Technical Report: HPCNL-KICS-TR-02

An Extensible Infrastructure for Benchmarking Multi-Core Processors based Systems

M. Hasan Jamal, Ghulam Mustafa, Waqar Mahmood & Abdul Waheed

January 2009

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/

© Copyright 2009 HPCNL-KICS, UET, Lahore, PAKISTAN

An Extensible Infrastructure for Benchmarking Multi-Core Processors based Systems

Mohammad Hasan Jamal [1] *Ghulam Mustafa* *Waqar Mahmood* *Abdul Waheed*

Al-Khawarzimi Institute of Computer Science,
University of Engineering and Technology, Lahore, Pakistan
{hasan.jamal, ghulam.mustafa, waqar, awaheed}@kics.edu.pk

Abstract

Performance benchmarking has always been an obscure art characterized by specialized knowledge and development techniques. Each effort brings its own solutions to common problems, such as portable and accurate time measurement, experimental design, execution control and repetitions, statistical analyses of measurements, and presentation of results. This problem is further exacerbated due to wide adoption of multi-core processors based systems and complexity of accompanying parallel programming paradigms. In order to avoid re-inventing the solutions for all of the above common benchmarking problems, we introduce Multi-core Processor Architecture and Communication (MPAC) benchmarking library. It provides a common benchmarking infrastructure that can be leveraged to develop micro-benchmarks, application benchmarks, and load generators for state-of-the-art multi-core processors based computing and networking platforms. We demonstrate the efficacy of MPAC by implementing the specifications of several well-known micro-benchmarks including stream, netperf, pthreadbench, and bogomips. We use these benchmarks to validate MPAC based performance measurements on Intel, AMD, and Cavium multi-core processors based platforms. In addition, we extend these micro-benchmarks using MPAC library to measure the scaling characteristics of these multi-core processors based platforms.

Keywords: Performance measurement, benchmarking, load generation, Multi-core Processors, and experimental design, analysis, and control.

1. Introduction

Benchmarks are designed to address a particular performance issue of a system under test (SUT). Current benchmarking practice relies on two contradictory methodologies: either (1) using well-known industry-standard benchmarks; or (2) developing customized benchmarks. Many industry-standard micro-benchmarks and application benchmarks are available to provide performance baseline for a target sub-system or an entire platform. These benchmarks are complete programs that exercise the SUT in a pre-specified manner to measure its performance in a timely fashion. Monolithic structure does not allow a user to modify the workload generation specifications of these benchmarks in any way. With constantly evolving processor, memory, interconnection network, and storage architectures, system designers are increasingly relying on their custom-developed benchmarks for evaluating and comparing the performance of various design choices. A performance issue is initially identified, either intuitively or empirically, and a targeted piece of code is written to benchmark a prototype SUT. While such benchmarks implement customized workload specifications that are relevant to the prototype, they may not be reused for any other platform or application performance issues. Thus both methodologies have limited scope and do not suffice the needs of rapidly evolving computer sub-systems, including multi-core processors, complex memory sub-systems, high-performance interconnects, and state-of-the-art storage devices.

[1] This work was completed through an internship from Cisco-Pakistan.

An alternative of hard-coded and monolithic benchmarks is the use of specification-based benchmarking. Specification-based benchmarking methodology was pioneered by NAS Parallel Benchmarks for comparing the performance of parallel systems [1]. The primary objective of NAS Parallel Benchmark was to use a paper-and-pencil specification of a problem to be solved on the target system rather than using a specific benchmark code. That approach allowed the parallel system vendors to write an optimized code with their own choice of language, compiler, and run-time system for their target architecture. While the goal of NAS Parallel Benchmark was to provide an architecture-independent method of benchmarking, same approach can be adopted to develop specifications based micro-benchmarks, application-benchmarks, and load generators. Benchmarking practice in industry is often focused on establishing performance baselines, comparison of design alternatives, and troubleshooting. A specification-based extensible benchmarking methodology is appropriate to support these goals while avoiding a re-write of common benchmarking code modules for every new platform and application.

We need an extensible framework for specification-based benchmarks and load generators that support state-of-the-art multi-core architectures and high-performance networks. The framework should hide the complexities related to performance measurement of multi-core architectures as well as high speed networks. In addition, it should provide an Application Programming Interface (API) and implementation of common benchmarking functions, such as accepting experimental factors, precise interval timing, thread management, thread affinity, statistical analysis, and reporting of results. The benchmarking framework should allow the user to specify the workload interactions with the SUT and perform the measurements without implementing the above common functions for every benchmark.

Event-driven simulation frameworks, such as simCore [10], can be considered analogous to extensible benchmarking infrastructure. Event-driven simulation can also be accomplished through industry standard tools as well as custom-designed simulators. However, simulation frameworks allow an analyst to focus on specifying the behavior and interactions among various system components instead of mechanics of scheduling, time-keeping, queue management, random number generation, and so on. An extensible benchmarking framework will similarly allow a user to focus on specifying the measurement-based experiment and evaluating the results rather than the implementation of common benchmarking tasks.

Multi-core Processor Architecture and Communication (MPAC) library is specifically designed and developed to ease the development of micro-benchmarks kernels and load generators for state-of-the-art commercial multi-core systems. MPAC library contains general purpose benchmarking modules that could be used to implement a customized and concurrent micro-benchmark. Modularity and flexibility of usage is the main focus of this effort. It is an open source, POSIX complaint, library, developed for and is freely available under FreeBSD style licensing model.

