OPELL and PM: A Case Study on Porting Shared Memory Programming Models to Accelerators Architectures

Joseph B. Manzano, Ge Gan, Juergen Ributzka, Sunil Shrestha, and Guang R. Gao

Department of Electrical and Computer Engineering
University of Delaware
[jmanzano,gan,ggao]@capsl.udel.edu
[ributzka,sunilaachaju]@gmail.com

No Institute Given

Abstract. Limits on applications and hardware technologies have put a stop to the frequency race during the 2000s. Designs now can be divided into homogeneous and heterogeneous ones. Homogeneous types are the easiest to use since most toolchains and system software do not need too much of a rewrite. On the other end of the spectrum, there are the type two heterogeneous designs. These designs offer tremendous computational raw power, but at the cost of hardware features that might be necessary or even essential for certain types of system software and programming languages. An example of this architectural design is the Cell processor which exhibits both a heavy core and a group of simple cores designed as a computational engine. Even though the Cell processor is very well known for its accomplishments, it is also well known for its low programmability. Among many efforts to increase its programmability, there is the Open OPELL project. This framework tries to port the OpenMP programming model to the Cell architecture. The OPELL framework is composed of four components: a single source toolchain, a very light SPU kernel, a software cache and a partition / code overlay manager. To reduce the overhead, each of these components can be further optimized. This paper concentrates on optimizing the partition manager by reducing the number of long latency transactions. The contributions of this work are as follows.

1. The development of a dynamic framework that loads and manages partitions across function calls to bypass the problem with restrictive memory spaces.
2. The implementation of replacement policies that are useful to reduce the number of DMA calls across partitions.
3. A quantification of such replacement policies given a selected set of applications
4. An API which can be easily ported and extended to several types of architectures.

1 Introduction

During this decade, the multi / many core architectures have seen a renaissance, due to the insatiable hunger for performance. Limits on applications and hardware technologies have put a stop to the frequency race around 2006. Designs now can be divided into homogeneous and heterogeneous ones. Homogeneous designs are the easiest to use since most toolchain and system software do not need too much of a rewrite. On the other end of the spectrum, there are heterogeneous designs. These designs offer tremendous computational raw power, but at the cost of hardware features that might be necessary or even essential for certain types of system software and programming languages. An example of this architectural design is the Cell processor which will be explained in the next section.

1.1 The Cell Broadband Engine

The Cell B.E. has been placed in the public eye thanks to being a central component in one of the fastest super computer, being the main processing unit of the Sony's Playstation 3 videogame console, and being the bane of programmers everywhere. This architecture is a project in which three of the big computer / entertainment companies, IBM, Sony and Toshiba, worked together to create a new chip for the seventh generation of home video game consoles[1]. The chip possesses a heavy core, called the PowerPC Processing Element (or PPE for short), which acts as the system's brain. The workers for the chip are called the Synergistic Processing

Elements (or SPE for short) which are modified vector architectures which huge computational power. The SPE possesses 256 KiB of local memory and a Memory Flow Controller which takes care of external Input / Output operations.

Both processing elements coexist on the die with a ratio of 1 to 8 (one PPE to eight SPEs), but more configurations are possible. Finally, all the components are interconnected by a four-ring bus called the Element Interconnect Bus (or EIB for short). Figure 1 shows a high level overview of the Cell B.E. This chip is capable of around 200 Giga Floating Point Operations Per Seconds (FLOPS) for single precision and around 102.4 Giga FLOPS for double precision[1].

Although the heavy core possesses all the "standard" hardware components, the computational engine lacks many of these features. The SPEs exhibit limited local memory, lack caches of any type, and it has no virtual memory support. Communication between the host (PPE) and the computational engine (the SPEs) is achieved through explicit Direct Memory Access (DMA) operations between the main memory and the local memory of the computational engine. This puts more responsibilities on the system software, programmers and users to take advantage of the system raw computational power by orchestrating all components using the features of the computational engine.

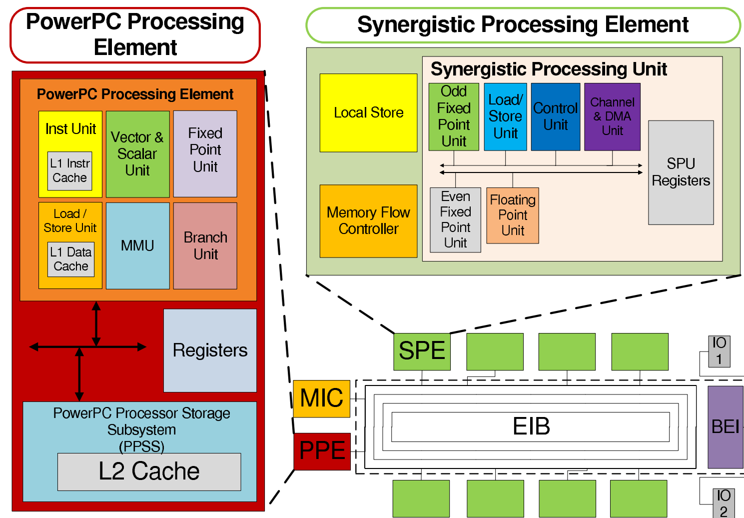


Fig. 1. Block Diagram of the Cell Broadband engine

1.2 Problem Formulation

The lack of programmability in the heterogeneous designs, especially in the Cell B.E., can be attributed to the loss of many hardware features, such as caches, reorder buffers, etc and a lack of runtime systems to take advantage of the architecture's performance. The need for new software stacks is evident. Due to this need, the OPELL framework was introduced. This framework tries to bring the OpenMP parallel programming model (De facto shared memory parallel programming paradigm) to the Cell architecture. The OPELL framework is composed of four components: a single source toolchain, a very light SPU kernel, a software cache and a partition / code overlay manager. This extra layer greatly increases the system's programmability, but it comes at the cost of additional overhead from the framework. To reduce the overhead, each of the components can be further optimized. This paper concentrates on optimizing the partition manager components by reducing the number of long latency transactions (DMA operations) that it produces. The contribution of this paper can be summarized as follows:

[1] These numbers come from the revised PowerXCell 8i Boards

1. The development of a dynamic framework that loads and manages partitions across function calls. In this manner, the restrictive memory problem can be alleviated and the range of applications that can be run on the co-processing unit is expanded.
2. The implementation of replacement policies that are useful to reduce the number of DMA calls across partitions. Such replacement policies aim to optimize the most costly operations in the proposed framework. Such replacements can be of the form of buffer divisions, rules about eviction and loading, etc.
3. A quantification of such replacement policies given a selected set of applications and a report of the overhead of such policies. Several policies can be given but a quantitative study is necessary to analyze which policy is best in which application since the code can have different behaviors.
4. An API which can be easily ported and extended to several types of architectures. The problem of restricted space is not going away. The new trend seems to favor an increasing number of cores (with local memories) instead of more hardware features and heavy system software. This means that frameworks like the one proposed in this paper will become more and more important as the wave of multi / many core continues its ascent. Moreover, the same concepts presented here can be extended to run on other heterogeneous accelerator type architecture like GPGPUs and FPGAs.

