# Optimizing IRS Benchmark for IBM Cell Processors

(Parallel Systems CSC 548)

**Balasubramanya Bhat**

**Suraj Kasi Satyanarayana**

## Table of Contents

Introduction ..... 3
The IRS Benchmark ..... 3
The Cell processor ..... 3
Cell memory architecture ..... 4
Project Goals ..... 5
Implementation Details ..... 5
1. Porting of IRS benchmark code into IBM Cell processors ..... 5
2. Run profiling on the IRS Benchmark ..... 5
3. Non-optimized IRS Benchmark Runtime Analysis on IBM Cell Processor ..... 5
4. Optimizations performed to the IRS Benchmark on IBM Cell Processor ..... 7
4.1. Changes to the MatrixSolve ..... 7
4.2. New SPU Program ..... 7
4.3. Prefetching the data over DMA: ..... 8
4.4. Using Asynchronous communication to queue all DMA requests ..... 9
4.5. Communication Computation Overlap: ..... 9
4.6. Reduction in the number of operations: ..... 9
4.7. Keeping the SPE context alive: ..... 9
4.8. Other optimizations: ..... 10
5. Performance Results: ..... 10
6. Timeline for the project ..... 11

# Introduction

IBM cell multiprocessor [5] as become a new direction for future systems. Chip multiprocessors are touted as an approach to deliver increased performance, now the adoption is increasing due to diminishing returns of the uni-processor designs.

The goal of this project is to select and implement a benchmark designed for evaluating the parallel systems on the IBM Cell processor using the multi-processing power of the same to reduce its execution time.

# The IRS Benchmark

We selected the IRS benchmark [1][2][3] from UC LLNL for our project. The name IRS stands for Implicit Radiation Solver. IRS solves a diffusion equation on a three-dimensional, block structured mesh. The single CPU instruction mix for this benchmark is roughly 40%load/store, 18% floating point, 31% fixed-point, and 11% branch.

# The Cell processor

The Cell engine is designed to address the diminishing returns available from a single core. This exploits parallelism at all levels, data level parallelism, instruction level parallelism, thread level parallelism and compute transfer parallelism. Key feature is to provide parallelism degrees at each level to ensure good utilization.

Data level parallelization offers an efficient method to increase the amount of computation. Power consumption is reduced due to sharing the execution between both scalar and SIMD computation elimination control and data path duplication. This also reduces the power requirement. When there is data parallelism the power spent is on the computation hence the power to performance ration of such architectures is high.

Instruction level parallelism avoids power inefficiency, because no execution units are added for performance increase. ILP increases average memory latency by concurrently servicing multiple outstanding cache misses. A processor continues execution across a cache miss to encounter clusters of cache misses. This allows to concurrently initiate the cache reload for several cache misses.

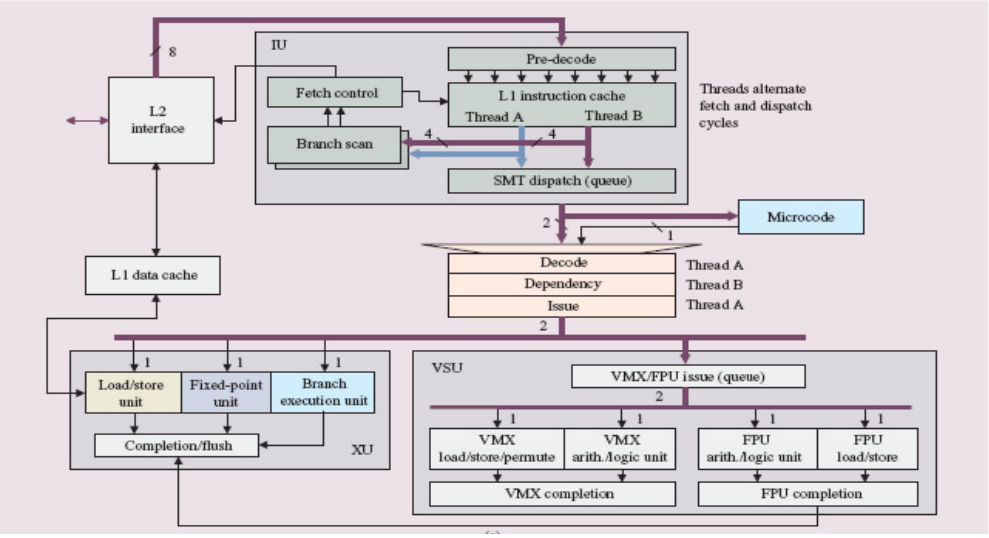
Thread level parallelism is support with a multithreaded PPE core and multiple SPE cores. This delivers a significant boost in performance it also provides a high power/performance efficiency. To exploit memory more efficiently transfer operations are inserted well before the computation is done, this ensures that the data is ready during execution.

