Design, Evaluation, and Implementation of an Accurate Benchmarking Methodology for Multi-Core Processors Based Systems

M. Hasan Jamal
December 2008

High Performance Computing and Networking Lab
Al-Khawarizmi Institute of Computer Science,
University of Engineering and Technology,
Lahore, PAKISTAN
Tel: +92-42-6841664
http://www.kics.edu.pk/hpcnl/
© 2008 M. Hasan Jamal

This technical report is based on a dissertation submitted December 2008 by the author for the Masters degree to The University of Engineering and Technology, Lahore, Pakistan.

Technical reports published by the High Performance Computing and Networking Lab, KICS, UET are freely available via the Internet:

http://www.kics.edu.pk/hpcnl/publications.php
Design, Evaluation, and Implementation of an Accurate Benchmarking Methodology for Multi-Core Processors Based Systems

Mohammad Hasan Jamal
December 2008

Keywords: performance benchmarking, performance evaluation, multi-core, multiprocessors, memory latency, memory throughput, execution time measurement, concurrent symmetric tasks.

ABSTRACT

The emergence of multi-core architectures has lead to the challenge of its efficient utilization. A comprehensive understanding of performance issues for these architectures is required to materialize any performance gains.

This research conducts measurement based performance benchmarking and evaluation for multi-core processors based systems. It comprises of four phases: (1) design of an accurate benchmarking methodology for multi-core processors based systems; (2) statistical comparisons and analyses of accuracy of this methodology with existing techniques; (3) development of selected benchmarks using this methodology; and (4) measurement based studies, which are driven by the workload generated through developed benchmarks.

This research work extends the state-of-the-art in existing benchmarks by improving their accuracy for benchmarking multi-core processors based systems. Results generated as an outcome of this research effort lends a greater insight into the interaction of multi-core architecture with system and user level software. Such insight is essential to design efficient parallel applications as well as to tune existing applications for multi-core architectures.

1 This research work is funded by National ICT R&D Fund, Ministry of Information and Technology, Pakistan. This work was completed through an internship from Cisco-Pakistan.
# TABLE OF CONTENTS

LIST OF FIGURES ......................................................................................................................v
LIST OF TABLES ............................................................................................................................vii

Chapter-1 INTRODUCTION ........................................................................................................1

Chapter-2 LITERATURE REVIEW .............................................................................................6

Chapter-3 BENCHMARKING METHODOLOGY .......................................................................10

3.1. Execution Time Measurement .........................................................................................10
3.2. Global Execution Time Estimation (GETE) Algorithm ....................................................11
3.3. Local Execution Time Estimation (LETE) Algorithm .......................................................12
3.4. Timing System Call Overhead ........................................................................................14
3.5. Clock Resolution ............................................................................................................14
3.6. Loop Overhead ..............................................................................................................15
3.7. Minimum Measurable Duration of a Task .......................................................................16
3.8. Thread Affinity ...............................................................................................................16
3.9. Synchronization .............................................................................................................17
3.10. k-NN Clustering ............................................................................................................18

Chapter-4 MEMORY PERFORMANCE BENCHMARKING ..................................................20

4.1. Experimental Setup .........................................................................................................20
4.2. SUT Architecture Overview ............................................................................................22
4.2.1. Intel based Architecture .............................................................................................22
4.2.2. AMD based Architecture ............................................................................................23
4.3. Experimental Design ........................................................................................................23
4.4. Validation of Measurements ...........................................................................................28
4.5. Comparison of GETE and LETE Algorithms based Methodologies ..............................28
4.6. Comparison of Precision ................................................................................................31

Chapter-5 Conclusion and Future Work ....................................................................................34

References ................................................................................................................................36
LIST OF FIGURES

Figure 1. Sequential and concurrent measurements of the execution times of a task specified in procedure(). Concurrent measurement of the same tasks across n symmetric threads can be conducted globally or locally within each thread. ..........................................................3

Figure 2. Global execution time estimation algorithm for a symmetric and concurrent task with customized barrier implementation. ...........................................................................11

Figure 3. Global Timing Methodology. ...................................................................................12

Figure 4. Local execution time estimation (LETE) algorithm. ...............................................12

Figure 5. Elapsed time for each thread per iteration using data size 1 MBytes. .....................13

Figure 6. The MINMAX approach. Minimum value from M maximum elapsed times will be picked after M rounds complete. ...........................................................................................13

Figure 7. Pseudo code to determine timing system call overhead. ........................................14

Figure 8. Code skeleton for estimating clock resolution. .......................................................15

Figure 9. Algorithm to determine loop overhead. ....................................................................16

Figure 10. (a) Histogram of a sample of 100 execution times showing an outlier using LETE algorithm for data size 16 Kbytes. (b) Data sample with mean M (c) Data sample after k-NN clustering method, green data points are considered to be outliers. ..................18

Figure 11. kNN clustering algorithm. .......................................................................................19

Figure 12. Pseudo code of memory copy benchmark. ............................................................21

Figure 13. Internal Architecture of Intel Xeon Quad Core Processor ...................................23

Figure 14. Internal Architecture of AMD Opteron Dual Core Processor .............................23

Figure 15. Result of 2k factorial design with six factors and their contributions to the execution time measurement on Intel based SUT. .................................................................24
Figure 16. Result of 2k factorial design with six factors and their contributions to the execution time measurement on AMD based SUT. ...............................................................25

Figure 17. Result of 2k factorial design with three factors and their contributions to the execution time measurement on Intel based SUT. .................................................................25

Figure 18. Result of 2k factorial design with three factors and their contributions to the execution time measurement on AMD based SUT. ...............................................................26

Figure 19. Result of 2k factorial experiment design of two factors and their contributions on Intel based SUT. ......................................................................................................................26

Figure 20. Result of 2k factorial experiment design of two factors and their contributions on AMD based SUT. ....................................................................................................................27

Figure 21. Result of 2k factorial experiment design of two factors and their contributions on Intel based SUT. ......................................................................................................................27

Figure 22. Result of 2k factorial experiment design of two factors and their contributions on AMD based SUT. ....................................................................................................................27

Figure 23. Memory throughput in Gbps across number of threads for data type double for different sizes of data on Intel based SUT. ..............................................................................29

Figure 24. Memory throughput in Gbps across number of threads for data type double for different sizes of data on AMD based SUT. ..............................................................................30

Figure 25. Histograms of execution time samples for memory-memory copy operations by 8 concurrent threads for different sizes of data on Intel based SUT. .............................32

Figure 26. Histograms of execution time samples for memory-memory copy operations by 4 concurrent threads for different sizes of data on AMD based SUT. .............................33
LIST OF TABLES

Table 1. Specifications of Systems under Test (SUTs) ..............................................................22

Table 2. Parameterization of steps of execution time measurement methodology on SUTs ........................................................................................................................................22

Table 3. Throughput in Gbps of memory-to-memory copy of 16 MB floating point data on SUTs ........................................................................................................................................28
CHAPTER-1

INTRODUCTION

In recent past, micro-architecture level performance improvement of single core processors was possible due to instruction level parallelism and higher clock rates. This was a consequence of Moore’s law, which says that the number of transistors on a chip doubles every 18 months. However, performance scaling has been diminishing lately due to excessive power consumption and resulting heat dissipation, wire delays, and memory access latencies.

