Project Report on

“Testing Database Performance with HelperCore on Multi-Core Processors”

Submitted by

Mayuresh P. Kunjir
M.E. (CSA)

Mahesh R. Bale
M.E. (CSA)

Under Guidance of

Dr. T. Matthew Jacob
Problem Definition

Study the performance of database queries on multi-core processor by measuring last level cache misses and then analyze the effect of using HelperCore[1] approach on the performance.

1. Introduction

The ability to pack billions of transistors on-chip has opened the doors to an important trend in building high-performance computer systems. Until recently, the trend for the development of microprocessors was by increasing their frequency and capability to extract more instruction level parallelism. This has led to severe limitations in terms of power consumption and design complexity. Therefore, this trend has changed to keeping the design simple while packing more processors onto the same die in order to achieve better performance. These chips are called Chip Multiprocessors (CMP) or Multi-cores. CMPs offer a higher granularity (thread/process level) at which parallelism in programs can be exploited by compiler/runtime support, rather than leaving it to the hardware to extract the parallelism at the instruction level on a single (larger) multiple-issue core. It is now possible to split a single application into multiple tasks and execute them in parallel. Nevertheless, an increasing number of tasks results in an increase in simultaneous memory requests and, therefore, the access to memory becomes a bottleneck. In addition, an increasing number of tasks may also result in larger synchronization penalty.

The use of multi-cores is limited to exploiting them for higher throughput. Current database management systems like PostgreSQL don't have any sophisticated techniques to exploit the benefits offered by the multi-core technology. What is currently possible is issuing queries in parallel i.e. on different processors. But due to the synchronization required by the different query executions in order to maintain database consistent, the parallelization will not be beneficial when the number of execution cores increases beyond a certain point. In addition, the increasing number of memory accesses to shared resources results in a bottleneck. Taking this into consideration, [1] have proposed a new approach to exploit the cores of a multi-core architecture in an alternative way. Instead of executing application code on all cores, some cores execute code that indirectly results in a performance benefit for the application. We call these the ‘Helper Cores’. An example of such a use of the multi-core chip is depicted in Figure 1, which presents a high-level diagram of the execution of an application on a 16-core multi-core chip where 12 cores are used for the execution of the application and 4 cores as Helper Cores.

![Figure 1. Multicore utilization scenario.](image-url)
It is observed that, all the cores are not exploited by the applications in a typical database system. The performance of database applications can be improved by using the unused cores of processors by using them as ‘helper cores’. We plan to use this approach on PostgreSQL database system and study the performance by measuring the last level cache (L2) misses.

The report is organized as follows. Section 2 summarizes the related work in the area. In section 3, we explain the idea of HelperCore approach in detail. The implementation details are given in section 4. Section 5 gives details of experiments carried out and results. The conclusion is given in section 6.

2. Related Work

Data pre-fetching is a technique that reduces memory latency by fetching data into the cache before it is requested by the CPU. Pre-fetching can be broadly classified into two categories: hardware-based pre-fetching and software-based pre-fetching. In the first category, the pre-fetch requests are generated automatically by the hardware. The hardware monitors the misses in the caches, predicts the next data to be requested and issues appropriate pre-fetch requests. The second pre-fetch category requires the programmer or compiler to explicitly insert pre-fetch instructions in the code given that the microprocessor supports such instructions.

An alternative to traditional pre-fetching is known as preloading which uses an extra thread created by the programmer or the system. Zhou et al. [2] proposed this technique. A thread is used as helper thread which executes a subset of the code ahead of the execution processor. This execution results in data being pre-fetched to the shared level of memory hierarchy so that the other working threads will almost always find the required data and hence executes faster.

The query processing in database systems can be made faster by exploiting intra-operator as well as inter-operator parallelism for typical database operators like join. [8] proposed the idea of encapsulating relational operators for efficiently utilizing the resources. But the problem of identifying and encapsulating the possible operator parallelism is critical here.

3. HelperCore Idea

The issues with HelperCoreDB design are:
(1) It must fetch the data from memory before the database application uses it but just early enough as to avoid being evicted before its use.
(2) Synchronization between the HelperCoreDB and database threads should result in a minimal overhead.
(3) The implementation of HelperCoreDB should be simple.

To fetch the data before the application needs it, we need to consider what code to execute on the HelperCore. Memory access latency is one of the most serious bottleneck in memory resident database systems. We use a form of data pre-fetching to tackle it. The pre-fetching is performed on the HelperCore, while this pre-fetched data is used by worker processes running on other cores. So the HelperCore and the Worker cores must share some level of memory hierarchy. We use L2 as the level of hierarchy shared by them.

