Measuring CPU Time from Hyper-Threading Enabled Intel® Processors

Scott Johnson
TeamQuest Corporation

Recent Intel processors support a technology called hyper-threading. This allows a single physical processor to support two independent architectural states thus turning it into two logical processors. While this improves performance in many instances, some simple tests suggest CPU service time measurements for programs are dependent on processor load. Multiple executions of the same program result in different measurements of CPU service time depending upon other activity in the system. This behavior has an impact for chargeback and capacity planning activities for these types of systems.

1. Introduction

Recent processors in the Intel® Xeon™ and Pentium® 4 product lines include a technology called hyper-threading. The goal of this technology is to increase processor performance by more fully utilizing certain internal processor resources. The ability to enable or disable hyper-threading is often implemented as a BIOS option. When hyper-threading is enabled, a single physical processor exposes two logical processors to the operating system.

Various tests, such as those described in [LENG02] and [MAGR02], have been run with workloads comparing the performance of a particular processor with hyper-threading enabled to the processor with hyper-threading disabled. Not surprisingly, the results indicate that many workloads benefit from hyper-threading. However, the amount of benefit is dependent on the particular workload.

Among the goals of hyper-threading discussed by [MARR02] was minimizing the die area cost for implementing this capability. [MARR02] indicates that this goal was accomplished as the first implementation used less than 5% of the total die area. This implementation resulted in most internal processor resources being shared by logical processors with a small number of replicated resources for the logical processors.

Another goal of the hyper-threading implementation was to allow a processor with only one active thread to use all of the physical processor resources and thereby have the thread run at the same speed as if hyper-threading were disabled. [MARR02] explains that this was accomplished by implementing different execution states for the physical processor. The physical processor could be in either a multi-task (MT) state or one of two single-task (ST0 or ST1) states.

The physical processor is in the MT state when a thread is running on each of the logical processors that it supports. In the MT state, each logical processor shares the resources of the physical processor. The physical processor is in one of the single-task states when a thread is running on only one of the logical processors. As the logical processors are numbered 0 and 1, the physical processor is in the ST0 state when only a single thread is running and it is running on logical processor 0. In the ST0 and ST1 states, the active logical processor has access to all the resources of the physical processor.

The ability of the physical processor to be in different processing states brings to mind the definition of a load dependent server from network queuing theory. A load dependent server has a service rate that is dependent on the number of customers or queue length. The design of the hyper-threading processor suggests that a single, single-threaded program will run faster in single-task state than in multi-task state. The other side of this coin is that a set of programs or a single multi-threaded program should run faster in multi-task state with hyper-threading enabled than on a system with hyper-threading disabled.

The sharing of physical processor resources by logical processors suggests that there may be some queuing behavior internal to the physical processor. Since the logical processors are visible to the operating system, it seems likely that this internal queuing time will be included in the measured busy time for the logical processor from the perspective of the operating system.

Figure 1 is a block diagram that attempts to identify the components of the expected behavior when hyper-threading is enabled for a processor and multiple threads are active (MT state). The physical processor is divided into two logical processors (Logical Processor 0 and Logical Processor 1). Each logical processor has some resources it alone uses (Non-shared resources) and some resources that must be shared by both logical processors (Shared resources). The shared resources are controlled by one or more queues (Qx) within the physical processor.

Two programs are running in this system (Program 1 and Program 2). The operating system sees two processors and has a queue for them (Q1). The operating system is able to measure the elapsed time for each program. The elapsed time includes the time spent waiting in an operating system queue (Q1) and time spent using the processor. From the operating system's perspective, the CPU time is measured by how long the logical processor is being used. The logical processor is in use when the Non-shared resources are used by the program, when the Shared resources are used by the program, and when the program needs to wait in an internal queue for the Shared resources (Qx). Thus, the CPU time measurement for one program in MT state has a variable component, time waiting in Qx, that is influenced by the activity of another program.

