STL on Limited Local Memory (LLM) Multi-core Processors

by

Di Lu

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved February 2012 by the
Graduate Supervisory Committee:

Aviral Shrivastava, Chair
Partha Dasgupta
Karamvir Chatha

ARIZONA STATE UNIVERSITY

May 2012

ABSTRACT

Limited Local Memory (LLM) multicore architectures are promising power-efficient architectures will scalable memory hierarchy. In LLM multicores, each core can access only a small local memory. Accesses to a large shared global memory can only be made explicitly through Direct Memory Access (DMA) operations. Standard Template Library (STL) is a powerful programming tool and is widely used for software development. STLs provide dynamic data structures, algorithms, and iterators for vector, deque (double-ended queue), list, map (red-black tree), etc. Since the size of the local memory is limited in the cores of the LLM architecture, and data transfer is not automatically supported by hardware cache or OS, the usage of current STL implementation on LLM multicores is limited. Specifically, there is a hard limitation on the amount of data they can handle. In this article, we propose and implement a framework which manages the STL container classes on the local memory of LLM multicore architecture. Our proposal removes the data size limitation of the STL, and therefore improves the programmability on LLM multicore architectures with little change to the original program. Our implementation results in only about $12-17\%$ increase in static library code size and reasonable runtime overheads.

To my family

ACKNOWLEDGEMENTS

Foremost, I would like to express my sincere gratitude to my advisor Prof. Aviral Shrivastava for the continuous support of my master study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my master study.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof. Karamvir Chatha and Prof. Partha Dasgupta, for their encouragement, insightful comments, and hard questions.

I thank my fellow labmates in CML Group: Ke Bai, Chuan Huang, Jian Cai, Fei Hong, Reiley Jeyapaul, Bryce Holton, Jing Lu, Mahdi Hamzeh, Yooseong Kim, Jared Pager, Tushar Rawat, Abhishek Rhisheekesan, Shashank Reddy Kaareddy, and Russel Dill, for the stimulating discussions and for all the fun we have had in the last two years. Also I thank my friends in ASU: Weijia Che, Fengze Xie, and Yunji Zhong.

Last but not the least, I would like to thank my parents, for giving birth to me at the first place and supporting me spiritually throughout my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Single core architectures can no longer meet the demands of simultaneous high performance and low power consumption. Multicore architectures provide a way to improve the total throughput of the system without much increase in the power consumption [19]. In addition to the challenge of power-efficiency, multicore architectures also help in tackling several other challenges, e.g., of reliability, and temperature at a much higher level of granularity [20].

As we transition from single to many cores, maintaining the illusion of a single unified memory in multicore architecture becomes a major challenge. This is not only because the power and performance overhead of maintaining data coherency increases as we scale the number of cores [18], but also because the overhead of automatic memory management, i.e., using caches is becoming prohibitive in terms of power consumption. Even in a single-core processor, caches consume more than half of the total chip power [9], but in multicore systems, owing to the coherency traffic, caches are expected to consume a much larger fraction of the total processor power.

To avoid the fixed design-time and minimize the run-time overheads of a shared memory system, multicore architectures are including scratch-pad memories in the cores. New DSP multicore architecture TMS320C6472 [48] and ADSP-BF561 [1] feature a local memory inside each processing core. More boldly, IBM Cell BE [37] uses only a local memory without any hardware cache on each processing core. Such multicore architectures are called Limited Local Memory (LLM) multicore architecture. In LLM multicores, the processing core can only access the code and data in the local memory. Access to global memory, and the memory of other cores has to be explicitly done through the use of Direct Memory Access (DMA) commands. If all the code and data required by a core fits into the local memory, then extremely power-

efficient execution is achieved – and this is the promise of LLM multicores. In fact, the peak power-efficiency of the IBM Cell processor is 5.1 Giga operations per second per watt. Contrast this with the power-efficiency of traditional shared memory multicores, e.g., the Intel core 2 quad is only 0.35 Giga operations per second per watt [32].

However, if the code or data required by the task executing on the core does not fit into the local memory, then the memory management must be done explicitly in the application. This explicit data management is one of the biggest hurdle in the success of LLM architectures. One important manifestation of this limitation is in using Standard Template Library for programming. Standard Template Library (STL) is a part of the C++ standard library, that is a powerful programming tool and is widely used for software development [46]. STLs provide dynamic data structures, i.e., their size can be changed at runtime, allows data type binding at the compile-time, i.e., programmers can use any data type when using the STL, and therefore greatly improves programming through code reuse [46, 39].

Unfortunately, STL implementations assume the presence of unlimited memory on each core, which is not true for Limited Local Memory (LLM) cores. As a result, if the size of the dynamic data structure is larger than the size of the local memory, the program using STL will not work. In this article, we propose a scheme to automatically perform data management for STL for LLM multicores. When more data is instantiated, our STL implementation automatically moves some part of the data to the global memory through the use of DMAs inside our STL implementation, transparent to the programmer. We preserve the syntax and semantics of the STL functions. Our "completely in software" technique can enable seamless use of STL on LLM architectures, improving their programmability. A major 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 also propose a scheme to resolve these pointers correctly.

The rest part of thesis is organized as follows. In Chapter 2, we will introduce the background of Limited Local Memory (LLM) architecture and Standard Template Library (STL). In Chapter 3, we explain why we want to extend STL on the LLM architecture. In Chapter 4, we list the challenges in extending the STL on the local memory of LLM cores. In Chapter 5, we list the related works which implement or extend the template library on different memory architecture. We have an in-depth discussion about why the software cache cannot solve our problem, and also why the current scratchpad memory data management works do not work in our case. In Chapter 6, we will go into the detail our implementation and methodology to modify the STL. In Chapter 7, we show that our extended STL significantly increased the capacity of the STL on the local memory of LLM core.

Chapter 2

BACKGROUND

2.1 Limited Local Memory architecture

The Limited Local Memory (LLM) multicore architecture is an emerging memory hierarchy for high power efficiency. Figure 2.1 shows an example of an LLM architecture. LLM multicore architectures are programmed in MPI-like task-based programming style. Each PE runs a single task with some inputs distributed by main core. In the LLM architecture, there is a local memory inside each Processing Element (PE) and a large global memory on main core. Since there are no hardware caches the data transfers among different memories need to be specified in the software. These data transfers are done through Direct Memory Access (DMA) commands inserted inside the application code. To use DMA, programmers need to provide the destination address, the source address, the number of bytes to be transferred, and the DMA channel, etc. Normally, there are some restrictions when using DMA operation, which are that the destination address and source address must be aligned to a multiple of some numbers, e.g. 16 or 128. In addition, the number of bytes to be transferred should also be a multiple of some specific numbers. These restrictions increase the complexity of programming on multicore architecture. In addition, Message Passing/Signal Notification can also be used for inter-process communication and synchronization. Each PE can send messages (usually a few bytes) to another PE or main core.

2.2 Standard Template Library (STL)

Standard Template Library is a generic programming tool and is a component of the C++ standard library. It uses *template* and abstract data structures which improve the reusability of the code. *Template* allows user to specify data type in program and performs the binding at the compile-time. It is very useful when it is used in memory constrained system, because the compiler can optimize the identical specialization of

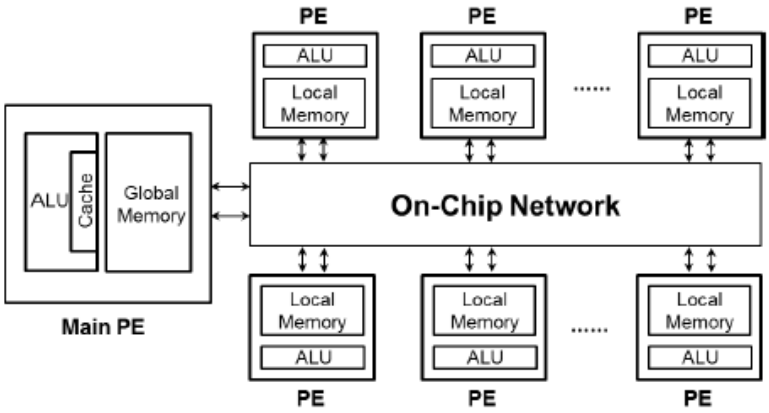


Figure 2.1: LLM is a distributed memory architecture, including a main core and execution PEs. The main core has a large global memory and OS running on it. It distributes data and tasks to the PEs. On each PE, there is a limited local memory, accessible by itself. DMA is specifically needed by it to access to global memory or other local memories. Among PEs, there is a on-chip network for data communication among different PEs.

different data types [**?**]. The *abstract data structure* in STL is called *container* which can store a collection of data objects, such as *vector*, *list*, and *stack*. *Container* automatically manages the memory for the data and users do not have to worry about the memory management. When there is an insertion or deletion, the *container* class will compare the allocated memory size and used memory size to determine if an allocation or de-allocation operation is necessary. *Container* uses another STL component *allocator* to allocate and de-allocate the memory. The STL *allocator* calls the system function *malloc* to allocate memory. The problem arises as the *allocator* assumes that there is only one memory and it is sufficiently large for the program execution. When the STL is moved to the LLM architecture, the assumption is no longer practical since the available memory size is very small.

Chapter 3

Limitations of STL on LLM multicore architecture

The current STL container classes still have limitations when it is used on LLM multicore architecture, since such architecture only has a small size of local memory and therefore the container class cannot contain large amount of data as it is for large memory. Normally, the application will conceptually be divided into 4 segments — code, global data, heap data and stack data. The size of code and global data is fixed after compilation. However, stack and heap are dynamic and they grow towards each other. The data in the container and storage data structure reside in the heap region. If the program on the PE keeps inserting data into container, there may be a hazard that heap would grow into stack segment and overwrite stack data. This may leads to a program crash in the best case, incorrect results in the worst case.

