Exploring Software Cache on the Cell BE Processor

Ganapathy Senthil, Sasikanth Gudla*, Pallav Kumar Baruah**
Sri Sathya Sai University
Prasanthi Nilayam - 515134, Andhra Pradesh, India
{*sasikanthgudla,** baruahpk}@gmail.com

Abstract

Software Cache promises to increase programmability and performance in certain applications such as those with irregular memory references on multi-core architectures like the cell processor where on chip memory is a precious resource. We discuss the need for software cache, design and implementation of a simple software cache on the Cell Broadband Engine. We present the performance of a histogram application using the software cache. We propose a static analysis tool that takes memory references generated by a given application and gives the optimal cache parameters that can be used to configure the software cache for the application to run on the Cell Broadband Engine. We also present the results of analysis done with this tool for the trace generated by a heap sort application

1 Introduction

Multi-core architectures are here to stay. Heterogeneous architectures like the Cell Processor have constraints in the amount of on chip memory available for them. The SPUs in the Cell processor are equipped with 256 KB Local storage memory. The programmer generally does a DMA for the SPUs to access memory. Normally cyclic buffering is used to hide the DMA latency thus overlapping computation with communication. However in applications such as those with irregular memory references this may become very difficult. Software Cache can be useful both in terms of increased programmability and increased performance in such scenarios.

1.1 Need for Software cache

Consider the following code to be executed by the SPU

```
for(i = 0; i < 100000; ++i)
        a[i] = b[i] + c[i] * d[f(i)];
```

In this case, b and c can be prefetched, but the access pattern in d cannot be predicted. d[f(i)] must be fetched on each iteration, resulting in a huge slow-down of the loop. Every access to d requires a high-latency access to main memory. While the SPEs have no hardware cache, it is possible to implement a small software cache by explicitly referring to it. For instance, the above example can be implemented as follows:

```
for(i = 0; i < 100000; ++i) {
        t = cache_lookup(d[f(i)]);
        a[i] = b[i] + c[i] * t;
}
```

A sample implementation of cache_lookup might look like this:

```
inlinevector cache _lookup(addr) {
    /* fetch the value if we haven't got it already */
if (cache_directory[addr&key_mask] != (addr&tag_mask))
    miss_handler(addr)
    /* return the value */
return cache_data[addr&key_mask][addr&offset_mask];
}
```

*student author: Sasikanth Gudla

1.2 Issues and Challenges

The software cache functions add some computation overhead compared to ordinary DMA data transfers and therefore ordinary DMA transfer is preferable in case the data access pattern is sequential [1], [4]. The two challenges in using the Software Cache effectively would be:

- The amount of space occupied by the software cache should be as minimum as possible as the local storage is a precious resource for the SPU
- For the Software Cache to be of useful in terms of performance, the cycles lost in lookup and other bookkeeping work should be more than compensated by the cycles gained by ensuring that required cache line is almost all the time present in the cache

1.3 Advantages

Software Cache has the following advantages:

- Better performance in some applications owing to the principle of locality of reference by saving redundant data transfers if the corresponding data is already in the Local Storage.
- Reconfigurable Topology and behavior which implies that it can be easily optimized to match data access patterns (unlike most hardware caches).

2 Design and Implementation of our Software Cache

We implement the software cache, using CELL SDK 2.1, in essentially four modules:

1. cache init:
   This module receives various user parameters like cache size, cache line size, and set size, computes bits required for line offset, set index and comparand.
2. cache lookup:
   This module takes the effective as an argument and extracts tag, line offset, set index. It uses these parameters to index into the cache to find a match. It returns the corresponding LS address if it finds a match else calls the cache miss handler.
3. cache miss handler:
   This module first selects a line to be evicted using the Round robin strategy, queues a DMA to flush the line if the dirty and valid bits are set and queues another DMA to fetch the required line. Fenced DMA is used to maintain consistency of the data.
4. cache flush:
   This module runs in a loop through the cache data array and queues a DMA to the flush the cache line to main memory if , if the valid and the dirty bits are set.