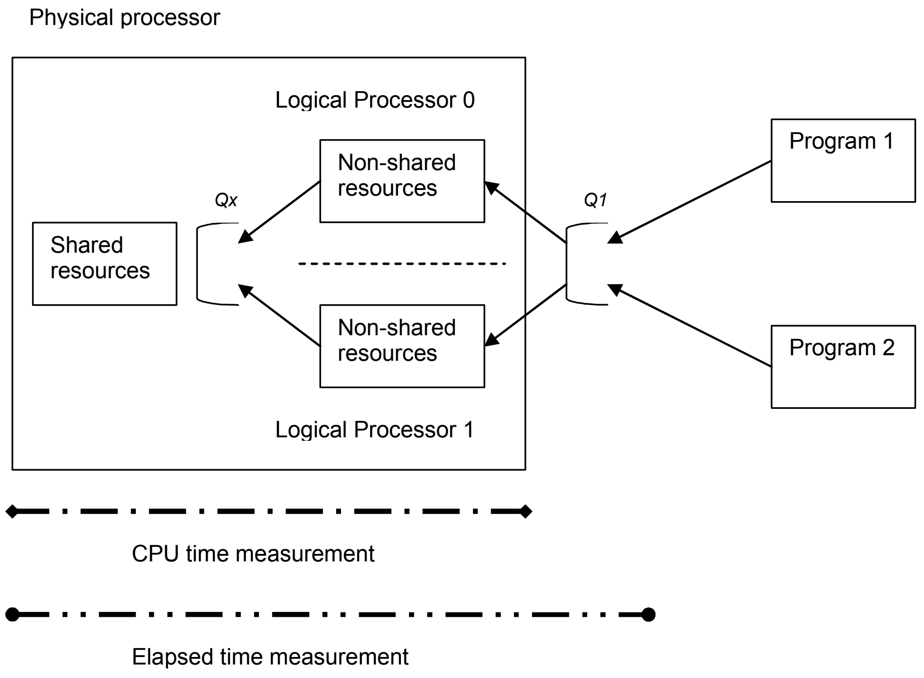

Figure 1: Hyper-threading enabled (MT state)

Figure 2 is a block diagram of the expected behavior when hyper-threading is disabled for a physical processor. The same types of resources exist within the chip. However, since only one thread can be active, there is effectively no queuing for the Shared resources. Again, two programs are running in this system (Program 1 and Program 2). The operating system sees one processor and has a queue for it (Q1). The operating system is able to measure both elapsed time and CPU time as before. In this case, there is no variable CPU time component since there is no contention for internal Non-shared resources.

