# Cooperative Multithreading on the Cell/B.E.

Vicenç Bertran[1], David Carrera[2], Jordi Torres[1,2], Eduard Ayguadé[1,2]

[1]Barcelona Supercomputing Center

Nexus II – Jordi Girona, 29 – Barcelona. Spain

[2]Department of Computer Architecture

Technical University of Catalonia

Building C6 – Campus Nord – Barcelona. Spain

**Abstract**

The Cell/B.E. processor has proved that heterogeneous multi-core systems can provide a huge computational power with high efficiency for a wide range of applications. The simple design of the computation units and the use of small managed local memories, rather than traditional super-scalar out of order processors with large caches, is the key to achieve this efficiency and performance at the same time. However, this simple and efficient design has one important drawbacks that cannot be underestimated: the increase in the code complexity to achieve good performance. The code written to run in this kind of processors must deal with several issues such as code vectorization, loop unrolling, management of data transfers from main memory to the local storage, etc. Some of this issues such as vectorization or loop unrolling can be partially solved by the compiler, but others such as the overlapping of memory transfers and computation must be manually addressed by the programmer. In this paper we propose our user level threading library, which allows the concurrent execution of several threads inside each SPU, to hide memory latencies and to overlap computation and trasfer times without increasing the code complexity.

## 1 Introduction

CellMT is a threading library for the Cell Broadband Engine Architecture(TM) that enables the concurrent execution of multiple threads inside each SPE processor. The focus of the library is to make the most of the SPE processors while improving its programability. The CellMT library is specially well suited for complex applications with unpredictable memory access that cannot easily use prefetch techniques such as double

buffering or multi buffering. Although these techniques effictively overlap transfer times and computation times, they also requiere non trivial code modification that are not always feasible. Instead, the CellMT library provides a familiar and well understood programming model that is already used to split work across SPUs, so it does not increase the complexity of the application.

The library provides a cooperative multithreading model, so it relies on the threads themselves to relinquish control once they are at a context switch point. This cooperative multithreading model is a perfect fit for any processor with a managed local store, such as the Cell processor, because the context switch points are easily identified. In fact, all the applications written for the Cell have this points explicitly identified by the memory flow control (MFC) operations used to wait for DMA request or Mailbox messages.

The CellMT library include the core threading library libsafecellmt_spu.a that can be directly used from the SPEs, and a dynamic library (libspe2mt.so) that wraps the original libspe2 library and provides a more convenient way to use the threading capabilities directly from the PPU side. The rest of the paper is organized as follows: Section 2 presents the related work. Section 3 introduces the Cell Broadband Engine processor and studies the latency of DMA operations. Section 4 presents our threading library for the Cell SPUs. In section 5, we performa a brief performance evaluation of our threading library. Finally section 6 draws the conclusions and present the future work.

## 2 Related work

Techniques such as double–buffering or multi–buffering [3] have been widely used in the Cell/B.E. to hide DMA latencies. Although both techniques are very effective, they must be used on a case–by–case basis, because these techniques require non–trivial and error–prone code modifications which are only suitable for applications with very predictable memory accesses. Other techniques have been proposed in the literature to hide memory latencies. In [5], authors propose a prefetching technique for I/O intensive applications which is effective for applications with huge working sets that do not fit in main memory. In [4] a software cache is proposed for irregular access, that improve the performance of some applications. Although all the afore–mentioned techniques are valid and effective for some specific applications, we need a more generic solution that can be effectively implemented in the runtime system, without increasing the overall system complexity. To this end we have investigated the use of multi–threading to hide memory latencies on the Cell/B.E. The SPUNK [1] nano–kernel provides a micro–threading model to increase the utilization of the Cell/B.E. resources while simplifying the programming model. The main goal of SPUNK is also to overlap DMA latencies with computation, but its context switch time of 4 ticks (compared to the 2.9 and 3.9 ticks of a DMA request of size 128 and 2048 bytes respectively) make it unsuitable for most applications. In contrast,

the CellMT library has an context switch overhead of only 0.5 to 0.8 ticks, which makes its applicability broader. We have augmented our execution runtime system with the CellMT library to effectively hide DMA latencies, and improve overall system performance in a transparent way.