This paper is divided as follows. Section 2 introduces relevant related work. Section 3 introduces the OPELL framework and each of its components. Section 4 shows the partition manager framework and its features. Section 5 presents the results for the partition manager different features. Finally, Section 7 shows the conclusions and future work.

2 Related Work

There have been many attempts to increase the programmability in the Cell B.E. The most famous ones are the ALF and DaCS[3] frameworks and the CellSS project[2]. The ALF and DaCS frameworks are designed to facilitate the creation of tasks and data communication respectively for the Cell B.E. The Accelerator Library Framework (ALF) is designed to provide a user-level programming framework for people developing for the Cell Architecture. It takes care of many low level approaches (like data transfers, task management, data layout communication, etc). The DaCS framework provides support for process management, accelerator topology services and several data movement schemas. It is designed to provide a higher abstraction to the DMA engine communication. Both frameworks can work together and they are definitely a step forward from the original Cell B.E. primitives. They are not targeted to Cell B.E. application programmers, but to library creators. Thus, the frameworks are designed to be lower level than expected for an OpenMP programmer.

The Cell SuperScalar project (the CellSS) [2] is designed to automatically exploit the function parallelism of a sequential program and distribute them across the Cell B.E. architecture. It accomplishes this with a set of pragma based directives. It has a locality aware scheduler to better utilize the memory spaces. It uses a very similar approach as OpenMP. However, it is restricted to task level parallelism in comparison to OpenMP that can handle data level parallelism. Under our framework, the parallel functions are analogous to CellSS tasks and the partition manager is their scheduler. Many of the required attributes of the tasks under CellSS are hidden by the OpenMP directives and pragmas which make them more programmable.

Finally, there have been efforts to port OpenMP to the Cell B.E.. The most successful one is the implementation in IBM's XL compiler[7]. The implementation under the XL compiler is analogous to the OPELL implementation with very important differences. The software cache under the XL compiler is not configurable with respect to the number of dirty bytes that can be monitored in the line. This allows the implementation of novel memory models and frameworks as shown in [4]. The other difference is that the partition manager under the XL uses static GCC like overlays. Under OPELL, the partitions can be dynamically loaded anywhere in the memory which is not possible under the XL compiler.

3 Opell Framework

The Open Source OpenMP on CELL (or Open OPELL for short) developed at the University of Delaware [6] is a porting of a very popular high performance parallel language to a heterogeneous accelerator type architecture. Its main objective is to provide an open source OpenMP framework for the Cell B.E. architecture. It is composed of an OpenMP toolchain, which produces Cell B.E. code from a single OpenMP source

tree; and a runtime that hides the heterogeneity of the architecture from the user. The framework provides the following features: a single source compiler, a simple micro kernel, software cache, and the partition / overlay manager.

3.1 Single Source Compilation.

The Cell B.E uses two distinct toolchains to compile its code for the architecture. This adds more complications to an already complex programming environment. In OPELL, the OpenMP source code is read by the driver program. The driver clones the OpenMP source for both toolchains and calls the respective compiler to do the work. The PPU compiler continues as expected, even creating a copy of the parallel function (which is the body of the parallel region in OpenMP) and inserting the appropriate OpenMP runtime function calls when needed. The SPU compiler has a different set of jobs. First, it keeps the parallel functions and discards the serial part of the source code. Second, it inserts calls to the SPU execution handler and its framework to handle the parallel calls and OpenMP runtime calls. Third, it inserts any extra function calls necessary to keep the semantics of the program. Finally, it creates any structures needed for the other components of the runtime system, links the correct libraries and generates the binary. After this step is completed, the control returns to the driver which merges both executables into a single one. Figure 2 shows a high level graphical overview of the whole single source process.

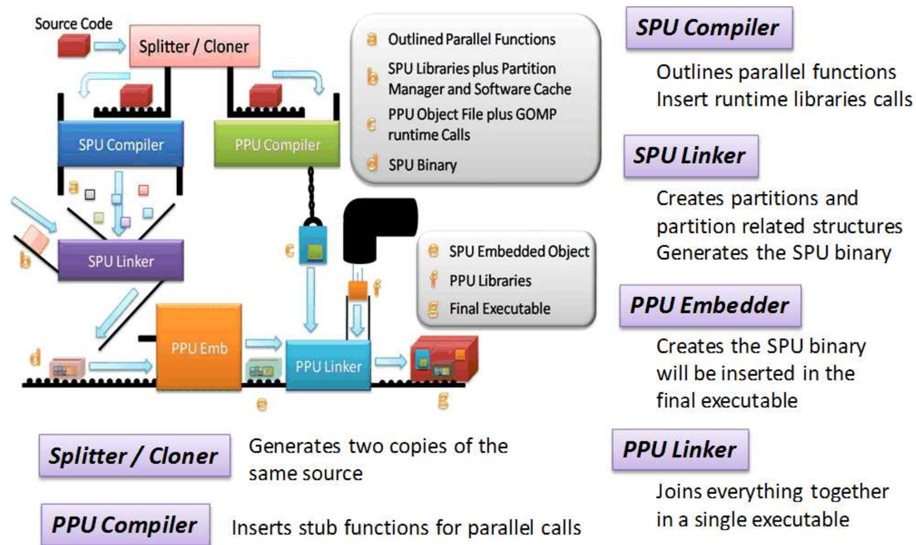


Fig. 2. A high level overview of the single source toolchain. Under this framework the SPU Embedder will "generate" a new SPU binary (i.e it wraps it with a special API) so it can communicate with the host