2.1 Software cache in Cell SDK 3.0

In this section we briefly discuss the software cache provided by the IBM SDK 3.0 The IBM Cell SDK 3.0 provides a software cache as a macro implemented library which can be used by application programmers in two modes, a synchronous mode and an asynchronous mode[1]. The software cache can be configured based on Associativity, Access mode (Read-only or read-write), Cache line size, Number of lines, Data type.

The synchronous mode (or safe mode) provides the programmer with a set of functions to access data simply by using the datas effective address. The software cache library performs the data transfer between the LS and the main memory transparently to the programmer and manages the data that is already in the LS.

The asynchronous interface (or unsafe mode) enables the programmer to hide the memory access latency by overlapping data transfer and computation. This mode provides a more

efficient means of accessing the LS compared to the safe mode. The software cache provides functions to map effective addresses to the LS addresses. The programmer should use those LS addresses later to access the data, unlike in safe mode where the effective addresses are used.

There is also a provision to define multiple caches, each configured differently to suit the needs of the programmer.

3 Performances and Results

We used the Histogram application to study the performance of our implementation of the software cache. A given gray scale image has pixels each assigned a gray level from 0-255. The histogram of an image is an array of 256 elements where each element gives the frequency of occurrence of the gray value in the image. In order to construct the histogram the algorithm traverses the image linearly and for each pixel increments the value of the element in the histogram array corresponding to the gray level of the pixel. Figure 1 shows that hit ratio increases almost linearly as cache size increases since histogram calculation is localized and also large cache size leads to less cache line replacements.

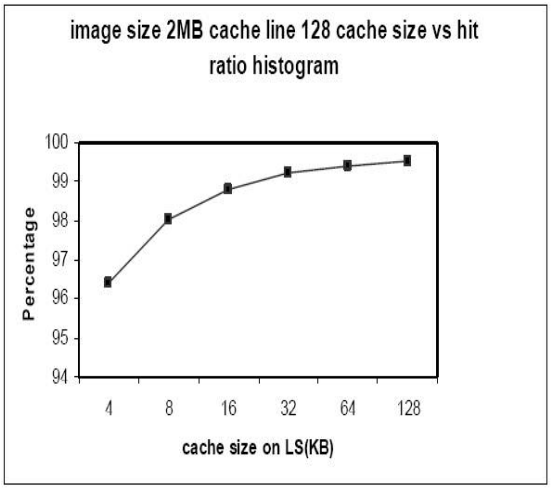

Figure 1: **Cache Hit ratio of Histogram Application with various cache sizes**

Figure 2 reiterates the point that as cache size increases time taken by application decreases, but ideal cache size depends on the size of application code because Local Storage is a very precious resource for the SPE.

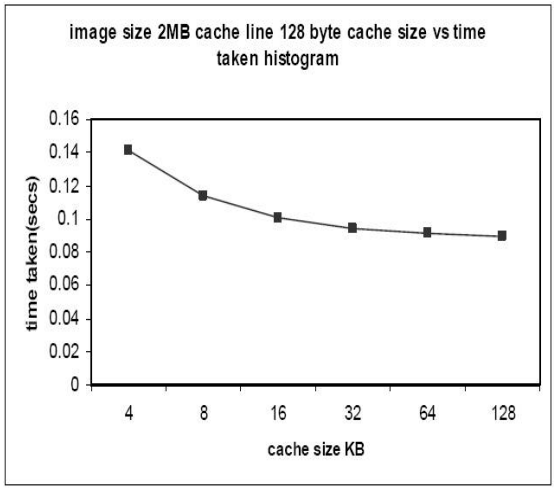

Figure 2: **Timing of Histogram Application with different Cache sizes**

4 Analysis Tool

As mentioned in the introduction the software cache can be configured based on parameters like line size, cache size, associativity, data type and access mode. We propose an utility tool that determines the optimal cache parameters that should be used to configure the cache for better performance for a given application. We first generate the memory access trace for an application by adding print statements in the code and writing the references on to a file. This file forms the input to the utility tool which would then evaluate the optimal cache parameters based on various factors like:

- The frequency of reference of a given cache line
- The access stride for a cache line which is the number of lines referenced between two successive references of a given cache line

- A comparative analysis of these metrics for different configurations of the cache thus enabling the user to make a best possible choice for a given application and a given input size of the application

We intend to take various applications, obtain the memory access traces and run the utility tool to obtain quantitative results in terms of performance gains with the insights provided by the tool. The tool could also provide valuable information regarding the memory access patterns of various applications.
Here we present the results of analysis done with the trace of heap sort application provided by the IBM SDK 3.0, which sorts a set of floating point numbers using the heap sort algorithm. The input could be given from a file or generated randomly by the heap sort application itself. We have obtained traces for various input sizes where input size corresponds to the number of floating point numbers being sorted. We then used the analysis tool to obtain graphs corresponding to frequency of references vs Line Number in the Effective Address space. The Line Number in the Effective Address space of a given memory reference corresponds to the Effective Address divided by the intended cache line size. The four graphs correspond to intended cache line sizes of 128 Bytes, 256 Bytes, 512 Bytes and 1024 Bytes. The graphs show that only the first few lines are referenced very frequently and there is a sharp decline in the number of references. Also, a longer Cache line size results in better performance for this heap sort application.

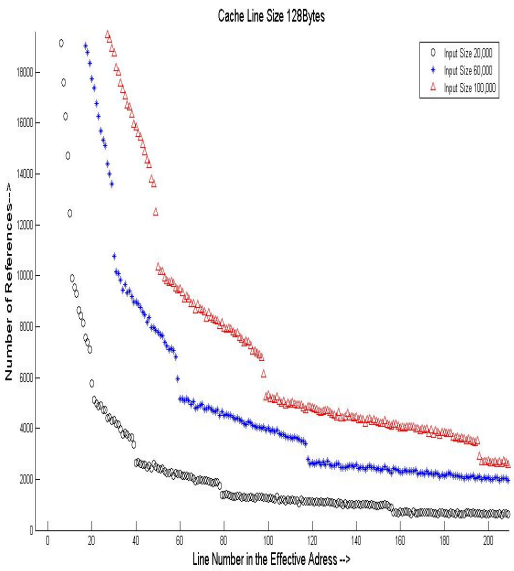

Figure 3: **Plot of No.of References vs Line No. for the Heap sort application, Cache Line Size is 128 Bytes**

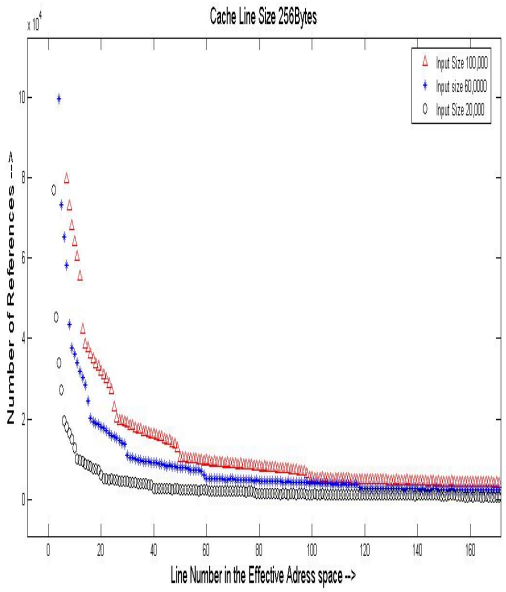

Figure 4: **Plot of No.of References vs Line No. for the Heap sort application, Cache Line Size is 256 Bytes**

