CACHE MANAGEMENT FOR MULTI CORE ARCHITECTURE IN REAL TIME SYSTEM

Debasis Maji[1], Mainak Biswas[2], Supriyo Nath[3], Milan Mukherjee[4], Sushovan Bhaduri[5]
[1,2,5]Master of Electrical Engineering
[3]Master of Illumination Engineering
[4]Master of Material Engineering
Jadavpur University
Kolkata, India.

Abstract: In case of multi core processors several tasks are running simultaneously. The performance of co-running tasks are affected due to the interference problem in shared cache memory which depends on present task load on processors. So need of proper job scheduling in different cores. Using of multithreading paradigm exploit the instruction level parallelism. With a huge overhead of context switching, the use of thread comes. On the other hand, the task calculates dynamically if it is necessary to create different threads of execution or not. Without going through the overhead of Thread creation or unnecessary context switching if not required, it uses the Thread Pool in order to distribute the work.
In this paper, the task of multiplication is taken. The traditional matrix multiplication using threads are compared with the approach of using the Task Parallel Library (TPL).

Keywords: Multi-core, Multi-thread, Thread Pool, Task Parallel Library, and TLP.

I. INTRODUCTION

Multi threading computer central processing units have hardware support to efficiently execute multiple threads. These are distinguished from multiprocessing systems (such as multi-core systems) in that the threads have to share the resources of a single core: the computing units, the CPU caches and the translation look aside buffer (TLB).

Where multiprocessing systems include multiple complete processing units, multithreading aims to increase utilization of a single core by using thread-level as well as instruction-level parallelism. As the two techniques are complementary, they are sometimes combined in systems with multiple multithreading CPUs and in CPUs with multiple multithreading cores.

II. MULTI-CORE

A multi-core processor is a single computing component with two or more independent actual central processing units (called "cores"), which are the units that read and execute program instructions. The instructions are ordinary CPU instructions such as add, move data, and branch, but the multiple cores can run multiple instructions at the same time, increasing overall speed for programs amenable to parallel computing. Manufacturers typically integrate the cores onto a single integrated circuit die (known as a chip multiprocessor or CMP), or onto multiple dies in a single chip package. Multi-core processors are widely used across many application domains including general-purpose, embedded, network, digital signal processing (DSP), and graphics. The improvement in performance gained by the use of a multi-core processor depends very much on the software algorithms used and their implementation. In particular, possible gains are limited by the fraction of the software that can be run in parallel simultaneously on multiple cores; this effect is described by Amdahl's law. In the best case, so-called embarrassingly parallel problems may realize speedup factors near the number of cores, or even more if the problem is split up enough to fit within each core's cache(s), avoiding use of much slower main system memory. Most applications, however, are not accelerated so much unless programmers invest a prohibitive amount of effort in re-factoring the whole problem. The parallelization of software is a significant ongoing topic of research. Various methods are used to improve CPU performance. Some instruction-level parallelism (ILP) methods such as superscalar pipelining are suitable for many applications, but are inefficient for others that contain difficult-to-predict code. Many applications are better suited to thread level parallelism (TLP) methods, and multiple independent CPUs are commonly used to increase a system's overall TLP. A combination of increased available space (due to refined manufacturing processes) and the demand for increased TLP led to the development of multi-core CPUs.

1. **THREADS**
   A thread is defined as the execution path of a program. Each thread defines a unique flow of control. If the application involves complicated and time consuming operations like database access or some intense I/O operations, then it is often helpful to set different execution paths or threads, with each thread performing a particular job.
2. **IMPLEMENTING THREADS**
   In this project matrix multiplication is implemented using multi-threading in ASP.NET Framework. Threads are lightweight processes. One common example of use of thread is implementation of concurrent programming by modern operating systems. Use of threads saves wastage of CPU cycle and increase efficiency of an application. So far programs are written where a single thread runs as a

 Copyright 2013.All rights reserved.

single process which is the running instance of the application. However, this way the application can perform one job at a time. To make it execute more than one task at a time, it could be divided into smaller threads. In .NET, the threading is handled through the System. Threading namespace. Creating a variable of the System. Threading. Thread type allows creating a new thread to start working with. It allows creating and accessing individual threads in a program.