Database systems organize data in blocks. A block is a construct of a specified fixed size that generally holds tuples of data and control information. Blocks reside on permanent storage as pages and in main memory as buffers. In the database applications, for certain algorithms, such as
the Sequential Scan, whenever a buffer is accessed most of the data on that buffer is likely to be used at that time. Taking advantage of this fact, HelperCoreDB is designed to pre-fetch at the buffer granularity. Consequently, the worker process (the process executing the database application) needs to communicate to the helper process (the process pre-fetching the data) only the next block to be processed by the query plan. This leads to both simpler implementation and less synchronization overhead, thus satisfying criteria (2) and (3).

After being triggered to fetch a block, the helper process chooses to fetch ‘d’ consecutive blocks with the assumption that those blocks will be used in the near future. The number d of consecutive blocks that are requested is called pre-fetch distance. In order to satisfy criteria (1), d needs to be carefully determined as a small value of d may not bring enough data to overcome the overheads and a large value of d may bring more data than necessary at that point, thus potentially evicting other useful data. Different queries should have different values of d.

The HelperCore may be implemented either by creating a separate helper thread and then using a busy-wait loop or a blocking synchronization primitive, such as the POSIX thread pthread_cond_wait() or the Linux specific mutex() system call. Blocking synchronization adds unacceptable overhead to the worker thread, so this technique is not suitable. Busy-wait synchronization is possible, but we need a helper thread for each worker process which is the number of clients of database application. Another way of sharing memory is by running a background helper process. Other client processes share the block number to be read and the relation(table) number from which to read it with this helper process by using ‘shm’ API. The synchronization between this data is achieved by using semaphores (‘sem’ API). We implemented this technique in PostgreSQL database, the details of which are given in next section.

4. Implementation

This section first describes the ‘shm’ API which is used for sharing the data between helper and worker processes in subsection 4.1. In subsection 4.2, we give an overview of the performance monitoring tool ‘perfmon2’ which is used to count L2 cache references and misses for analysis of our project. Then in subsection 4.3, the modifications done in PostgreSQL database system are listed.

4.1 Shared Memory

Shared memory is a method of inter-process communication (IPC), i.e. a way of exchanging data between programs running at the same time. One process will create an area in RAM which the other processes can access. In our project we need to share data between a worker process and the helper process. We used the API for shared memory provided by UNIX system 5. The corresponding function prototypes are defined in header file <sys/shm.h>

A process can create a shared memory segment using shmget(). Once created, a shared segment can be attached to a process address space using shmat(). It can be detached using shmdt(). The attaching process must have the appropriate permissions for shmat(). Once attached, the process can read or write to the segment, as allowed by the permission requested in the attach operation.
4.2 Perfmon tool

All modern processors have a hardware performance monitoring unit (PMU) which exports a set of counters, called hardware performance counters to collect micro-architectual events such as the number of elapsed cycles or the number of L1(or L2) cache misses etc.. Exploiting these counters to analyze the performance of key applications and operating systems is becoming common practice.

perfmon2[4] is a hardware based performance monitoring interface for Linux. Perfmon needs the kernel support. The interface is available as a standalone kernel patch. This patch is needed to be applied to the kernel in order to use the perfmon2 interface. We applied perfmon patch for linux kernel version 2.6.27.

Perfmon provides the options of profiling at various levels of granularity. One can either profile the entire system or a single process or all processes on a single CPU or even various functions within a single process. It counts the various micro-architectual events like CPU cycles, cache misses, cache references, TLB accesses etc.

4.3 Modifications to PostgreSQL

We used PostgreSQL [5,6] version 8.3.4 for our work.

In PostgreSQL 'postmaster' process is the main process. Postmaster spawns a new process whenever new client request arrives. Let us call it a WorkerProcess. Postgres runs some background processes along with that e.g. auto-vacuum garbage collector, write-ahead logger, statistics collector. We added one more background process called the HelperProcess which is also a child of postmaster.

We modified the postmasterMain() function of postmaster so that it creates the shared memory segment (using shmget()). The shared memory segment contains variable workers(number of workers) and two arrays rel_no[MAX_WORKERS] and block_no[MAXWORKERS]. MAX_WORKERS is a macro indicating maximum number of Worker Processes. Both the arrays have an entry each for each WorkerProcess. Array rel_no is used to store the requested relation number while the array block_no for requested block number.