Diminishing returns on performance scalability and increasing power consumption have led to the introduction of multi-core processor architectures featuring reduced clock rate, power consumption, and heat dissipation. With the development of multi-core technology, Moore’s law still holds true as the number of cores on a chip is increasing with efficient interconnects. The emergence of multi-core processor architectures has led to new challenges related to software performance. Software performance on traditional single-core processors used to improve with the introduction of new processors through higher clock rates without any effort by the programmers. However, this is no longer the case with multi-core processors because applications require parallelization and tuning to materialize any performance gains.

It is emphasized that the future is the multi-core technology where the number of cores on a chip will continue to grow according to Moore’s law. Growing number of cores will provide new challenges for memory bandwidth due to increased bus contention. Thus studying the behavior of memory accesses to avoid memory contention and to reduce latency is an active area of research [21], [22], [23], [24], [27], [39], [40].
To efficiently utilize the multi-core technology, there is a need for a comprehensive understanding of performance issues, such as identifying performance gaps between different subsystems, highlighting subsystems that create bottlenecks, and developing techniques to overcome these performance issues in user and system software. In general, four types of approaches are used in evaluating performance of multi-core architectures, which include: statistical modeling, machine simulation, measurements, and profiling.

This research performs measurement based performance evaluation for multi-core processors based systems. Typically, benchmarks are used for measurement-based evaluation of processor architectures and their impact on system and user level performance. For designing an accurate benchmarking methodology for multi-core processor based systems, precise measurement of execution time of concurrent, symmetric and short tasks plays a vital role. Measuring the elapsed time since the start of a task until its completion is a straightforward procedure in the context of a sequential task. In this case, the execution time is simply the difference between time stamps taken at the start and completion of the task. This procedure becomes complex when the same task is executed concurrently by \( n \) threads on \( n \) distinct processors or cores. In this case, it is not guaranteed that all tasks start at the same time or complete at the same time. Therefore, the measurement is imprecise due to concurrent nature of the tasks. To avoid overly complicating this scenario, we assume that all \( n \) threads are distinct and do not have any synchronization or locking before the execution of the symmetric task completes. At completion, each task joins the main thread. As shown in Figure 1, execution time can be measured either globally or locally. In the case of global measurement, execution time is equal to the difference of time stamps taken at global fork and join instants. Local times can be measured and recorded by each of the \( n \) threads and after joining, the maximum of all these individual execution times provides an estimate of overall execution time.

Regardless of using global or local approach, the execution time measurement is not precise and merely provides an estimate for global task execution time. Several factors contribute to the lack of precision:

1. Despite independent executions of the same code (symmetric and concurrent tasks), some tasks finish before others causing global measurements to represent the worst-case timing rather than precise timing;
2. The imprecise timing problem is exacerbated when symmetric task durations are comparable to the clock resolution;

3. With large number of cores or processors, a global clock may not exist to provide a global synchronization medium;

4. When tasks are executed repeatedly to obtain the desired confidence level for execution time measurement, the overhead of calling the timing function and indexing the loop could be comparable to the duration of the task itself (short task), which can make the measurement more imprecise; and

5. Due to hardware level resource conflicts, some timing results may be skewed due to outliers.

![Figure 1. Sequential and concurrent measurements of the execution times of a task specified in procedure().](image)

Precise and repeatable execution time measurement is essential for calculating cache, memory, and network protocol level throughputs and latencies. Such measurements provide the basis for resource management decisions, quality-of-service (QoS) guarantees, and service level agreement (SLA) monitoring. At a system level, precise and repeatable execution time measurements can be used for efficient scheduling of periodic real-time tasks. Above mentioned complexities persist, even if we use high resolution timers, as the non-deterministic nature of concurrent individual task completion times persists. In the case of multi-core systems, repetitive tasks also become concurrent using multiple threads or processes. Precise and repeatable task execution time measurement becomes a challenge in this scenario due to non-
deterministic execution paths of each symmetric and concurrent task. It is not possible to identify or force exact start and completion points for all of the tasks due to the concurrent nature of these tasks. This problem is exacerbated when task durations are close to clock resolution. Examples of such tasks include cache and memory accesses. Thus, time measurements in these scenarios are only estimates with limited precision. In case of multi-core architectures, suitable benchmarks are not available. Therefore, this thesis work emphasizes the need for an accurate benchmarking methodology to evaluate the performance of multi-core-processor-based systems, design such methodology and implement this methodology by extending existing benchmarks and developing new benchmarks. The developed benchmarks are validated with existing benchmarks.

Multi-threaded parallel programming paradigm is widely used on Symmetric Multiprocessor (SMP) and Chip Multiprocessor (CMP) architectures, which is an extension of shared memory multiprogramming using POSIX Threads [16]. In this paradigm, a process runs multiple threads, which execute symmetric tasks in a fork-and-join fashion. These threads use multiple available processors or cores, share the address space, and control access to the shared memory through locks. This programming paradigm is appropriate for developing benchmarks for multi-core-processors-based systems. For benchmarking applications, multiple threads execute symmetric tasks without any data sharing and lock-based synchronization. Almost all existing benchmarks implement the global time algorithm depicted in Figure 1, using C-library timing functions (such as gettimeofday()) for portability. Different timing functions provide multiple levels of portability and measurement resolution. We can use any one of the three available timing functions: (1) clock_gettime function from POSIX real-time library; (2) gettimeofday from C-library; or (3) Read Time Stamp Counter (RDTSC) x86 processor instruction with highest possible clock resolution. Our benchmarking methodology does not depend on the choice of time measurement function employed. Even with highest time measurement resolution, the complexities of precise estimation of concurrent and symmetric task execution time persist.