## Cell memory architecture

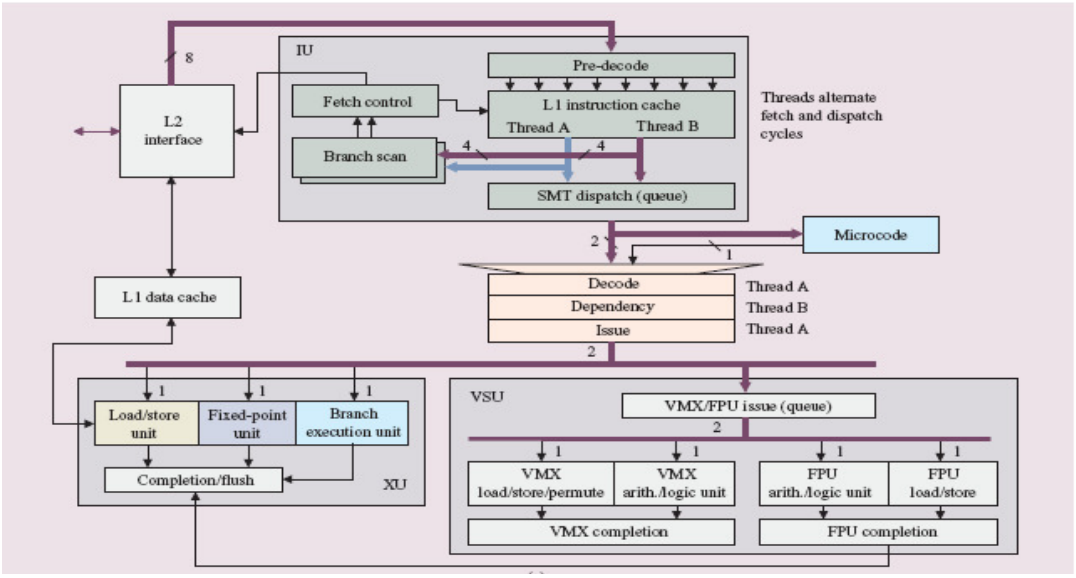
The key to exploiting memory bandwidth is to increase the number of simultaneous cache misses to reduce the average service time and the memory bandwidth. By putting multiple threads on a chip the latency of one core waiting for data can be covered by the other computing core.

Many data rich applications can predict block of data based on program structure, scheduling data transfer before data will be accessed by the program.

**The PPE architecture is as shown below.**




**The SPE architecture is as shown.**



# Project Goals

The goal of this project is to run the IRS (Implicit Radiation Solver) parallel benchmark [1][2][3] on multiple IBM cell processor [5] nodes using MPI and accelerate the runtime of the benchmark on each node using the multiprocessing capability available within the IBM cell processor (PPE and SPEs).

# Implementation Details

## 1. Porting of IRS benchmark code into IBM Cell processors

We downloaded the IRS benchmark version 1.0 from the LLNL website. We modified different Makefiles and some perl scripts to change the compilation to use powerpc compilers, compiler / linker options, include / library paths etc. We built the working IRS benchmark which could run on the PSXX machines.

Made Changes to the build system to incorporate following functinoalites:

1. The makefile was changed to run the irs application on the powerpc architecture. This change uses the gcc compiler to compile the irs application.
2. We later changed the makefile to use mpicc compiler which internally invokes ppu-gcc to compile the ppu and the spu code.
3. A new directory was created to write the spu code, and a makefile to compile the spu code is written in that.
4. During compilation and the linking phase we had to change the library paths to use the cell sdk libraries and other compiler options.
5. A new make file is created to compile, link and embed the SPU program into the final IRS executable.