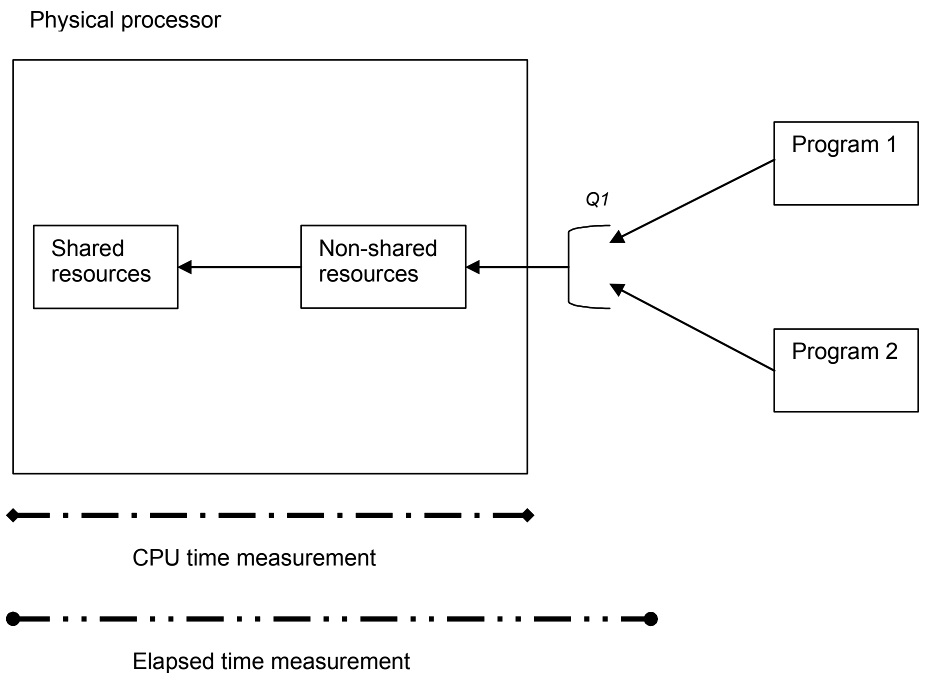

Figure 2: Hyper-threading disabled

This paper attempts to answer the following questions:

1. On a processor with hyper-threading enabled, does a single program run faster in single-task state than in multi-task state?
2. How is processor time measured for a program on a processor with hyper-threading enabled? Is it dependent on the number of other programs active on the system?

2. Methods

A simple test program to consume all available CPU cycles was constructed. The source for this program is included in the appendix. The program is configurable for the number of threads it creates and the fixed amount of work each thread performs.

The test program provides three measurements – elapsed time, kernel processor time, and user processor time. Elapsed time is measured using the time() function. This function returns values at the granularity of one second. Kernel and user processor time are measured using the GetProcessTimes() function. This function returns values at the granularity of 100 nanoseconds. The values returned by the GetProcessTimes() function include the activity of all threads created by the process.

The test system was a dual processor Compaq ML530 with 2.4GHz Intel® Xeon™ processors. The system was running Microsoft Windows 2000 Server Service Pack 3. A BIOS utility provided the ability to enable and disable the hyper-threading capability.

Two test scenarios were developed. The first test scenario ran one instance of the test program and varied the number of threads created by the test program. The number of threads tested was 1, 2, 3, 4, 8, and 16. The sequence of test cases was run on the system with hyper-threading enabled and with hyper-threading disabled.

The second test scenario varied the number of instances of the test program. Each instance of the test program created only one thread. The number of instances was 1, 2, 3, 4, and 8. The sequence of test cases was run on the system with hyper-threading enabled and with hyper-threading disabled.

3. Results

The tests were run on a nearly idle system. Other than the test programs, only normal operating system processes and a performance data collector were running.

Table 1 shows the results from the first test scenario. The results from the test cases with hyper-threading disabled are consistent with what one would expect from a two processor system.

- The average CPU time per thread was nearly the same in all cases. It ranged from 211.1 seconds to 213.4 seconds – a difference of about 1%.
- In the test cases for one and two threads, the elapsed time was nearly equal to the average CPU time per thread. Each thread was using one

processor with little or no contention from other activity on the system.

- In the test case for four threads, the elapsed time was about twice the average CPU time per thread. In this case, one would expect the average behavior to be one thread executing on each processor and one thread waiting for each processor. Thus one would expect the elapsed time to be very close to equation (1).

```
(1) Elapsed time =
    (number of threads
     / number of physical processors)
    * average CPU time per thread
```

  This relationship holds for the test cases with eight and sixteen threads as well.