In addition to memory and bus bandwidth issues, a global clock with increasing number of cores may not be practical. This can further complicate the execution time measurement problem for symmetric and concurrent tasks. Our methodology does not depend on a global clock and can utilize a clock that is local to a processor or core.
In Chapter-2, we discuss the approaches of performance evaluation and related work on performance evaluation of single as well as multi-core architectures. In Chapter-3, we describe our accurate benchmarking methodology. Chapter-4 presents a case study of measuring memory subsystem performance of a multi-core-processor-based system using existing and our benchmarking methodology to compare their precision and repeatability. We conclude in Chapter-5 with a discussion of the scope and future directions of this research.
CHAPTER-2

LITERATURE REVIEW

This section reviews some of the existing literature on performance studies related to multi-core processor based systems. Generally four types of approaches are used for evaluating the performance of a system under test (SUT): (1) analytical modeling; (2) machine simulation; (3) measurement based evaluation and (4) on-chip counters based profiling.

Analytical modeling based evaluation contributes to the understanding of the complexity and impact of architecture on system performance. Chandra et al. [5] propose an inductive probability performance model to predict the impact of L2 cache sharing on the scheduled threads. Kim et al. [18] study the fairness in cache sharing between threads in multi-core architecture, propose static and dynamic L2 cache partitioning algorithms that optimize fairness, and analyze the relationship between fairness and throughput. Petoumenos et al [29] propose an analytical model of a Chip Multiprocessor (CMP) shared cache, and a fine-grained mechanism to manage such caches.

Ding and Kennedy [8] observe that application performance is now primarily limited by memory bandwidth. They describe an approach based on estimating and monitoring of memory bandwidth consumption, which results in achieving accurate and efficient performance, by tuning the source code and compile time prediction. In another paper [9], they discuss the improvement of cache performance in dynamic applications through reorganization of data and computation at run time, and further define bandwidth based performance model that helps demonstrate the serious performance bottleneck due to the lack of memory bandwidth. In another study [10], they describe a new set of compiler optimizations for reducing bandwidth...
consumption of programs. Donald and Martonosi [11] present a programming methodology for converting existing uniprocessor simulators into parallelized multiple-core simulators. Goldberg and Hennessy [13] present a tool for Performance Debugging of Shared Memory Multiprocessor Applications. Analytical modeling does not directly depend on precise measurement of execution times. However, workload characterization, which drives analytical calculations, can utilize precise timing to accurately parameterize the analytical models.

Machine simulation and compiler based tools are also employed for architecture level performance evaluation. Burger et al. [4] present the SimpleScalar tool set, which is a collection of publicly-available simulation tools for modern processor architectures. Chang et al [6] present simulation of Cooperative Caching for CMPs, which is a combination of the strengths of private and shared cache organizations. The results show that cooperative caching schemes improve the performance of multithreaded commercial workloads compared to shared and private cache organizations. Event-driven simulation is based on advancing a global clock in response to multiple tasks utilizing shared resources. Thus, execution times are always precise and repeatable.

Measurement based performance benchmarking and characterization of processor architectures is a mature area of research. Benchmarks are obtained by mimicking a particular type of workload on a component or entire system. Microbenchmarks measure performance of specific hardware subsystems or software modules. Microbenchmarks are characterized by symmetric and repetitive tasks. Precise execution time measurement is critical for benchmarks to report repeatable and useful results.

Several micro-benchmarks are available for measuring the performance of memory subsystems. STREAM (see [21] and [22]) is a well known microbenchmark, which measures sustainable memory throughput and the corresponding computation rate on uniprocessors, vector processors, and shared and distributed memory systems. However it is a single-threaded microbenchmark, which is limited to unit stride vector kernel, and exercises a single data type on dataset larger than cache sizes to measure memory bandwidth for uncached accesses. STREAM2 [23] microbenchmark, an extension of STREAM microbenchmark, measures sustained bandwidth at all levels of the cache hierarchy but is limited to a single data type with unit stride vector kernel. LMBench [24] is a portable microbenchmark suite intended to evaluate OS
performance, which provides a subset of tests that evaluate memory bandwidth and latency. Hbench:OS [1] is a revised version of LMBench, which provides a detailed analysis and decomposition of the performance of the operating system primitives. Cachebench [27] evaluates and parameterizes the performance multiple level of caches present on and off the processor in terms of bandwidth for a unit stride vector kernel but is designed for uniprocessor systems. C2CBench (see [39] and [40]) evaluates the performance of different levels of cache hierarchy and is based on a runtime system that uses the Producer-Consumer execution model, using different block and stride sizes of data. HPC Challenge benchmark suite [15], [20], designed to help define the performance boundaries of future Petascale computing systems, examine the performance of HPC architectures using kernels with memory access patterns. The suite is composed of several benchmarks consisting of STREAM, RandomAccess [38], and bandwidth/latency tests (b-eff) [19], [30], [31], that measure systems memory performance. RandomAccess measures the rate of integer updates to random memory locations. Latency/Bandwidth tests measure communication patterns of increasing complexity between as many nodes as is time-wise feasible. SPEC benchmarks [34] are commonly used to evaluate performance for commercial computer systems today. They consist of benchmarks that are more sensitive to memory bandwidth and latency but these benchmarks are not specifically designed for performance evaluation of the memory system. Apex-map (see [1], [35], [36], and [37]) measures global data access performance. Apex-map is designed based on parameterized concepts for temporal and spatial locality and generates a global data access stream according to specified levels of these measures of locality. Apex-map allows fine grained and highly automated analysis and profiling of a memory subsystem. It allows random access streams as well as regular access streams. A. B. Yoo et al. present memory benchmarks for SMP based high performance parallel computers that evaluate the performance of different level of memory hierarchy varying vector strides [43]. None of these benchmarks has to deal with the problem of measuring execution time across multiple concurrent tasks.

Hardware counters based performance profiling tools are also used for accurate performance measurements of applications and characterization of system architecture. Hardware counters are used to measure processor events, such as clock ticks, cache misses, TLB misses, bus utilization and conflicts, and branch mispredictions. Profiling tools tie these low-level measurements to user and system level
code to allow architecture-aware performance evaluation and characterization. Some existing processor architecture level performance profiling and monitoring tools are Intel VTune profiler, PAPI [3] and PCL [28].