# 3 Cell/B.E. Architecture

The Cell Broadband Engine Architecture (CBEA) [2] is a single chip heterogeneous multiprocessor. The design goals of the Cell processor were to address the fundamental challenges facing modern microprocessor development: high memory latencies and on-core power dissipation. Until now, microprocessors have achieved performance improvements through higher clock frequencies and deeper pipelines, but the fundamental problem that current processors face is the memory wall [6]. On modern processors significant amounts of time are spent waiting in memory stall, due to the large difference between the processor and the memory speed. Large memory latencies make it difficult to obtain further performance gains with traditional processor designs based on hardware caches. The Cell processor approaches this problem in a different way, providing a heterogeneous processor with explicit memory management. This approach potentially improves the throughput of the processor, but also increase the effort to efficiently implement a program.

Figure 1 shows the three basic components of the Cell processor. First, the PowerPC Processor Element (PPE), which is primarily intended to manage global resources. Second, the Synergistic Processing Elements (SPE) that are specialized vectorial processors. Finally, the communication between the PPE, the SPEs, main memory, and external devices is realized through an Element Interconnect Bus (EIB).

## The Synergistic Processing Element

SPEs are specialized vectorial units with an instruction set similar (but not compatible) to the PPE VMX. The main difference is found in the memory hierarchy, which is divided into three levels: the main memory, the local stores and a large unified register files. The large unified register file makes it possible to hold the majority of operands directly inside the CPU core without having to spill values onto the stack. A 256Kb local store is used to store the SPE code and temporary data. SPEs can perform asynchronous DMA transfers between their local stores and main memory

SPEs are designed to execute regular computationally intensive programs rather than general purpose software. This allows the system to hide memory latencies without having to employ complex hardware mechanisms such as branch-prediction, out-of-order execution, and deep pipelining, often used in superscalar processor cores. This permits the reduction of the hardware required to obtain a high throughput and hardware utilization on regular programs, but at the cost of requiring a considerable effort in order to

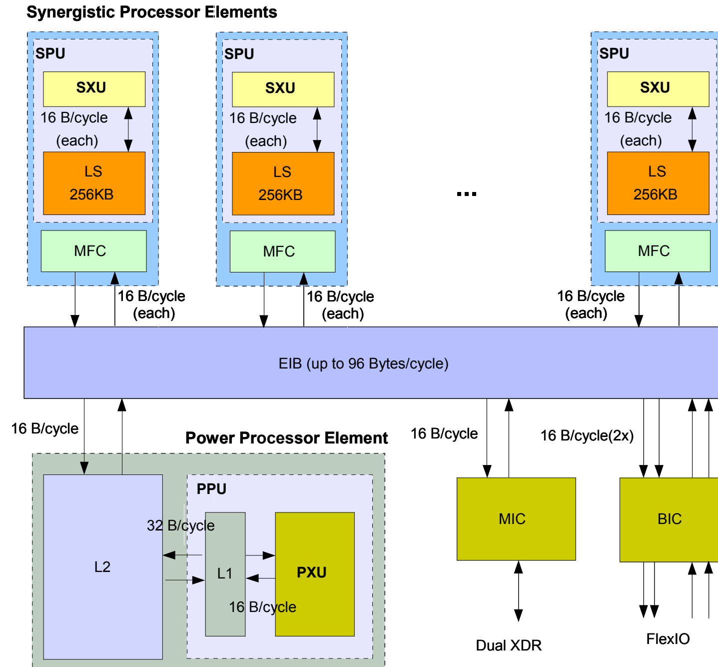


Figure 1: The Cell Broadband Engine Architecture

optimize non-regular programs in such a way that an acceptable performance is obtained. The simple design of the SPE cores makes it possible to pack together up to eight SPU in a single multi-core-die.

## The Element Interconnect Bus

Communication between SPEs and the Element Interconnect Bus (EIB) is realized through the SPEs Memory Flow Controller (MFC). The MFC of each SPE can enqueue up to 16 DMA commands, which implies that the whole system can process more than 100 DMAs simultaneously. It also provides memory mapped I/O registers (MMIO) and channels to monitor DMA commands, SPU events and facilitate interprocess communication via mailboxes and signal-notification. Mailboxes are a set of queues that support exchanges of 32-bit messages between an SPE and other devices. Two one-entry mailbox queues are provided for sending messages from the SPE. The EIB is a 4-ring structure (2 clockwise, 2 counterwise) for data, and a tree structure for commands with an internal bandwidth of 96 Bytes per cycle. The EIB has two external interfaces, the MIC, which is the interface between main memory and EIB and the BEI, which allows data transfer between EIB and I/O devices.