The results from the test cases with hyper-threading enabled show the following:

- The average CPU time per thread was not the same in all test cases. It increased from the one-thread case to the four-thread case. This increase would be consistent with the behavior of the physical processors running in single-task state for one or two threads and in multi-task state for four or more threads.
- In the test cases for one, two, three, and four threads, the elapsed time was nearly equal to the average CPU time per thread. Since the operating system now sees four processors, this would be the expected behavior. Each thread would have access to a logical processor with little or no competition from other activity on the system.
- In comparing the elapsed times from the one- and two-thread case with the four-thread case, the results show an increase of 37% and 30% rather than the expected 100%. This behavior suggests that this workload was able to take advantage of the additional physical processor resources available when hyper-threading was enabled and operating in multi-task state.
- In the test case for eight threads, the elapsed time was about twice the average CPU time per thread. In this case, one would expect the average behavior to be one thread executing on each processor and one thread waiting for each processor. Thus, one would expect the elapsed time to be very close to equation (2).

```
(2) Elapsed time =
    (number of threads
      / number of logical processors)
    * average CPU time per thread
```

  This relationship holds for the test case with sixteen threads as well.

There are some additional observations worth noting in regard to the first test scenario.

- One would expect the elapsed times for the one-thread and two-thread test cases to be very similar when comparing hyper-threading enabled and hyper-threading disabled. While the elapsed times for the two thread case are within 2% of each other, the elapsed times for the one thread case show about a 5% difference.
- With hyper-threading enabled, the amount of processor time required to complete the fixed number of instructions in the test code varies with the level of activity as shown in the test cases for one, two, three, and four threads. This variability may be due to the logical processors competing for shared physical processor resources.

	Hyper-threading enabled				Hyper-threading disabled			
Number of threads	Elapsed time (sec.)	Total kernel CPU time (sec.)	Total user CPU time (sec.)	Average CPU time per thread (sec.)	Elapsed time (sec.)	Total kernel CPU time (sec.)	Total user CPU time (sec.)	Average CPU time per thread (sec.)
1	201	0.0	201.6	201.6	211	0.0	211.1	211.1
2	212	0.0	425.5	212.8	215	0.0	426.5	213.3
3	250	0.1	728.5	242.9	323	0.0	640.1	213.4
4	276	0.1	1092.9	273.3	428	0.0	852.4	213.1
8	550	0.2	2184.3	273.1	856	0.0	1705.9	213.2
16	1096	0.1	4372.7	273.3	1707	0.0	3406.3	212.9

Table 1: Results from first test scenario

Table 2 shows the results from the second test scenario. The results from the test cases with hyper-threading disabled are very similar to the results from the first test scenario. There appears to be little difference in the behavior between a single process with multiple threads and multiple single-thread processes.

The results from the second test scenario test cases with hyper-threading enabled show some of the same behaviors as the first test scenario:

- The average CPU time per process was not the same in all test cases. It increased from the one-process case to the four-process case.
- In the test cases for one, two, three, and four processes, the elapsed time was nearly equal to the average CPU time per thread.
- In the test case for eight processes, the elapsed time was about twice the average CPU time per process.

However, there was a notable difference in this test scenario. In test cases for two, four, and eight processes, the elapsed times were noticeably higher with hyper-threading enabled than with hyper-threading disabled.

	Hyper-threading enabled				Hyper-threading disabled			
Number of processes	Maximum elapsed time (sec.)	Total kernel CPU time (sec.)	Total user CPU time (sec.)	Average CPU time per process (sec.)	Maximum elapsed time (sec.)	Total kernel CPU time (sec.)	Total user CPU time (sec.)	Average CPU time per process (sec.)
1	201	0.0	201.6	201.6	213	0.0	212.2	212.2
2	245	0.0	489.8	244.9	216	0.0	426.7	213.4
3	327	0.0	968.0	322.7	323	0.0	639.6	213.2
4	474	0.0	1882.7	470.7	424	0.0	851.8	213.0
8	940	0.0	3749.1	468.6	827	0.0	1704.1	213.0

Table 2: Results from second test scenario

4. Conclusions

The test program used in this experiment provided a fixed amount of work for one or more threads created by the program. The results of the test scenarios provide some insight into the questions posed in the introduction.