Whenever new client is created, postmaster calls a function BackendStartup() which does some initialization work and sets up the client. We attach the shared memory for the WorkerProcess in this function using function shmat().

Whenever the WorkerProcess needs a block of data, it asks the buffer manager for the same. The file bufmgr.c provides these functionalities. There is a function ReadBuffer(), which is the single point of access to the data. It takes the relation number and block number which is to be fetched and finds the corresponding memory buffer that holds that block. If the buffer does not exist in main memory, it reads it from permanent storage. We modified the function ReadBuffer, so that in addition to fetching the required block it writes the relation number and block number to shared memory locations rel_no[i] and block_no[i]. Here ‘i’ refers to the index of the WorkerProcess. Each relation is identified by a structure RelFileNode and the relation number is given by its field relNode. The variable blockNum contains the block number to be fetched. Each block is of size 8kB which is the default block size used by PostgreSQL.

The HelperProcess maintains two private arrays old_rel_no[] and old_block_no[] and a macro D which is the pre-fetch distance. The entries old_rel_no[i] and old_block_no[i] refer to
the relation number and block number last requested by WorkerProcess ‘i’. The HelperProcess is a continuous loop. In each iteration of the loop, it successively checks the locations in shared arrays rel_no and block_no i.e. if any of the worker processes has requested a new block that was not pre-fetched from its last request. If a new request is found then the HelperProcess starts pre-fetching data blocks. Its objective is to read ‘D’ consecutive data blocks starting from the current block requested by the worker. This allows the HelperProcess to effectively pre-fetch the data blocks ahead of their use.

We have synchronized helper and worker processes by the use of semaphore. The sys/sem.h library provides the functions semget(), semctl() and semop() which are used to manage the semaphores. The code fragment for ReadBuffer function and the loop of HelperProcess is given below.

```c
/*Shared memory variables*/

//number of workers currently in execution
int *workers;

//relation numbers of the workers
int *rel_no[MAX_WORKERS];

//block number of the current block for each worker
int *block_no[MAX_WORKERS];

/*ReadBuffer function*/

//i is the worker process identifier
//reln is the relation structure pointer
//SemID is semaphore variable

//Wait(Empty)
semop(SemID, &WaitEmpty, 1);

//write into shared memory the relation number and block number
*rel_no[i] = reln->rd_node;
*block_no[i] = blockNum;

//Signal(Full)
semop(SemID, &SignalFull, 1);

//Fetch the current block
ReadBuffer_common(reln, blockNum, zeroPage, strategy);

/*Helper process*/

for(i=0 ; i<*workers ; i++)
{

    //Wait(Full)
    semop(SemID, &WaitFull, 1);

    //read from shared memory the relation number and block number
    tmp_rel_no = *rel_no[i];
    tmp_block_no = *block_no[i];

    //fetch next block if following condition holds
    if((tmp_rel_no.relNode != old_rel_no[i])
```
|| (tmp_block_no < old_block_no[i])
|| (tmp_block_no > old_block_no[i] + D ))
{
    for(k=1 ; k<=D ; k++)
    {
        //Read the next block
        if(My_ReadBuffer(tmp_rel_no, tmp_block_no+k, NULL) == -1)
        {
            break;
        }
    }
    //update old block number and relation number
    old_block_no[i] = tmp_block_no;
    old_rel_no[i] = tmp_rel_no.relNode;
}

//Signal(Empty)
semop(SemID, &SignalEmpty, 1);

5. Experimental Setup

5.1 System characteristics

All our experiments were executed natively on an Intel Core 2 Duo processor. The system’s characteristics are listed in Table 1. L1 is private to each core while L2 is shared.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Intel Core 2 Duo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Cores</td>
<td>2</td>
</tr>
<tr>
<td>Frequency</td>
<td>2 GHz</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>private to each core</td>
</tr>
<tr>
<td>L1 Data Cache Size</td>
<td>16KB</td>
</tr>
<tr>
<td>L1 Data Cache Line Size</td>
<td>64 Bytes</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>shared by both cores</td>
</tr>
<tr>
<td>L2 Cache Size</td>
<td>2048KB</td>
</tr>
<tr>
<td>L2 Cache Line Size</td>
<td>128 Bytes</td>
</tr>
<tr>
<td>Main Memory (RAM)</td>
<td>2GB DDR2</td>
</tr>
</tbody>
</table>

*Table 1. System characteristics*

The experiments were carried on linux kernel version 2.6.27 with performance monitoring support, for which we applied the perfmon patch to the kernel.