The EIB has a theoretical peak data bandwidth of 204.8GB/s, but the DMA operations with main memory are limited to 25.6GB/s. Moreover, the data transfer times form main memory to a SPU local storage have a delay of at least 1000 processor cycles that is not negligible for small data transfers. Figure

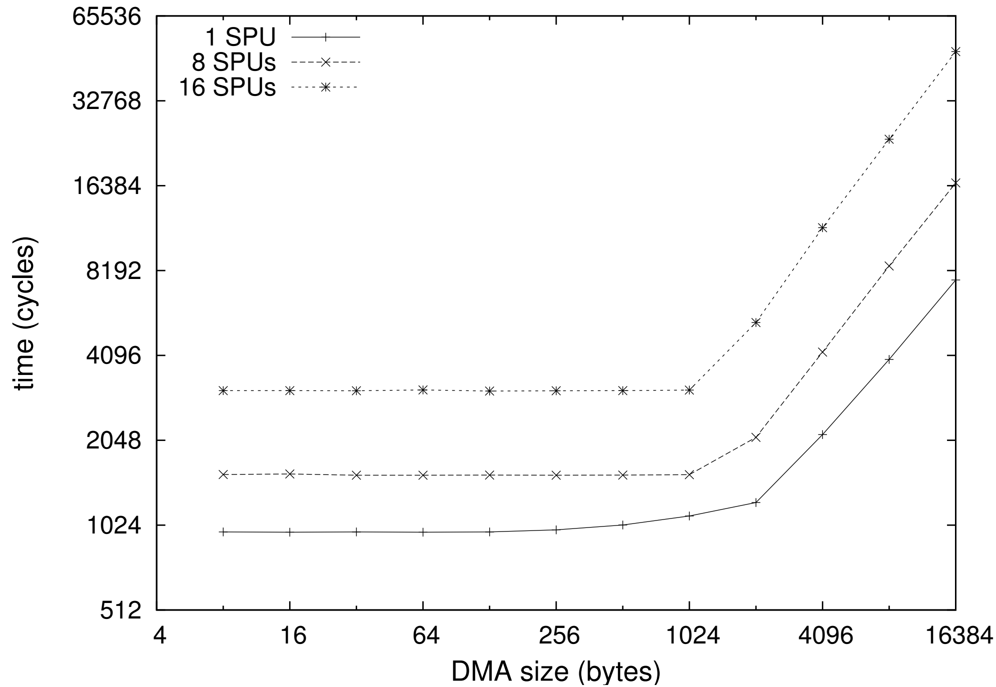


Figure 2: DMA latencies of the Cell/B.E.

2 show the transfer time of DMA read operations with block sizes that ranges from 8 bytes to 16 Kbytes in the x-axis. The y-axis measures the transfer time in processor cycles. The ideal transfer time of a DMA read operations are composed of a initial delay plus the DMA block size divided by the memory bandwdith. This initial delay dominate the transfer times of DMA read operations for block sizes of up to 1024 bytes. In Figure 2 there are three different configurations evaluated. The first measures the performance of DMA operations when only one SPU is active, the second configuration measures the performance of DMA read operations of eight concurrent SPUs. Finally, the last configuration shows the performance when all the 16 SPUs of a QS20 are evaluated. As we can see, the bus congestion increases the latency from 1000 cycles for one SPU alone to more than 3000 for the 16 SPUs configuration. The initial DMA transfer delay is not amortized until we use DMA block sizes of at least 2048 bytes, when the transfer time becomes dominated by the DMA block size.

# 4 CellMT library

The cooperative multi-threading library is implemented on a core library that provides all the features and flexibility required to run complex multi-threaded application inside the SPUs. This core library, which is described in detail in section 4.1, provides a low-level threading API that can be directly called from the

SPU application code. This low-level API is useful to write applications with complex interactions between threads, but its flexibility can also increase the complexity of the applications itself. To address this issue, the CellMT library also provides an auxiliar library described in section 4.2, that simplifies the development of applications that follow a common threading patterns. This auxiliar library is a wrapper to the standard libspe2 library that is used from the PPU side, and provides a high level abstraction to use the SPU threads.

## 4.1 The libcellmt library

The libcellmt library is a self-contained SPU library with all the required features to run a multi-threaded application. Listing 1 shows the functions that this library provides to the programmer. The most important functions are *run_thread(...)*, *wait_for(..)* and *yield()*. The first one allows the programer to spawn a new thread that will start executing the function *int f(...)* with the specified parameters and using the a custom stack pointer. This function return error if the number of threads has reached the maximum or 0 on success. The thread id of the new thread is returned on the *int *th_id* parameter. The function *wait_for(...)* is used to wait for the end of a previously created thread. Finally, the *yield()* function is used to transfer the execution to another thread. If there is only one active thread this function do not have any effect.