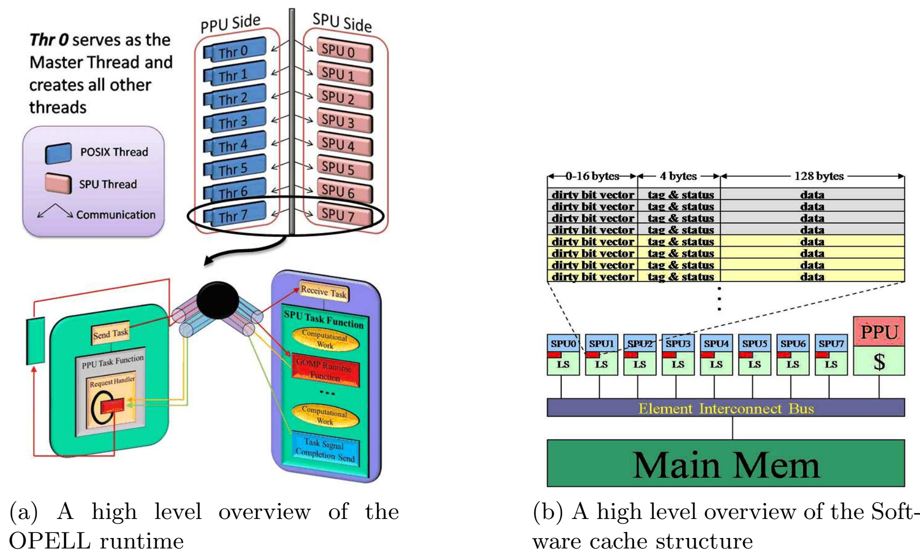
3.2 Simple Execution Handler

This small piece of code[2] deals with the communication between the PPU and SPU during runtime and how runtime and parallel function calls are handled. Since each of the SPUs have very limited memory, it is in everybody best interest to keep the SPU threads very light. To achieve this, the SPU thread will be loaded only with a minimal set of the code (the simple execution handler and a set of libraries). This SPU resident code does not include the parallel regions of the OpenMP code nor the OpenMP runtime libraries. Since both are needed during runtime, they are both loaded or executed on demand, but by different mechanisms. The parallel regions are loaded and executed by another component, i.e. the partition manager, which loads and

[2] In this paper, the terms simple execution handler and SPU micro kernel will be used interchangeably

overlays code transparently. The OpenMP runtime libraries require another framework to execute. Under this framework, there exists an extra command buffer per thread that is used to communicate between the SPE and PPE frameworks. Moreover, there exists a complementary PPE thread for each SPE thread which is called the mirror or shadow threads which services all the requests from its SPE.

When a SPE thread is created[3], the simple execution handler starts and goes immediately to polling. When a parallel region is found by the master thread (which runs on the PPE), a message is sent to the simple execution handler with the identifier's ID and its arguments' address. When it is received, the SPU calls the code in the parallel region (through the partition manager). The SPU continues executing the code, until an OpenMP runtime call is found. In the SPU, this call creates a PPU request to the command buffer. This request is composed of the operation type (e.g. limit calculations for iteration space) and its arguments. While the SPU waits for the results, the PPU calls the runtime function and calculates the results. The PPU saves the results back to the Command buffer and sends a signal to the SPE to continue. Finally the SPU receives the signal and reads the results. The SPU thread ends polling when the PPU shadow thread sends a self terminate signal, effectively ending the thread's life. Figure 3a shows a graphical representation of the SPE micro kernel and communication framework.



(a) A high level overview of the OPELL runtime

(b) A high level overview of the Software cache structure

Fig. 3. Components of the Simple Execution handler and the Software cache

3.3 Software Cache

As stated before, the SPU component of the Cell B.E. does not have caches (at least not across the SPU local storages) or any other way to maintain coherence. This presents a peculiar problem for the pseudo shared memory which Open OPELL presents[4]. This heterogeneity hindrance is resolved by the software cache. This framework component is designed to work like a normal hardware cache with the following characteristics. It has 64 sets with 4-way associativity and a cache line of 128 bytes (most efficient size for DMA transfers). Its total size is 32 KiB and it has a write back and write allocate update policy. As a normal cache, each line possesses a dirty-bit vector which keeps track of the modified bytes of the line. When the effective (global) address is found in the cache, a hit is produced and the operation is performed, i.e. read or write.

In case that the effective address is not in the cache, a miss is produced. A read miss or write miss causes an atomic DMA operation to be issued to load the desired value from memory and may produce a write back operation if any of the bits in the dirty bit vector are set. The write process only touches the dirty

[3] which happens before the application code is run

[4] Open OPELL is designed to support OpenMP which is a shared memory programming model

bytes and leaves the clean ones untouched. A graphical overview of the software cache is presented by figure 3b.

This component has been used in the testing and creation of weak memory models presented in [4].

3.4 Overlay / Partition Manager.

As the software cache is used for data, the partition manager is used for code. This small component is designed to load code on demand and manage the code overlay space when needed. When compiling the source code, certain functions are selected to be partitioned (not loaded with the original source code in the SPU memory). The criteria to select these functions are based on the Function Call Graph, their size and their runtime purpose, e.g. like parallel regions in OpenMP. Finally, the partitions are created, descriptive structures are formed and special calling code is inserted when appropriate. During runtime, the function call proceeds as usual (i.e. save registers, load parameters, etc), up to the point of the actual call. Instead of jumping to the function, the control is given to the partition manager runtime and several decoding steps are done. With information extracted from the actual symbol address, a loading decision is made and the code is loaded into memory or not (if the code already resides in the overlay). Then, the partition manager runtime passes control to the function. When the function finishes, the control returns to the partition manager so any cleaning task can be performed, like loading the caller partition if it was previously evicted. Finally, the partition manager returns to its caller without leaving any trace of its activities.