## 2. Run profiling on the IRS Benchmark

We also enabled MPI and the gprof [6] instrumentation on the IRS and got the execution profile of the benchmark on IBM Cell processor. We show the performance of the un-optimized IRS Benchmark in later sections.

## 3. Non-optimized IRS Benchmark Runtime Analysis on IBM Cell Processor

After the IRS benchmark is ported to the IBM Cell processor, we ran the same using ‘mpirun –np 2’, with gprof profiling enabled. Following command is used to run the benchmark using MPI on IBM cell processor:

```
mpirun -np 2 ./irs /home/skasisa/irs.1.0/decks/zrad3d -child_io_off -k 00008MPI -
def NDOMS=8
```

Following are the top 10 functions in terms of the execution times on the IBM cell processor as recorded by gprof.

```
Each sample counts as 0.01 seconds.
%     cumulative  self        self        total
time  seconds     seconds     calls       s/call      s/call      name
48.70 20.96       20.96       3304        0.01        0.01        rmatmult3
19.73 29.45       8.49        6           1.42        5.54        MatrixSolveDriver
4.00  31.17       1.72        3280        0.00        0.00        icdot
3.67  32.75       1.58        24          0.07        0.07        DiffCoef
3.14  34.10       1.35        76          0.02        0.02        volcal3d
2.32  35.10       1.00        4908        0.00        0.00        norml2
1.23  35.63       0.53        48          0.01        0.01        newMatrix
1.18  36.14        0.51       8284        0.00         0.00       setpz1
```

For a complete list of log files and commands used for running IRS, please see the following webpage

http://sites.google.com/site/irsoncell/deliverables/on-ibm-cell

Based on the above timing details obtained from the gprof the functions were analyzed for parallelism. The analysis is as given below.

MatrixSolveDriver: The call graph for this function is as shown below.

```
[6]      79.5   14.61   36.50       6         MatrixSolveDriver [6]
                0.00   27.60    6608/6608        rmatmult [7]
                3.21    0.00    6560/6560        icdot [12]
                2.30    0.02    9816/9816        norml2 [14]
                1.37    0.03    3328/3436        rbndcom [18]
                1.08    0.00   16328/16568       setpz1 [22]
                0.87    0.00   16424/16664       setpz2 [24]
                0.00    0.03      48/48          cblkbc [102]
                0.00    0.00      60/19129       FunctionTimer_finalize [103]
                0.00    0.00      60/19129       FunctionTimer_initialize [116]
                0.00    0.00      48/152         TimeStepControl_find [217]
```

It is very clear that the majority of this function time corresponds to the rmatmult function that is shown in the gprof output.

1. The function rmatmult calls rmatmult3 or rmatmult2 based on the dimensions and this functions are responsible for matrix multiplication. In the above case rmatmult3 is called because the test ran with a dimension greater than 2. This function is called 6608 times and hence it is a potential hot spot, by parallelizing this function we can get better performance.

2. icdot: Although called multiple times this cannot be parallelized because parameters that are passed on depends on the previous icdot calculation. Hence icdot is ruled out.

3. norml2: Even this one has dependencies between iterations hence cannot be used. There are loops inside the function that can be parallelized but the overhead of data transfer might supersede the benefits of parallelism.

4. rbndcom: Used for MPI communication hence ignored.

The 2 functions we chose account for 50% of the total execution time, hence parallelizing these would lead to better performance.

## 4. Optimizations performed to the IRS Benchmark on IBM Cell Processor

### 4.1. Changes to the MatrixSolve

MatrixSolve is the function is the entry point for rmatmult which is called by MatrixSolveDriver. MatrixSolve acts as out ppu program. Inside MatricSolve, the rmatmult function is called in a for loop as shown below.

```
for ( iblk = 0 ; iblk < my_nblk ; iblk++ ) {
        rmatmult( &domains[iblk], &rblk[iblk], x[iblk], r[iblk] ) ;
}
```

Here each calculation of rmatmult is independent of each other, i.e the memory accesses don't overlap hence they can be called in parallel. The value of my_nblk, that determines the number of executions of the for loop does not exceed value 4 hence it can be easily be parallelized in spu.
We create my_nblk number of spu thread and load the rmatmult context into it. These get executed parallely. So each iteration of rmatmult is executed on different SPUs.