3. **CREATING THREADS**
   A thread is created by creating a Thread object, giving its constructor a Thread Start reference.
   ThreadStart childthread = new ThreadStart(childthreadcall);
4. **THE THREAD LIFE CYCLE**
   The life cycle of a thread starts when an object of the System. Threading. Thread class is created and ends when the thread is terminated or completes execution. Following are the various states in the life cycle of a thread:
   (a) The Un-started State: it is the situation when the instance of the thread is created but the Start method has not been called.
   (b) The Ready State: it is the situation when the thread is ready to run and waiting CPU cycle.
   (c) The Not Run able State: a thread is not run able, when:
      i. Sleep method has been called.
      ii. Wait method has been called.
      iii. Blocked by I/O operations.
   (d) The Dead State: it is the situation when the thread has completed execution or has been aborted.
5. **THE THREAD PRIORITY**
   The Priority property of the Thread class specifies the priority of one thread with respect to other. The .Net runtime selects the ready thread with the highest priority. The priorities could be categorized as:
   (a) Above normal
   (b) Below normal
   (c) Highest
   (d) Lowest
   (e) Normal

   Once a thread is created its priority is set using the Priority property of the thread class.
   NewThread.Priority = ThreadPriority.Highest;
6. **THREAD PROPERTIES AND METHODS**
   Thread properties and methods are defined as per table 1:
7. **ORGANIZATION OF THE REPORT**
   In part 3, multi-threading scheduling schemes are discussed. In part 2, the problem and methodologies are discussed. In part 5, implementation details and the results of Performance Monitor and Resource Monitor are discussed. In part 6,future work is mentioned and concluded.

Property	Description
Current Context	Gets the current context in which the thread is executing
Current Culture	Gets or sets the culture for the current thread
Current Principle	Gets or sets the thread's current principal (for role-based security)
Current Thread	Gets the currently running thread
Current UI Culture	Gets or sets the current culture used by the Resource Manager to look up culture-specific resources at run time
Execution Context	Gets an Execution Context object that contains information about the various contexts of the current thread
Is Alive	Gets a value indicating the execution status of the current thread
Is Background	Gets or sets a value indicating whether or not a thread is a background thread
Is Thread Pool Thread	Gets a value indicating whether or not a thread belongs to the managed thread pool
Managed Thread Id	Gets a unique identifier for the current managed thread
Name	Gets or sets the name of the thread
Priority	Gets or sets a value indicating the scheduling priority of a thread
Thread State	Gets a value containing the states of the current thread

Table 1: Thread Properties and Methods.

III. LITERATURE SURVEY

Several researches are going on to improve the cache utilization, processor performance and to meet the task deadline.

1. **REAL-TIME SYSTEMS**
   A real-time operating system (RTOS) is an operating system (OS) intended to serve real-time application requests. It must be able to process data as it comes in, typically without buffering delays. Processing time requirements (including any OS delay) are measured in tenths of seconds or shorter.

   A key characteristic of an RTOS is the level of its consistency concerning the amount of time it takes to accept and complete an application's task; the variability is jitter. A hard real-time operating system has less jitter than a soft real-time operating system. The chief design goal is not high throughput, but rather a guarantee of a soft or hard performance category.

   An RTOS that can usually or generally meet a deadline is a soft real-time OS, but if it can meet a deadline deterministically it is a hard real time OS.An RTOS has an advanced algorithm for scheduling. Scheduler flexibility

 Copyright 2013.All rights reserved.

enables a wider, computer-system orchestration of process priorities, but a real-time OS is more frequently dedicated to a narrow set of applications. Key factors in a real-time OS are minimal interrupt latency and minimal thread switching latency; a real-time OS is valued more for how quickly or how predictably it can respond than for the amount of work it can perform in a given period of time.

2. **HARD-REAL TIME SYSTEM**
   An overrun in response time leads to potential loss of life and/or big financial damage. Many of these systems are considered to be safety critical. Sometimes they are "only" mission critical, with the mission being very expensive. In general there is a cost function associated with the system.
3. **SOFT REAL-TIME SYSTEM**
   Deadline overruns are tolerable, but not desired. There are no catastrophic consequences of missing one or more deadlines. There is a cost associated to overrunning, but this cost may be abstract. It often connected to Quality-of-Service (QoS).
4. **FIRM REAL-TIME SYSTEM**
   The computation is obsolete if the job is not finished on time. Cost may be interpreted as loss of revenue. Typical examples are forecast systems.
5. **WEAKLY HARD REAL-TIME SYSTEM**
   Systems where m out of k deadlines has to be met. In most cases feedback control systems, in which the control becomes unstable with too many missed control cycles. Best suited if system has to deal with other failures as well (e.g. Electro Magnetic Interference EMI).
6. **NON REAL-TIME SYSTEM**
   In most cases the (soft) real-time aspect may be constructed (e.g. acceptable response time to user input).Computer system is backed up by hardware (e.g. end position switches). Quite often this type of system has simply oversized computers.