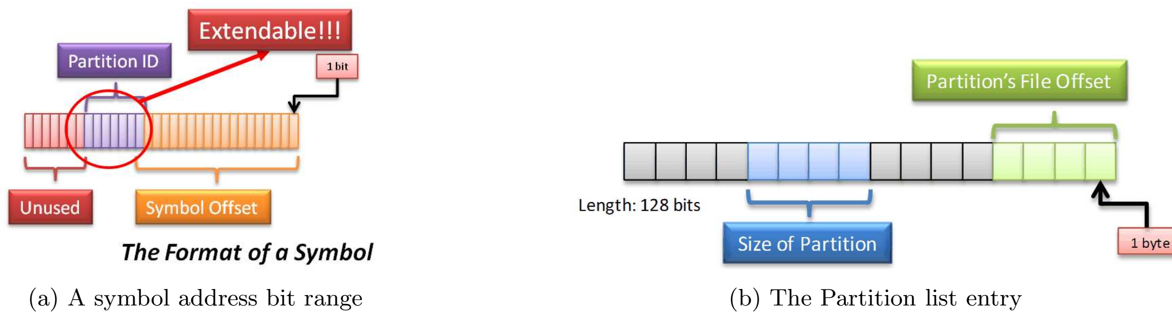
A more detailed description of this component is given in the next section.

4 The Partition Manager

The Partition Manager framework depends on four structures and some binary image changes. Some of them are created by the compiler, while others are created and maintained during runtime. The partition manager major components are described next.

4.1 Major Toolchain Changes

Under the Partition Manager framework, all partitionable code's symbols will be modified. These symbols represents the offset in bytes of the given symbol in its partition. The symbol's partition id is saved in the upper 14 bits. If the symbol is not in a partition, the upper 14 bits are zero and the lower bits represents the absolute address of the function. The format of symbol is described in figure 4a



(a) A symbol address bit range

(b) The Partition list entry

Fig. 4. The symbol address bit range and the Partition list entry format

4.2 The Partition List

This structure is created by the toolchain. It consists of two parts which defines the partition offset on the file and the partition size. Moreover, the partition list resides in the computational element local memory;

just after the program's data section. Under this framework, a partition is defined as a set of functions for which their code has been created to be position independent (PIC); thus they can be moved around the memory as the framework sees fit. The actual partition code is not loaded with the program, but left in the global memory of the machine. The partition offset part of a list element shows the offset (in bytes) from the binary entry point. Finally, the size section of the entry contains the size in bytes of the partition on the memory image. Under this model, each of the partitions is identified by a unique number that index them into this list. When a partition is required, the element's partition is loaded using the partition list information and the correct buffer state is set before calling the function. The format and the bit range of the partition list entries are described in figure 4b

4.3 The Partition Stack

The Partition Stack is a meta-structure which records the calling activity between partitions. It was designed to solve a very simple problem: how to return from a function call which was called from another partition? By keeping the partition stack, the framework can know who the caller of the current function call is, load the partition back if it is required and save function state, i.e. registers which must be saved across partition manager calls. Although the partition code is defined as PIC, when returning from a function call, the framework must load the partition back to its original position. If this is not the case then a large amount of binary rewriting and register manipulation is needed to ensure the correct execution of the function. This is true even for PIC code since registers might have stale addresses to the original sub-buffer location.

4.4 The Partition Buffer

The Partition Buffer is a special region of the local memory, in which the partition code is swap in and out. It is designed to have a fixed value per application, but it can be divided into sub-buffers if required. Moreover, it contains certain state, like the current Partition index and the lifetime of the code in this partition; which is used for book-keeping and replacement policies. This buffer is managed by the partition manager kernel.

The partition buffer and the partition list modifies the SPE binary image a bit. It adds the list to the end of the data segment and the buffer after the interrupt table. The modified image compared with a normal SPE image is given in figure 5

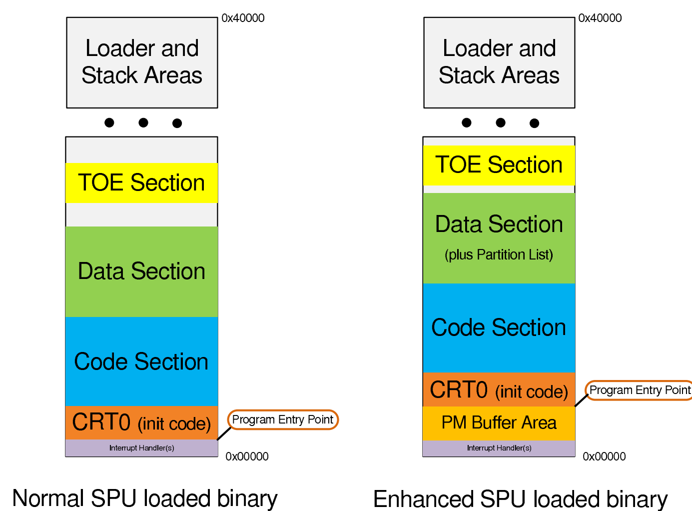


Fig. 5. A comparison between the modified SPE binary image and a normal one

4.5 The Partition Manager Kernel

At the center of all these structures lies the Partition Manager. This small function takes care of the loading and management of the partitions in the system. During initialization, the partition manager may statically divide the partition buffer so that several partitions can co-exist with each other. It also applies the replacement policy to the buffers if required. The sequence of operations involve in a simple partition manager call is presented in Figure 6

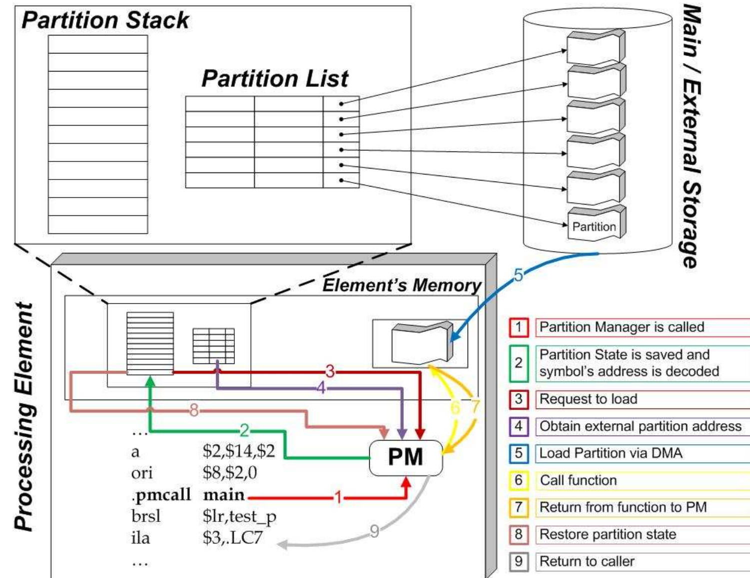


Fig. 6. A typical partition manager call

The next section explains a replacement policy and an enhancement which is applied to the partition manager framework and its effect on the number of operations.