Analytical modeling based approach provides efficient performance estimates but it lacks realistic performance of systems because they rely on probabilistic assumptions. Similarly simulation is not always suitable as it increases the cost with its capability of in-depth evaluation. Additionally, it is not always portable due to its architecture dependencies. Measurement based studies are most suitable in achieving realistic performance evaluation of any system because they reflect real testing environment.

Typically, benchmarks are used for measurement-based evaluation of processor architectures and their impact on system and user level performance. Some existing benchmarking efforts are mentioned in the previous section; however, in case of multi-core architectures, suitable benchmarks are not available. Therefore, we extend existing benchmarks and develop new ones to fill this gap. We use these multi-core benchmarks to characterize the performance of multi-core processor based systems.
This section describes our accurate benchmarking methodology. Following are the steps of our benchmarking methodology:

1. Compute timing system call overhead
2. Compute loop overhead
3. Compute minimum number of iterations required to measure the task execution time
4. Create thread and thread local data initialization
5. Implement Barrier to synchronize threads
6. Stamp the thread local start time
7. Benchmark the workload
8. Stamp the thread local end time
9. Repeat step 5-8 for N no. of iterations
10. Use the MINMAX approach to obtain execution time interval
11. Repeat step 1-10 to get 100 data samples
12. Use k-NN clustering method to get 60 out of 100 samples closest to the mean

In the following subsections, we discuss the above steps in detail. We also provide algorithmic level details to understand the implementation of the above MINMAX timing algorithm.

3.1. Execution Time Measurement

Execution time measurement of a sequential program is a simple operation. A task can be timed by stamping the time instants before start and after completion of its operation. However, timing in multiprocessing and multi-threading environments is
not that simple. General perception is that in a multithreaded application, the operations can be timed before thread creation and after thread joining. Although this process may appear simple, the measurements may not be accurate or repeatable. As an example, consider benchmarking memory performance where our requirement is to time the memory-to-memory data transfer operation. In order to perform this operation, we have to initialize the memory with some random thread-local data. In this example, if the time is stamped before fork and after join, the benchmarking results will be totally inaccurate. This is due to greater length of random initialization of data compared to relatively short duration of data transfer operation itself. Therefore, to achieve reliable results, it is necessary that we measure the execution time of data transfer operation alone.

We generalize the above discussion in the following two subsections and present two timing algorithms for symmetric and concurrent tasks: (1) Global Execution Time Estimation (GETE) algorithm; and (2) Local Execution Time Estimation (LETE) algorithm. GETE is a minor extension of commonly used global timing algorithm depicted in Figure 1 while LETE algorithm is the one proposed for our methodology.

3.2. **Global Execution Time Estimation (GETE) Algorithm**

![Figure 2. Global execution time estimation algorithm for a symmetric and concurrent task with customized barrier implementation.](image)

Figure 2 shows the pseudo code for GETE algorithm. This algorithm is used for known cases where symmetric concurrent task execution times are measured, such as microbenchmarks. In this algorithm, a global timer is used. The starting time is taken at the barrier (for synchronization of K threads) followed by the operations to be timed, for a given number of iterations. The ending time is taken at another barrier as shown in Figure 3. The time for a single iteration is obtained by dividing the total latency with the number of iterations. Multiple iterations are needed to ensure that overall execution time is comparatively larger than the clock resolution.
3.3. Local Execution Time Estimation (LETE) Algorithm

Figure 4 presents the LETE algorithm for symmetric and concurrent tasks. This algorithm works on the basis of multiple iterations of a symmetric task, which are repeated for M rounds. The threads are synchronized at the beginning of an iteration using a barrier after which the local starting times of the individual threads are measured locally. After the task execution, the local completion times of individual threads are measured as shown in Figure 4. Note that each thread will determine different task execution time to completion.

Thread_local_processing:

\[
\begin{align*}
    &k = \text{my\_thread\_id}; \\
    &\text{FOR each round } i \text{ out of } M \\
    &\quad \text{CALL Barrier} \\
    &\quad \text{CALL Get\_Clk RETURNING starttime} \\
    &\quad \quad \text{FOR each iteration } j \text{ out of } N \\
    &\quad \quad \quad \text{CALL procedure} \\
    &\quad \quad \quad \quad \text{CALL Get\_Clk RETURNING endtime} \\
    &\quad \quad \quad \quad A[k][i] = (endtime - \text{starttime})/N \\
    &\quad \text{END FOR} \\
    &\text{Main\_thread\_MINMAX\_processing:} \\
    &\text{FOR each round } i \text{ out of } M \\
    &\quad B[i] = \text{max of } A[k][i] \text{ for all } k=0,\ldots,L-1 \\
    &\text{END FOR} \\
    &\text{LETE = min of } B[i] \text{ for all } i=0,\ldots, M-1 \\
    &\text{Return LETE}
\end{align*}
\]

We apply the LETE algorithm to determine the latency of memory-to-memory copy operation of 1 Mbytes of data, which is replicated across eight threads on a quad-core processor based system. It is observed that each thread takes a different time to
complete the memory operation as shown in Figure 5; local time measurements can greatly differ across threads and rounds of repetitions.

![Figure 5](image_url)  
*Figure 5. Elapsed time for each thread per iteration using data size 1 MBytes.*

Local execution time measurements are repeated for M rounds to determine the result with a high confidence level. Using this algorithm, we have a collection of (rounds * K) number of test latencies, where K is the number of threads. In order to determine the overall throughput of the system, a MINMAX approach is used, which is elaborated in Figure 6. The MINMAX approach picks the minimum latency across all rounds among the maximum latencies of individual threads from each round. The MINMAX is a global operation, which needs to be handled by the main thread.

![Figure 6](image_url)  
*Figure 6. The MINMAX approach. Minimum value from M maximum elapsed times will be picked after M rounds complete.*
3.4. Timing System Call Overhead

Often concurrent and symmetric tasks are observed to be of a very short duration relative to the clock resolution. In such cases, system call overhead to determine system time becomes comparable to the execution time. This system call overhead can be computed and subtracted from global or local time estimates for each timing system call to accurately measure the execution time. The pseudo code for determining the system call overhead extracted from [32] is shown in Figure 7.

```
Time stamping overhead:
No. of Iterations = M;
starttime = Get_Clk();
while (ndone < M)
{
ncalls++;
currenttime = Get_Clk();
diffs[ndone] = currenttime - starttime;
if (diffs[ndone] > 1.0E-9)
{
ndone++;
starttime = currenttime;
}
}
for (j = 0; j < M; j++)
sum += diffs[j];
Avr = total/M;
Return Avr;
```

Figure 7. Pseudo code to determine timing system call overhead.