A. DIFFERENT SOLUTIONS

1. **MEMORY PROFILING AND COLORED LOCK-DOWN [1]**
   Multi-core architectures are shaking the fundamental assumption that in real-time systems the WCET, used to analyze the schedulability of the complete system, is calculated on individual tasks. This is not even true in an approximate sense in a modern multi-core chip, due to interference caused by hardware resource sharing. In this work proposed as (1) a complete framework to analyze and profile task memory access patterns and (2) a novel kernel-level cache management technique to enforce an efficient and deterministic cache allocation of the most frequently accessed memory areas. In this way, a powerful tool is provided to address one of the main sources of interference in a system where the last level of cache is shared among two or more CPUs.
2. **WCRT MINIMIZATION [5]**
   In a multi core platform, the inter-thread cache interferences can significantly affect the worst-case execution time (WCET) of each real-time task, which is crucial for schedulability analysis. At the same time, the worst-case cache interferences are dependent on how tasks are scheduled to run on different cores, thus creating a circular dependence. An offline real-time scheduling approach on multi core processors by considering the worst-case inter-thread interferences on shared L2 caches is done. This scheduling approach uses a greedy heuristic to generate safe schedules while minimizing the worst-case inter-thread shared L2 cache interferences and WCET.
3. **THREAD SCHEDULING LIBRARY [3]**
   Here four libraries are compared for efficiently running threads when the performance of a CPU cores are degraded. First, brute performance of the libraries when all the CPU resources are available are considered and second, measuring how the scheduling strategy impacts also the memory management in order to revisited. It is well known that work stealing, when done in an anarchic way, may lead to poor cache performance. It is also known that the migration of threads may induce penalties if they are too frequent. At the processor level, the memory management in order to find trade-offs between active thread number that an application should start and the memory hierarchy.
4. **INTER THREAD CACHE INTERFERENCE [2]**
   In a multi core platform, the inter-thread cache interferences can significantly affect the worst-case execution time (WCET) of each real-time task, which is crucial for schedulability analysis. At the same time, the worst-case cache interferences are dependent on how tasks are scheduled to run on different cores, thus creating a circular dependence. In this solution, an offline real time scheduling approach on multi core processors by considering the worst-case inter-thread interferences on shared L2 caches is presented. The scheduling approach uses a greedy heuristic to generate safe schedules while minimizing the worst-case inter-thread shared L2 cache interferences and WCET.

IV. MULTI-THREADING

Some advantages of multi-threading include:

1. If a thread gets a lot of cache misses, the other thread(s) can continue, taking advantage of the unused computing resources, which thus can lead to faster overall execution, as these resources would have been idle if only a single thread was executed.
2. If a thread cannot use all the computing resources of the CPU (because instructions depend on each other's result), running another thread can avoid leaving these idle.
3. If several threads work on the same set of data, they can actually share their cache, leading to better cache usage or synchronization on its values.

 Copyright 2013.All rights reserved.

Some criticisms of multithreading include:

1. Multiple threads can interfere with each other when sharing hardware resources such as caches or TLBs.
2. Execution times of a single thread are not improved but can be degraded, even when only one thread is executing. This is due to slower frequencies and/or additional pipeline stages that are necessary to accommodate thread-switching hardware.
3. Hardware support for multithreading is more visible to software, thus requiring more changes to both application programs and operating systems than multiprocessing.

The Task Parallel Library (TPL) is based on the concept of a task, which represents an asynchronous operation. In some ways, a task resembles a thread or Thread Pool work item, but at a higher level of abstraction. The term task parallelism refers to one or more independent tasks running concurrently. Tasks provide two primary benefits:

1. More efficient and more scalable use of system resources. Behind the scenes, tasks are queued to the Thread Pool, which has been enhanced with algorithms that determine and adjust to the number of threads and that provide load balancing to maximize throughput. This makes tasks relatively lightweight, and can create many of them to enable fine-grained parallelism.
2. More programmatic control than is possible with a thread or work item. Tasks and the framework built around them provide a rich set of APIs that support waiting, cancellation, continuations, robust exception handling, detailed status, custom scheduling, and more.