5 The N Buffer: The Lazy Reuse Approaches

Since the partition buffer might be mostly empty most of the time, it can be broken down into sub-buffers to further utilize the hardware resources. This opens many interesting possibilities on how to manage the sub-buffers to increase performance. Even though this area is not new, these techniques are usually applied in hardware. The techniques applied for replacement in this buffer are cache like in which that they try to take advantage of partition locality. The first technique is when the buffer subdivisions are treated as FIFO (first in first out) structures. In this context, this technique is called *Modulus* due to the operation used to select the next replacement. The second one is based on one of the most famous (and successful) cache replacement policies: Least Recently Used (LRU). First, we need to introduce the challenges of dividing the buffer under our framework and how it affects each component.

The partition buffer is enhanced by adding extra state. Each sub-buffer must contain the partition index residing inside of it and an extra integer value to help achieve advanced replacement features (i.e. the integer can represent lifetime for LRU or the next partition index on a pre-fetching mechanism). Moreover, the partition that resides in local memory becomes stateful under this model. A partition now can be *active*, *in-active*, *evicted* or *evicted with the opportunity of reuse*. For a description of the new states and their meanings, please refer to table 1.

Every partition begins in the *evicted* state in main memory. When a partition is used, the partition is loaded and becomes *active*. From this state the partition can become *in-active*, if a new partition is needed and this one resides into a sub-buffer which is not replaced; back to *evicted*, if it replaced and it doesn't

State	Location	Description
Evicted	Main Memory	Partition was not loaded into local memory or it was loaded, evicted and it will not be popped out from the partition stack.
Active	Local Memory	Partition is loaded and it is currently in use
In-active	Local Memory	Partition is not being used, but still resides in local memory
EWOR	Main Memory	Evicted With the Opportunity of Reuse. This partition was evicted from local memory but one of the element of the partition stack will pop its partition id in the near future.

Table 1. The Four States of a Partition

belong to the return path of a chain of partitioned function calls; or *Evicted with an Opportunity to Reuse*, in the case that a partition is kicked out but it lies on the return path of a chain of partitioned function calls. An *in-active* partition may transition to evicted and $EWOR$ under the same conditions as an active one. An $EWOR$ partition can only transition to an *active* partition.

These states can be used to implement several levels of partitioning. One of them is described in Section 5.3.

When returning from a chain the partition function calls, the partition must be loaded into the same sub-buffers that they were called from. To achieve this, the partition stack node must know where the partition originally resided. Thus, this structure must save the sub-buffer id.

5.1 Replacement Policies: The Modulus Approach

Under this approach, sub-buffers form a type of First-In First-Out (FIFO) structure in which the oldest partition is always replaced. It follows the normal formula in which the next sub-buffer to be replaced is selected by the formula $next = (next + 1) mod NSB$ where the $next$ is the sub-buffer in which the new partition is loaded and NSB represents the total number of sub-buffers.

5.2 Replacement Policies: The LRU Approach

Under this approach, each of the sub-buffers has a lifetime counter which decrements every time that a function is called on another partition. The formula to select the next buffer to be replaced becomes $next = MIN(LTA)$ where $next$ is the sub-buffer where the next partition is put and LTA is the Lifetime Array of values. In case that the minimum of the array is a set, this group of elements is managed as if it was a FIFO buffer across different calls of the replacement policy functions. It is important to note that by having multiple sub-buffers, duplication might be possible, the partition framework disallows this. In this way, the framework would not get “confused” when figuring out which sub-buffer to jump in. In the case that a partition is duplicated (for example when returning from a function call into a different sub-buffer), the framework moves the partition to the correct sub-buffer and nullify its old locations. This move saves a load to main memory or prevents the need to adjust all the address in the partition to match the new sub-buffer.

5.3 The Victim Cache for the Partition Framework

Under this framework, the victim cache is a dynamically allocated piece of memory that is created when EWOR partition are called. The EWOR partition is recognized by setting a bit in a partition mask (which has support for 128 partition indexes) every time that a partition stack frame is pushed. When the partition stack frame is popped, the bit on the mask is unset[5]. When a new partition is being loaded into the main memory, the evicted partition index is checked against the partition mask. If they match, the partition code which resides on the sub-buffer is copied to a newly allocated memory block. When an EWOR partition is needed back, the victim cache is checked and the partition is copied back to the sub-buffer if found. Under the current implementation, there is only a single entry on the victim cache. This means that it can only

[5] This might create false positives in long chain of functions, but it is acceptable in practice

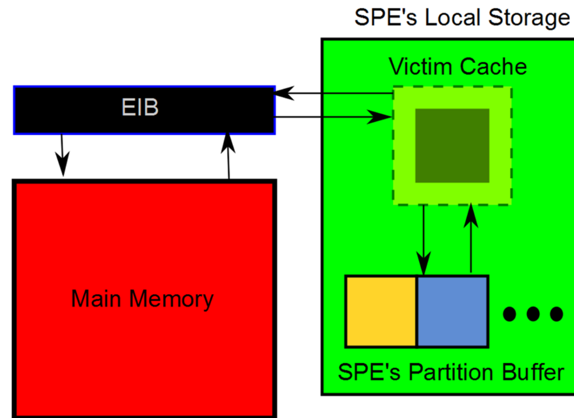


Fig. 7. The victim cache scheme

provide support for the most recent EWOR partition on the function chain. A high level overview of the victim cache is given in figure 7.

Since the victim cache can be created dynamically, it can also be brought down in the same way. The framework offers two wrappers for the memory allocators (i.e. malloc and free) which can check the memory pool for availability. If the pool is empty or near it, the victim cache can be brought down to free up memory for the application.

6 Experimental Testbed and Results

The partition manager framework uses a small suite of test programs dedicated to test its functionality and correctness. The testbed framework is called Harahel and it is composed of several Perl scripts and test applications. The next subsections will explain the hardware and software testbeds and presents results for each of the test programs.