Besides this three functions, there is also other auxiliar functions such as *get_thread_id()* that return the thread id of the current thread or *get_free_stack_space()* that returns the available stack space of the current thread. Finally, there are several non blocking functions used to wait for common events such as DMA completions or SPU channel activity.

Currently the maximum number of SPU threads supported are 16, although this number can easily increased if necessary. The library is completly embedded with the user application at compilation time and it does not need to initialize any dinamyc data structure or variable.

Listing 1: Core CellMT API

```
#define MAX_NUM_THREADS 16

int get_thread_id();

int get_free_stack_space();

void yield(void);

int wait_for(int thread_id, int *ret);

void run_thread(unsigned long long id, unsigned long long argp, unsigned long long envp,
                int (*f)(unsigned long long, unsigned long long, unsigned long long),
                void * thread_stack_pointer,
                unsigned int stack_size,
                int *th_id);

static inline void mfc_mt_wait_all (const int tag){
    const unsigned int mask = 1 << tag;
    unsigned int ret;
    do {
        yield();
        mfc_write_tag_mask(mask);
        mfc_write_tag_update(MFC_TAG_UPDATE_IMMEDIATE);
        ret = mfc_read_tag_status();
```

```
    } while (unlikely((ret & mask)) == 0) ;
}

static inline int spu_mt_read_in_mbox(void){
    while(spu_stat_in_mbox() == 0) yield();
    return spu_read_in_mbox();
}

static inline void spu_mt_write_out_mbox(int data) {
    while(spu_stat_out_mbox() == 0) yield();
    spu_write_out_mbox(data);
}
```

## 4.2 The libspe2mt library

This library is intended to ease the development of applications with a common multi-threaded pattern, which is the same pattern already used to split work across several SPEs. It follows the philosophy of the standard libspe2 library, but extends its functionality to support the execution of multiple threads in each SPE. The complet API of the libspe2mt are in listing 2. Each of these functions are wrappers to the original libspe2 functions. There is only one new function in this library: the *spe_mt_context_add_thread(...)* which can be used to specify the number of threads that will run on a *spe_mt_context*. Each of the configured thread will execute the *main()* function of this *spe_mt_context* with the specified arguments.

Listing 2: Libspe2mt API

```
typedef struct spe_mt_context* spe_mt_context_ptr_t;

spe_context_ptr_t get_context(struct spe_mt_context *ctx);

spe_mt_context_ptr_t  spe_mt_context_create(unsigned  int flags, spe_gang_context_ptr_t gang);

int spe_mt_program_load (spe_mt_context_ptr_t spe, spe_program_handle_t *program);

int spe_mt_context_add_thread(spe_mt_context_ptr_t  spe, void *argp, void *envp);

int spe_mt_context_run(spe_mt_context_ptr_t  spe, unsigned int *entry, unsigned
               int runflags, spe_stop_info_t *stopinfo, unsigned int stack_size);

int spe_mt_context_destroy (spe_mt_context_ptr_t  spe);
```

## 4.3 A "Hello World" example with the CellMT library

This section explains the basic steps required to run a program with the CellMT library.

Listing 3: ppu_main.c file

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <libspe2mt.h>
#include <pthread.h>

extern spe_program_handle_t mt_hello_world_spu;

#define MAX_SPU_THREADS     16

#define NUM_ITERS 16
#define NUM_MT_THREADS 4
#define STACK_SIZE 0 // 0 => Use all the available space
```

```
void *ppu_pthread_function(void *arg) {
  spe_mt_context_ptr_t ctx;
  unsigned int entry = SPE_DEFAULT_ENTRY;

  ctx = *((spe_mt_context_ptr_t *)arg);

  const int num_iters = NUM_ITERS / NUM_MT_THREADS;
  unsigned int i;
  for(i=0; i<NUM_MT_THREADS; i++) {

    int ret = spe_mt_context_add_thread(ctx, 0 /*argp*/, (void*)num_iters /*envp*/);
    if(ret != 0){
      perror ("Failed add thread");
      exit(EXIT_FAILURE);
    }
  }

  int ret = 0;
  if ((ret = spe_mt_context_run(ctx, &entry, 0, NULL, STACK_SIZE) < 0)) {
    perror ("Failed running context");
    exit (1);
  }
  pthread_exit(NULL);
}