For both of these reasons, in the .NET Framework, TPL is the preferred API for writing multi-threaded, asynchronous, and parallel code. Objective of this project work is to show the comparative results of multi-threading and TPL by using the task of matrix multiplication.

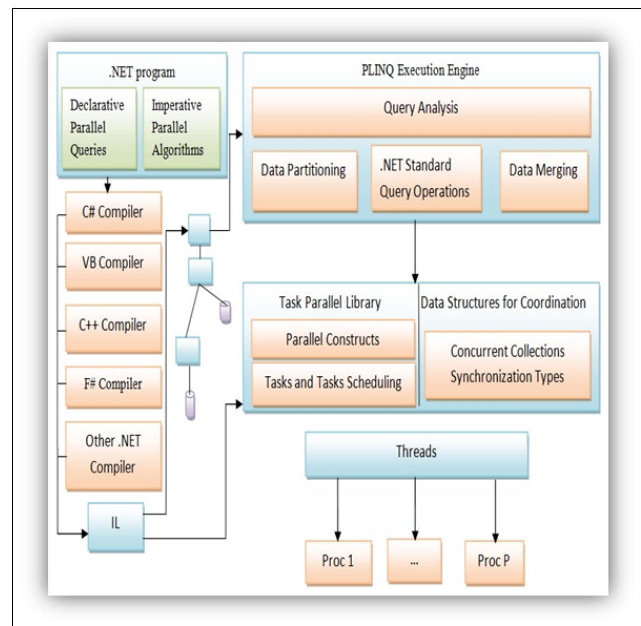

Figure 1: A high-level overview of the parallel programming architecture in the .NET Framework 4

V. PROPOSED WORK

1. **INTRODUCTION**
   The Task Parallel Library (TPL) is a set of public types and APIs in the System. Threading and System. Threading. Tasks namespaces in the .NET Framework 4. The purpose of the TPL is to make developers more productive by simplifying the process of adding parallelism and concurrency to applications. The TPL scales the degree of concurrency dynamically to most efficiently use all the processors that are available. In addition, the TPL handles the partitioning of the work, the scheduling of threads on the Thread Pool, cancellation support, state management, and other low-level details. By using TPL, it is possible to maximize the performance of code while focusing on the work that the program is designed to accomplish.
2. **PARALLEL PROGRAMMING IN .NET FRAMEWORK**
   Many personal computers and workstations have two or four cores (that is, CPUs) that enable multiple threads to be executed simultaneously. Computers in the near future are expected to have significantly more cores. To take advantage of the hardware of today and tomorrow, it is possible to parallelize code to distribute work across multiple processors. In the past, parallelization required low-level manipulation of threads and locks. Visual Studio 2010 and the .NET Framework 4 enhance support for parallel programming by providing a new runtime, new class library types, and new diagnostic tools. These features simplify parallel development so that it is possible to write efficient, fine-grained, and scalable parallel code in a natural idiom without having to work directly with threads or the thread pool. Fig 5.1 A high-level overview of the parallel programming architecture in the .NET Framework 4.
3. **TASK PARALLEL LIBRARY**
   Starting with the .NET Framework 4, the TPL is the preferred way to write multithreaded and parallel code. However, not all code is suitable for parallelization; for example, if a loop performs only a small amount of work on each iteration, or it doesn't run for many iterations, then the overhead of parallelization can cause the code to run more slowly. Furthermore, parallelization like any multi-threaded code adds complexity to program execution.
4. **DATA PARALLELISM USING TPL**
   Data parallelism refers to scenarios in which the same operation is performed concurrently (that is, in parallel) on elements in a source collection or array. In data parallel operations, the source collection is partitioned so that multiple threads can operate on different segments concurrently.

```
// Sequential version
foreach (var item in sourceCollection)
```

 Copyright 2013.All rights reserved.

```
{
Process(item);
}
// Parallel equivalent
Parallel.ForEach(sourceCollection, item => Process(item));
```

When a parallel loop runs, the TPL partitions the data source so that the loop can operate on multiple parts concurrently. Behind the scenes, the Task Scheduler partitions the task based on system resources and workload. When possible, the scheduler redistributes work among multiple threads and processors if the workload becomes unbalanced.