Figure 3.1 shows an example program running on LLM multicore architecture. The PE thread in Figure 3.1 (b) is initiated from thread on the main core in Figure 3.1 (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' ".

Container Class	Approx. Code Size (in Bytes)	Approx. Data Size (in Bytes)
Vector	138388	32768
Deque	139364	102908
Set	141284	11464
List	134924	21976

Table 3.1: The footprint of current container classes and the maximum sizes of data they can support. These data are collected from IBM Cell B.E. processor, SPE core, which is a typical LLM architecture. The size of the SPE local memory is 256 KB, which consists of heap region, stack region, global and static data region, and code region.

```
main(){
    pthread_create(…spe_context_run(peID)…);
}
```
(a) Main PE code

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

Figure 3.1: Outline of a threaded program on the LLM multicore architecture: (a) Main core creates threads on each PE, and STL vector class is used in the PE program. (b) In this example, the PE program works fine if N is small. But, if N is large enough, the PE program will consume up the local memory and therefore crash.

Table 3.1 shows the library memory footprint of current STL and the maximum size of data that each container can contain as used in a program which is similar to Figure 3.1. The maximum size is got by keeping adding data to the container till the program crashes. Firstly, the footprint of STL library code is large. All containers have more than 130KB code size, which means the size of the program that uses STL container will be at least larger than 130KB. Secondly, as it shows, other than the container *deque* which can contain more than 100KB of data, other containers only can have less than 32KB data. Even, the container *set* can only contain about 11KB data. The main reason they can contain so less data is that 1) The reallocation process of the vector class needs more than 3 times of memory size of actual data — one for the actual data, and reallocation process will allocate a double-size space each time. 2) The data structure of storage for *list*, and *set* is large. For *list*, if the template data type is *int*, each node needs to use three times of space of data size. And for *set*, the underlying support data structure is red-black tree, besides the data, each node needs to store the information of node color, address of parent node, and addresses of left child node and right child node. Therefore, for the template data type *int*, it needs 5 times of data size to store an integer. Finally, the current STL library do not have the functionally

that the data in the container can be transferred between the small local memory and larger global memory. In this paper, we remove this restriction by moving most of the container data into global memory and therefore enable unlimited use of container data.

Chapter 4

Challenges in Implementing STL on LLM Core

Our target is to extend the programmability of STL framework on LLM and preserve the syntax and semantics of the original STL framework. As an example of LLM programming, the Cell B.E programming model is to develop program on a distributed memory architecture. The difference in our works is that we have the restriction that each SPE only has a small local memory and we have a large global memory to use. Our solution is to use the global memory as the main storage for the container data, and use a software cache on the SPE local memory to access the container data. Our framework manages the inter-processor communication to support the extended programmability.

There are some issues that we need to solve:

- **Preserve syntax and semantics**: Our objective is to relieve the programmers' burden in programming for the architectural difference. Developing a new programming interface for programmers to use is to shift the programmers' burden from one form to another. Our extended STL preserves all the original syntax and semantics.

  For the member function of *container* class, it is easy to hide the implementation modification inside the interface. However, it is difficult to change the pointers to adapt to the change of the memory architecture. Because the container data are placed on the global memory, the pointers which point to the start and end of the data storage now have a global address instead of local address. For iterator which is a pointer, the de-referencing may cause the segmentation fault since the memory controller cannot access a global address in local memory.

  Template in the STL implementation can be specialized for any data type. The data type can be specified during the user program with STL. Our modification

also have to consider the cases in which the *template* may be a generic data type. The modified container class may not function correctly if our modification is not general for all data types. For example, if the *template* of a container is another container, we need ensure the container which is the member data of another container can work correctly. The information of the container should be preserved.

- **Cache data and utilize DMA**: As the main storage for container data is on global memory, each time the container retrieves the container data, it needs to transfer it from global memory. Each DMA transfer involves instruction overhead and high transfer latency. To maximize the performance, we can cache data in local memory to reduce the number of DMA operation, and overlap the DMA transfer with the computation to minimize the transfer latency.

  To decide which data should be cached in the local memory, we need a caching scheme for the container elements. There has been many caching schemes proposed for reducing the number of data transfer, no one can be the best in all cases. Another problem is that when caching data in the local memory, all different components of the STL will share the local memory. We should ensure the data of one container will not overwrite the buffer of another container. Moreover, the program should ensure the data consistency, different parts of the program should only see unique copy of data during the execution. A third problem is that, we may have one container as the element of another container, and how to cache the data in this scheme is still a question.

  Besides, DMA operations can be overlapped with computation to improve the performance. In this situation, how to schedule the data transfer becomes a problem, e.g. when to evict them to global memory, when to issue the DMA operation, when to synchronize the data transfer.

- **Dynamic global memory allocation**: To leverage the global memory for data storage, we need a dynamic memory allocation/de-allocation scheme for STL *allocator* on global memory. As the thread on execution PE cannot directly allocate/de-allocate on the global memory, extra implementation needs to be added.

  There is a static way to do global memory allocation, but it does not apply in extending the *allocator*. The program on main core can allocate a large chunk of memory, and then pass the first address of the allocated memory to the thread on execution PE. When the execution PE thread is initiated, it can pass the first address of the allocated memory to STL allocator, and the allocator can allocate memory from this pool to the containers. However, this method has drawbacks in terms of inflexibility. First, if the program uses too few data, then the memory space is under-utilized, if the execution PE thread uses large data set, it may simply overwrite other area which causes the execution PE thread fails. Second, it poses a big challenge to the allocator design if the STL container needs to allocate and de-allocate the memory space frequently. In such circumstances, the allocator needs to collect the fragmented area in the memory and reuse them well. One case is that the vector may keep re-allocating for larger memory space. Another case is that the map or set may insert and delete data frequently. Therefore, we need a dynamic allocation scheme for allocator.

- **Resolve hazard of the external pointers**: In LLM architecture, only part of the container data can be buffered in the local memory. If the new requested data coming in the local buffer, the old data will be evicted to the global memory to make room for new incoming data. Moreover, there are cases that pointers can point to the container data. As shown in Figure 4.1, no matter how we implement the caching logic, the container data that are pointed by an *external pointer*[1]

[1]We call the pointer which is outside the STL framework but is pointing to an container element as

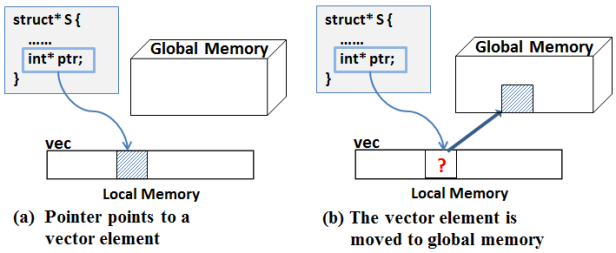


Figure 4.1: (1) No matter what scheme is implemented, older data need to be evicted to the main memory to make space for new data when the local memory is full. (2) The pointer becomes invalid when the data pointed by it have been evicted.

will be evicted to the global memory due to the constrained memory size. The pointers in the local memory will become invalid since the pointer content has been moved. This case exists since there is more than one memory space in the system. To enable the use of STL *container* in the LLM multicore architecture, we need some mechanism to ensure the validity of these pointers.

external pointer.

Chapter 5

RELATED WORKS

There are three categories of previous works related to our works. The first is the parallel generic programming library. These works propose solution to develop generic library on shared memory architecture and distributed memory architecture. The second is the software cache (SC). Software cache can manage data between local memory and global memory, which includes automatic transfer requested data into local memory. The third is the data management on scratchpad memory. These works propose techniques in managing different regions of the local memory. Their solutions also leverage the global memory for extra data storage.

5.1 Previous works on parallel generic programming library

The STL is designed for traditional Von Neumann memory architecture. All *container* elements are stored in one memory space. There should be only one thread accessing a *container* if the thread intends to modify the *container*. In parallel programming, *container* classes may be accessed by multiple threads and the *container* elements may be distributed over several memory spaces. If the STL *containers* are used for parallel programming, concurrent access to a *container* by different threads simultaneously will cause the incorrect execution.

There are previous works that ensures the concurrent access to the *container* classes will be executing correctly. The location of the container data and the memory architectural differences are all hidden inside the programming interface.

The previous work are done on two types of architectures: 1) shared memory architectures and 2) distributed memory architectures. On shared memory architecture, the previous work implemented the template library in a way that it could be concurrently accessed by multiple threads. The underlying thread synchronization has been

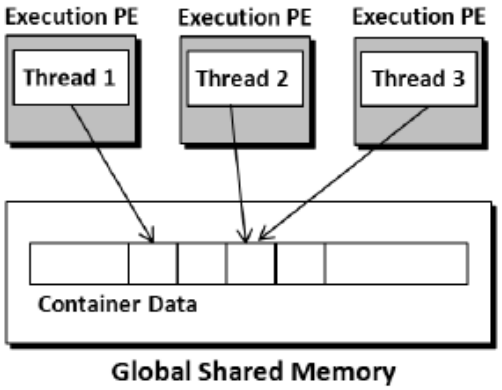


Figure 5.1: In the shared memory architecture, all threads on the execution PE can directly access the global memory. In this example, thread 1, 2 and 3 on different execution PEs can access the same container data on the global shared memory. Here is a race condition between thread 2 and thread 3, since they access the same container element. The inter-thread synchronization can be done on hardware or software.

implemented to ensure the correct execution. For example, in Figure 5.1, all threads in different execution PE access the same container in global memory, and the race condition will be resolved by the library. And once thread 2, 3 simultaneously the same container element, their exclusive accesses to the element are scheduled by the library and can be executed one after another. Intel TBB [24] developed both algorithms and containers on the shared memory architecture, with which the containers can be concurrently accessed by multiple threads. MPTL [6] and MCSTL [43] extended the STL algorithms for parallel processing.

