**University of Delaware**
**Department of Electrical and Computer Engineering**
**Computer Architecture and Parallel Systems Laboratory**

# Code Partition and Overlays: A reintroduction to High Performance Computing

Joseph B. Manzano

Ge Gan
Juergen Ributzka
Sunil Shrestha
Guang R. Gao

**CAPSL Technical Memo 108**
August 31st, 2011



University of Delaware • 140 Evans Hall • Newark, Delaware 19716 • USA
http://www.capsl.udel.edu • ftp://ftp.capsl.udel.edu • capsladm@capsl.udel.edu

**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

Future many core designs need to make certain sacrifices to achieve their strict power and performance requirements. Many of the current hardware structures, that consume a lot of power, are the first ones to be simplified or even cut from their designs. However, software systems and high level programming languages depends on such structures and their elimination creates shockwaves on the field. This allows many interesting research opportunities for software systems. Dataflow systems, overlay techniques, dynamic scheduling among others, are prime to resurface. Overlay techniques can be used to help alleviate the problem of explicit memory hierarchies. Moreover, partition frameworks allow novel ideas (like code / data percolation) to be implemented in the new High Performance Computing (HPC) field.

This paper presents a partition manager framework that implements several hardware ideas under a software implementation. The framework implements fine grained partitions which allows efficient code movement across computational components. This paper is divided as follow. Section 2 shows an introduction to the hardware infrastructure which is used. Section 3 shows the related work for this framework. Section 4 presents Open OPELL, a framework which uses the partition manager to implement OpenMP in a heterogeneous high performance chip.

Section 5 shows the components of this current implementation of the partition framework and its enhancements. Finally, sections 7 and 8 show the experimental results and the conclusions and future work.

## 2 The Cell Broadband Engine

The Cell B.E. architecture is an architecture created by IBM, Sony and Toshiba worked to be the heart of their seventh generation home game console and high end products[1]. The chip has a PowerPC core (i.e. PowerPC Processing Element or PPE for short), which is the system's brain. The computational's engine of this architecture is composed of eight Synergistic Processing Elements (or SPE for short). SPEs 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.

Finally, every components are interconnected by a four-ring bus called the Element Interconnect Bus (or EIB for short). Figure 1 shows a block diagram for the Cell B.E. This chip achieves around 200 Giga Floating Point Operations Per Seconds (FLOPS) for single precision and around 102.4 Giga FLOPS for double precision[1].

The PPE core possesses all the common hardware features and can run unmodified Operating Systems and software toolchains. On the other hand, the SPEs lacks many of these components and needs specialized (i.e heavily modified) software runtimes and toolchains.The SPEs exhibit limited local memory, lack caches (both instruction and data), and it has no virtual memory support.

External communication between global memory and the SPE internal memory is achieved through explicit Direct Memory Access (DMA) transfers. This decreases the general "productivity" of the architecture by forcing the software to orchestrate all the data and code movement across its components.

## 3 Related Work

The most famous frameworks to increase the productivity for the Cell B.E. are the ALF and DaCS[3] libraries and the CellSS project[2]. The ALF and DaCS frameworks help to create tasks and to facilitate data communication. The Accelerator Library Framework (ALF) was created to provide a user-level programming framework to develop task based programs for the Cell Architecture. It abstracts many of the low level aspects of Cell B.E. programming, i.e. data transfers, task management, data layout communication, etc. The main objective of ALF was a generalized view of task programming. On the other hand, the DaCS framework takes care of accelerator related problems, such as topology services, data movement schemas and process management. One of its main objectives is to provide a higher abstraction to the