### 4.2. New SPU Program

The rmatmult3 function which was implemented sequentially in the original IRS benchmark is now rewritten to run on the SPU. We faced several challenges to run on the matrix multiplication program on the SPU. Following is a list of some of the challenges faced and the solutions we used.

#### 4.2.1. The complex structure of matrix multiplication.

The program structure of matrix multiplication in IRS is extremely complex. It uses about 60 different arrays to compute the final result of matrix multiplication. Each array is accessed in a non-sequential manner with the indices in the range 0-24000. Each element in the array is a double. The program running on the SPU cannot access any PPU memory without DMA. This coupled with the alignment problems discussed in the later section poses a tremendous challenge to perform the calculations within the SPE.

**Solution:**
See solutions to other problems. The same solution applies hare.

#### 4.2.2. Limited memory available in each SPUs

Cell has only 256Kb of internal memory. It is not possible to allocate memory for all the array members that are involved in the operation as the total amount of memory required sum up to several mega bytes, hence the option of copying the complete data on to SPU and then executing is ruled out.

**Solution:**
As we cannot completely copy the data required for computation into the SPU memory we have to do this in stages. We bring in the required data in/out of SPUs only when it is being used within SPU.

#### 4.2.3. Memory Alignment Requirements for DMA

The DMA engine in SPU requires the source and destinations used in the transfers to be 16 byte aligned. The IRS matrix multiplication accesses more than 60 huge arrays and in non-contiguous manner which are dynamically allocated. It is virtually impossible to ensure that all memory accesses are 16 byte aligned. This problem was one of the major hurdles we faced during the project.

**Solution:**
Each DMA request is validated for the alignment of the source and destination addresses. If they are not aligned, we find out the preceding and succeeding 16 byte boundary and we read and write to this range instead of only the requested range.
Logic:

```
If (req_address is not 16 byte aligned)
{
        find the lower bound address < req_address which is 16 byte aligned
        find the upper bound address > req_address which is 16 byte aligned
        read the data b/w upper and lower bounds
        change the requested bytes in the fetched data
        write the entire block (lower – upper bound) back to destination
}
else
{
        write the requested block onto destination
}
```

### 4.3. Prefetching the data over DMA:

As discussed before, we do the memory transfers small block by block instead of everything at the beginning. Before we enter the innermost iteration in the matrix multiplication, we fetch all the data required for that iteration. This means that the computation within the loop has to wait till the data is ready. The DMA would become major bottleneck and we won't get the performance improvement that is expected. To solve this problem we are using data pre-fetching. At each iteration, we pre-fetch the data for the next iteration in an asynchronous manner. The data required for the current iteration will be perfected during the previous iteration. As we will see later, this greatly enhances the performance.

### 4.4. Using Asynchronous communication to queue all DMA requests

At each iteration during matrix multiplication, we pre-fetch the data for the next iteration in an ***asynchronous*** manner to avoid blocking while performing the DMA.

### 4.5. Communication Computation Overlap:

At each iteration during matrix multiplication, we pre-fetch the data for the next iteration in an ***asynchronous*** manner and during this time we perform the computations for the current iteration. So DMA transfer time and computation times greatly overlap improving the performance.

### 4.6. Reduction in the number of operations:

Consider the below 'for' loop, here in the above for loop we can see that the value of i is calculated based on the the values of ii, jj and kk but the value of jj*jp and kk*kp don't change in every iteration for ii, hence it is an over head.

```
for ( kk = kmin ; kk < kmax ; kk++ ) {
        for ( jj = jmin ; jj < jmax ; jj++ ) {
                for ( ii = imin ; ii < imax ; ii++ ) {
        i = ii + jj * jp + kk * kp ;
                }
        }
}
```

This will be changed to :

```
for ( kk = kmin ; kk < kmax ; kk++ ) {
        kk_intermediate = kk * kp;
        for ( jj = jmin ; jj < jmax ; jj++ ) {
                jj_intermediate = jj * jp
                for ( ii = imin ; ii < imax ; ii++ ) {
                        i = ii + jj_intermediate + kk_intermediate.
                }
         }
}
```

Here, we have moved these additions that are performed in the inner most loop repeatedly, outside the loop. This actually will save large amount of additions that are repeatedly performed due to large iteration values inside the inner loop.