In the distributed environment, the library needs to manage the data distribution which includes storing and searching a container element, and handling the data transfer. The implementation of data distribution and communication is transparent to the user. For example, in Figure 5.2, when the thread 2 wants to access an element on the remote memory, it will first transfer it from the local memory of execution PE 1 to its local memory, and then accesses the data. POOMA [27], AVTL[41], STAPL[47], PSTL[28], and Parallel Boost Graph Library (BGL) [22] implemented container classes which can distribute container data over different memory spaces on distributed memory architecture. The classes can be accessed by different threads concurrently.

In shared memory architecture, each execution PE can access the global memory rather than its own small local memory on LLM multicore architecture. Similarly, in distributed memory architecture, 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. If an execution PE is assigned a computation task, its local memory is sufficient for the execution of program. In LLM architecture, if the *container* class uses too much data in the local memory, it may cause the program crashes.

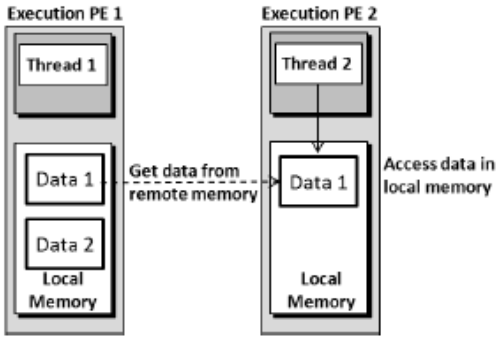


Figure 5.2: In distributed memory architecture, the container class is capable of managing data on different memory space. When a thread on a different execution PE, the library function will first transfer the data from a remote memory, and then use it from the local memory. All these steps are transparent to the user.

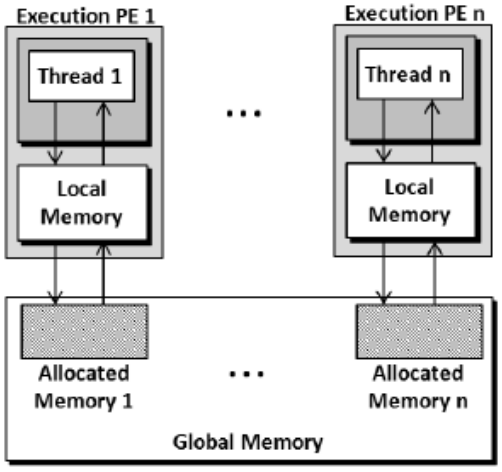


Figure 5.3: In LLM architecture, execution PE can use the global memory for data storage so that the local memory can only keep a small part of data and avoid the hazard when using large amount of data.

In LLM architecture, if a program uses STL on execution PE which requires more space than the size of available local memory, the execution will crash without data management. In order to utilize the PEs, programmers need to take care of the sizes of data in container classes and may have to repartition the program and the data for execution PEs. This severely slows down the program development progress. To solve this problem, we keep the the original program and do the dynamic data management in runtime as Figure 5.3 shows. The data distribution and communication complexities are encapsulated inside the function interface.

5.2 Previous works on Software Cache

In traditional memory architecture, the data management between cache and global memory is done through hardware support. In LLM architecture, there are cases that we use the data from global memory in the local memory. However, there is no hardware support between local memory and global memory. Operations like pointer dereferencing cannot be done as the local memory controller cannot access an address which is not local. Software cache is an easy-to-use programming interface which can map the data from the global memory address to a local memory address. It com-

prises the functionality of cache lookup and the cache miss handling which automatically use the DMA operation to transfer requested data into local memory. Figure 5.4 is an example of how to use a global pointer in the local memory with software cache. *arr* is a pointer which has a global memory address. When we use this global pointer, it is passed into the software cache interface, and the software cache interface *cache_access()* will return a local memory address where the data resides in. And the PE thread can use *temp_int* for program execution.

IIC [23], ESC [40], MDSC [3] worked on improving the performance of software cache through different caching data structures and caching logic. Works [17, 25, 11] investigated the scheme of using compiler techniques to overlap the communication and computation in order to reduce the overhead of software cache. Works [12, 21] tried to remove the unnecessary use of software cache by using a *direct buffer*. Work [42] built a software cache on the shared memory of GPU which has no hardware coherence support. Work [5] optimized the software cache implementation for computing the H.264 motion compensation. Work [10] used prefetching technique on its software cache to further improve the performance.

Software cache can automatically manage the data between local memory of execution cores (PE) and the global memory on the main core. However, the existing

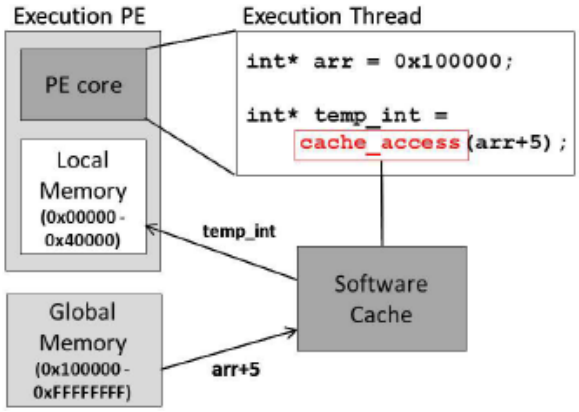


Figure 5.4: Software cache can map a global memory address *arr+5* into a local memory address *temp_int*.

```
tree_node* ptr, ptr2;
int* d2;
void F() {
    tree_node*& d1 = cache_access(ptr)->left;
    *(cache_access(d2)) = val2; // d1 evicted
    /* Some code executed */
    d1 = ptr2;
}
```

Figure 5.5: The function *cache_access()* is a software cache interface. It takes a pointer with global address as parameter, and returns a pointer to a local address with the corresponding data from global memory. In the example, reference *d1* is on the same memory address in the local buffer of software cache as *left* pointer of tree node *ptr*, so the operation performed on *d1* will be performed on the *left* of tree node *ptr*. Problem arises as the *ptr* is evicted from local buffer of software cache, *d1* is still on the same address as the *left* pointer of *ptr* was. When there is a later modification to *d1*, it will be written to a wrong place, but not *ptr* left pointer.

software cannot be directly used to transform the STL code and remove the programming restriction of STL container on execution PE. There are two reasons: first, the software cache does not manage the memory space on global memory, and is only used to map the container data from global memory to local memory. However, software cache does not prevent the growth of the heap region when the allocator keeps allocating in the local memory for container data. If the heap region keeps growing, it will still have the maximum data limit, and the hazard of overwriting the stack region still

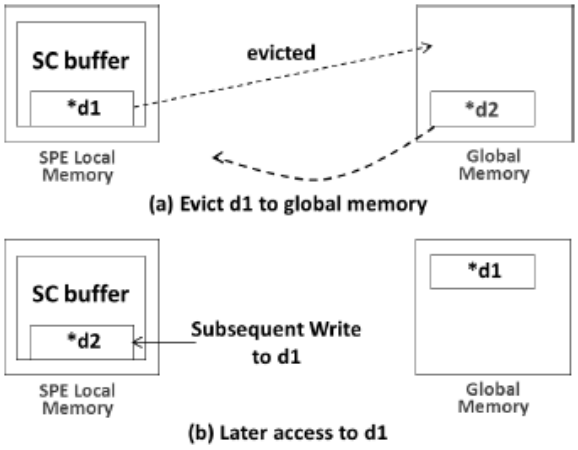


Figure 5.6: This program shows how the reference pointer *d1* will cause a problem for referencing a value in software cache local buffer. When the original content of *d1* is replaced by *d2* in the local memory, the later update to *d1* will cause incorrect results.

exists. If we have an allocator which can allocate the memory on the global memory, but not local memory, then we can move heap growing from local memory to global memory. Since the available memory size of global memory can be assumed to be infinite, we can increase the data limit of the container class significantly. Second, the proposed software cache does not deal with the error which may be introduced by C++ *reference*. This could happen when the program makes a reference to a location in the software cache buffer. We use an example code in Figure 5.5 to explain this scheme.

In this example, *d1* is a reference pointer, the original pointer is the *left* pointer of the tree node *ptr*. *d1* is initialized to the *left* pointer in the software cache buffer. In the original STL, the modification to *d1* is intended for *left*. Figure 5.6 shows what happens in the software cache buffer. A later software cache access evicts the *ptr*, and replace its memory in local buffer with the content of another pointer *d2*. Since the *d1* now accesses the same memory address *d2*, any write to *d1* is updated to *d2* which should be updated to *left* pointer of *ptr*, these writes will introduce incorrect results.

5.3 Data Management for Scratchpad Memory

The execution PE of LLM architecture leverages scratchpad memory as the small local memory. In each execution PE, the heap region, stack region, static and global data region, and code region shares the whole local memory. Since there is no hardware support or OS support for the data management between the local memory and global memory, if there is too much data in the local memory, program will crash. The overwrite to heap and stack region is the most common case of incorrect execution or program crash during the runtime. The heap region may overwrite the stack region if there is too much memory allocated to the program. The stack region may overwrite the heap region if the recursive function call is too many. The static and global region, and the code region may take too much memory space, but it can be detected during the compile-time, and can be statically analyzed for optimization. In order to enable

the program run correctly, the programmer needs to program the memory management logic for data which are in different memory space.