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

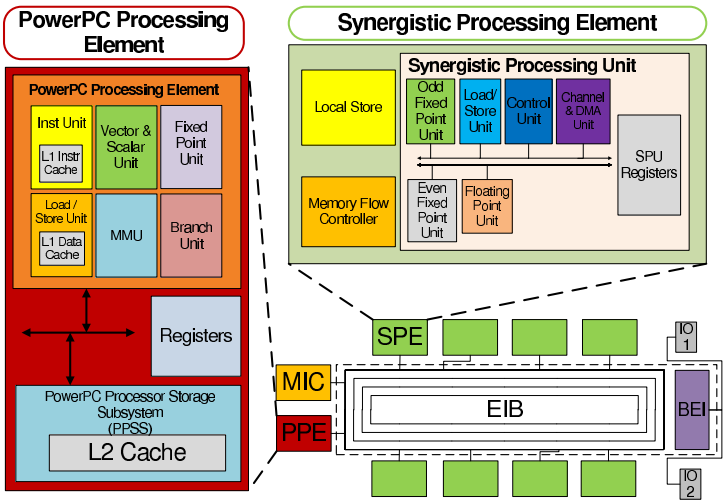



Figure 1: Block Diagram of the Cell Broadband engine

DMA engine communication. Both ALF and DaCS can work together to create high level libraries and frameworks to increase the productivity of the Cell B.E. Thus, these frameworks are designed to be building blocks to create runtimes for high level programming language.

The Cell SuperScalar project (the CellSS) [2] exploits the function parallelism, using pragmas, and schedules them across the Cell B.E. architecture. It is composed of a locality aware scheduler that utilizes the memory spaces more efficiently. It is very similar to OpenMP in the way that it expresses parallelism. However, it is restricted to task parallelism instead of data parallelism (like OpenMP supports). In the partition manager, the body of the parallel functions (partitions) are analogous to the CellSS tasks and the partition manager schedules them.

Finally, there have been efforts to port high level programming languages, like OpenMP, to the Cell B.E.. The most successful one is the IBM's XL compiler implementation of OpenMP [6]. The implementation under the XL compiler mirrors the OPELL implementation with very important differences. The XL's software cache is not configurable in any way. Under OPELL, the line's dirty bits are fully configurable and it allows the implementation of novel memory models[4]. Another difference is that the XL's partition manager 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. This allows implementing several schedulers.

# 4 The OPELL Framework

The importance of porting high level parallel programming languages to the new many core designs has become apparent. For this purpose, the OpenMP on CELL framework, or OPELL for short, was created. It was developed in the University of Delaware and it virtualized all the necessary components of a shared memory system for OpenMP programmers to use.

An effort such as OPELL requires extensive efforts across all the software stack. The major modifications and additions are briefly explained below.

### 4.1 Single Source Compilation.

Under OPELL, a source code is read and its parallel regions are extracted. The serial sections are sent to the host's toolchain; while the parallel regions are arranged[2] and sent to the accelerator's compiler. The necessary code is inserted on both compilation paths when needed. This includes calls to the OpenMP runtime and the insertion of software cache and partition manager inlets. Several structures are created (like partition lists, overlay buffers, etc), and the binary format (i.e. ELF) is modified to support them. After the linking has been completed, an extra step will combine the two binaries to create a single executable. Figure 2 shows a high level graphical overview of the whole single source process.