5 Conclusions and Future Work

The analysis tool proposed is a precursor to a novel implementation of the software cache. Software cache has an advantage over hardware cache that it is highly reconfigurable, flexible in implementation. We intend to extend the idea of configuring the cache based on the memory access patterns to implement a cache based on

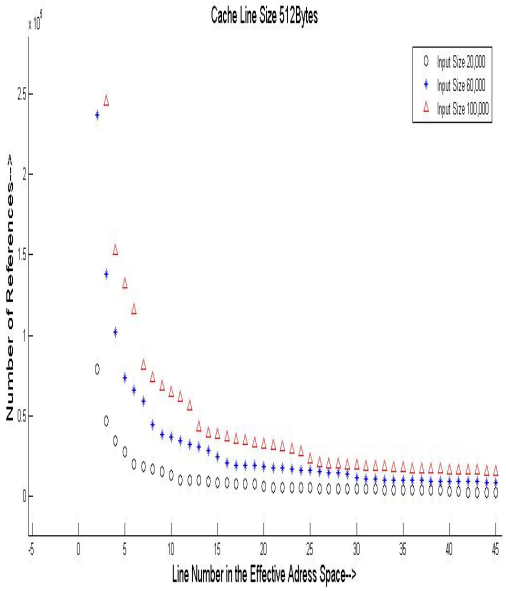

Figure 5: **Plot of No.of References vs Line No. for the Heap sort application, Cache Line Size is 512 Bytes**

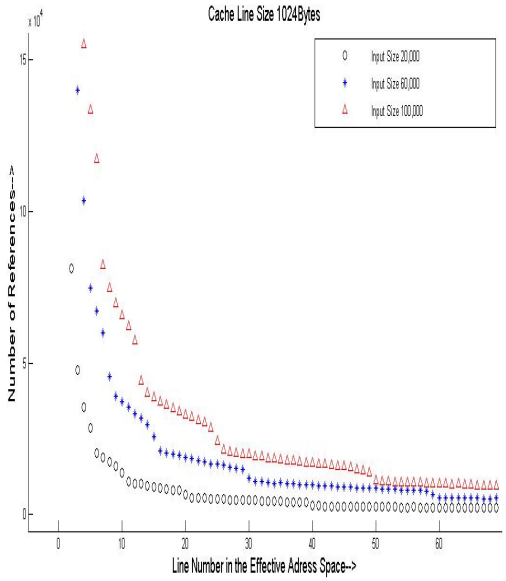

Figure 6: **Plot of No.of References vs Line No. for the Heap sort application, Cache Line Size is 1024 Bytes**

the usage of the cache lines. Since, we are not restricted by hardware limitations and we need not stick to the traditional view of cache. Instead we can implement the cache purely based on access patterns and thereby substantially improve the cache performance. We also intend to study various applications' performance based on our implementation of the Software Cache for the Cell Processor.

References

[1] Programming the Cell Broadband Engine Architecture Examples and Best Practices: *www.redbooks.ibm.com/redbooks/pdfs/sg247575.pdf*

[2] Jairo Balart , Marc Gonzalez, Xavier Martorel, Eduard Ayguade, Zehra Sura, Tong Chen, Tao Zhang, Kevin Obrien, Kathryn Obrien: A Novel Asynchronous Software Cache Implementation for the Cell-BE Processor: *Proceedings of the 2007 Workshop on Languages and compilers for Parallel Computing, pages 125-140, 2007*

[3] Jie Tao, Siegfried Schloissnig, and Wolfgang Karl: Analysis of Spatial and Temporal Locality in Data Accesses: *International Conference on Computational Science ,pages 502-509, 2006*

[4] Tong Chen, Tao Zhang, Zehra Sura, Marc Gonzalez Tallada, Kathryn OBrien, Kevin OBrien: Prefetching Irregular References for Software Cache on Cell: *CGO, pages 155-164, 2008*