Much works has been done in utilizing scratchpad memory efficiently. [44, 2, 51, 35, 52, 45, 15, 16, 49, 26, 50, 36] propose the techniques that are used to manage code region. The techniques of managing static and global data region, stack region, and heap region are proposed in [44, 4, 52, 31, 30, 33, 49, 50] , [4, 35, 38, 33, 49], and [38, 34, 14] respectively.

There are also some works that work on LLM architecture. [7] manages the heap data in the local memory of execution core. It implements the memory allocation function and the address translation functions which can map from global address to local address, and vice versa. And the address translation function can also automatically transfer data between local memory and global memory. However, this work is only for C program, it cannot deal with some C++ features such as private member of class. [8] and [29] manages the stack data and code overlay for local memory respectively. However, these tools cannot handle the data management for STL containers without changing the internal program of STL.

Chapter 6

OUR APPROACH

The small size of local memory of execution PE is the main reason that restricts the programmability of STL on LLM architecture. Even if there is no overhead of code region, stack region, global and static region, and no storage data structure overhead, the STL container data can only contain a few hundreds of bytes data. Our work leverages the global memory as main storage for container data. Then the available size of memory becomes the size of the global memory which is normally a virtual memory space and can be very large.

In order to use global memory, we need to modify the container, iterator, allocator, and algorithms. Our modified allocator is capable of allocating and de-allocating memory space on global memory dynamically. Besides, we modify the container, iterator, and algorithms to utilize the allocated space on global memory. To support the extended functionality of STL components, we develop a memory allocation tool and a software cache tool for performing operation on container data on global memory. The memory allocation tool is used in the modified allocator. It can allocate memory on global memory, and return the start global address. The software cache will take a global address as input, bring the data on that global address into local memory, and finally return its address on local memory. As container, iterator, and algorithm uses pointers and reference to access the container data, the software cache is used for pointers which point to the global memory.

6.1 STL framework implementation

There are six major components in STL: *containers*, *algorithms*, *iterators*, *function objects*, *adaptors*, and *allocators* [13]. *Containers* are classes of data structures which can contain collection of any data types. There are 4 data structures for container

classes: dynamic array (vector), double-ended queue (deque), linked list, and red black tree. All the container data and the storage are dynamically created in the heap region. *Algorithms* include searching and sorting algorithms. There are also some auxiliary algorithm functions which can perform data copy, data assignment, and arithmetic operations. *Iterator* is a class which can be used like a smart pointer. It is the connection between *Algorithm* and *Container* in the ways that *Algorithm* manipulates the data in the *Container* through *Iterator*. *Allocators* are classes which take care of memory management functionality for *Container* classes. *Function Objects* are functions which are normally used as arguments to be passed into *Algorithms* for performing arithmetic operations and *Containers* as comparison functions for ordering the elements. *Adaptor* is a component which can take another components, and transforms their interfaces into new interfaces. There are three kinds of *adaptor*: container adaptor, function adaptor, and iterator adaptor [13]. For example, stack and queue both are container adaptor for deque (Double-Ended Queue). *Queue* transforms the deque to a FIFO sequence container, with the contrary that stack transforms it to a LIFO sequence container.

Our solution is to put most of the data in STL container class to the global memory and use local memory as a buffer to the managed data (like a software cache). To extends the programmability of the STL, we need to modify *containers*, *algorithms*, *iterators*, and *allocators*. We will discuss the details in Section 6.

We modify the underlying functionality support for STL framework and preserve the original interface and semantics. We modify the allocator with the heap allocation tool, so that the container can have member data placed on global memory. Also, we use software tool to support container, iterator, and algorithm to access the container data on global memory. The purpose of using software cache is to translate the global address into a local address for pointer, and also transfer the data into local memory. As STL is written in an object-oriented programming model, we can encapsulate all the code transformation inside the original interfaces.

Allocator

Allocator manages the memory allocation and deallocation for container class. The original allocator of STL on LLM is using the new allocator. Because the new allocator allocates the local memory, it will cause the heap region keep growing in the local memory, and the container class will have a max data limit to use in the local memory. Therefore, we need to transfer the increase of heap region to the global memory from local memory. In order to do so, we use the memory allocation tool to allocate memory space on global memory. Different container will specialize the allocator for allocating different types of data structures. For example, *vector* will specialize the allocator allocating for template data type, and *red-black tree* will specialize the allocator for allocating the tree node with the template data type. Each time the allocator is called, it returns a global address, and the container will use the global address for memorizing the location of its member data.

The allocator is used for allocating and de-allocating container instances and container data on heap region. Normally, container instances is on stack region, and its data are on global memory. Accessing the container data can be done by simply passing the data address in global memory to the software cache and uses the returned local address. However, there are cases that container instances are the member data of another container.[1] In such cases, the memory space of these container instances will be allocated on heap region which is in the global memory. For container instances on heap region, we need to apply software cache to the *this* pointer of the container class, and we can use this global address of *this* pointer for accessing the container instances.

[1] In rest parts of the paper, we call this case as "container-to-container" cases.

Container

Different from the STL container, our new container classes store the container data on global memory and access the data through software cache. The container data uses the modified allocator as the same way as the original allocator. Since the modified allocator allocates memory space on global memory, the container data is on the global memory. The extended container uses the global addresses instead of the local address to keep track of the container data. Accordingly, functions and operators of container classes which access and perform operations on container data will be changed to use the software cache interface in order to access the container data on global memory.

For container *vector*, the start and end address of the allocated memory space for data are maintained by pointers *_M_start* and *_M_end_of_storage*. *_M_finish* specifies the position after the last element. During allocation and re-allocation, the allocator will return an allocated global address for *_M_start* and *_M_end_of_storage*. Since it does not retrieve the data, there is no need to use the software cache. For the operations that the *vector* needs to access the data through *_M_start* or *_M_finish*, the software cache needs to be applied for fetching the vector elements. The global address of the element G_{re} can be calculated easily by:

$$G_{re} = \_M\_start + Index_{re} \times (element\_size - 1)$$

where G_{re} is the global address of the requested element and $Index_{re}$ is the offset of the requested element relative to the first element. For example, assume we have an vector of integers, if the *_M_start* is 0x10000 and we want to fetch the 133th element, then the global address of 133th element is 0x10210. The software cache can then retrieve the element by using the global address 0x10210.

Container *deque* is similar to vector. It uses a two dimensional pointer which is called *map pointer* to store the elements. The first level pointer stores the location

of the blocks, and the second level of pointers point to the actual elements. The data structure of *deque* is shown in Figure 6.1. The *deque* class uses two iterators *first* and *last* to remember the first element, and the position after the last element. The *map pointer* maintains the global address of the first-level table which is also stored in global memory. In the member function of *deque*, it uses *map pointer* for accessing the second-level blocks for memory allocation and de-allocation.

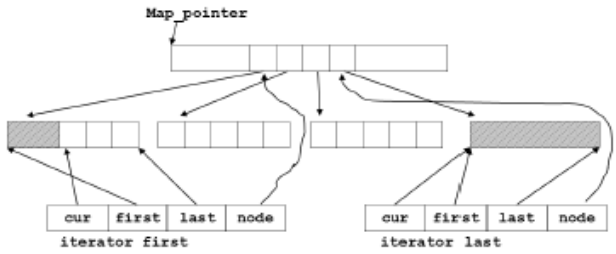


Figure 6.1: The map pointer contains the pointers point to the elements. The iterator of *deque* has four fields, *cur* which points to the current accessed element, *first* and *last* pointers point to the first element and last element inside the element array, *node* points to the corresponding pointer in the first level.

We apply the software cache on the *map pointer*, we can do the following transform:

```
_Tp** cur;
*((_Tp**)cache_access(cur)) = this->_M_allocate_node();
```

where *cur* is a *map pointer* and *cache_access()* is the software cache interface. The allocated global address for a block is stored in the first-level table on the global memory. *deque* uses its iterator class to access the container elements, and the implementation modification is encapsulated in the iterator operators.

For container class *list* (linked list) and *red black tree*, the modification are more complicated than vector. It is because *list* and *red black tree* (RB tree) are node-based data structure. The container class *list* contains a node as the class member which is the end of the linked list. Similar to list, class *red black tree* also contains a tree node as class member which is the end node for *red black tree iterator*. Normally, if the *list*, *RB tree* class is used as a variable in program to contain basic data types, they can work well. Because the instances of *list* and *RB tree* is placed either on global and static variable region, or on the stack region, in both ways, the addresses of the instances are not changed. However, if they were used in a ”container-to-container” situation, the instances of list and red black tree are likely to be initiated on heap region. As we

mentioned before, the container data are placed on the heap region on global memory, and are brought into local memory through software cache interface. A global address may be mapped to different local addresses after its content is swapped out from, and then brought into the software cache. The address change of the list node in class *list* and tree node in *RB tree* class may introduce incorrect results. In many cases, the program simply would not stop since the algorithm cannot find these end nodes. To resolve this problem, we change the list node in class list and the tree node in RB tree class, so that all the nodes are accessed by using global addresses.

Iterator and Algorithm

Iterator is a class which can be used as a pointer. For each type of container data structure, it will implement the pointer operation based on the data structure. For container *vector*, the functionality of the iterator is the same as a pointer, only the iterator contains the global address to the container element. Our modification to the *vector* iterator is to overload the * operator with software cache tool, so that it can fetch the container elements. Since the address of vector elements can be easily calculated, there is no need to change other iterator operators.

The iterator of *deque* maintains a *map pointer* which points to the current accessed second-level node, and three pointers *_M_cur*, *_M_first* and *_M_last* which are used in the accessing block. The structure of iterator is shown in Figure 6.1. *_M_cur* is used to point to the current accessed element, and *_M_first* and *_M_last* are pointing to the first and last element in the current accessed node. We need to change the * operator for fetching deque element, and the function which switch the *map pointer* to a second-level block.