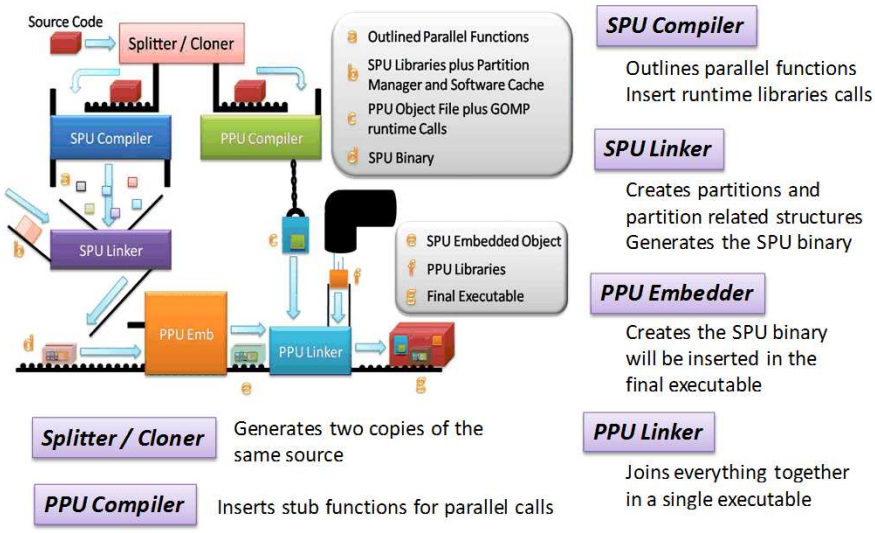



Figure 2: A high level overview of the single source toolchain. Under this framework the SPU Embedder will "generate" a new SPU binary (i.e it wraps it with a special API) so it can communicate with the host

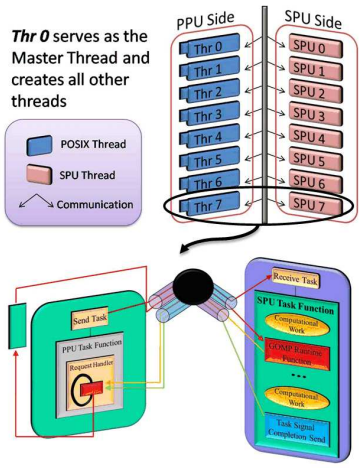
### 4.2 Simple Execution Handler

Each accelerator component is activated at the beginning of execution. Then, it is put to poll for available work. In this way, the cost of turning on and off the accelerator components is mitigated. Under OPELL, this represents our workers' pool. Each worker talks with its host by a distinct buffer communication buffer and a signal[3]. The buffer contains data required for a host's task to execute and the signal will alert the communicating parties of the task identity

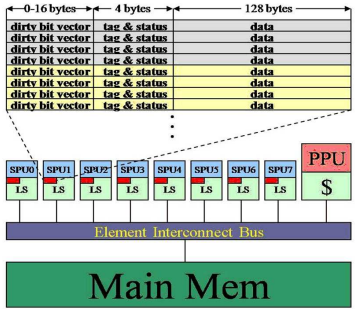
[2]By inserting glue code

[3]in the case of the Cell B.E., this is achieved through mailboxes

or state (aborted or completed). When the host wants to send a parallel job to the accelerator, it will send a signal, with the task id embedded in it, and the accelerator will begin executing. When the worker completes, or requires help (like in the case of a high level synchronization construct like a barrier), it sends a message back to the host with its requirements. When the application ends, the host will send a kill signal to everyone, effectively ending the worker's life. To take advantage of any extra parallelism that the host might exhibit, each communication buffer is managed by a distinct (POSIX) thread on the host side. These are called shadow or mirror threads. 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

Figure 3: Components of the Simple Execution handler and the Software cache

### 4.3 Software Cache

To simulate the shared memory environment and to overcome explicit and/or restricted memory hierarchies, the OPELL framework implemented software caches and partition managers. In the case of the software cache, it sports all the features of its hardware brethrens. It has 4-way associate 64 sets with a cache line size of 128 bytes. It has a total size of 32 KiB and it has write back and write allocate update policies. The main difference between this cache and others is the finer control over the granularity of the dirty bits per line. Each line can support up to 128 dirty bits (one for each byte) and ist configuration can be changed to support bytes, quarter words, half words, words, double words and quad words. In this way, many novel memory models can be simulated on top of this framework, as shown in [4]. A graphical overview of the software cache is presented by figure 3b.

