Cache Effect Analysis of Different Types of Bit Reversal Algorithms

Jose E. Torres Berrocal
Advisor: Dr. Jaime Seguel

Electrical & Computer Engineer Department
University of Puerto Rico, Mayagüez Campus
Mayagüez Puerto Rico 00681-5000
jetorres@ece.uprm.edu

Abstract

The Recursive, Cyclic, Jeong, Elster and a type of cache aware bit reversal algorithms were studied in terms of cache response on one Intel and Sparc architecture.

1. Introduction

In this paper I will focused on describe the actual project, and assume that is already known the purpose of the Bit Reversal algorithms, and the way a Cache works, but a brief description on how the cache is involved is presented. I will mention the objectives of the project and explain how it was performed, then will present the results obtain, mostly in graphical form. Now I start presenting the objectives of the project.

2. Objectives

Since one part involved in the processing of a program is the memory, and it is intervene by the Cache, even though is used mostly to improve performance it sometimes affect the execution of the programs adversely, caused by the fact that data references are usually not cache-oblivious [Prokop99] and misses start to grow as the problem size grows. In this project is analyze experimentally the effect caused by the Cache on Bit Reversal algorithms, will obtain by regression a numerical relation of cache misses in function of size (N) of the problem, and compare the cache response for different types of Caches, for the algorithms studied. This objectives were accomplished by running some Bit Reversal algorithms for the problem sizes from $2^1$ up to $2^{30}$ on two Cache types, with the following methodology.

3. Methodology

As mentioned before some Bit Reversal algorithms were studied, these are the following 5 different algorithms mentioned: Recursive [Cormen90], Cyclic which is an alternative for the Recursive with the temporary array eliminated using a cyclic method, Elster [Elster89], Jeong [Jeong92] which is another recursive algorithm with a special cyclic method, and Aware which is an alternative for Jeong with a cache segment aware method. The Cache types were simulated using a brotherly run program that receives address references parallel from the Bit reversal programs, and the cache modeled were an UltraSparc version [Cockcroft95] and an Intel version [Intel]. The UltraSparc has a 16Kb size, a 32 bytes cache line, 32 bit addresses handling, with a Direct Map scheme which by default uses a FIFO replacement policy. The Intel has a 256Kb size, also has a 32 bytes cache line and 32 bit addresses handling, but with a 8-way Set Associativity using a LRU replacement policy. The cache simulator program do the calculations describe on the Figure 1.

![Figure 1. Cache Simulator Operation](image)

For the simulator to work it needs to receive the reference addresses, in order to do that there is an address reference acquisition process. This process is done invasively on the Bit Reversal codes. Using the C++ language capability to do dereference we were able to obtain the address references. In the system the addresses are 32 bit long, so we declare address variables unsigned long (which is the biggest integer type provided by the language, and gives enough space to hold address up to 32 bits), then the bytes size of the data variable value is determine, which is machine dependent (ex. dval = sizeof(double)), and
we take the address references of the bit reversal array only. I assume that values in an array are contiguous starting on a base address (base = &array[0], add = base + ind*dval). The address obtained is sent to the std out of the Bit Reversal program were the simulator is waiting for it on its stdin (cout << add). Also is assume that reads are done before writes, so if for example we have a code line like “array[1] = array[2] + array[1]” then we write after the code line these lines:

“cout << base + 2*dval; cout << base + 1*dval;”
“cout << base + 1*dval;”

Note the Figure 2 for a specific example.

```c
void elster(int *B, int size){
    int c, i = 2, d = 1, k;
    unsigned long Bad = (unsigned long) B;
    B[0] = 0; cout << endl << Bad;
    B[1] = c = (size >> 1); cout << endl << Bad + size;
    while((c = c >> 1) >= 1){
        for(k = 0 ; k < (d << 1); i++, k++){
            B[i] = B[k] | c;
            cout << endl << Bad + size*i;k;
            cout << endl << Bad + size*i;
        }
        d = k;
    }
}
```

Figure 2. Code Example (Elster)

With this methodology I got the following results, described on the next section.

4. Results

First I present the results obtain for the Cache UltraSparc version, then for the Intel version, and finally a comparison of both cache types is presented. The regression analysis were performed using MS Excel 5.0

4.1 Results - Cache Sparc

The following Table 1 summarizes the regression analysis for the five Bit Reversal algorithms with the Sparc cache simulation, and the Figure 3 shows a comparison of the curves drawn from the data output of the cache simulator.