The iterator of *red black tree* and *list* keeps the information of the current accessed node. The operators of iterator use the functions for *tree* traversal and *list* traversal. The software cache is applied for not only the element retrieving, but also the ele-

ments traversal. The complexity of modification is from the pointer which points to the next node. There may be several *left* and *right* pointer used in *red black tree*, and multiple *next* and *previous* pointer in *list*. In these cases, we need to separate the original statement into multiple statements. For an example code

```
tree_node* parent;
tree_node* tmp_node = parent->right->left;
```

we will need to transform it to

```
tree_node* tmp_ptr = ((tree_node*)cache_access(parent))->right;
tree_node* tmp_node = ((tree_node*)cache_access(tmp_ptr))->left;
```

Here, the software cache are applied to a single pointer access in a statement.

For algorithms, we needs to apply the software cache interface to the iterators for which are pointers. The problem is that we cannot simply apply the software cache tool to the original code. Because if the iterator is not pointer, then the compiler will emit error and stop compiling. These happens when the iterator is a pointer. In Figure 6.2, we show what we need to change in the base algorithm fill(). We use the *__is_pointer*() function separate the implementation for pointer type iterators, then we can apply software cache to the iterators. For the transformed code, there will not have compiler error.

6.2 Software Cache

The software cache (SC) tool is used to perform data management on execution PE local memory. Software cache can retrieve the data on global memory which is pointed to by a pointer which has a global address. The interface of our software cache are shown in Table 6.1. The *init_cache*() is used to initialize the caching data structure, and allocates memory for cache blocks. The *cache_access*() is used to access the data on

```
template<bool>
struct __fill {
        template<typename _ForwardIterator, typename _Tp>
        static void fill(_ForwardIterator __first, _ForwardIterator __last,
                         const _Tp& __value) {
                for (; __first != __last; ++__first)
                        *__first = __value;
} };
```

(a) Original STL code

```
template<bool>
struct __fill {
        template<typename _ForwardIterator, typename _Tp>
        static void fill(_ForwardIterator __first, _ForwardIterator __last,
                         const _Tp& __value) {
                typedef typename std::__is_pointer<_ForwardIterator>::__type
                        _Integral;
                fill_aux_1(__first, __last, __value, _Integral());
} };
template<typename _ForwardIterator, typename _Tp>
void fill_aux_1 (_ForwardIterator __first, _ForwardIterator __last,
                        _Tp& __value, __true_type){
        for (; __first != __last; ++__first)
                *((_ForwardIterator)cache_access((uint32_t)__first) = __value;
} };
```

(b) Transformed code for applying software cache interface

Figure 6.2: To enable algorithm code to handle the pointer type iterator, we need to separate the implementation with pointer type iterator from the original implementation. In this case, the software cache is used in the fill_aux_1() function.

global memory. It will automatically transfer the data on the requesting global address into local memory, and return the pointer to access the data.

The software cache is a direct-map cache. The data structure of the local memory storage is a hash table with an FIFO linked list. The FIFO linked list serves as a victim cache. The hash table and FIFO linked list use the same cache block data structure. For each cache block, it contains the global address of the first byte in the cache block, a valid flag, and a pointer to the actual cache block data.

For the cache lookup, SC will first check the hash table to see if the requested data is in the local memory. If there is a miss, then it will go to the linked list to see if there is cache hit. If there is a cache miss, it will take this block to replace a block in the hash table, and put the replaced block to the head of the linked list. If there is still a miss, we need to transfer a block of data which contains the requested data into SC buffer. For a cache miss, we first inserted the new requested block into the hash table. Then, the original corresponding entry in the hash table will be inserted into the head of the linked list. Finally, the last block in the FIFO linked list will be transferred to global memory by using a non-blocking DMA.

Our software cache views the data in global memory in blocks. The addresses within the range of a block can be all hashed to the address of the first byte of the block. To check if the requested address is in a cache block, the SC will compare the hashed value of the requested address with the address of the first byte of the block. If two addresses are the same, then it is a hit, otherwise, it is a miss. As discussed in [40], choosing the block size in the power of 2 can simplify the calculation of hashing an

Function Interface	Functionality
void init_cache()	Initialize the table for direct-map cache and the FIFO linked list
void* cache_access(uint32_t ref_ppu_addr)	This interface will accept a global address ref_ppu_addr as the input, it will bring in the data at the requested global address, and return the pointer to its corresponding local address in cache.
void* ppu_addr(void* ptr)	If the address of *ptr* is in software cache buffer, this functino will convert this address into a corresponding global address. This function is used in resolve the external pointer problem.

Table 6.1: Software cache interfaces

address. We use a block size that is in the power of 2. The following equation is used to to calculate the first address of a block

$$block\_address = input\_address\&(\sim (block\_size - 1))$$

The *block_address* is the address of the first byte of the cache block, *input_address* is the requested data address, and *block_size* is the size of a data block. After the *AND* operation, we can get an address which is the power of 2. For example, if the cache block size is 16, *input address* is 0x01008, then the *block_address* is 0x01000.

6.3 Dynamic Global Memory Allocation

Since the containers store container data on global memory, our allocator allocates the memory space on global memory instead of allocating the memory space on local memory. In order to minimize the change to the original code, the allocation interface is implemented separately. The allocation interface set includes three functions $pmalloc()$, $pfree()$, and $prealloc()$, which is used for allocating, de-allocating, and reallocating memory on global memory. The new allocator uses $pmalloc()$ to substitute for operator *new*, and uses $pfree()$ to substitute for operator $delete$. Since the STL containers use $construct()$ function from standard library to initialize the objects on the allocated memory, this change will not introduce incompatibility problem.

The heap allocation tool allocates the memory space on global memory for the STL container. Since PE program cannot allocate memory on global memory, we use a thread on the main PE core and allocate the memory for execution PE program. We used both the DMA and message passing for communication. The major functionality of message passing is to ensure the execution order of execution PE thread and main core thread during the allocation process. DMA is used to transfer additional parameters when necessary. The main core thread starts before the execution PE thread is initiated. After it starts, it initiates a data structure $msgStrut$ for transferring infor-

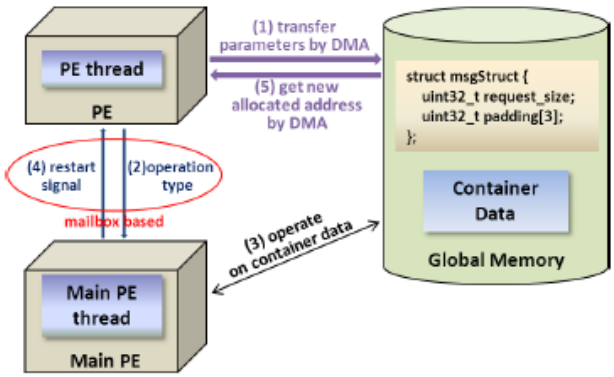


Figure 6.3: (1) The whole process for memory reallocation is shown and the number means the order of steps. (2) Important information for reallocation is contained in a data structure named *msgStruct*, which is located in the global memory. It is 16 bytes large, but elements can be different depending on the type of operation.

mation with execution PE thread through DMA channel, and then waiting on a read operation for the new request message.

The inter-processor communication process includes five steps:

1. Execution PE thread informs the Main PE thread of requested task by mailbox, starts waiting for the response mailbox message
2. Main core thread receives message from the mailbox
3. Main core thread does the allocation/de-allocation/re-allocation
4. Main core thread sends back task completed message through mailbox and start waiting for the next request message on mailbox
5. Execution PE thread receives the message from mailbox and continues executing

Figure 6.3 shows an example of allocation. The *msgStruct* is a data structure instance located in global memory. Its global address is the same during the execution PE executes. As the execution PE program needs to allocate a new piece of memory on global memory. It transfers the request memory size by DMA to the *msgStruct* on global memory. Then, it sends an integer as a message to main core thread which indicates the type of operation it is requesting. After the main core thread is waked up from waiting for request message, it reads the message from the channel and understand it as an allocation operation. The main core thread reads the allocation size from *msgStruct* and perform the allocation process. If the allocation is successful, the start address is placed in the *msgStruct*. After that, the main core thread sends back an integer which indicates the allocation complete to PE thread. Finally, the PE thread reads the transfer the start address into execution PE and use it.

For most DMA engine, there is a typical requirement for the start and end transferring address to be aligned. The allocation program handles the alignment problem both in execution PE side and main core side. In execution PE thread, we ensure that the allocation size is aligned to an transferable address for DMA engine. To alleviate

this problem, we maintain a memory pool which is an allocated memory space on main core thread. For memory pieces which have an equal or smaller size than software cache block size, the main core thread allocates the memory from the memory pool and aligns the address. Since all the allocated memory pieces from memory pool are aligned, we do not have fragmentation problem. After one memory pool is exhausted, the allocation program will allocate a new memory pool. All the allocated memory pool are stored in a linked list, so there is no memory leak problem. For larger allocation size, the main core allocation program still uses $malloc()$ to allocate larger memory space, and aligns the start address to an aligned address.

6.4 External pointers

In the traditional memory architecture, since pointers and the container data are all in a same memory location, there is no hazard for pointers. However, in LLM architecture, as the container element needs to place in the global memory, the pointer may become invalid as the data which it is pointing to no longer exist. Our idea to ensure the validity of the external pointer is to let the pointer maintain the global address of the container element. We propose a technique which requires the code transformation to be performed on the original code. When the pointer content is needed, the de-referencing is done by the software cache. There are two steps in the solution for the external pointers. First, we need to identify all the *potential* external pointers which may point to the container data. Then, the code transformation is performed in order to apply the software cache to the external pointers.