### 4.4 Overlay / Partition Manager.

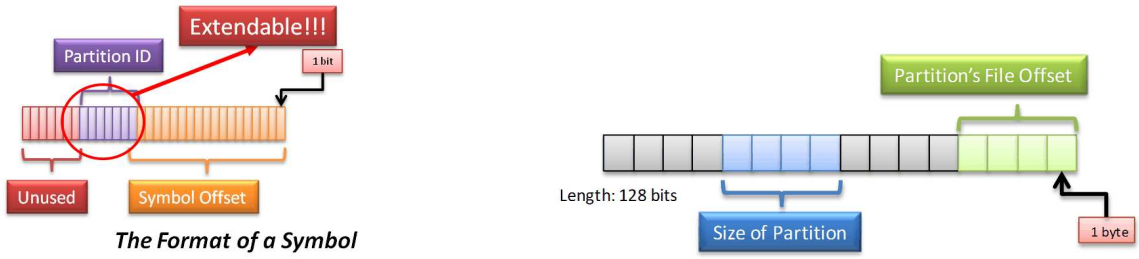
The other component used to mask the heterogeneity of the memory hierarchy is the partition manager. This small component takes care of loading and managing code when it is needed. It is similar to way virtual memory loads pages, but it supports different sizes partition and overlays. It also has several replacement policies. Finally, its partition can be moved, saved and reallocated anywhere in the program space as the framework sees fit. A function call will be intercepted by the partition manager when needed and its code loaded (if needed) to a selected memory region. All this process is invisible to the programmer and none of the partition manager work will leave any trace. A more detailed description of this component is given in the next section.

## 5 The Partition Manager

To support the partition manager framework, changes on all the software stack components are required. These include changes to the compiler, assembler, linker and the runtime itself. The major changes to each component are described briefly in the next paragraphs.

### 5.1 Major Toolchain Changes

Symbols for partitionable functions are modified to include its resident partition identification number. Under the framework, the identification number of zero represents not partitionable code. This implies resident code and allows external libraries to be linked against the executable without any changes. The format of a partitionable symbol is described in figure 4a Several




(a) A symbol address bit range

(b) The Partition list entry

Figure 4: The symbol address bit range and the Partition list entry format

new assembly directives are added to the compiler and assembler to convey information about partitions and aid in code generation. This includes directives to mark code as in a partition, directives which will prevent the deletion of important information on the symbols, directives which will disable the partition call if the analysis tells us that it is not needed, among others. A comprehensive list of the directives and a brief description can be found in table 1.

Table 1: Overview of New Assembly Directives

| Directive | Description |
|---|---|
| .partition *pn* | Define the next code section to belong to partition pn |
| .pm *caller* | The following function call might be interpartitional |
| .libcallee | The following function call will always target partition zero |
| .funcpointer | The following load or store loads a function pointer. Thus, DO NOT clean the upper 15 bits. |

### 5.2 The Partition List

It is a list of where to find the partitionable code and its size. Under OPELL, it is added to the end of the ELF accelerator's data segment. During runtime, the partition manager will access this list. The global address is calculated by adding the program entry point with the offset under this entry, and a transfer is initialized. Since all the code in these regions is Position Independent Code (PIC), the loaded location of the code is decided during runtime by the manager. The index to this array is extracted from the symbol's address (which has its containing partition id). The format and the bit range of the partition list entries are described in figure 4b.

### 5.3 The Partition Stack

The partition stack is a structure which is analogous to the function call stack in normal software applications. It keeps track of the caller / callee relationships among partitions. This is necessary when dealing with long function call chains which span several partitions. This structure also keep important meta information about the partitions involved in the chain (such as lifetime, partition relocation information, important runtime information, etc). Although, relocation is allowed for all new partitions, partitions involved in the returning path of the partition stack are not allowed to be relocated. They must resume execution at the same memory region in which they were originally placed. The reason for this is in the case that the reallocation is allowed, registers might contain stale jumping information. This can be corrected by substantial binary rewriting and analysis on the part of the partition manager framework which will make its overhead unacceptable.