int main(){

  int i, spu_threads;
  spe_mt_context_ptr_t ctxs[MAX_SPU_THREADS];
  pthread_t threads[MAX_SPU_THREADS];

  spu_threads = MAX_SPU_THREADS;

  for(i=0; i<spu_threads; i++) {
    /* Create context */
    if ((ctxs[i] = spe_mt_context_create (0, NULL)) == NULL) {
      perror ("Failed creating context");
      exit (1);
    }
    /* Load program into context */
    if (spe_mt_program_load (ctxs[i], &mt_hello_world_spu)) {
      perror ("Failed loading program");
      exit (1);
    }
    /* Create thread for each SPE context */
    if (pthread_create (&threads[i], NULL, &ppu_pthread_function, &ctxs[i]))  {
      perror ("Failed creating thread");
      exit (1);
    }
  }

  /* Wait for threads to complete execution.  */
  for (i=0; i<spu_threads; i++) {
    if (pthread_join (threads[i], NULL)) {
      perror("Failed pthread_join");
      exit (1);
    }
    /* Destroy context */
    if (spe_mt_context_destroy (ctxs[i]) != 0) {
      perror("Failed destroying context");
      exit (1);
    }
  }

  return (0);
```

Listing 4: mt_hello_world_spu.c file

```
#include <stdio.h>
#include <cellmt.h>

int cellmt_main(unsigned long long id,
                unsigned long long argp,
                unsigned long long envp)
{
    printf("Thread %d, Hello World!\n", get_thread_id());

    unsigned long long i;
    for(i=0; i<envp; i++) {
        printf("Thread %d, iteration: %llu\n", get_thread_id(), i);
```

```
        yield(); // <—— context switch
    }

    return 0;
}
```

Listing 5: PPU Makefile

```
########################################################################
#                        Local Defines
########################################################################

INCLUDE         := -I $(CELLMT_TOP)/include

IMPORTS         +=  $(CELLMT_TOP)/lib/libspe2mt.so
```

Listing 6: SPU Makefile

```
########################################################################
#            Local Defines
########################################################################

INCLUDE         := -I $(CELLMT_TOP)/include/spu

IMPORTS         += $(CELLMT_TOP)/lib/spu/libcellmt_spu.a $(CELLMT_TOP)/lib/spu/libspe2mt_spu.a
```

### 4.4 Multithreading vs. double buffering

In this section we compare the code complexity of our multi-threaded technique vs. the code complexity of double buffering techniques. Listing 7, shows the simplest code required to encrypt a data buffer resident in main memory. As we can see the steps required to encrypt a buffer of an arbitrary size are straitforward. The original buffer is transfered to the local storage in blocks of size *local_buffer_size*. Each of these block are then encrypted and the resulting block is copied back to main memory. This process is repeated until all the data has been encrypted. The main drawback of this code is that we are not overlapping data transfer times with computation times.

Listing 7: AES simple buffering

```
int aes_simple_buffering(unsigned long long buffer,
                          const unsigned int buffer_size,
                          const unsigned int local_buffer_size,
                          const AES_KEY *key,
                          const int mode){

    unsigned char local_buffer[local_buffer_size] align16;
    const int iters = buffer_size/local_buffer_size;
    int tag = mfc_tag_reserve();

    int i;
    for(i=0; i<iters; i++){

        mfc_getb(local_buffer, buffer, local_buffer_size, tag, 0, 0);
        mfc_wait_all(tag);

        AES_ecb_encrypt_fast((const vec_uchar16 *)local_buffer,
                             (vec_uchar16 *)local_buffer,
                             (const unsigned long)local_buffer_size,
                             key, mode);

        mfc_put(local_buffer, buffer, local_buffer_size, tag, 0, 0);
        buffer += local_buffer_size;

    }
```

```
    mfc_wait_all(tag);
    mfc_tag_release(tag);

    return 0;
}
```

To improve the performance of the code shown in Listing 7 we can use double buffering techniques. With double buffering we can overlap the computation time of the current block with the transfer time of the next block. Listing 8 shows the double buffering version of the original code. As we can observe, the complexity of the loop has increased. Now we need an epilogue and a prologue to correctly process the first and last blocks of the buffer. Moreover, the loop must be unrolled to process two block per iteration. We also need two times more space in the local storage to allocate the buffers required to do double buffering. Altough this code is more efficient than the first version, it is also more complex and error-prone.

Listing 8: AES double buffering