To identify the *potential* external pointers, we first need to identify the pointers which are getting the address of a container element. Then we will do some simple pointer analysis to track the possible pointers which may also point to a container element. The member function or operator of a container class does not return a pointer to the container elements. They may either return an iterator which can finally return a ref-

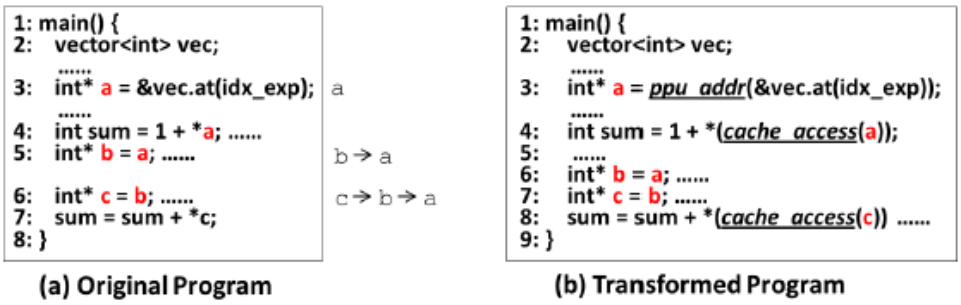


Figure 6.4: (a) The *potential* pointers will first be identified. (b) The program will be transformed to use the software cache interface and *ppu_addr()* which is used to extract the global address of a container element.

erence to the container element, or they may directly return the reference of a container element. Therefore an external pointer may point to a container element by getting the address of a returned element reference. For example, in the Figure 6.4 (a) line 3, the pointer *a* can point to a vector element at offset *idx_exp*. In Table 6.2, the member functions and operators of *vector* and *map* which return a reference to the container element are listed. When the source code is analyzed, each use of these functions and operators are tracked. If the address of their reference is assigned to a pointer, then it is a *potential* external pointer. Furthermore, for all pointers which are assigned by the value of *potential* pointers are also *potential* pointers. As shown in Figure 6.4 (a), *a* becomes a *potential pointer* is because it gets the address from a container element reference. *b* becomes a *potential pointer* because it is assigned by *a*'s address. The same for *b* which is assigned by *b*'s address. After the analysis, we get a set of *potential pointers*.

Vector & Deque	List & Queue	Stack	Map
operator[]	front()	top()	operator[]
at()	back()		
front()			
back()			

Table 6.2: Member functions and operators that return a reference to elements of containers

The code transformation to the source code consists of two steps: first, the initial reference to a container object needs to be converted into global address. Second, software cache needs to be applied for the de-referencing of *potential pointers*. We illustrate the code transformation in Figure 6.4 (b). The *ppu_addr()* function is used in the line 3, because the local address of the element is first assigned to the *potential* external pointer *a*. Function *ppu_addr()* extracts and returns the global address for the vector element at offset *idx_exp*. Then, the *cache_access()* function is applied to line 4 and 8 to perform the pointer de-referencing. The line 7, 8 can be executed safely since they do not use the data on global memory.

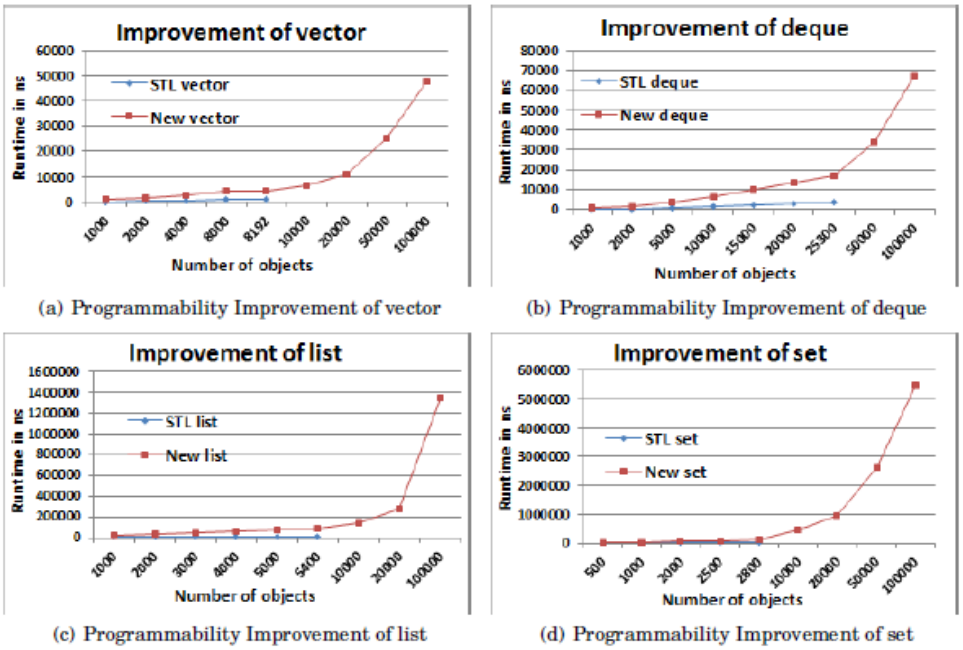
Chapter 7

Experimental Evaluation

Our experiments are done on PS3 which uses IBM Cell B.E. Processor. IBM Cell is a LLM multicore architecture in which each core has a local memory of 256KB. We installed Fedora 9 and IBM Cell B.E. SDK 3.1 on PS3. The benchmark that we used is shown in Table 7.1. We use $\_mftb()$ and $time()$ to measure the run time of the SPE program.

7.1 Enabling use of STLs

We have proposed techniques to manage the STL data on the limited memory of the cores of an LLM multicore architecture. Note that without our technique, STL can be used, however, they can only support certain data size, and after that, it crashes. Our modifications to the STL library, perform DMAs to the global memory and enable STLs to manage any amount of data. Figure 7.1 shows the experimental proof that



(a) Programmability Improvement of vector

(b) Programmability Improvement of deque

(c) Programmability Improvement of list

(d) Programmability Improvement of set

Figure 7.1: The effectiveness of our solution are shown in the above four figures.

Benchmark	Description	Containers	Data Size (in Bytes)
Heapsort	The heapsort algorithm	vector	4,000,000
Dijkstra	The Dijkstra shortest path algorithm	vector, queue	4,008,000
Edmonds-Karp	The Edmonds-Karp max flow algorithm	vector, list, queue	8,909,584
Kruskal	The Kruskal's minimum spanning tree algorithm	list, vector	8,023,984
anagram	A program which finds out all the anagram in the dictionary	map, list, string	833,684
MMints Compress	An image compression scheme by estimating the current cell based on the neighbors values	vector	4,000,000
MMints Wavelet	Debaucles 4-Coefficient Wavelet filter for image compression	vector	6,000,000
Basicmath	Perform simple mathematical calculation	vector	24
Olden power	Olden power pricing benchmark	vector	512
List merge	Merge two list containers	list	8000
CRC32	Compute the 32-bit CRC checksum	vector	100,000

Table 7.1: Benchmarks for Experimental Evaluation

our technique can enable any amount of STL data management in the vector, deque, list and set containers. In each sub-graph, the Y-axis is the runtime when the number of elements (on X-axis) are inserted. What we can see is that when using the original STL, there is a limit on how many elements can be inserted, but our implementation supports unlimited number of elements. In specific, Figure 7.1 (a) shows that original vector class could only handle 8192 elements, while our implementation does not have such a limitation. Therefore, after using our implementation of STL, programmers do

Container	Original Code Size	Approx. Code Size (in Bytes)	Perc of Increase
Vector	138388	155036	12%
Deque	139364	156132	12%
Set	141284	166228	17.7%
List	134924	151228	12%

Table 7.2: Extra Static Code Overhead

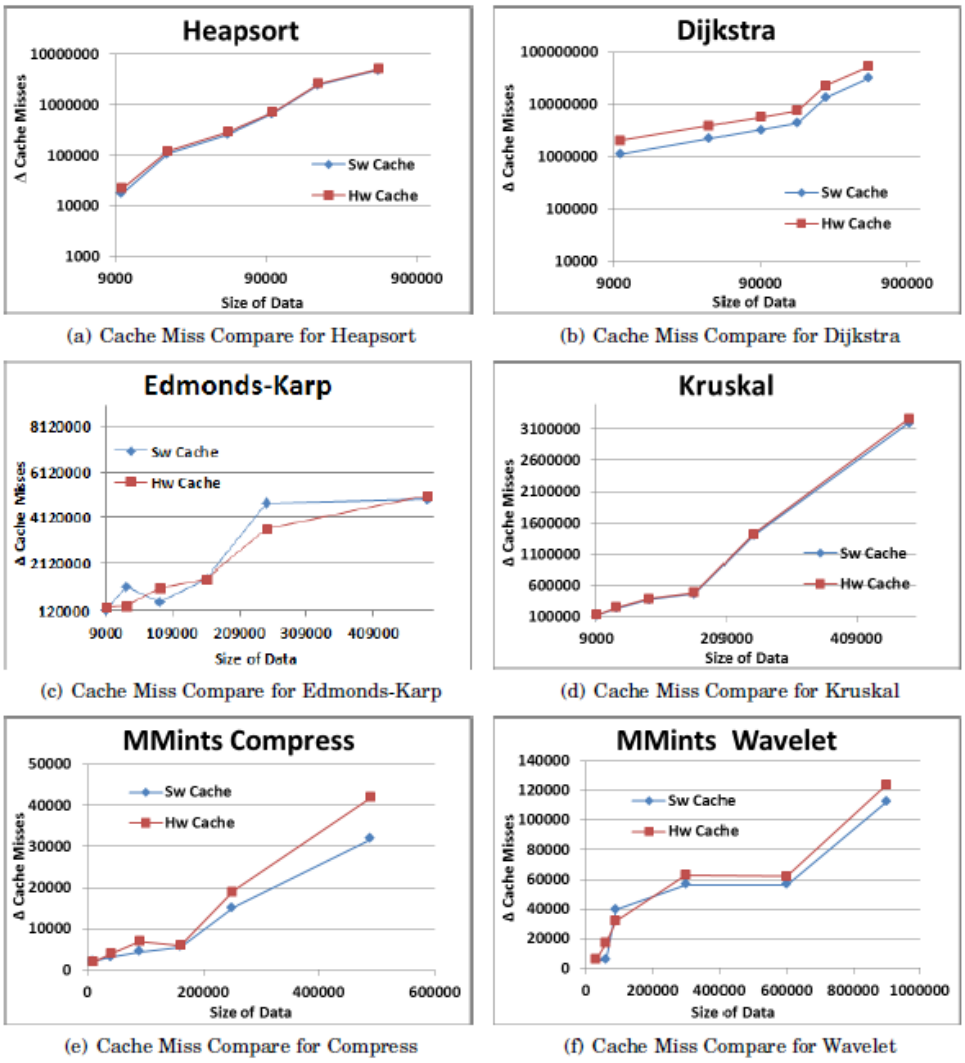
not have to worry about the number of elements in their dynamic data structures – they can program without that worry, and our implementation will manage any amount of data. We clearly improve the programmability of LLM multicores, but this comes at some codesize and performance overhead. The rest of this section characterizes and breaks down these overheads.