### 5.4 The Partition Buffer

All the partition code is loaded into the partition buffer which is a special memory region designated to swap code in and out. It is managed by the partition manager kernel and it can

be divided into several sub-regions of different sizes. Each region has its state used for book keeping or replacement policies (such as lifetime, children ids, etc). Both the buffer and the list are added to the accelerator's binary image. This produces the image shown in figure 5. The buffer is placed just after the interrupt table of the image.

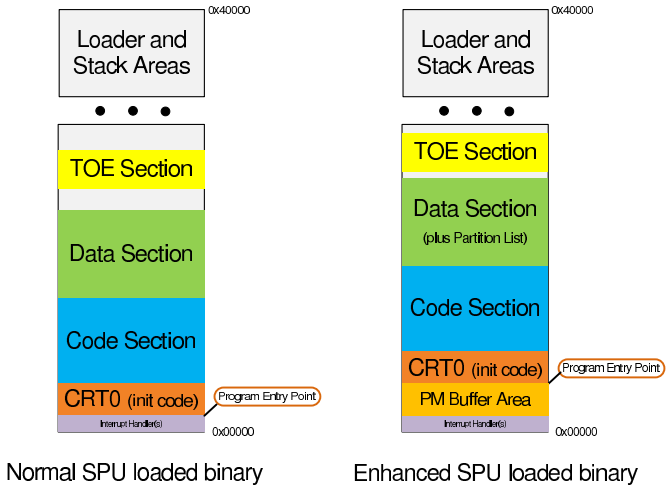



Figure 5: A comparison between the modified SPE binary image and a normal one

### 5.5 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

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.

## 6 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

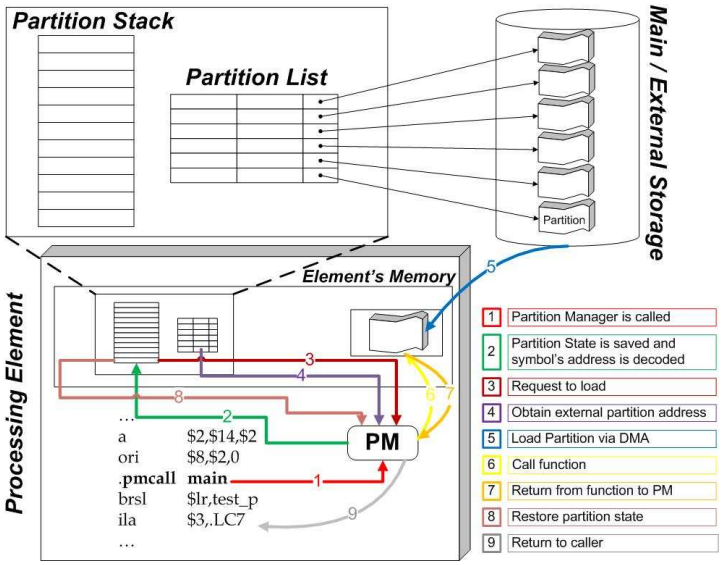



Figure 6: A typical partition manager call

| 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 2: The Four States of a Partition

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 2.

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 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 6.3.

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.

## 6.1 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.

## 6.2 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.

## 6.3 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[4]. When a new partition is being loaded into the main memory, the evicted partition

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

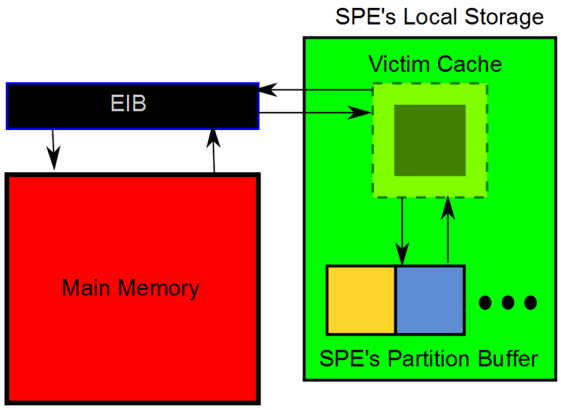



Figure 7: The victim cache scheme

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 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.