```
int aes_double_buffering(unsigned long long buffer,
                          const unsigned int buffer_size,
                          const unsigned int local_buffer_size,
                          const AES_KEY *key,
                          const int mode){

    unsigned char local_buffer[2][local_buffer_size] align16;
    const int iters = (buffer_size/local_buffer_size)/2;
    int tag[2] = {mfc_tag_reserve(), mfc_tag_reserve()};

    mfc_get(local_buffer[0], buffer+(local_buffer_size*0), local_buffer_size, tag[0], 0, 0);
    mfc_get(local_buffer[1], buffer+(local_buffer_size*1), local_buffer_size, tag[1], 0, 0);

    int i;
    for(i=0; i<iters-1; i++){
        mfc_wait_all(tag[0]);
        AES_ecb_encrypt_fast((const vec_uchar16 *)local_buffer[0],
                             (vec_uchar16 *)local_buffer[0],
                             (const unsigned long)local_buffer_size, key, mode);
        mfc_put (local_buffer[0], buffer+(local_buffer_size*0), local_buffer_size, tag[0], 0, 0);
        mfc_getb(local_buffer[0], buffer+(local_buffer_size*2), local_buffer_size, tag[0], 0, 0);

        mfc_wait_all(tag[1]);
        AES_ecb_encrypt_fast((const vec_uchar16 *)local_buffer[1],
                             (vec_uchar16 *)local_buffer[1],
                             (const unsigned long)local_buffer_size, key, mode);
        mfc_put (local_buffer[1], buffer+(local_buffer_size*1), local_buffer_size, tag[1], 0, 0);
        mfc_getb(local_buffer[1], buffer+(local_buffer_size*3), local_buffer_size, tag[1], 0, 0);

        buffer += (2*local_buffer_size);
    }

    mfc_wait_all(tag[0]);
    AES_ecb_encrypt_fast((const vec_uchar16 *)local_buffer[0],
                         (vec_uchar16 *)local_buffer[0],
                         (const unsigned long)local_buffer_size, key, mode);
    mfc_put(local_buffer[0], buffer+(local_buffer_size*0), local_buffer_size, tag[0], 0, 0);

    mfc_wait_all(tag[1]);
    AES_ecb_encrypt_fast((const vec_uchar16 *)local_buffer[1],
                         (vec_uchar16 *)local_buffer[1],
                         (const unsigned long)local_buffer_size, key, mode);
    mfc_put(local_buffer[1], buffer+(local_buffer_size*1), local_buffer_size, tag[1], 0, 0);

    mfc_wait_all(tag[0]);
    mfc_tag_release(tag[0]);
    mfc_wait_all(tag[1]);
    mfc_tag_release(tag[1]);

    return 0;
}
```

Finally, Listing 9 shows the multi-threaded version of the original code. In this version we achieve a true overlaping of data transfer times and computation transfer times through the use of multiple threads. Each threads works on its own portion of the original buffer. When one thread is going to wait for a DMA transfer to be completed, the thread relinquish the control to another thread. The only change required is the substitution of the *mfc_wait_all(tag)* macro by the *mfc_mt_wait_all(tag)* macro. The rest of the code remains totally unchanged. The steps required to initialize and split the work into several threads do not increase the user code complexity because all the Cell/B.E. programs are already designed to split the work across several SPUs. In the next section we study the cost of a context switch to prove the feasibility of our multi-threaded approach.

Listing 9: AES multi-threading

```
int aes_mt_simple_buffering(unsigned long long buffer,
                            const unsigned int buffer_size,
                            const unsigned int local_buffer_size,
                            const AES_KEY *key,
                            const int mode){

    unsigned char local_buffer[local_buffer_size] align16;
    const int iters = buffer_size/local_buffer_size;
    int tag = mfc_tag_reserve();

    int i;
    for(i=0; i<iters; i++){
        mfc_getb(local_buffer, buffer, local_buffer_size, tag, 0, 0);
        mfc_mt_wait_all(tag);