5. **TASK PARALLISM USING TPL**
The Task Parallel Library (TPL) is based on the concept of a task, which represents an asynchronous operation. In some ways, a task resembles a thread or Thread Pool work item, but at a higher level of abstraction. The term task parallelism refers to one or more independent tasks running concurrently. Tasks provide two primary benefits:
More efficient and more scalable use of system resources. Behind the scenes, tasks are queued to the Thread Pool, which has been enhanced with algorithms that determine and adjust to the number of threads and that provide load balancing to maximize throughput. This makes tasks relatively lightweight

6. **HARDWARE ENVIRONMENT**
   - Processor - Core i5 (3rd Generation)
   - Variant - 3337U
   - Chipset - Mobile HM76 Express
   - Brand - Intel
   - Clock Speed - 1.8 GHz with Turbo Boost Up to 2.7 GHz
   - Cache - 3 MB
   - System Memory - 6 GB DDR3
   - Hardware Interface - SATA
   - RPM - 5400
   - HDD Capacity - 500 GB

7. **SOFTWARE ENVIRONMENT**
   - Microsoft Visual Studio Professional 2010
   - Implemented in ASP.NET which is a web development platform in Microsoft
   - .NET Framework
   - Visual C and HTML is used
   - Operating System Windows Ultimate 7 Type 64-bit and can be created many of them to enable fine-grained parallelism.

More programmatic control than is possible with a thread or work item. Tasks and the framework built around them provide a rich set of APIs that support waiting, cancellation, continuations, robust exception handling, detailed status, custom scheduling, and more.

For both of these reasons, in the .NET Framework, TPL is the preferred API for writing multi-threaded, asynchronous, and parallel code.

8. **BLOCK SCHEMATIC**

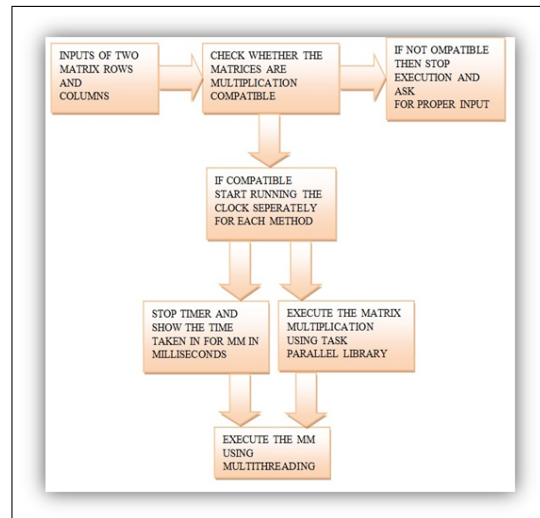

Figure 2: Block Schematic

Performance Monitor and Resource Monitor is used to show Multi-core Processor Utilization and Cache Utilization.

9. **EXPERIMENTAL SETUP** With the software and hardware mentioned the code is executed. The input is given in the constructed GUI and results are compared.

EXPERIMENT 1 For the same size of matrices the multi-

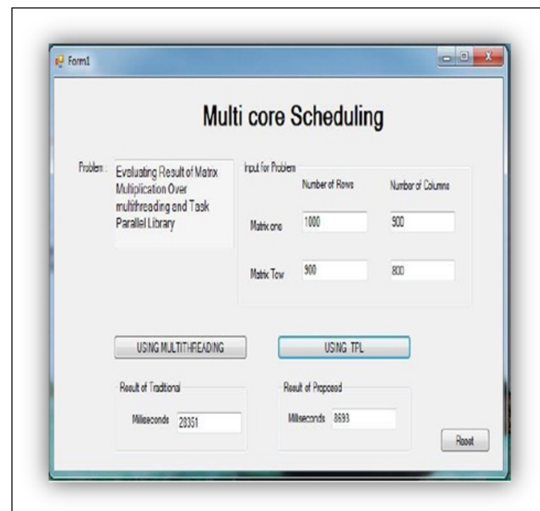

Figure 3: The GUI to execute MM

Copyright 2013.All rights reserved.

threading takes 28351 milliseconds but the MM using TPL takes 8693 milliseconds. The result is analyzed using Resource Monitor and Performance Monitor. The X-axis shows the