### 4.7. Keeping the SPE context alive:

As we can see from the gprof output, the matrix multiplication is called several thousand times

from various locations. Creating he SPU context each time and reloading the program is a big overhead. We locally cache the SPE contexts to reuse for future invocations.

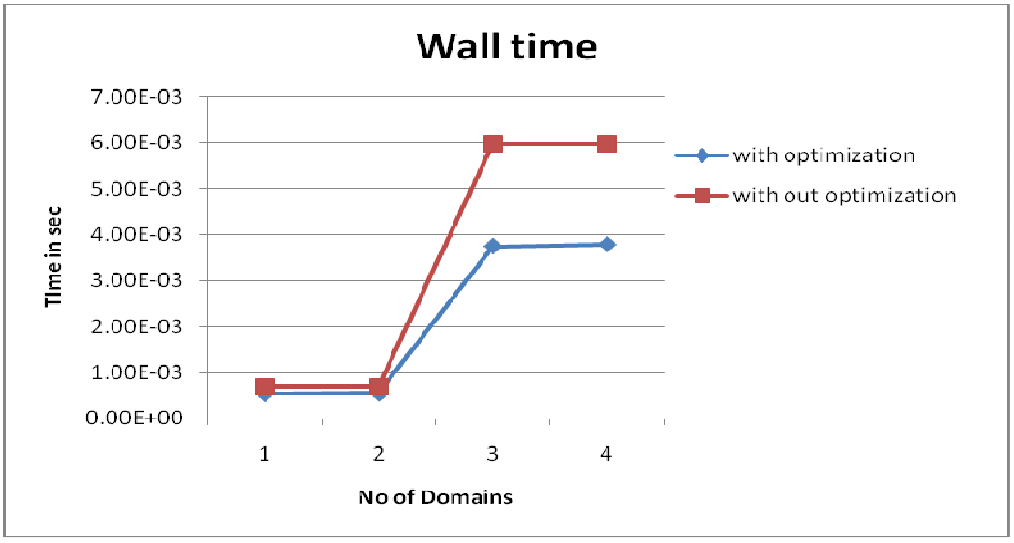
### 4.8. Other optimizations:

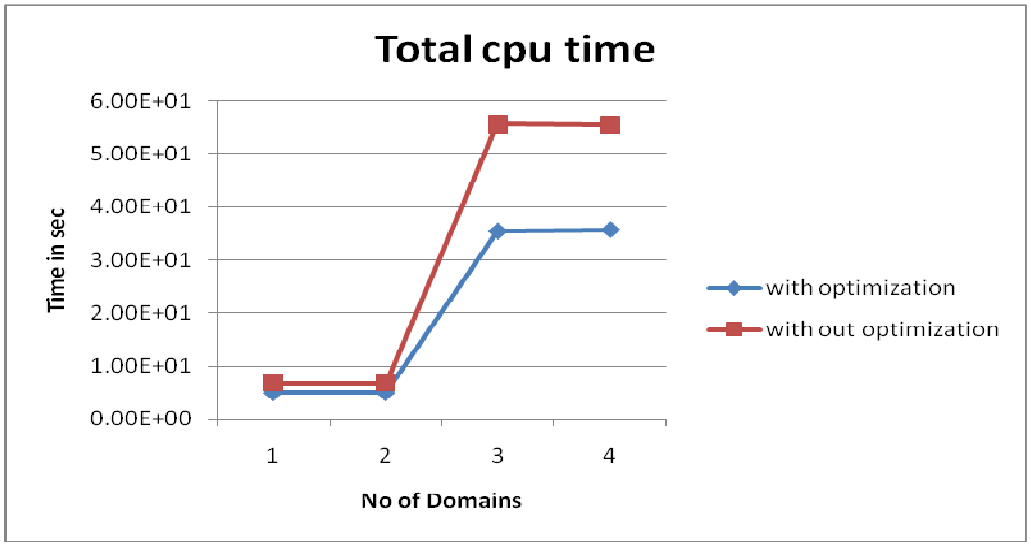
We also performed several alignment related optimizations which account for about 2% improvements.