        AES_ecb_encrypt_fast((const vec_uchar16 *)local_buffer,
                             (vec_uchar16 *)local_buffer,
                             (const unsigned long)local_buffer_size,
                             key, mode);

        mfc_put(local_buffer, buffer, local_buffer_size, tag, 0, 0);
        buffer += local_buffer_size;
    }

    mfc_mt_wait_all(tag);
    mfc_tag_release(tag);

    return 0;
}
```

Listing 10: mfc_wait_all vs. mfc_wait_mt_all

```
static inline void mfc_wait_all(const int tag){
    mfc_write_tag_mask(1 << tag);
    mfc_read_tag_status_all();
}

static inline void mfc_mt_wait_all(const int tag){
    const unsigned int mask = 1 << tag;
    unsigned int ret;
    do {
        yield();
        mfc_write_tag_mask(mask);
        mfc_write_tag_update(MFC_TAG_UPDATE_IMMEDIATE);
        ret = mfc_read_tag_status();
    } while (unlikely((ret & mask)) == 0) ;
}
```

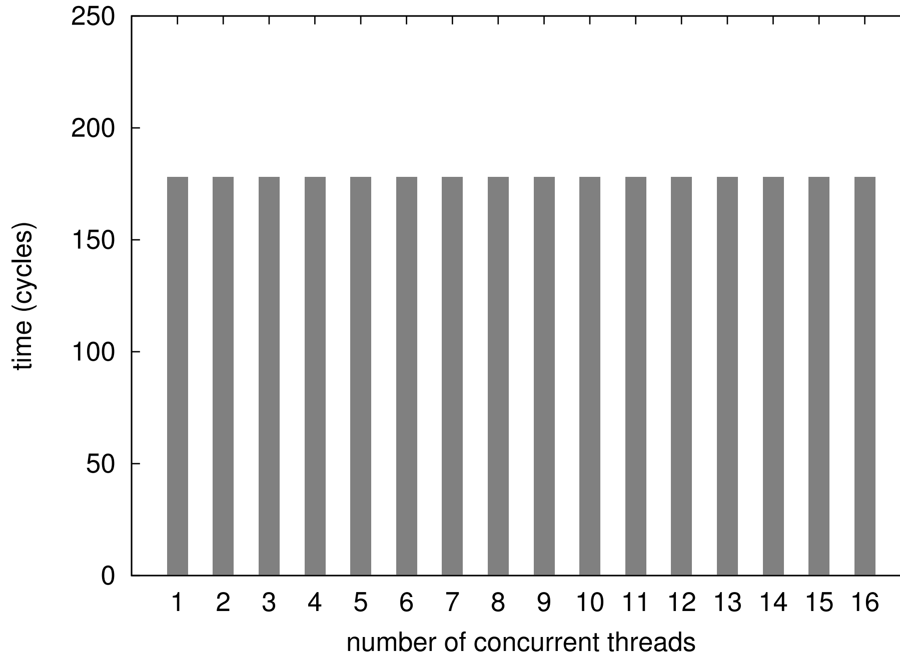


Figure 3: Context Switch Overhead

# 5 Evaluation

In this section we have evaluated the context switch overhead of our threading library. All the experiments have been conducted on Cell/B.E. machines available at the Barcelona Supercomputer Center. They are QS20 blades powered with two Cell processors clocked at 3.2 GHz with 1 GB of RAM memory. They have the IBM Cell SDK 3.0 installed. All experiments have been ran within the default Linux (version 2.6.24.7-92.fc8) execution environment, with the default virtual page size of 4096 bytes. All experiments have been executed several times and we have verified that the standard deviation of the experiments is very low.

Figure 3 shows the context switch overhead of our threading library. The x–axis refers to the number of concurrent threads from 1 to 16 and the y–axis measure the context switch overhead in processor cycles. As we can observe the overhead of a context switch –around 180 cycles– is independent of the number of running threads. The source code used to obtain this numbers are listed in Listing 11. To measure with precision the overhead of the *yield()* function we first measure the time that takes to execute $n$ times an arbitriary piece of code, which will be our reference time. Then we run the same code but extended with a call to the *yield()* function. Finally we substract to the obtained time the time of the original code, so we get the cost of the *yield()* function. We repeat this process with the 16 thread configurations, so each thread executes $n$ / *num_threads* iterations.

Listing 11: Context Switch Overhead Test