6.1 Hardware Testbed

For these experiments, we use the Playstation 3's CBE configuration. This means a Cell processor with 6 functional SPE, 256 MiB of main memory, and 80 GiB of hard drive space. The two disabled SPEs are used for redundancy and to support the hypervisor functionality. Besides these changes, the CBE processor has the same facilities as high end first generation CBE processors. We take advantage of the timing capabilities of the CBE engine. The CBE engine has hardware time counters which ticks at a slower rate than the main processor (in our case, they click at 79.8 MHz). Since they are hardware based, the counters provided minimal interference with the main program. Each of the SPEs contains a single counter register which can be accessed through our own timing facilities.

6.2 Software Testbed

For our experiments, we use a version of Linux running on the CBE, i.e. Yellow Dog with a 2.6.16 kernel. Furthermore, we use the CBE toolchain version 1.1 but with an upgraded GCC compiler, 4.2.0, which was ported to the CBE architecture for OpenOPELL purposes.

The applications being tested include kernels used in many famous benchmarks. This testbed includes the GZIP compression and decompression application which is our main testing program. Besides these applications, there is also a set of micro-benchmarks designed to test certain functionality for the partition manager. For a complete list, please refer to 2.

In the next section, we will present the overhead of the framework using a very small example.

Name	Description
DSP	A set of DSP kernels (a simple MAC, Codebook encoding, and JPEG compression) used at the heart of several signal processing applications.
GZIP	The SPEC benchmark compression utility.
Jacobi	A benchmark which attempts to solve a system of equations using the Jacobi method.
Laplace	A program which approximate the result of an integral using the Laplace method.
MD	A toy benchmark which simulates a molecular dynamic simulation.
MGRID	A simplified program used to calculate Multi grid solver for computing a 3-D potential field.
Micro-Benchmark 1	Simple test of one level partitioned calls.
Micro-Benchmark 2	Simple chain of functions across multiple files.
Micro-Benchmark 3	Complete argument register set test.
Micro-Benchmark 5	Long function chain example 2.
Micro-Benchmark 6	Long function chain example 3: Longer function chain and reuse.
Micro-Benchmark 7	Long function chain example 4: Return values and reuse.
Micro-Benchmark 8	Long function chain example 5: Victim cache example.

Table 2. Applications used in the Harahel testbed

6.3 Partition Manager Overhead

Since this framework represents an initial implementation, the main metric on the studies presented will be the number of DMA transfer produced by an specific replacement policy or/and partition feature. However, we are going to present the overhead for each feature and policy.

The first version represents the original design of the partition manager in which every register is saved and the sub-buffer is not subdivided. The improved version is with the reduction of saved registers but without any subdivision. The final sections represent the policy methods with and without victim cache.

On this model, the overhead with the DMA is between 160 to 200 monitoring cycles. Although this is a high number, these implementations are proof of concepts and they can be greatly optimized. For this reason, we concentrate on the number of DMA transfers since they are the most cycle consuming operation on the partition manager. Moreover, some of these applications will not even run without the partition manager.

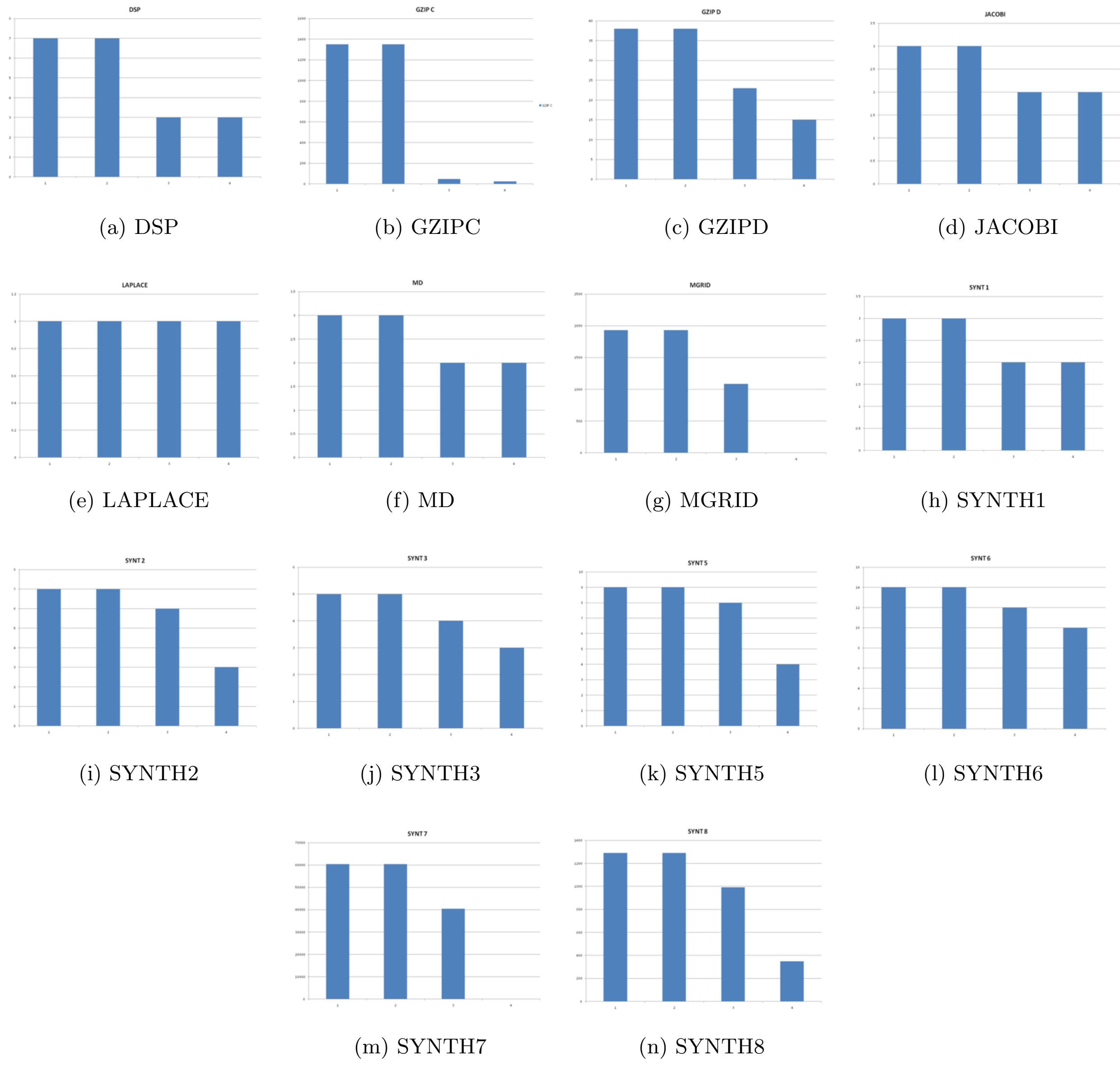
6.4 Partition Manager Policies and DMA counts