7.2 Static Code Size Overhead

We evaluated the code size overhead for our new STL framework. The evaluated code is the same as the code used for evaluating the code size in Table 3.1. Table 7.2 shows that the code size of our new STL container generally increases by 12%, except that of the container list increases by 17.7%. The increase of code size is mainly determined by how much the container class uses the pointer to access data. The storage data structure of container class set is red black tree, whose original implementation contains a large number of pointer access. That's the reason why the increase of code size of set is larger than other containers.

7.3 Data Management Overhead

There are two parts of overhead in the software cache: (i) **data transfer overhead:** overhead due to data transfers between the local memory and the global memory, and (ii) **dynamic instructions overhead:** overhead due to extra instructions that must be executed as a part of software cache implementation to find out if the data is present in the software cache or not, and if not, getting it from the global memory. Note how-



(a) Cache Miss Compare for Heapsort

(b) Cache Miss Compare for Dijkstra

(c) Cache Miss Compare for Edmonds-Karp

(d) Cache Miss Compare for Kruskal

(e) Cache Miss Compare for Compress

(f) Cache Miss Compare for Wavelet

Figure 7.2: Comparison of cache misses between our software cache for STL and hardware direct-map cache.

ever, that the overhead of actual data transfer is not calculated in this, as it is already computed in (i).

To find the data transfer overhead, and compare it to what we would have faced in a cache based system, we compare the number of DMAs required by our technique to the number of cache misses on a cache based architecture. To estimate the later, we used the *Cachegrind* tool in the *Valgrind* to measure the hardware cache miss. For both, our software cache, and the hardware cache in Cachegrind, we use a direct-map cache of 32KB and line size of 128B. We run the executions on the Cell processor, and

simulations on Cachegrind for several data sizes of the applications. Since these are two different systems (different libraries, and ABI), the actual number of cache misses are not comparable. As a result, in figure 7.2 we plot the increase in the number of cache miss on the current data size over the number of cache misses for the next data size. For example, we run Heapsort with 10000 elements, and then run it for 20,000 elements. We plot *cache-misses-for-20000-elements - cache-misses-for-10000-elements* for both Cell SPU and cache based system. We see from the graphs that the number of DMAs required is quite similar to that of the number of cache misses.

The dynamic instruction overhead comes from the extra instructions that software cache needs to perform the cache lookup and cache miss handling. For this experiment, we runs the same copies of SPE code which uses the STL container and the new containers separately, and compare the instruction counts for different copies. The input data size is reduced so that we can use STL library. The experiments are conducted on the PS3 full system simulator. Figure 7.3 shows that the instruction count overhead for different benchmarks. In this experiment, we have two sets of benchmarks. In Figure 7.3(a), the STL container are intensively used for computation. Under the extensively use, the additional overhead from the new STL can be up to 15 times of the original instruction count. In Figure 7.3(b), the container class are used normally in the application, and the additional instruction overhead can be only 25% of the total original execution instruction. Also, the dynamic instruction overhead depends on the complexity of the container implementation and the amount of usage of container member functions. For benchmarks *heapsort*, *wavelet* that use vector have a large increase in instruction count. This is because vector has simple implementation in retrieving elements, so most of the overhead is in software cache operations. On the other hand, for benchmarks $edmonds-karp$, *kruskal*, and $list-merge$, the extra overhead appears to be smaller in percentage. This is because the container member functions implementation are more complex. For benchmark *basicmath*, $olden\_power$, and $list\_merge$,

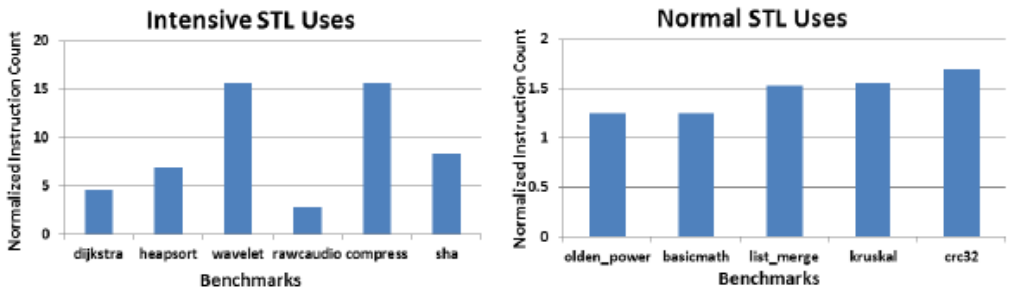


Figure 7.3: The instruction overhead of the new STL library. We separate the benchmarks into two sets: intensive use benchmarks and normal uses benchmarks.

the use of STL member function is not much, and therefore, the additional instruction overhead is small.

7.4 Scalability on Multicore

In this experiment, we run the benchmarks on different number of cores to see how our approach scales to multiple cores. For this experiment, we use input data sizes of 360,000 Bytes, 1,276,800 Bytes, 1,280,000 Bytes, and 75,500 Bytes for Dijkstra, Krusal, Edmonds-Karp, and Anagram respectively. As shown in Figure 7.4, three benchmarks which only use the vector can scale well from 1 core to 6 cores. For benchmarks Anagram, Dijkstra, Edmonds-Karp and Kruskal, the runtime increases

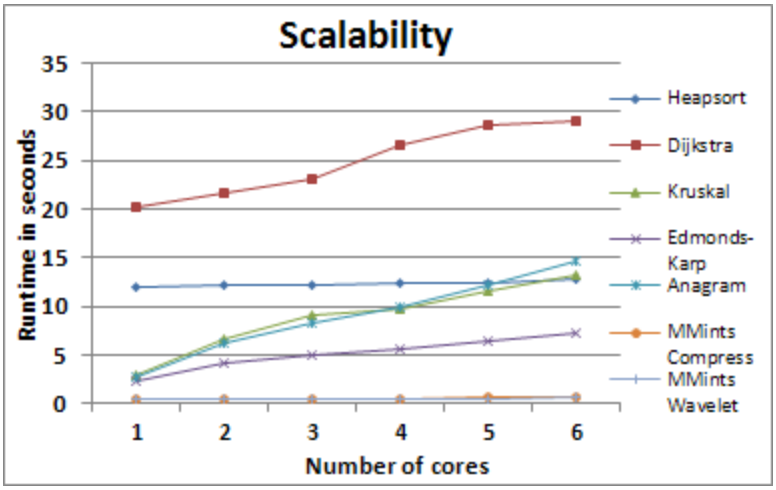


Figure 7.4: The scalability of the benchmarks is mainly depends on the types of container used.

proportionally to the number of cores execution. This is because these two benchmarks heavily use the pointer operation. The pointer operation usually access the memory address inconsecutively, and introduces many DMA operations.

Chapter 8

Conclusions

In this article, we enable the usage of C++ STL library on the Limited Local Memory (LLM) multicore architectures. LLM multicore architectures are power-efficient, and feature scalable memory design. However, the programmability of STL library on LLM architecture is limited as the local memory of each core is small. We improve the programmability of the STL library by placing the container data on the global memory instead of using only the local memory. Our experiment shows that our techniques allow using STL for any unlimited data size. The static code size of our changes to the STL logic increases the size of STL library by 12-17%. The number of DMAs required are comparable to the number of cache misses that would have occurred, with a equivalent cache size. Finally, although in the worst case, the extra instruction overhead can be as high as 15 times that of the original, on typical applications, the overhead is much more moderate and tolerable.

REFERENCES

[1] Analog Devices, Inc, One Technology Way, Norwood, Mass 02062. *ADSP-BF561 Blackfin Processor Hardware Reference*, revision 1.1 ed. edition.

[2] F. Angiolini, F. Menichelli, A. Ferrero, L. Benini, and M. Olivieri. A post-compiler approach to scratchpad mapping of code. In *Compilers, architecture, and synthesis for embedded systems*, pages 259–267, New York, NY, USA, 2004. ACM Press.