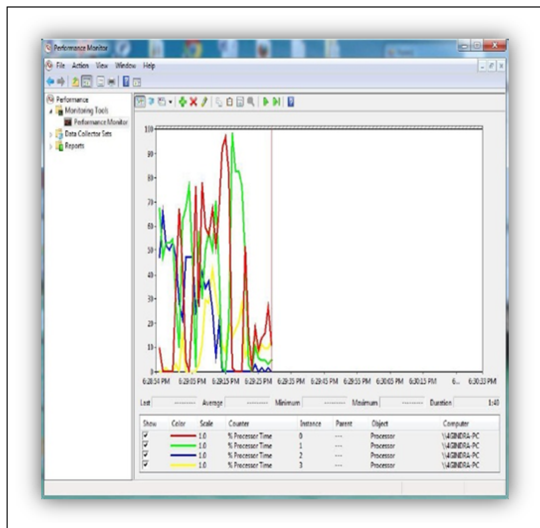

Figure 4: utilization of 4 cores while executing MM using multi-threading

time taken in seconds and the Y-axis shows the percentage of processor utilization. While the MM executes the percentage of processor utilization varies from time to time. In the fig the utilization mostly varies from 30 to 80 percent. While

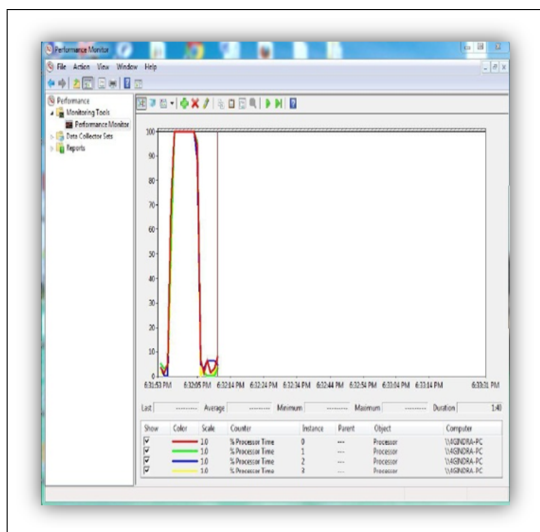

Figure 5: utilization of 4 cores while executing MM using TPL

using TPL the percentage utilization of all the processors are increased and reached to the peak for most of time of task utilization. When the MM finishes execution then the utilization decreases. The instances 0,1,2,3 are four counters and working as processor object.

EXPERIMENT 2 The result is shown in Resource Monitor.

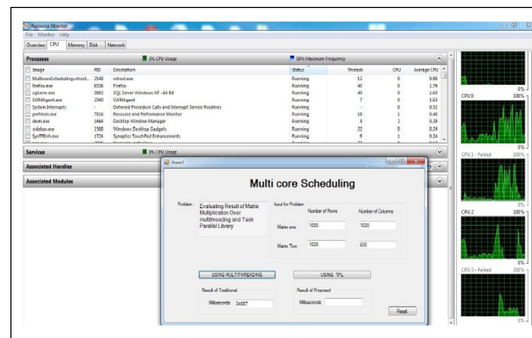

Figure 6: Utilization of individual cores while MM using multithreading

It is possible to input any no. of rows and columns in the two matrices but no of columns in the first matrix and no of rows in the second matrix should be same so that the two matrices are compatible for MM. When the MM using multithreading is executing the utilization of individual processors in the system are shown. Here the CPU performances are random. So most of the processors are underutilized and processors are idle for some time. For the same input of matrices M1(1000*1020)

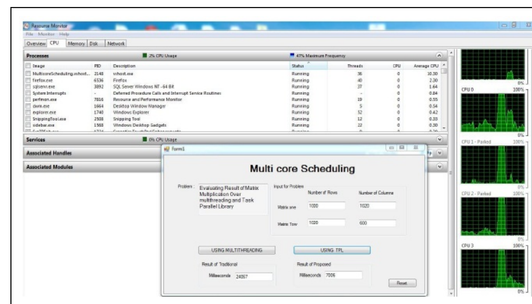

Figure 7: Utilization of individual cores while MM using TPL

and M2(1020*600) the multithreading takes 24067 milliseconds and TPL takes 7006 milliseconds. The individual processor utilization of all processors is almost 100% for the time the MM was executing using TPL. Hard Faults is when a program has asked for an address and the page it resides on is no longer located in main memory as it has either been swapped to disk or has to be referenced from the original source file on disk somewhere. This metric can be used to find performance problems. The less hard faults an application generates, the more often the information it is requesting is found in RAM and the less often it needs to access slower storage mechanisms. In fig 5.8 while MM using multithreading is executing the cached memory used is 1304 MB and the graph shows some

 Copyright 2013.All rights reserved.