3.5. Clock Resolution

The precision of the execution time measurements largely depends on the clock resolution. We determine the minimum number of iterations, which need to run long enough to be measured precisely. The pseudo code to determine the clock resolution is shown in Figure 8.

We perform register operations, which start with two iterations. The number of iterations increases exponentially at each step until the difference between the execution times of the register operations and time overhead becomes positive. The register operations are used because they are constrained to the CPU and are deterministic. They execute in one or more clock cycles and are repeatable. These operations are designed such that all the operations are performed in the CPU registers assuring no memory access. To avoid compiler optimization, logical operations are performed to force the compiler to execute every iteration.
Furthermore, the variables are printed using dummy functions so that the compiler does not skip the assignment operations.

```c
Unsigned char d = 0x1a;
register int N = 2, sum = 0, c, y = d;
time_overhead = time_stamping_overhead();
do
{
    starttime = Get_Clk();
    for (j=0; j<N; j++)
    {
        c = j;
        if (c == y)
            sum = sum + c;
        else
            y = y + 1;
    }
    endtime = Get_Clk() - starttime;
    N = N << 1;
}while (endtime - time_overhead < 0);
// to avoid compiler optimizations
Print_dummy(c, y, sum);
Minimum Iterations = N;
```

Figure 8. Code skeleton for estimating clock resolution.

To determine the clock resolution, LMBench [33] uses a compute_enough module that increases the timing interval until small variations in measured work produce small variations in measured time, reducing the timing error to less than one percent.

### 3.6. Loop Overhead

For some short duration tasks, the overhead of the “for” loop can be significant. The loop overhead is computed by the algorithm shown in Figure 9. Register operations are performed and the time stamping overhead is subtracted from its execution time. This step is repeated by varying the number of iterations.

After obtaining a set of iterations and their corresponding overhead, this dataset is used for curve fitting to achieve a linear equation of the form

\[ y = mx + c \quad (3.6.1) \]

where ‘m’ is the slope and ‘c’ is the intercept along y axis. If ‘c’ in this equation is not zero, it represents the time taken for the register operations and using this value loop overhead is calculated by subtracting this time from the overhead time computed in Figure 9.
This result can be validated by the technique used in LMBench [33], which uses two loops, the first with one instance of the register operations and the second with two instances of the register operations, giving two equations.

\[ T_1 = N(\text{loop\_overhead} + \text{register operations}) \]  
\[ T_2 = N(\text{loop\_overhead} + 2 \text{ register operations}) \]

In above equations, \( T_1 \) and \( T_2 \) are measured execution times and \( N \) is the number of loop iterations. The loop overhead can be computed as follows.

\[ \text{loop\_overhead} = \frac{2T_1}{N} - \frac{T_2}{N}. \]

### 3.7. Minimum Measurable Duration of a Task

It is observed that some tasks do not execute for substantial time intervals and in those cases timing and loop overheads have significant impact on the measurements. In our methodology, we define the lower limit of the duration of a task to be 10 times the minimum iteration time minus the overheads.

Minimum task duration = 10 x (Minimum iteration time - loop\_overhead - timing\_overhead).  

### 3.8. Thread Affinity

Concurrently executing symmetric tasks can migrate from one core to another after context switching. It is possible to instruct the kernel to schedule a particular thread...
on a specific core from available choices. Thread affinity ensures that unrelated latencies due to contention for shared L2 cache or bus among a group of cores does not impact measurements in an unexpected manner [41]. Thread affinity can thus improve the precision and repeatability of measurements. 

Thread affinity can also be employed in the case of using on-chip performance counters for measurements to match the counter values with a particular thread. The Read Time Stamp Counter (RDTSC) instruction for x86 architectures provides high resolution time measurements. However, there is a potential issue while using the RDTSC instruction on multi-core-processor-based systems. The clock cycle values are read from the local CPU register. If the timing function is called multiple times from a thread, it is not guaranteed that the value will be taken from the same core’s register due to context switching of a thread from one core to another. Furthermore, multiprocessor systems do not guarantee synchronization of their cycle counters between cores. This problem is exacerbated when combined with modern power management technologies that idle and restore various cores at different times, which results in the cores typically being out of synchronization. For an application, this generally results in glitches or potential crashes as the thread jumps between the processors and gets timing values that result in large deltas, negative deltas, or halted timing [42]. To overcome these issues, the threads are assigned affinities and are bound to run on a specific processor core.

Using thread affinity with the LETE algorithm is feasible as each thread is using its local clock, taking the value from the same processors register. However, with the GETE algorithm, timing a task becomes cumbersome when forcing the global timer to read the value from the same register, at the start and at the end.

### 3.9. Synchronization

Symmetric tasks may need to accomplish some initialization tasks prior to starting their main operation that needs to be timed. The initialization period is dependent on the problem being solved and its duration is likely to vary across threads due to the non-deterministic nature of execution paths at different cores. As a result some threads may still be initializing while the other threads start performing the required task. This leads to imprecise execution time measurements. To precisely measure the execution time, it is required that all the threads start their main task at the same time after initialization. Thus initial thread synchronization is necessary.
We use barrier synchronization in our methodology. When a thread arrives at a barrier, which is pre-defined to a specified count (equal to the number of threads) \( K \), it waits until \( K-1 \) other threads also reach that barrier. In the case of the GETE algorithm, at this stage the time is measured, after which the threads are signaled to perform their primary task. In the case of LETE algorithm, the barrier is used only for synchronizing the main task start times.

### 3.10. k-NN Clustering

Concurrent task execution time intervals are sometimes non-repeatable. Out of multiple experiments performed, some execution time measurements are significantly different than most of the other measurements. We consider such measurements as outliers. These outliers significantly impact the precision and repeatability of execution time measurement for concurrent tasks.

![Histogram of 100 Samples](image)

![Data Sample with Mean M](image)

![Data Sample after k-NN Clustering Method](image)

Figure 10. (a) Histogram of a sample of 100 execution times showing an outlier using LETE algorithm for data size 16 Kbytes. (b) Data sample with mean M (c) Data sample after k-NN clustering method, green data points are considered to be outliers.
Previous studies perform multiple experiments and take the average or median to achieve the final result. In case where outliers are observed in the measurements, such strategies cannot be considered robust. In addition, the outlier issue will become more severe with large number of concurrent and short symmetric tasks.