[3] Ben Juurlink Arnaldo Azevedo. A multidimensional software cache for scratchpad-based systems. *International Journal of Embedded and Real-Time Communication Systems (IJERTCS)*, 1:1–20 pp, 2010.

[4] Oren Avissar, Rajeev Barua, and Dave Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Trans. on Embedded Computing Sys.*, 1(1):6–26, 2002.

[5] Arnaldo Azevedo and Ben Juurlink. An efficient software cache for h.264 motion compensation. In *Proceedings of the 11th international conference on System-on-chip*, SOC'09, pages 147–150, Piscataway, NJ, USA, 2009. IEEE Press.

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

[7] Ke Bai and Aviral 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.

[8] Ke Bai, Aviral Shrivastava, and Saleel Kudchadker. Stack data management for limited local memory (llm) multi-core processors. In *Proceedings of the Interna-*

tional Conference on Application Specific Systems, Architectures and Processors (ASAP), 2011.

[9] R. Banakar, S. Steinke, Bo-Sik 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.

[10] Ping Chao and Youn-Long Lin. An elastic software cache with fast prefetching for motion compensation in video decoding. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES/ISSS '10, pages 23–32, New York, NY, USA, 2010. ACM.

[11] Tong Chen, Haibo Lin, and Tao Zhang. Orchestrating data transfer for the cell/b.e. processor. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 289–298, New York, NY, USA, 2008. ACM.

[12] Tong Chen, Tao Zhang, Zehra Sura, and Mar Gonzales Tallada. Prefetching irregular references for software cache on cell. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 155–164, New York, NY, USA, 2008. ACM.

[13] G. Derge D. Musser and A. Saini. *STL Tutorial and Reference Guide, 2nd Edition*. Addison-Wesley Professional, 2001.

[14] A. Dominguez, S. Udayakumaran, and R. Barua. Heap data allocation to scratchpad memory in embedded systems. *Embedded Computing*, 1(4):521–540, 2005.

[15] Bernhard Egger, Chihun Kim, Choonki Jang, Yoonsung Nam, Jaejin Lee, and Sang Lyul Min. A dynamic code placement technique for scratchpad memory using postpass optimization. In *CASES '06: Proceedings of the 2006 international*

conference on Compilers, architecture and synthesis for embedded systems, pages 223–233, New York, NY, USA, 2006. ACM.

[16] Bernhard Egger, Jaejin Lee, and Heonshik Shin. Scratchpad memory management for portable systems with a memory management unit. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 321–330, New York, NY, USA, 2006. ACM.

[17] A. E. Eichenberger, J. K. O'Brien, K. M. 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, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo. Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture, 2006.

[18] A.E. Eichenberger et al. Using advanced compiler technology to exploit the performance of the cell broadband enginetm architecture. *IBM Syst. J.*, 45(1):59–84, 2006.

[19] David Geer. Industry trends: Chip makers turn to multicore processors. *Computer*, 38:11–13, May 2005.

[20] W. Gibbs. A split at the core. *Scientific American*, Nov 2004.

[21] Marc Gonzàlez, Nikola Vujic, Xavier Martorell, Eduard Ayguadé, Alexandre E. Eichenberger, Tong Chen, Zehra Sura, Tao Zhang, Kevin O'Brien, and Kathryn O'Brien. Hybrid access-specific software cache techniques for the cell be architecture. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 292–302, New York, NY, USA, 2008. ACM.

[22] Douglas Gregor and Andrew Lumsdaine. The parallel bgl: A generic library for distributed graph computations. In *In Parallel Object-Oriented Scientific Computing (POOSC*, 2005.

[23] Erik G. Hallnor and Steven K. Reinhardt. A fully associative software-managed cache design. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 107–116, New York, NY, USA, 2000. ACM.

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

[25] Andhi Janapsatya, Aleksandar Ignjatović, and Sri Parameswaran. A novel instruction scratchpad memory optimization method based on concomitance metric. In *ASP-DAC '06: Proceedings of the 2006 conference on Asia South Pacific design automation*, pages 612–617, Piscataway, NJ, USA, 2006. IEEE Press.

[26] Andhi Janapsatya, Aleksandar Ignjatović, and Sri Parameswaran. A novel instruction scratchpad memory optimization method based on concomitance metric. In *ASP-DAC '06: Proceedings of the 2006 conference on Asia South Pacific design automation*, pages 612–617, Piscataway, NJ, USA, 2006. IEEE Press.

[27] et al John Reynders. Pooma: A framework for scientific simulations on parallel architectures. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 553–594. MIT Press, 1996.

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

[29] Seungchul Jung, Aviral Shrivastava, and Ke Bai. Dynamic code mapping for limited local memory systems. In *Proceedings of the International Conference on Application-specific Systems Architectures and Processors (ASAP)*, pages 13–20, July 2010. ISSN 1063-6268.

[30] Mahmut T. Kandemir, J. Ramanujam, and Alok N. Choudhary. Exploiting shared scratch pad memory space in embedded multiprocessor systems. In *DAC*, pages 219–224, 2002.

[31] Mahmut T. Kandemir, J. Ramanujam, Mary Jane Irwin, Narayanan Vijaykrishnan, Ismail Kadayif, and Amisha Parikh. Dynamic management of scratch-pad memory space. In *DAC*, pages 690–695, 2001.

[32] Yongjoo Kim, Jongeun Lee, Aviral Shrivastava, and Yunheung Paek. Operation and data mapping for cgras with multi-bank memory. In *Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems*, LCTES '10, pages 17–26, New York, NY, USA, 2010. ACM.

[33] Lian Li, Lin Gao, and Jingling Xue. Memory coloring: a compiler approach for scratchpad memory management. In *PACT*, pages 329–338, Sept. 2005.

[34] R. Mcllroy, P. Dickman, and J. Sventek. Efficient dynamic heap allocation of scratch-pad memory. In *ISMM'08:The 7th international symposium on Memory management*, pages 31–40, New York, NY, USA, 2008. ACM Press.

[35] Nghi Nguyen, Angel Dominguez, and Rajeev Barua. Memory allocation for embedded systems with a compile-time-unknown scratch-pad size. In *CASES*, pages 115–125, 2005.

[36] Amit Pabalkar, Aviral Shrivastava, Arun Kannan, and Jongeun Lee. SDRM: Simultaneous determination of regions and function-to-region mapping for scratch-pad memories. In *Int'l Conference on High Performance Computing (HiPC)*, December 2008.

[37] D. Pham, S. Asano, M. Bolliger, M.N. Day, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and

K. Yazawa. The design and implementation of a first-generation cell processor. In *ISSCC '05: IEEE Solid-state circuits*, pages 184–592, 2005.

[38] Francesco Poletti, Paul Marchal, David Atienza, Luca Benini, Francky Catthoor, and Jose Manuel Mendias. An integrated hardware/software approach for run-time scratchpad management. In *DAC*, pages 238–243, 2004.

[39] Barbara G. Ryder, Mary Lou Soffa, and Margaret Burnett. The impact of software engineering research on modern progamming languages. *ACM Trans. Softw. Eng. Methodol.*, 14:431–477, October 2005.

[40] Sura Z. Sangmin Seo, Jaejin Lee. Design and implementation of software-managed caches for multicores with local memory. In *IEEE 15th International Symposium on High Performance Computer Architecture (HPCA'09)*, 2009.

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

[42] Mark Silberstein, Assaf Schuster, Dan Geiger, Anjul Patney, and John D. Owens. Efficient computation of sum-products on gpus through software-managed cache. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 309–318, New York, NY, USA, 2008. ACM.

[43] Johannes Singler, Peter Sanders, and Felix Putze. Mcstl: The multi-core standard template library. In Anne-Marie Kermarrec, Luc Boug, and Thierry Priol, editors, *Euro-Par 2007 Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, pages 682–694. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-74466-5-72.

[44] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Design, automation and test*, page 409, Washington, DC, USA, 2002. IEEE Computer Society.

[45] Stefan Steinke, Nils Grunwald, Lars Wehmeyer, Rajeshwari Banakar, M. Balakrishnan, and Peter Marwedel. Reducing energy consumption by dynamic copying of instructions onto onchip memory. In *ISSS '02: Proceedings of the 15th international symposium on System Synthesis*, pages 213–218, New York, NY, USA, 2002. ACM.

[46] Bjarne Stroustrup. Evolving a language in and for the real world: C++ 1991-2006. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 4–1–4–59, New York, NY, USA, 2007. ACM.

[47] Gabriel Tanase, Antal Buss, Adam Fidel, Harshvardhan Harshvardhan, Ioannis Papadopoulos, Olga Pearce, Timmie Smith, Nathan Thomas, Xiabing Xu, Nedal Mourad, Jeremy Vu, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger. The stapl parallel container framework. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 235–246, New York, NY, USA, 2011. ACM.

[48] Texas Instruments Incorporated, Texas Instruments, Post Office Box 655303, Dallas, Texas 75265. *TMS320C6472 Fixed-Point Digital Signal Processor Technical Brief (Rev. B)*, July.

[49] Sumesh Udayakumaran, Angel Dominguez, and Rajeev Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *Trans. on Embedded Computing Sys.*, 5(2):472–511, 2006.

[50] M. Verma and P. Marwedel. Overlay techniques for scratchpad memories in low power embedded processors. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 14(8):802–815, Aug. 2006.

[51] M. Verma, L. Wehmeyer, and P. Marwedel. Cache-aware scratchpad allocation algorithm. In *Design, automation and test*, page 21264, 2004.

[52] Manish Verma, Klaus Petzold, Lars Wehmeyer, Heiko Falk, and Peter Marwedel. Scratchpad sharing strategies for multiprocess embedded systems: A first approach. In *ESTImedia*, pages 115–120, 2005.