Figure 9 and 8 show the relation between the number of DMA and the number of cycles that the application takes using a unoptimized buffer (saving all register file), optimized one buffer (rescheduled and reduction of the number of registers saved), optimized two buffers and optimized four buffers. For most applications, there are a correlation between a DMA's reduction and a reduction of execution time. However, for cases in which the number of partition can fit in the buffers, the cycles mismatch like in Synthetic case 1 and 6.

Figure 10 show the ratio of Partition manager calls versus the number of DMA transfers. The X axis represents the applications tested and the ratios of calls versus one, two and four buffers. As the graph shows, adding the extra buffers will dramatically lower the number of DMA transfers in each partition manager call.

Figure 11 selects the GZIP and MGRID applications to show the advantage of using both replacement policies. In the case of MGRID, both policies gives the same counts because the number of partitions is very low. In the case of the GZIP compression, the LRU policy wins over the Modulus policy. However, in the case of decompression, the Modulus policy wins over the LRU one. This means that the policy depends on the application behavior which opens the door to smart application selection policies in the future.

Finally, in Figure 12, we show that the victim cache can have drastically effects on the number of DMA transfers on a given application (Synthetic case 8). As the graph shows, it can produce a 88x reduction in the number of DMA transfers.



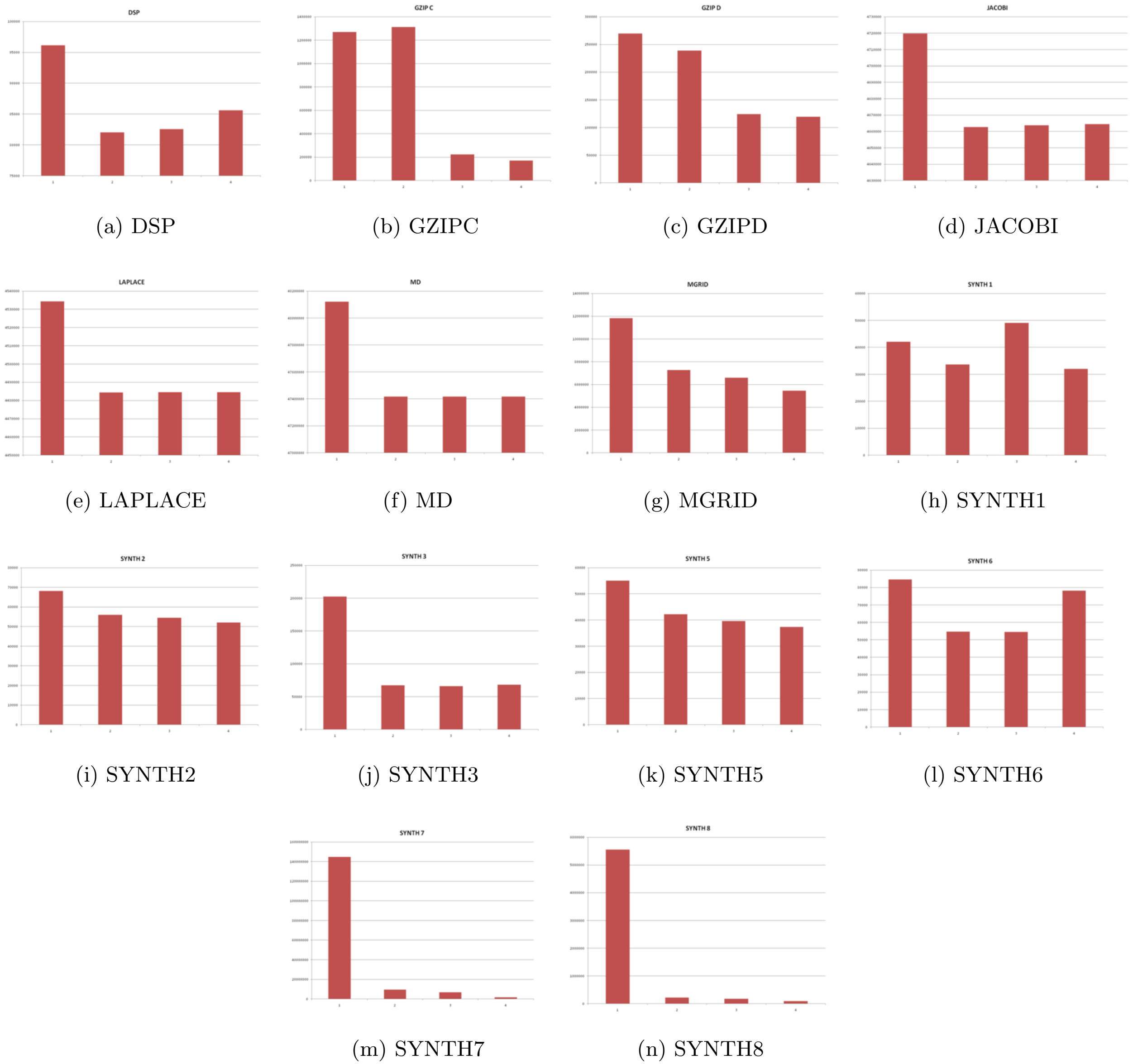
(a) DSP (b) GZIPC (c) GZIPD (d) JACOBI

(e) LAPLACE (f) MD (g) MGRID (h) SYNTH1

(i) SYNTH2 (j) SYNTH3 (k) SYNTH5 (l) SYNTH6

(m) SYNTH7 (n) SYNTH8

Fig. 8. DMA counts for all applications for an unoptimized one buffer, an optimized one buffer, optimized two buffers and optimized four buffer versions