1. The test cases showed a measurable difference in behavior for the hyper-threading enabled processors comparing single-task state to multi-task state. In single-task state (test cases for one or two threads and one or two processes), the fixed amount of work completed faster than in multi-task state (test cases with four or more threads or processes).
2. With hyper-threading enabled, the operating system measures what it sees – the logical processors. The test results show that processor time measurements for a fixed amount of work depend on the level of activity for each logical processor. The level of activity determines whether the physical processor operates in single-task state or multi-task state. When in multi-task state, wait time due to the contention of logical processors for shared physical processor resources appears to be included in the measurement of logical processor time.

The results suggest that the amount of work that an application can complete in a second of logical processor time is not consistent in an environment with hyper-threading enabled due to the different amount of processor resources available which is determined by the execution state. This has some interesting implications from the perspective of chargeback and capacity planning.

Suppose that a multi-threaded application runs on a system where processor time is one of the measurements used for chargeback. The least expensive option to get work done is to run the application with the same number of threads as physical processors and during a time when there is no other activity on the system (single-task state). However, this alternative may not get the work done as quickly as when the application uses the same number of threads as logical processors (multi-task state).

Many capacity planning activities are based on knowing the profile of resource usage for a transaction. The profile usually includes information such as the number of seconds of processor time and the number of seconds of I/O device time. Since a second of logical processor time in the hyper-threading enabled environment does not necessarily represent a consistent amount of work, the capacity planning process becomes a bit more complicated.

In summary, hyper-threading processors seem to behave as advertised. As with many new technologies, the amount of benefit will vary with specific workloads. Similarly, the process of reaping that benefit may introduce some additional complexity for those of us that must manage that technology.

5. References

[BORO02] John Borozan, "Microsoft Windows-Based Servers and Intel Hyper-Threading Technology", Microsoft Corporation, April, 2002.

[LENG02] Tau Leng, et. al., "A Study of Hyper-Threading in High-Performance Clusters", Dell Power Solutions, November, 2002.

[MAGR02] William Magro, Paul Petersen, Sanjiv Shah, "Hyper-Threading Technology: Impact on Compute-Intensive Workloads", Intel Technology Journal Q1, 2002.

[MARR02] Deborah T. Marr, et. al., "Hyper-Threading Technology Architecture and Microarchitecture", Intel Technology Journal Q1, 2002.