# 7 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.

## 7.1 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

| 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 3: Applications used in the Harahel testbed

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.

### 7.2 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 3.

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

### 7.3 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.

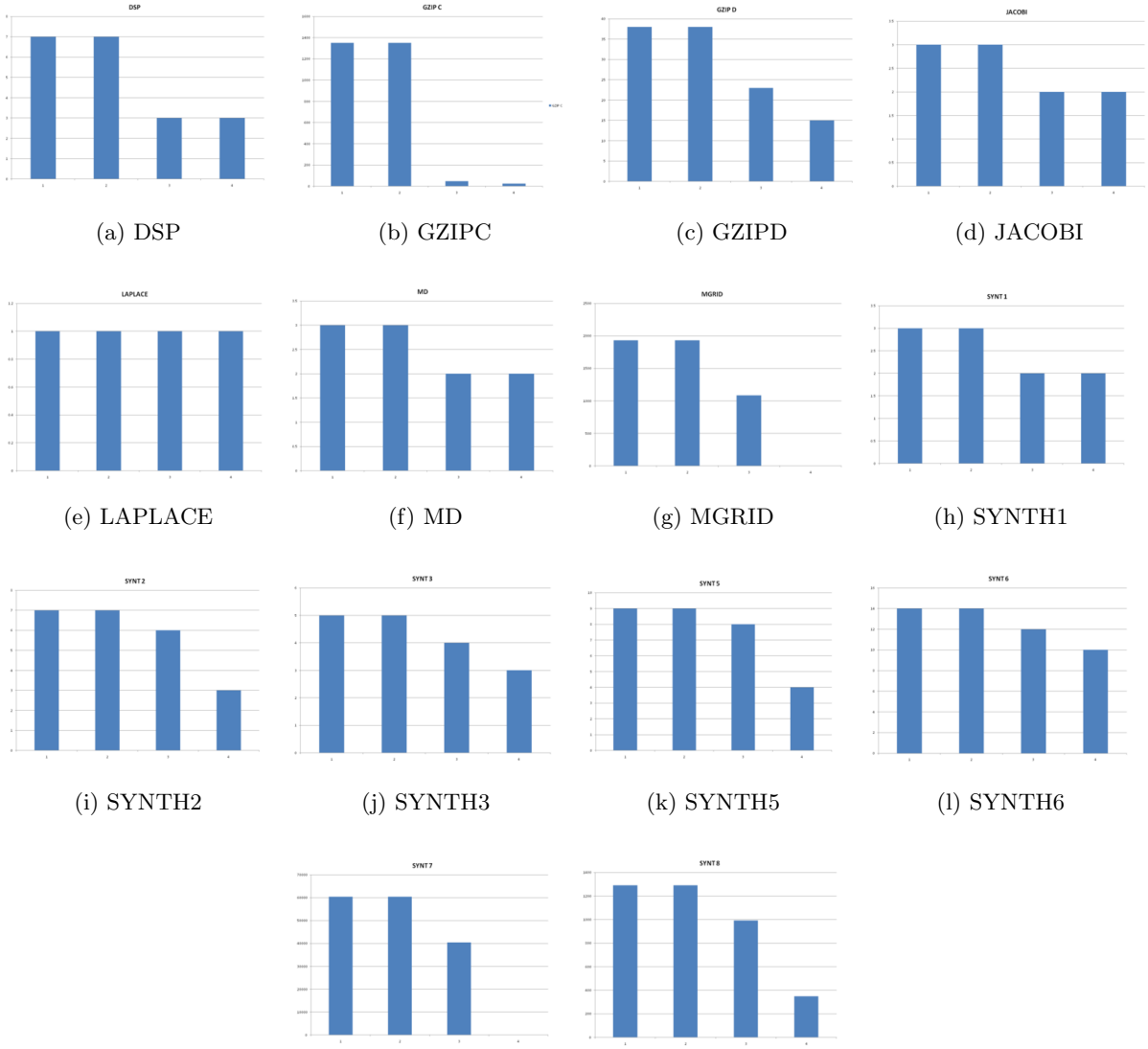
### 7.4 Partition Manager Policies and DMA counts