### Table 1. Bit Reversal on Sparc

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Regression function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elster</td>
<td>$y = 3.3818x - 3091.1$, $R^2 = 1$</td>
</tr>
<tr>
<td>Recursive</td>
<td>$y = 3E-06x^2 + 11.092x - 38645$, $R^2 = 0.9996$</td>
</tr>
<tr>
<td>Jeong</td>
<td>$y = 8E-09x^2 + 0.6461x - 43.295$, $R^2 = 1$</td>
</tr>
<tr>
<td>Cyclic</td>
<td>$y = 1E-05x^2 + 21.422x - 98126$, $R^2 = 0.9994$</td>
</tr>
<tr>
<td>Aware</td>
<td>$y = 2E-08x^2 + 2.2636x - 2078.8$, $R^2 = 1$</td>
</tr>
</tbody>
</table>

![Figure 3. Misses with size $N$ on Sparc](image)

Is shown on Table 1 that the cache misses grow linearly with size on all algorithms, and from Figure 3, is noted the order of growth of each algorithm. Notice that even that Aware knows the cache segment it has more misses that the original Jeong version, and the Cyclic is close to have twice the misses than the original Recursive version. Elster is much less that the original Recursive but has more misses than Jeong.

On the previous figure is shown a plot for the same data series as the Figure 3 but is scale to an X-axis of $k$, where is noticed that there is a common point were all algorithms start to grow misses. Next is shown a plot on the Figure 5 were the Total reference of address is compare to all algorithms instead of the misses. It shows the same pattern, with the same order of growth, so the misses are directly proportional to the total count of references.
Last on this section there is a comparison with the
miss ratio instead of the total misses, in this
comparison is noted the real effect of cache on the
algorithms, and is shown on the Figure 6.

In the Figure 6 is noted that in average there is a size
problem were all the algorithms grow rapidly, also
that there is a minimum rate for each and there is a
maximum rate for Aware, Jeong, and Elster. Also in
this figure is noted the order in which the algorithms
are affected by the cache. At the beginning the
Aware, Jeong and Elster experience the same effect,
the Cyclic and Recursive their own, but at the point of
common grow they start to cross over. For instance,
Elster has the highest miss ratio on the first group,
and Cyclic has the higher on the second group, as the
problem size increases.

4.2 Results - Cache Intel

Using this cache type the algorithms behave with the
same pattern but different factors. Now I will show
the correspondent tables and figures for this type of
cache.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Regression function</th>
<th>R²</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elster</td>
<td>( y = 1.6243x - 9647.6 )</td>
<td>1</td>
</tr>
<tr>
<td>Recursive</td>
<td>( y = 4E-08x^2 + 8.5108x - 1E+06 )</td>
<td>0.9997</td>
</tr>
<tr>
<td>Jeong</td>
<td>( y = -3E-12x^3 + 0.6253x - 2335.1 )</td>
<td>1</td>
</tr>
<tr>
<td>Cyclic</td>
<td>( y = 4E-07x^2 + 25.896x - 3E+06 )</td>
<td>0.9994</td>
</tr>
<tr>
<td>Aware</td>
<td>( y = -4E-09x^2 + 1.1996x - 70677 )</td>
<td>0.9997</td>
</tr>
</tbody>
</table>

4.3 Results - Cache Comparison

This section presents various plots comparing for
each algorithm the response obtain on both cache
types, the first plots (Figure 11) compare them in
terms of misses, and the last ones (Figure 12)
compare them in terms of miss ratio. By looking the Figure 11 is noticed that all the algorithms but Jeong are affected by the cache type, and can be seen that for each algorithm there is a point were a rapid grow starts. On the miss ratio plots (Figure 12) is noted also that the cache types affect the miss ratio from a common start point for each algorithm, but for Jeong that is affected only for a range of problem sizes later are nearly the same ratio. Following are the figures.

4.4 Results - General Discussion

Finally after the previous results were presented I come up with the following general results: For both cache types the regression analysis for all algorithms gives a lineal function. The slope factor determines the grow rate of each algorithm. Aware, Jeong, and Elster have a similar slope factor. Natural recursive have a bigger slope, and Cyclic has the biggest slope. There is a range on problem sizes where all the algorithms have an average response dependent on the cache type. (2^13 Sparc, or 2^19 Intel). I could say that the slope factor is determine by the total count of address references. Comparing response by cache type the Aware, Elster, Cyclic, Natural recursive algorithms show to be dependent, but Jeong algorithm show to be independent. Also I find that there is a minimum Miss Ratio for all algorithms and seems to be also a maximum, since for Elster, Jeong and Aware already can be accepted, and for Cyclic and Recursive there is a tendency. My opinion is that the Jeong algorithm is the best from the point of cache response.

References


