Figure 13, *radixsort* can be improved by simply increasing the block size of the buffer. Because it accesses data sequentially in nature, a large block size takes advantage of data prefetching and higher data transfer bandwidth. However, most benchmarks can achieve the optimal performance for the block sizes between 16 and 64, since there is a trade-off between the transfer granularity and data locality. When the block size is small, increasing the block size can increase the reuse the data. After some time, when increasing the block size, the data locality is not increased too much but the overhead introduced by the larger transfer size increases a lot.

6.3.3 *Impact of Associativity*

In this experiment, we evaluate the effectiveness of different buffer associativity in different benchmarks. In all investigations of buffer associativity, the SPE buffer of vector is set to 32 blocks with 16 elements in each block. Since this experiment only compares the effectiveness of buffer associativity, we do not use all the local memory of SPE. The purpose of using a buffer with a higher associativity is to decrease the miss rates and reduce the number of DMA transfers. In the implementation of software buffer, a higher associativity will incur higher computations, such as the computation spent on looking up the management data structure. On the other hand, higher associativity may have a better hit ratio. If the benefit brought by high hit ratio beats the computation overhead, higher associativity is better. Otherwise, higher associativity will degrade the performance. The reason here explains all results shown in Figure 14. However, some benchmarks are explained in more details.

In the benchmark *heap sort*, *direct map buffer* performs the best. It is because the cheaper computation cost on looking up blocks compensates the performance degradation caused by higher data miss. 4-way associative buffer performs better than 2-way associative buffer, since it has a higher hit ratio with little increase of computation cost. 8-way associative buffer has the highest hit rate, since the increase of computation overhead exceeds the decrease in communication cost that higher hit ratio brings. In the experiments of *FFT* and *invfft*, associative buffer always performs better. All associativities have the same miss rate, and their runtime increase with the increase in associativity due to higher computation cost. Direct map buffer almost has 2 times of runtime as that of associative buffer, since it has only about 2 thirds of hit rate as that of associative buffer. In addition, We observe that associativity will not affect the benchmark *radixsort*, and conversely introduces more computation overhead. The data access pattern of this algorithm is sequential, and the program will always have a miss for the first access in a block due to the limit size of SPE buffer. Therefore, associative buffer performs worse than direct-map buffer as it has a higher computation cost as providing the same hit rate.

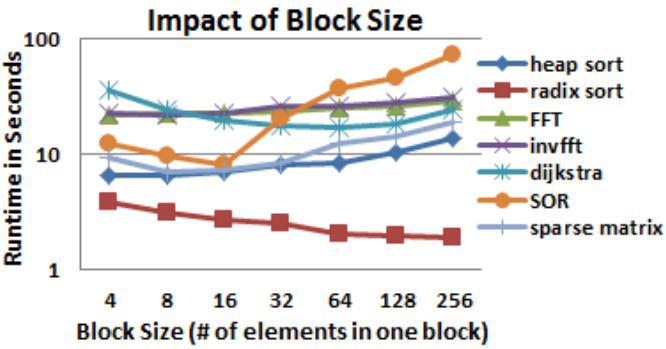

Figure 13: The buffer size is fixed at 512 and the block size is changed from 4 to 256.

6.4 Scalability

In this experiment, we evaluate the scalability of benchmarks that uses our vector class. Vector class used in the benchmarks is implemented with direct map buffer which has 32 blocks and block size is 16. We run identical program over different number of cores, and compare the runtime to evaluate the scalability.

As shown in Figure 15, *dijkstra* exhibits a good scalability which only has a 1% increase in runtime when we scale up the number of execution cores. *FFT* also only has 10% increase in runtime when the number of cores is increased to 6. However, *heap sort* and *radix sort* show a higher percentage of increase in runtime. Our analysis of benchmarks reveals that the additional overhead comes from cores competition over shared resources, e.g. DMA engines.

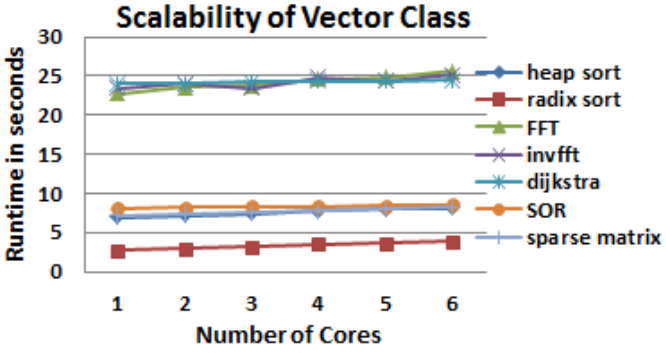

Figure 15: Comparison of runtime when execute benchmarks on different number of cores.