```
#define KILO                (1024LU)
#define MEGA                (KILO * KILO)
#define DECR_COUNT          (0xFFFFFFFF)
#define TICKS_PER_SECOND    (14318000LLU)
#define CYCLES_PER_SECOND   (3200LLU*1000000)
#define STACK_SIZE          (4096)
#define NUM_ITERS           (16*MEGA)

typedef int (*bench_func) (unsigned long long);

int benchmark(unsigned long long num_iters){
    unsigned long long i;
    unsigned long long sum=0;
    for(i=0; i<num_iters; i++){
        sum += i;
    }
    return sum;
}

int benchmark_mt(unsigned long long num_iters){
    unsigned long long i;
    unsigned long long sum=0;
    for(i=0; i<num_iters; i++){
        sum += i;
        yield();
    }
    return sum;
}

int main(unsigned long long id,
         unsigned long long a,
         unsigned long long b) {


    unsigned int i;
    int ids[MAX_NUM_THREADS] align16;
    char stack[MAX_NUM_THREADS][STACK_SIZE] align16;

    bench_func funcs[2] = {benchmark, benchmark_mt};

    // Start decrementer
    spu_writech( SPU_WrDec, DECR_COUNT);
/*      Single threaded reference execution      */
    unsigned long long end_time;
    unsigned long long start_time = spu_readch( SPU_RdDec );
    funcs[0](NUM_ITERS);
    end_time = spu_readch( SPU_RdDec );
    unsigned long long reference_ticks = start_time-end_time;
    printf("Reference␣time:␣%llu␣ticks\n", reference_ticks);
/******************************************** */
    unsigned long long ticks[MAX_NUM_THREADS+1];
    ticks[0] = reference_ticks;


/*  Multithreaded threaded  execution           */
    unsigned int num_threads;
    for(num_threads=1; num_threads<=MAX_NUM_THREADS; num_threads++){

        unsigned long long num_iters = NUM_ITERS/num_threads;
        start_time = spu_readch( SPU_RdDec );
        for(i=0; i<num_threads-1; i++){
            run_thread(num_iters, a, b, funcs[1], (void *)stack[i], STACK_SIZE, &ids[i]);
        }
        // All the threads are ready
        funcs[1](num_iters);

       for(i=0; i<num_threads-1; i++){
            int th_ret, ret;
            ret = wait_for(ids[i], &th_ret);
        }
        end_time = spu_readch( SPU_RdDec );
        ticks[num_threads] = start_time-end_time;
    }
/******************************************** */
    unsigned long long iters = NUM_ITERS;
    for(num_threads=1; num_threads<=MAX_NUM_THREADS; num_threads++){
        unsigned long long overhead_ticks = ticks[num_threads]-reference_ticks;
        unsigned long long cycles = (CYCLES_PER_SECOND*overhead_ticks) /
                                    (TICKS_PER_SECOND*iters);
        printf("Context␣switch␣overhead␣for␣%d␣threads:␣%llu␣cycles\n", num_threads, cycles);
    }

    return 0;
```

```
}
```

## 6 Conclusions and future work

In this paper we have presented our CellMT threading library. This library provides a cooperative user–level threading model that can be used to run multit–threaded applications inside the SPUs of the Cell/B.E. The main goal of this library is to provide a natural way to overlap DMA transfers times and computation times, without increasing the code complexity. With the help of the CellMT library we can write efficient programs that do not requiere the use of double buffering or multi buffering techniques. We have compared the multi–threaded approach with the double buffering approach with an ilustrative example. Finally we have measured the context switch overhead of the current implementation that is around 180 processor cycles. In the future we pla to further investigate the suitability and feasibility of our threading library for complex applications with irregular memory access. We will also try to further reduce the current context switch overhead to improve the performance of our library for a wide range of applications.

## References

[1] Mohamed F. Ahmed, Reda A. Ammar, and Sanguthevar Rajasekaran. Spenk: adding another level of parallelism on the cell broadband engine. In *IFMT '08: Proceedings of the 1st international forum on Next-generation multicore/manycore technologies*, pages 1–10, New York, NY, USA, 2008. ACM.

[2] Thomas Chen, Ram Raghavan, Jason Dale, and Eiji Iwata. Cell Broadband Engine Architecture and its first implementation. *IBM DeveloperWorks*, November 2005.

[3] Tong Chen, Zehra Sura, Kathryn O'Brien, and John K. O'Brien. Optimizing the Use of Static Buffers for DMA on a CELL Chip. In *LCPC*, pages 314–329. Springer Berlin / Heidelberg, 2006.

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

[5] M. Mustafa Rafique, Ali R. Butt, and Dimitrios S. Nikolopoulos. Dma-based prefetching for i/o-intensive workloads on the cell architecture. In *CF '08: Proceedings of the 2008 conference on Computing frontiers*, pages 23–32, New York, NY, USA, 2008. ACM.

[6] Wm Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. Technical report, University of Virginia, Charlottesville, VA, USA, 1994.