6. Trademarks

Appendix – Test Program

```
// ****************************************************************
// CPULoopT
//
// The purpose of this program is to sit in a loop and chew up
// CPU time. Two command line options are provided by this program:
//
// /l <# loops>
//   Specifies the number of outer loops for the program.
//   One outer loop uses about 1 second of CPU time on a
//   1.0GHz Pentium III.
//
// /t <# threads>
//   Specifies the number of threads that are created to run the
//   looping procedure.
//
// /o <filename>
//   Specifies the name of a file to write measurement results to.
//
// Upon completion, the program displays the following
// measurements:
//
// Elapsed seconds
//    Elapsed time that the program has run. Computed using the
//    time function. Measured to the nearest second.
//
// Kernel seconds
//    Seconds of kernel (privileged) CPU time used by the program.
//    Obtained from the GetProcessTimes function. Granularity of
//    function values is 100 nanoseconds. This includes kernel CPU
//    time for all threads created by the program.
//
// User seconds
//    Seconds of user CPU time used by the program. Obtained from
//    the GetProcessTimes function.  Granularity of function values
//    is 100 nanoseconds. This includes user CPU time for all threads
//    created by the program.
//
// Example:
//
// To run the program and specify 20 loops:
//
// cpuloop2 /l 20
// ****************************************************************

#include <afxmt.h>          // MFC Multithreading Support
#include "stdio.h"
#include "time.h"
#include "windows.h"
#include "malloc.h"

#define MT_MAX_THREAD   32  // max number of threads


int  ThreadNo    = 0;       // instance of thread giving output
int  ThreadCount = 1;       // user specified thread count
int  LoopCount   = 1;       // user specified loop count
FILE *OutFile    = stdout;  // output file
```

```
int parse_options (int argc, char* argv[])
   {
   int  i;
   char FileName[80];

   // loop through the command line options

   if (argc > 1)
      {
      for (i=1; i<argc; i++)
         {
         if ((stricmp(argv[i], "/l") == 0) || (stricmp(argv[i], "-l") == 0))
            {
            LoopCount = atoi(argv[++i]);
            }
         else if ((stricmp(argv[i], "/t") == 0) || (stricmp(argv[i], "-t") == 0))
            {
            ThreadCount = atoi(argv[++i]);
            if (ThreadCount > MT_MAX_THREAD)
               ThreadCount = MT_MAX_THREAD;
            }
         else if ((stricmp(argv[i], "/o") == 0) || (stricmp(argv[i], "-o") == 0))
            {
            strcpy(FileName, argv[++i]);
            OutFile = fopen(FileName, "w");
            }
         }
      }

   return 0;
   }

UINT ThreadProc(LPVOID)
   {
   double i, j, k, m;
   int m_ThreadNo = ++ThreadNo;

   // burn some CPU

   for (m = 1; m <= LoopCount; m++)
      for (i = 1; i <= 102; i++)
         for (j = 1; j <= 1000000; j++)
            {
            k = j + 1;
            }

   fprintf(OutFile,"     thread %d complete\n", m_ThreadNo);
   return 0;
   }



int main(int argc, char* argv[])
   {
   DWORD         dwThread[ MT_MAX_THREAD ];  // Thread ID
   HANDLE        hThreads[ MT_MAX_THREAD ];  // Thread Handle
   FILETIME      CreationTime, ExitTime, KernelTime, UserTime;
   LARGE_INTEGER UserTime3, KernelTime3;
```

```
double        ElapsedTime, UserSeconds, KernelSeconds;
time_t        ElapsedStart, ElapsedEnd, LocalTime;

time(&ElapsedStart);

parse_options(argc, argv);

fprintf(OutFile, "CPU Looper started %s", ctime(&ElapsedStart));

// Spawn Threads

for( int i = 0; i < ThreadCount; i++ )
   {
   hThreads[i] = CreateThread( NULL,
                               0,
                               (LPTHREAD_START_ROUTINE)ThreadProc,
                               NULL,
                               0,
                               &dwThread[ i ] );
   }

fprintf(OutFile, "%4d threads have been started\n", ThreadCount);
fprintf(OutFile, "%4d loops\n", LoopCount);

// Wait for threads to finish, then clean up

WaitForMultipleObjects( (DWORD) ThreadCount,
                        hThreads,
                        TRUE,
                        (DWORD) INFINITE );

for( i = 0; i < MT_MAX_THREAD; i++ )
   {
   CloseHandle(hThreads[i]);
   }

time(&ElapsedEnd);

/* get cpu time */

GetProcessTimes( GetCurrentProcess() ,
                 &CreationTime,
                 &ExitTime,
                 &KernelTime,
                 &UserTime );

// convert cpu time to seconds

UserTime3.LowPart = UserTime.dwLowDateTime;
UserTime3.HighPart = UserTime.dwHighDateTime;
KernelTime3.LowPart = KernelTime.dwLowDateTime;
KernelTime3.HighPart = KernelTime.dwHighDateTime;

UserSeconds = ((double)UserTime3.QuadPart) / 10000000.0;
KernelSeconds = ((double)KernelTime3.QuadPart) / 10000000.0;

// calculate elapsed time

ElapsedTime = (double)(ElapsedEnd - ElapsedStart);
```

```
// display results

fprintf(OutFile, "%4.0f elapsed seconds\n", ElapsedTime);
fprintf(OutFile, "%4.1f kernel seconds\n", KernelSeconds);
fprintf(OutFile, "%4.1f user seconds\n", UserSeconds);
time(&LocalTime);
fprintf(OutFile, "CPU Looper ended %s", ctime(&LocalTime));

return 0;
}
```