In our methodology, we implement a *k nearest neighbor* (k-NN) clustering technique [7] to eliminate outliers for robust and stable execution time measurement. Figure 10 (a) shows outliers from our test data sample while Figure 10 (b) and (c) provide a generic concept of an outlier.

A common rule of thumb in statistics is that for a sample size greater than 30, a 95% confidence interval should be achieved [25]. Therefore, we ensure one hundred repetitions of the concurrent task measurements, take the mean of the results, and select sixty outcomes out of the hundred samples that are closest to the mean. In this way the outliers in the results are removed. Our algorithm for eliminating the outliers is shown in Figure 11.

```plaintext
// No. of Samples = 100
Data D [No of Samples];

FOR each D i to No of Samples
    d[i] = D[i] - M(D);
END FOR

FOR each D i to No of Samples
    IF d[i] <0
        Negative [] ← d[i];
    ELSE
        Positive [] ← d[i];
    END IF
END FOR

SORT (Negative [],'descend');
SORT (Positive [],'ascend');
i = j = 1;

WHILE i+j <= ((No of samples in Cluster) + 1)
    IF Positive[i] < ABS (Negative[j])
        Cluster [] ← Positive[i];
        i=i+1;
    ELSE
        Cluster [] ← Negative[i];
        j=j+1;
    END IF
END WHILE

//Data clustered around the Mean
Cluster [] = Cluster [] + M (D);
```

Figure 11. kNN clustering algorithm.
In this section, we apply our MINMAX timing algorithm to determine the memory subsystem performance of a multi-core-processor-based system. Our objective is to quantitatively compare the precision of this algorithm with the simplistic global timing algorithm using a memory-to-memory copy of an array as a symmetric, concurrent, and short duration task executed through multiple threads. This case study is inspired from the STREAM benchmark [22], which uses sequential task-based timing measurements. In fact, we shall use the STREAM benchmark results as a reference for single thread-based usage of our MINMAX algorithm for validation. Figure 12 below provides the outline of memory-to-memory copy benchmark, which is implemented as a part of STREAM benchmark. We also use the same operations for this case study but employ GETE and LETE timing algorithms with our methodology.

4.1. Experimental Setup

The specifications of the two SUT used for our work are listed in Table 1. The first SUT is a quad core Intel Xeon E5405 processor based server with clock speed of 2 GHz, 1333 MHz Front Side Bus (FSB), 32 KB L1 instruction and data caches, and 2 x 6 MB L2 caches, as our first system under test (SUT). The SUT is equipped with two processors, making the total count of cores equal to eight with four 2 GB, DDR2 667MHz ECC, fully-buffered memory RAM modules. The system runs the Fedora Core 8 kernel 2.6.23.1-42.fc8 SMP operating system. GCC 4.1.2 is used for compiling all the codes using the -O3 optimization flag.
The second SUT is equipped with dual core AMD Opteron 2212HE processor with clock speed of 2 GHz, 64 KB L1 instruction and data caches, and 2 x 1 MB L2 caches. This SUT is equipped with two processors, making the total count of cores equal to four with four 2 GB, DDR2 667MHz ECC, fully buffered memory RAM modules. The system runs the Fedora Core 8 kernel 2.6.23.1-42.fc8 SMP operating system. GCC 4.1.2 is used for compiling all the codes using the–O3 optimization flag.
Table 1. Specifications of Systems under Test (SUTs)

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Intel Xeon (IX)</th>
<th>AMD Opteron (AO)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CPUs</td>
<td>2x4 cores</td>
<td>2x2 cores</td>
</tr>
<tr>
<td>CPU Speed</td>
<td>2.0 GHz</td>
<td>2.0 GHz</td>
</tr>
<tr>
<td>L1 D Cache</td>
<td>32 KB</td>
<td>64 KB</td>
</tr>
<tr>
<td>L1 I Cache</td>
<td>32 KB</td>
<td>64 KB</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>2 x (2 x 6) = 24 MB</td>
<td>2x(2x1) = 4 MB</td>
</tr>
<tr>
<td>DRAM Size</td>
<td>8 GB</td>
<td>8 GB</td>
</tr>
<tr>
<td>OS Version</td>
<td>2.6.23.1-42, Fedora core 8</td>
<td>2.6.23.1-42, Fedora core 8</td>
</tr>
<tr>
<td>Compiler</td>
<td>gcc 4.1.2, -O3</td>
<td>gcc 4.1.2, -O3</td>
</tr>
</tbody>
</table>

Table 2 parameterizes various timing attributes that our methodology takes into account for the SUT.

Table 2. Parameterization of steps of execution time measurement methodology on SUTs

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Intel</th>
<th>AMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time system call overhead</td>
<td>538 ns</td>
<td>1240 ns</td>
</tr>
<tr>
<td>Clock Resolution</td>
<td>Minimum Iteration (N) = 1024, Time = 1420 ns</td>
<td>Minimum Iteration (N) = 32, Time = 1830 ns</td>
</tr>
<tr>
<td>Loop overhead</td>
<td>y = 1.41x + 235.68</td>
<td>y = 1.02x + 543</td>
</tr>
<tr>
<td></td>
<td>(1420 – 538 – 236 = 646 ns)</td>
<td>(1830 – 1240 – 543 = 47 ns)</td>
</tr>
<tr>
<td>Minimum Measurable Duration of Task</td>
<td>10 * (1420 – 538 – 646) = 2.36 µs</td>
<td>10 * (1830 – 1240 – 47) = 5.43 us</td>
</tr>
<tr>
<td>Thread Affinity</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Synchronization Type</td>
<td>Barrier</td>
<td>Barrier</td>
</tr>
<tr>
<td>Clustering (data size = 16 KB)</td>
<td>Before Clustering: Mean = 1.1059 µs, Variance = 5.25e-04</td>
<td>Before Clustering: Mean = 1.671 µs, Variance = 1.72e-03</td>
</tr>
<tr>
<td></td>
<td>After Clustering: Mean = 1.107 µs, Variance = 8.05e-05</td>
<td>After Clustering: Mean = 1.659 µs, Variance = 1.14e-04</td>
</tr>
</tbody>
</table>

4.2. SUT Architecture Overview

4.2.1. Intel based Architecture

The block diagram of IX is shown in Figure 13. IX comprises of four cores on a single chip. Each core has a separate L1 data and instruction cache of size 32 KB each. The L2 caches of 6 MB each are shared between two cores. Each core has a clock speed of 2 GHz. The main memory is accessed using a shared system bus of speed 1333 MHz.
4.2.2. AMD based Architecture

The block diagram of AO is shown in Figure 14. AO comprises of two cores on a single chip. Each core has a separate L1 data and instruction cache of size of 64 KB each. Each core has a separate L2 cache of 1 MB. The clock speed of each core is 2 GHz. Separate low latency high bandwidth interconnects are used for I/O and memory operations to reduce bus contention. A memory controller is integrated into the processor to reduce the memory bandwidth bottleneck. It operates at the speed of processor core.