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

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

# 8 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


(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

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

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

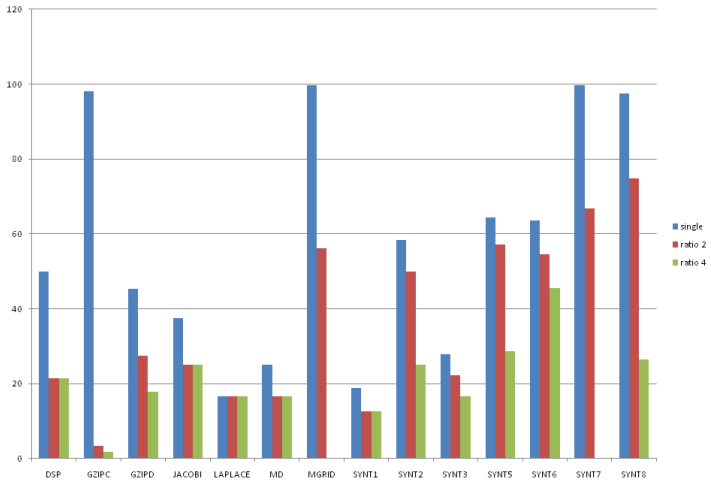



Figure 10: Ratio of Partition Manager calls versus DMA transfers

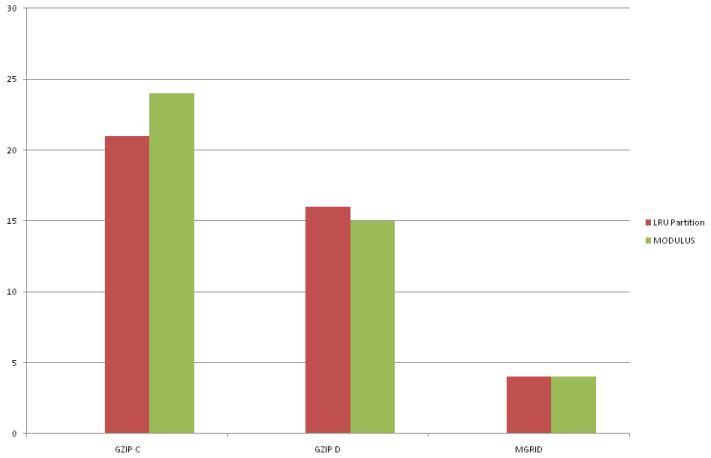



Figure 11: LRU versus Modulus DMA counts for selected applications

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.

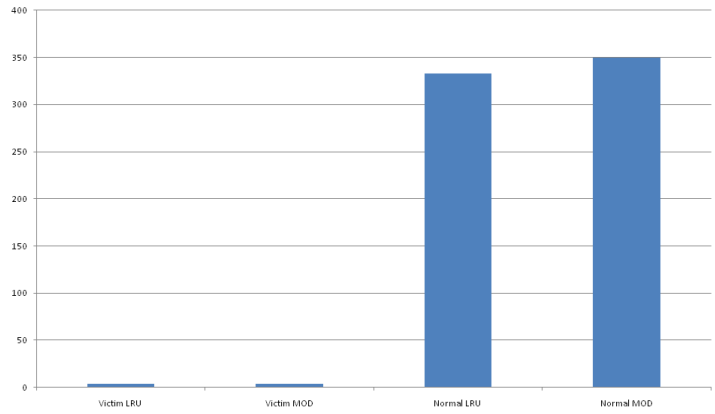



Figure 12: The victim cache comparison with LRU and Modulus policies

[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] 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.