7. CONCLUSION AND FUTURE WORK

In this paper, we propose a vector class with data management on Limited Local Memory (LLM) multi-core architecture. Our implementation provides the same interface as the STL vector class and the programming difficulties in data management are hidden inside the vector functions. Our experiments shows that our vector has improved programmability significantly. On the other hand, when the vector data is small enough to enable the execution of the original STL vector, our vector class has additional 7% runtime overhead. In addition, we implement a software-managed buffer to reduce the data transfers between the local memory of SPE and the global memory. Finally, we present a scheme to assure the validity of the pointers which point to a vector element. Our future work will focus on implementing other STL containers on LLM multi-core architectures. More flexible caching techniques on LLM architectures also will be investigated, including variable-length cache line and replacement policy. As future multi-core architectures are likely to have limited local memory, improving programmability on LLM architecture can greatly improve the programming efficiency.

8. REFERENCES

[1] http://en.wikipedia.org/wiki/dijkstra%27s_algorithm.

[2] http://users.cis.fiu.edu/~weiss/dsaa_c2e/sort.c.

[3] http://www.sgi.com/tech/stl/.

[4] D. Baertschiger. Multi-processing template library. Master's thesis, UniversitéŽSe de Gen'eve, 2006. http://spc.unige.ch/mptl.

[5] K. Bai and A. Shrivastava. Heap data management for limited local memory (llm) multi-core processors. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES/ISSS '10, pages 317–326, New York, NY, USA, 2010. ACM.

[6] K. Bai, A. Shrivastava, and S. Kudchadker. Stack Data Management for Limited Local Memory (LLM) Multi-core Processors. In *ASAP '11: Proceedings of the 2011 International Conference on Application-specific Systems, Architectures and Processors*, 2011.

[7] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *CODES'02:Hardware/software codesign*, pages 73–78, New York, NY, USA, 2002. ACM Press.

[8] A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger. Stapl: standard template adaptive parallel library. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, SYSTOR '10, pages 14:1–14:10, New York, NY, USA, 2010. ACM.

[9] T. Chen, T. Zhang, Z. Sura, and M. Tallada. Prefetching irregular references for software cache on cell. In *CGO'08:The sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 155–164, New York, NY, USA, 2008. ACM Press.

[10] A. E. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind. Optimizing compiler for the cell processor. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, pages 161–172, Washington, DC, USA, 2005. IEEE Computer Society.

[11] J. R. et al. Pooma: A framework for scientific simulations on parallel architectures. In G. V. Wilson and P. Lu, editors, *Parallel Programming using C++*, pages 553–594. MIT Press, 1996.

[12] B. Flachs, S. Asano, S. Dhong, H. Hofstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, B. Michael, H.-J. Oh, S. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, N. Yano, D. Brokenshire, M. Peyravian, V. To, and E. Iwata. The microarchitecture of the synergistic processor for a cell processor. *IEEE Solid-state circuits*, 41(1):63–70, 2006.

[13] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. Mibench: A free, commercially representative embedded benchmark suite. *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14, 2 Dec. 2001.

[14] Intel Corporation. *Reference for Intel Threading Building Blocks*, 2006.

[15] E. Johnson. *Support for Parallel Generic Programming*. PhD thesis, Indiana University, Indianapolis, IN, 1998.

[16] S. C. Jung, A. Shrivastava, and K. Bai. Dynamic Code Mapping for Limited Local Memory Systems. In *ASAP '10: Proceedings of the 2010 International Conference on Application-specific Systems, Architectures and Processors*, pages 13–20, 2010.

[17] Y. Kim, J. Lee, A. Shrivastava, and Y. Paek. Operation and data mapping for cgras with multi-bank memory. *SIGPLAN Not.*, 45(4):17–26, 2010.

[18] R. Pozo and B. Miller. Scimark 2.0. http://math.nist.gov/scimark2/.

[19] T. J. Sheffler. A portable mpi-based parallel vector template library. Technical report, 1995.

[20] J. Singler, P. Sanders, and F. Putze. Mcstl: The multi-core standard template library. In *Euro-Par 2007 Parallel Processing*.