4.3. Experimental Design

A number of experimental factors can impact the execution time measurement in the context of memory benchmarking. These factors include: timing function, timing algorithm, number of threads, thread affinity, duration of symmetric and concurrent task, and level of compiler optimization (e.g. –O2 or –O3). These factors should be
varied to quantify the dependence of execution time measurement on these factors. However, all factors do not significantly contribute to the results. Therefore, it is logical to try to reduce the number of experimental factors. In addition to using qualitative arguments to try to eliminate certain experimental factors, rigorous experimental design techniques can be employed to guide us through this process. One such technique is 2k-factorial design technique [17].

In order to apply the 2k-factorial experiment design technique to our memory benchmarking case study, we vary the selected experimental factors between two levels that represent a range of their possible values. These factors constitute our workload for this benchmarking effort. The selected factors are, optimization level (-O1 or –O3), thread Affinity (Yes or No), number of threads (1 to Maximum no of cores), timing algorithm (LETE or GETE), data size (16 KB or 16 MB), and timing function (gettimeofday or clock_gettime).

![Figure 15. Result of 2k factorial design with six factors and their contributions to the execution time measurement on Intel based SUT.](image)

Figure 15 and Figure 16 show the result of analyzing the significance of these factors to execution time measurement using the 2k factorial design analysis on Intel and AMD based systems respectively. Clearly data size is the single most important factor that affects the throughput and latency, followed by the number of threads and the combined effect of data size and number of threads. An important observation is that the timing function does not have any impact on the result indicating that the resolution of the timing function does not have significant impact on the execution times measurements. Furthermore the significance of timing algorithm, thread affinity and level of optimization are observed to be negligible.
Figure 16. Result of 2k factorial design with six factors and their contributions to the execution time measurement on AMD based SUT.

Figure 17 and Figure 18 show the result of the 2k-factorial experiment design analysis without consideration of less significant factors timing function, timing algorithm and thread affinity on Intel and AMD based systems respectively. Data size is the single most important factor that affects the execution time measurements, followed by the number of threads and combined effect of data size and number of threads. An important observation is that optimization level does not have any significant impact on the results.

A = Optimization Level
B = Thread Affinity
C = No. of Threads
D = Timing Algorithm
E = Data Size
F = Timing Function

Figure 17. Result of 2k factorial design with three factors and their contributions to the execution time measurement on Intel based SUT.
To see the significance of the less significant factors, the factor that had major contribution to the results were fixed to see the significance of these less significant factors. Figure 19 and Figure 20 show the result of the 2k-factorial experiment design analysis taking into consideration the less significant factors timing function, and thread affinity on Intel and AMD based systems respectively. Thread affinity is the single most important factor that affects the execution time measurements.

A = Timing Function  
B = Thread Affinity
To further see the impact of thread affinity, its 2k-factorial experiment design analysis was performed with data size. Figure 21 and Figure 22 show the result of the 2k-factorial experiment design analysis taking into consideration thread affinity and data size on Intel and AMD based systems respectively. Data size is the important factor that affects the execution time measurements.

Figure 20. Result of 2k factorial experiment design of two factors and their contributions on AMD based SUT

Figure 21. Result of 2k factorial experiment design of two factors and their contributions on Intel based SUT

Figure 22. Result of 2k factorial experiment design of two factors and their contributions on AMD based SUT
From our results based on 2k-factorial experiment design and analysis, the most significant factors are data size and number of threads; hence they are used as the workload for analyzing the execution time measurement methodologies in the context of memory benchmarking for multi-core-processor-based systems. The less significant factors are fixed in our experiments. We use –O3 optimization level, thread affinity, LETE timing algorithm and clock_gettime timing function for all experiments.

4.4. Validation of Measurements

In order to validate our methodology, we compare the memory throughput measured by MINMAX methodology with STREAM benchmark results on our SUTs. We employ both GETE and LETE based timing algorithms and use a single thread based execution to measure the memory-to-memory copy throughput of 16 Mbytes of float arrays to mimic the STREAM benchmark approach. Table 3 shows memory-to-memory copy throughputs of the STREAM benchmark as well as those of the GETE and LETE timing-based benchmarks on our two SUTs. The similarity validates the results based on our methodology. The STREAM benchmark measures slightly higher throughput because it initializes the array elements to constant values (zeros) while our methodology initializes to random numbers. We believe that our approach of using random numbers instead of a constant does not provide the compiler with any room for optimization.

Table 3. Throughput in Gbps of memory-to-memory copy of 16 MB floating point data on SUTs.

<table>
<thead>
<tr>
<th>SUT</th>
<th>STREAM benchmark</th>
<th>GETE based benchmark</th>
<th>LETE based benchmark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel</td>
<td>27.905</td>
<td>26.434</td>
<td>26.728</td>
</tr>
<tr>
<td>AMD</td>
<td>16.23</td>
<td>15.793</td>
<td>15.769</td>
</tr>
</tbody>
</table>

4.5. Comparison of GETE and LETE Algorithms based Methodologies

With concurrent symmetric threads copying data between two thread-local arrays on each core of a multi-core processor, one expects to see the memory-to-memory throughput scale with the number of threads. Figure 23 and Figure 24 shows memory throughput versus number of threads for GETE and LETE execution time
measurement algorithms using double data type for various data sizes on Intel and AMD based servers respectively.

With data sizes of 16 KB, 128 KB and 1 MB, most of the memory accesses should hit L2 caches rather than the main memory. It is observed in Figure 23 (a), (b), and (c) and Figure 24 (a), (b), and (c) that the throughput scales with the number of threads and thus validates our expectation. In addition, LETE based methodology measures higher throughputs for these shorter tasks because it is not impaired by slower threads due to localized measurements.