(a) DSP (b) GZIPC (c) GZIPD (d) JACOBI

(e) LAPLACE (f) MD (g) MGRID (h) SYNTH1

(i) SYNTH2 (j) SYNTH3 (k) SYNTH5 (l) SYNTH6

(m) SYNTH7 (n) SYNTH8

Fig. 9. Cycle counts for all applications for an unoptimized one buffer, an optimized one buffer, optimized two buffers and optimized four buffer versions

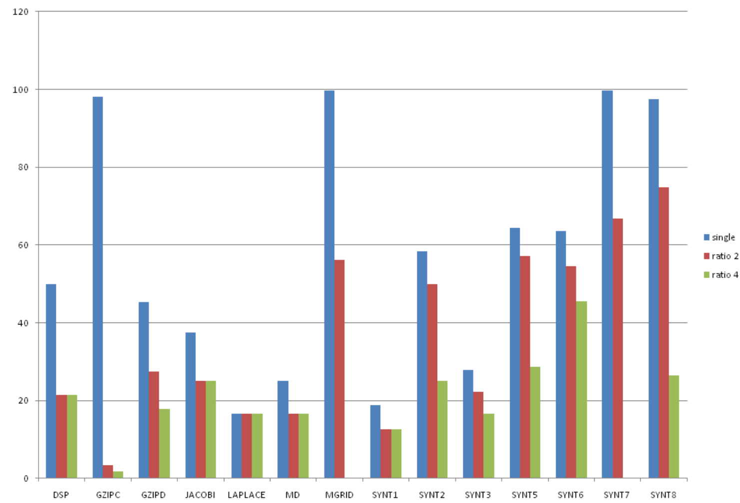


Fig. 10. Ratio of Partition Manager calls versus DMA transfers

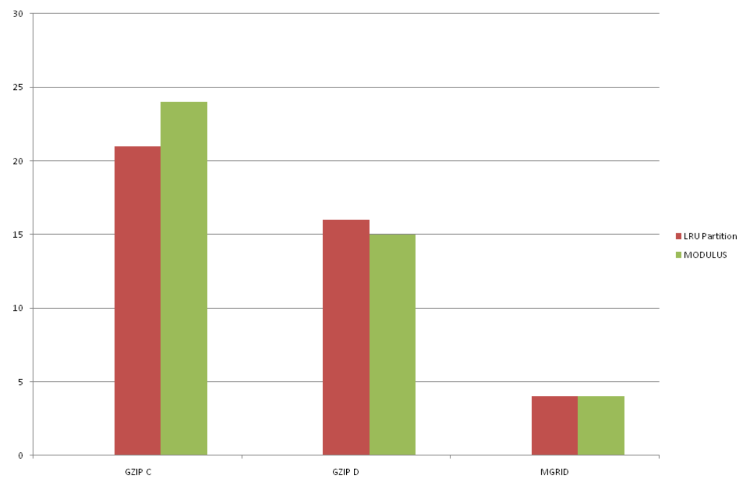


Fig. 11. LRU versus Modulus DMA counts for selected applications

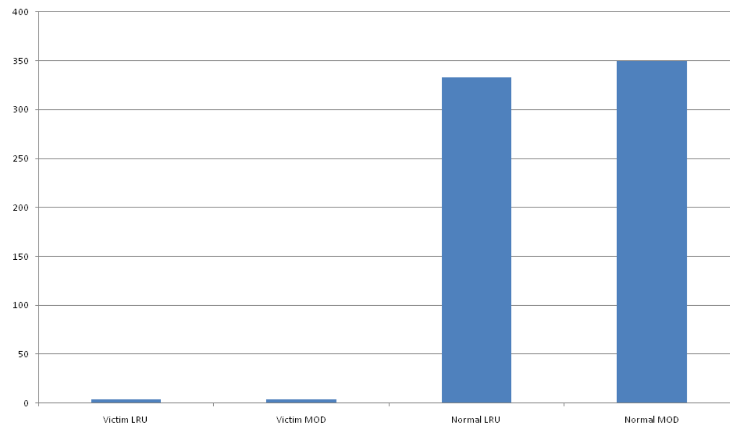


Fig. 12. The victim cache comparison with LRU and Modulus policies

7 Conclusions and Future Work

Ideas presented in this paper show the trend of software in the many core age: the software renaissance. Under this trend, old ideas are coming back to the plate: Overlays, software caches, dataflow execution models, micro kernels, among others. This trend is best shown in architectures like Cyclops-64[5] and the Cell B.E.'s SPE units. Both designs exhibit explicit memory hierarchy, simple pipelines and the lack of virtual memory. The software stacks on these architectures are in a heavily state of flux to better utilize the hardware. This fertile research ground allows the reinvention of these classic ideas. The partition manager frameworks rise from this flux.

This paper shows a framework to support the code movements across heterogeneous accelerators components. It shows how these effort spans across all components of the software stack. Moreover, it depicts its place on a higher abstraction framework for a high level parallel programming language. It shows the effect of several policies dedicated to reduce the number of high latency operations. Future work on this area include the creation of a partition based function call graph which can be used for pre-fetching schemes and the extension of task based framework that allows percolation of code.

References

1. *CBE Architectural Manual*.
2. Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. Cellss: a programming model for the cell be architecture. In *ACM/IEEE CONFERENCE ON SUPERCOMPUTING*, page 86. ACM, 2006.
3. Jordi Caubet. Programming ibm powerxcell 8i / qs22 libspe2, alf, dacs, may 2009.
4. Chen Chen, Joseph B. Manzano, Ge Gan, Guang R. Gao, and Vivek Sarkar. A study of a software cache implementation of the openmp memory model for multicore and manycore architectures. In *Euro-Par (2)'10*, pages 341–352, 2010.
5. Juan del Cuvillo, Weirong Zhu, Ziang Hu, and Guang R. Gao. Tiny threads: A thread virtual machine for the cyclops64 cellular architecture. *Parallel and Distributed Processing Symposium, International*, 15:265b, 2005.
6. Joseph B. Manzano, Ziang Hu, Yi Jiang, Ge Gan, Hyo-Jung Song, and Jung-Gyu Park. Toward an automatic code layout methodology. In *IWOMP*, pages 157–160, 2007.
7. Kevin O'Brien, Kathryn O'Brien, Zehra Sura, Tong Chen, and Tao Zhang. Supporting openmp on cell. *Int. J. Parallel Program.*, 36:289–311, June 2008.