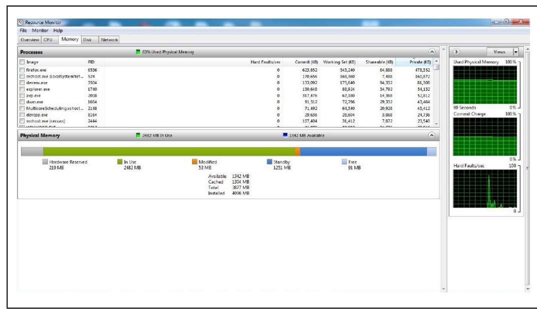

Figure 8: Cache used in MB and Hard Faults per second while using multithreading

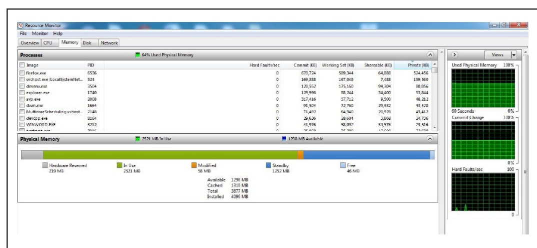

Figure 9: Cache used in MB and Hard Faults per second while using TPL

hard faults has occurred. When MM using TPL is executing the no of hard faults are less and the cache memory used here is 1310 MB. So TPL has much improved performance.

VI. CONCLUSION AND FUTURE WORK

From the experiments given it is possible to conclude that TPL is more efficient approach rather than using multithreading. Although multithreading uses Instruction level and thread level parallelism to improve performance but the creation of thread and execution is costly because it needs sharing of hardware. The TPL scales the degree of concurrency dynamically to most efficiently use all the processors that are available. The TPL is used and relative performance of MM is shown using performance monitor and resource monitor.

In next phase of work the ways to improve shared cache performance by encouraging or discouraging the co-scheduling of groups of tasks based on their expected cache impact is to be explored. Cache aware global-EDF (Earliest Deadline First) task scheduling algorithm is to be used.

References

[1] Renato Mancusoy, Roman Dudkoy, Emiliano Betti, Marco Cesati, Marco Caccamoy, and Rodolfo Pellizzoniz. Real-time and Embedded Technology and Applications Symposium (RTAS). *IEEE*, 2013.

[2] Yiqiang Ding, and Wei Zhang. On the Interactions Between Real-Time Scheduling and Inter-thread Cache Interferences of Multicore Processors Quality Electronic Design (ISQED). *14th International Symposium*, 2013.

[3] Christophe Cerin Hazem Fkaier, and Mohamed Jemni. Experimental study of thread scheduling libraries on degraded CPU. *14th IEEE International Conference on Parallel and Distributed Systems*, 2008.

[4] Xiaoya Xiang, Bin Bao, Chen Ding and Kai Shen. Cache Conscious Task Regrouping on Multicore Processors. *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2012.

[5] Huping Ding Yun Liang Tulika Mitra. Shared Cache Aware Task Mapping for WCRT Minimization. *Design Automation Conference (ASP-DAC),18th Asia and South Pacific*, 2013.

[6] Mayank Shekhar, Abhik Sarkar, Harini Ramaprasad, and Frank Mueller. Semi-Partitioned Hard-Real-Time Scheduling Under Locked Cache Migration in Multicore Systems. *24th Euromicro Conference on Real-Time Systems*, 2012.

[7] Hyoseung Kim, Arvind Kandhalu, and Ragunathan (Raj) Rajkumar. A Coordinated Approach for Practical OS-Level Cache Management in Multi-Core Real- TimeSystems. *25th Euromicro Conference on Real-Time Systems*, 2013.

[8] Chun-Yi Shih, Ming-Chih Li, Chao-Sheng Lin, Pao-Ann Hsiung, and Chih-Hung Chang. Adaptive Performance Monitoring for Embedded Multicore Systems. *International Conference on Parallel Processing Workshops*, 2011.

 Copyright 2013.All rights reserved.