![Figure 23](a) 16 KB  
![Figure 23](b) 128 KB  
![Figure 23](c) 1 MB  
![Figure 23](d) 16 MB

Figure 23. Memory throughput in Gbps across number of threads for data type double for different sizes of data on Intel based SUT.

Figure 23 (d), presents memory copy throughput for 16 MB of data size, which results in up to two orders of magnitude longer execution times compared to smaller data sizes on an Intel based SUT. In this case, memory copy throughput does not scale linearly with the number of threads. In contrast to data sizes of 16 KB, 128 KB, and 1 MB, which can fit in L2 caches, copying 16 MB require extensive memory accesses through shared bus in Intel based SUT. Thus, throughput is lower compared to the cases where accesses hit in L2 caches and saturates as the bus becomes a bottleneck. Memory copy throughput saturates at around 40 Gbps, which is half of the available
bus bandwidth (64 bits x 1333 MHz = 85.3 Gbps). Furthermore, throughput is constrained due to shared L2 cache conflicts for up to four cores, but then starts increasing as operations spread to other cores with thread affinity. This process continues until the bus becomes a secondary bottleneck. This result is consistent with the measurements reported in [41] for a similar dual quad-core based system. In addition, due to longer task durations, the relative difference between GETE and LESE based time measurements appears to be negligible.

Figure 24. Memory throughput in Gbps across number of threads for data type double for different sizes of data on AMD based SUT.

Figure 24 (d), presents memory copy throughput for 16 MB of data size, which results in up to two orders of magnitude longer execution times compared to smaller data sizes on an AMD based SUT. In this case, memory copy throughput scales from one to two threads after which the throughput drops with increase in number of threads. This linear increase from one to two threads is due to private L2 cache for individual core that limit the cache conflicts and due to the presence of memory controllers in AMD based SUT that operate at the speed of the processors and avoid being a bottleneck as compared to the system bus in Intel based SUT.
4.6. **Comparison of Precision**

In this subsection, we compare the precision and repeatability of GETE and LETE algorithms based methodologies through measurement of multi-threaded memory-to-memory data transfer throughput. We run the memory copy tasks for both algorithms hundred times for different data sizes. We apply kNN clustering algorithm to remove the outliers from the execution time data samples.

Figure 25 presents the execution time of 100 memory-memory copy operations with and without outlier removal using 8 threads on Intel based SUT and Figure 26 presents the execution time of 100 memory-memory copy operations with and without outlier removal using 4 threads on AMD based SUT. Each thread is bound to a specific core of the system. There are four data sizes that we consider for copy operation: 16 KB, 128 KB, 1 MB, and 16 MB. A sharper peak of the histogram with lesser spread represents higher precision and repeatability. We note the following from these results on both SUTs:

For shorter tasks, LETE algorithm is more precise but this difference diminishes as task durations increase, especially for the 16 MB data size case.

In all cases, LETE based MINMAX implementation results in lower latency and higher throughput measurements as it is less sensitive to non-deterministic task completions compared to GETE based implementation.

Applying kNN algorithm reduces variance in all cases.

These results also validate the throughput measurements presented in Figure 23 and Figure 24 for Intel and AMD based SUT respectively. Compared to 16 MB data size case, all other cases have two orders of magnitude shorter execution times. This is a typical difference between memory accesses that hit main memory compared to those hitting L1 or L2 caches.

This case study also points out the superiority of the LETE based MINMAX algorithm. It is more precise, less sensitive to non-deterministic concurrent executions, and does not rely on a global clock. Our validation with the STREAM benchmark empirically proves that it is at least as accurate as traditional global-sequential measurement. Thus, we can confidently use it for short duration tasks, such as cache accesses, register level processing, and high-end interconnection network level events.
Figure 25. Histograms of execution time samples for memory-memory copy operations by 8 concurrent threads for different sizes of data on Intel based SUT.
Figure 26. Histograms of execution time samples for memory-memory copy operations by 4 concurrent threads for different sizes of data on AMD based SUT.

(a) 16 KB /w outliers

(b) 16 KB w/o outliers

(c) 128 KB /w outliers

(d) 128 KB w/o outliers

(e) 1 MB /w outliers

(f) 1 MB w/o outliers

(g) 16 MB /w outliers

(h) 16 MB w/o outliers
CHAPTER-5

CONCLUSION AND FUTURE WORK

In this work, we present an accurate benchmarking methodology for multi-core processor based systems that is capable of precisely measuring the execution time of concurrent, symmetric, and short duration tasks. We validate this methodology through comparison with a known example from memory subsystem benchmarking and further demonstrate its use with multiple threads on multiple processor cores. We implement our benchmarking methodology by developing a benchmarking suite for multi-core processor based systems [26].

We present the capabilities of local measurements based LETE MINMAX methodology for measuring the execution times of symmetric, concurrent, and short tasks. When the duration of a task is close to the clock resolution with measurement overheads, global timing does not provide correct measurements. Our empirical evaluation shows that global measurement is reliable only in those cases where execution times are considerably higher than clock resolution and measurement overheads. Otherwise, global measurements will be skewed due to dissimilar task completion instants. On the other hand, local timing based methodology is not impacted by the concurrent nature of measurements. In addition, LETE based timing does not require a global clock, which is a significant advantage with massive level of parallelism in terms of number of cores or processors.

We plan to extend the current work by using on-chip counters to verify the impact of cache and bus conflicts for memory-to-memory measurements that we reported in this paper. We also intend to extend this study to systems with larger than eight cores. This benchmarking methodology will contribute future research endeavors. Presently, suitable benchmarks are not available for multi-core architectures. This benchmarking
methodology is expected to identify performance limits and scaling characteristics of selected commercial multi-core processor based systems.

Other than performance evaluation for any computer system, measuring execution times of various tasks is central to resource management as well as performance evaluation for networking systems. Depending on the type of networking system, such as a switch, router, firewall, software as a service (SAS) gateway, or application server, these tasks can be repetitive as well as short with real-time constraints. Precise and reliable time measurement can help efficient resource scheduling and sharing to ensure meeting a required level of service quality. Packet and message level throughput and latency measurements rely on precise execution time determination. Our methodology can be very helpful in these cases.
REFERENCES