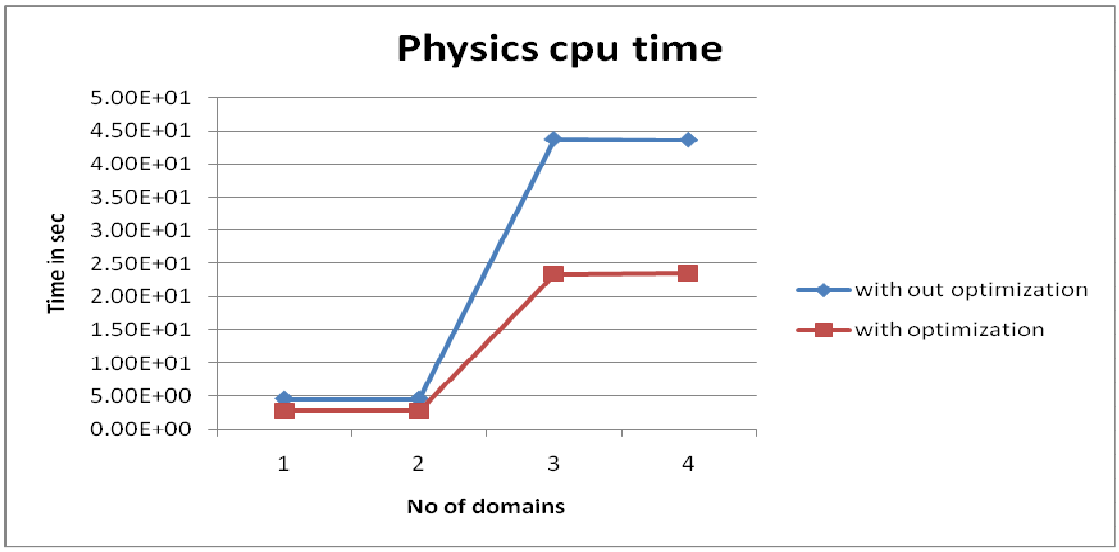
## 5. Performance Results:

We performed all the optimizations listed above and rerun the IRS benchmark on the Cell processor with different parameters. We clearly see that we achieve, more that 35% improvement in the overall execution time of the IRS benchmark.

The following graphs show the comparisons for the IRS benchmark on the IBM Cell processor, before and after the optimizations:






## 6. Open Problems

One of the open problems is that we have mainly concentrated on the rmatmult function and .
For example, the functions setpz1 and setpz2 are called multiple times inside MatrixSolveDriver as shown below.

```
for ( iblk = 0 ; iblk < my_nblk ; iblk++ ) {
        setpz1( p[iblk], 0.0, &domains[iblk] ) ;
        setpz1( t[iblk], 0.0, &domains[iblk] ) ;
        setpz2( t[iblk], 0.0, &domains[iblk] ) ;
}
```

The individual iterations are independent of each other. Hence they can be parallelized similar to rmatmult. This is left for future work.

The granularity of the DMA transfers can be increased to improve the performance. Even though we don't have an estimate of how much this would improve, we may experiment with the changes.

Restructure the IRS code more to use alignment buffers wherever possible to avoid wastage of DMA transfer bandwidth by over fetching buffers.

## 7. Timeline for the project

| Dates | Task | Status |
|---|---|---|
| Oct25 – Nov2 | Understanding the benchmark code | Completed |
| Nov3 – Nov9 | Porting the benchmark onto the IBM cell processors without any modifications to the code. | Completed |
| Nov10-Nov15 | Converting the hotspot functions into kernels and running it on the | Completed |

| | | |
|---|---|---|
| | cell processor. | |
| Nov16- Dec 2 | Testing and debugging the code. | Completed |
| Dec 3 – Dec 6 | Document the steps and write report. | Completed |

**Deliverables**

The deliverables for the project includes the modified source code for the IRS bench mark, project report, original and the improved timing metrics and graphs.

Please refer to the following website for all deliverables and log files etc.
http://sites.google.com/site/irsoncell/deliverables/on-ibm-cell

**References**

[1] https://asc.llnl.gov/computing_resources/purple/archive/benchmarks/irs/
[2] http://moss.csc.ncsu.edu/~mueller/cluster/cluster03/g1/irs.pdf
[3] https://asc.llnl.gov/computing_resources/purple/archive/benchmarks/irs/irs.readme.html
[4] http://moss.csc.ncsu.edu/~mueller/cluster/ps3/
[5] http://www-03.ibm.com/technology/cell/index.html