5.2 Workload

We used the database benchmark workload TPC-H for our experiments. As sequential scan is an operation for which pre-fetching would give the most benefit, we tested queries involving sequential scan on various relations of the TPC-H database. For each query we measured three parameters, viz.

1. Number of L2 references
2. Number of L2 misses
3. Execution time

The above three parameters were measured both with and without using HelperCore. We also varied the value of pre-fetch distance D. When D equals zero, it is as good as normal execution without using HelperCore support.

We counted these parameters by profiling the worker process executing the database queries. We are presenting the results for three queries below.

5.3 Results

We first present results of two queries which are given below.
Query 1: Select * from customer;
Query 2: Select * from part;

Figure 2 shows the effect of varying pre-fetch distance D on L2 cache references. It can be observed that number of L2 references doesn’t change much with change in D. The slight reduction is due to the fact that; as we increase D, we pre-fetch more and more number of blocks and worker process will find the blocks in L1 cache.

Figure 3 shows the effect of D on L2 cache misses. As we pre-fetch more and more data, the L2 misses should reduce. But for higher values of D, the newly pre-fetched blocks replace the earlier ones resulting in an increase in L2 misses. We have found from our experiments that pre-fetch distance of 3 is optimal. It approximately reduces number of misses to 2/3rd of the misses without pre-fetching (D=0). It is also observed that the values of D greater than 5 increase the cache misses.

![Figure 2: L2 cache references plotted against pre-fetch distance](image-url)
Figure 3: L2 cache misses plotted against pre-fetch distance

Figure 4: Execution time (ms) plotted against pre-fetch distance

Figure 4 shows the change in execution time as we vary D. There is no improvement observed. This is due to the fact that, helper process and worker process are synchronized resulting in waiting time on semaphore. Although, the time to fetch the data is reduced due to reduction in cache misses, the waiting time amounts to increase in total execution time. The execution without helper process support i.e. D=0 is found to take the least amount of time.

We also tested the TPC-H benchmark query template 6 which is a sequential scan on relation ‘lineitem’. The results are shown in Table1 below. We found same patterns in these results also.
QT6 : 
```
select sum(l_extendedprice * l_discount) as revenue
from lineitem
where l_extendedprice > 0
```

<table>
<thead>
<tr>
<th>D</th>
<th>L2 references</th>
<th>L2 misses</th>
<th>Time(seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>45106742</td>
<td>180475</td>
<td>12.24</td>
</tr>
<tr>
<td>1</td>
<td>44789333</td>
<td>143256</td>
<td>14.82</td>
</tr>
<tr>
<td>2</td>
<td>44120988</td>
<td>124532</td>
<td>13.05</td>
</tr>
<tr>
<td>3</td>
<td>43050196</td>
<td>113243</td>
<td>12.58</td>
</tr>
<tr>
<td>4</td>
<td>42242583</td>
<td>129876</td>
<td>13.67</td>
</tr>
<tr>
<td>5</td>
<td>42229009</td>
<td>168730</td>
<td>13.15</td>
</tr>
<tr>
<td>6</td>
<td>41100230</td>
<td>178593</td>
<td>13.46</td>
</tr>
<tr>
<td>7</td>
<td>39880133</td>
<td>219224</td>
<td>12.68</td>
</tr>
<tr>
<td>8</td>
<td>38896054</td>
<td>232451</td>
<td>13.25</td>
</tr>
<tr>
<td>9</td>
<td>37531957</td>
<td>255832</td>
<td>13.92</td>
</tr>
<tr>
<td>10</td>
<td>36791351</td>
<td>261783</td>
<td>12.84</td>
</tr>
</tbody>
</table>

*Table 2: QT6 results*

6. Conclusion

We implemented HelperCoreDB approach in PostgreSQL database system and analyzed the effects on performance. We found that L2 cache misses reduce approximately by 33% for optimal pre-fetch distance which we found to be 3 for our system configuration. The original paper found a reduction of about 22% in execution time. But our results show that there is no improvement in execution time due to synchronization overheads. The original paper used a separate helper thread for a client which enables sharing of global data. But this suggests creating separate helper thread for each client which is adding too much overhead on system. We chose to use a background process which will work as helper process for all the clients. Data sharing is achieved by shm API. As the results show, this idea does reduce the L2 cache misses but performance doesn’t improve because of synchronization and shared memory overheads.

References