This paper presents the architecture of MPAC library, describes its interface, and provides examples of its usage by implementing the specifications of four well-known benchmarks: bogomips, stream, netperf, and pthreadbench. We initially use these four benchmarks to validate the measurements through MPAC on three platforms based on Intel, AMD, and Cavium multi-core processors. After validating the MPAC measurements through these well-known benchmarks, we use MPAC library to execute current versions of these benchmarks to analyze the scaling characteristics of underlying platform architectures. While many parallel application benchmarks exist, micro-benchmarks for multi-core systems are not very common due to added complexity involved in concurrent measurements. Open source availability of MPAC library is an effort to try to fill that void.

A number of research efforts in this area are discussed in section 2. Section 3 describes the MPAC library's software architecture in detail. Reference MPAC benchmarks and measurements obtained on different multi-core architectures are presented in section 4. Section **5** concludes with a discussion of contributions of this effort to benchmarking multi-core processors based systems.

2. Related Work

System performance evaluation can be accomplished through analytical, simulation, and measurement based techniques. Performance benchmarking is a measurement based analysis technique, which is widely accepted and practiced in industry. We review various performance benchmarking efforts in this section.

A number of research efforts focus on development and usage of monolithic benchmarks to suit specific applications. Hanson et. al. study power/performance using performance benchmarks and focus on the problem of variability and its effect on system power management. They develop tools for collecting correlated power and performance data [4]. Gahvari et. al. present a benchmark for evaluating the performance of Sparse matrix-dense vector multiply (SpMV) on scalar uniprocessor machines [2]. Jensen explores the need for updated benchmarks that reflect a new workflow model derived from technology shift to parallel systems, and provide examples of how productivity can increase by running parallel applications [8]. Joshi et. al. describe a framework, BenchMaker, for constructing parameterized, scalable, synthetic benchmarks from a set of hardware-independent program characteristics and show that with a suitable choice of a few inherent program characteristics related to instruction mix, instruction-level parallelism, control flow behavior, and memory access patterns, it is possible to generate a synthetic benchmark whose performance directly relates to that of real-world application [9]. Vera et. al. present FAME, a new evaluation methodology aimed to fairly measure the performance of multithreaded processors. FAME can be used in conjunction with any of the metrics proposed for multithreaded processors like IPC throughput, weighted speedup, etc [21].

A number of micro-benchmarks are widely being used by the research community for performance benchmarking. STREAM [12] is a well known single-threaded micro-benchmark, which measures sustainable memory throughput and the corresponding computation rate on uniprocessors, vector processors, and shared and distributed memory systems. LMBench [13] is a portable micro-benchmark suite intended to measure performance bottlenecks in a wide range of system applications. It consists of a set of micro-benchmarks for different subsystems. Cachebench [15] evaluates and parameterizes the performance of multiple level of caches present on and off the processor in terms of bandwidth for a unit stride vector kernel. C2CBench [20] 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. Netperf is used to measure unidirectional network throughput and end-to-end network latency [5]. Pthreadbench [16] measures the multithreading performance of pthreads of a SUT.

In terms of commercial benchmarks there are two main leading players. The Standard Performance Evaluation Corporation (SPEC) [19] and Embedded Microprocessor Benchmark Consortium (EEMBC) [3] are working at micro-level as well as application level benchmarks and have developed a number of specific benchmarks. These benchmarks are suitable for comparing different systems but lack extensibility, which is required by an architect to compare various design alternatives through measurements.

A number of benchmarks have specifically targeted high performance parallel computing (HPC) systems. HPC Challenge benchmark suite [11] designed to help define the performance boundaries of future Petascale computing systems, examines the performance of HPC architectures using kernels with memory access patterns. 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 [24]. Other well-known parallel benchmarks include NAS parallel benchmark [1], parkbench [6], Java Grande [18], SPLASH [17] and C3I [14].

Almost all existing benchmarks are monolithic programs. NAS benchmarks, simCore [10] and AONBench [22] are some examples of a few exceptions that utilize specification based benchmarking to a limited extent. These efforts lack an infrastructure that an end user could utilize to develop their own benchmarks according to their specifications.

There is a need for an extensible framework for specification-based benchmarks that target multi-core architectures and high-performance networks. The framework should provide user friendly API that eases the user to develop benchmarks for performance measurement of multi-core architectures as well as high speed networks with minimal effort. Our work provides such an extensible framework.

3. Software Architecture of MPAC

MPAC library is designed to provide an extensible benchmarking infrastructure by leveraging hardware and operating system resources. MPAC Library uses multiple threads with a fork-and-join approach that helps simultaneously exercise multiple processor cores of a system under test (SUT) according to user specified workload.
MPAC library is not only beneficial for benchmarking recent multi-core processor architectures and high performance networking systems but can also be used for traditional single-core and symmetric multiprocessor (SMP) systems.
MPAC library includes different APIs related to concurrent benchmarking activities targeting various system resources, such as processors, memory, I/O devices, network, operating system, system software, and application. The software package[2] also includes sample reference benchmarks using this library.
Figure 1 provides an overview of MPAC's software architecture. It provides an implementation of some commons tasks, such as measurement of timer resolution, determination of loop overhead, accurate interval timers, and other statistical and experimental design related functions, which may be too time consuming to be written by a regular user. However, these ideas are fundamental to accurate and repeatable measurement based evaluation.

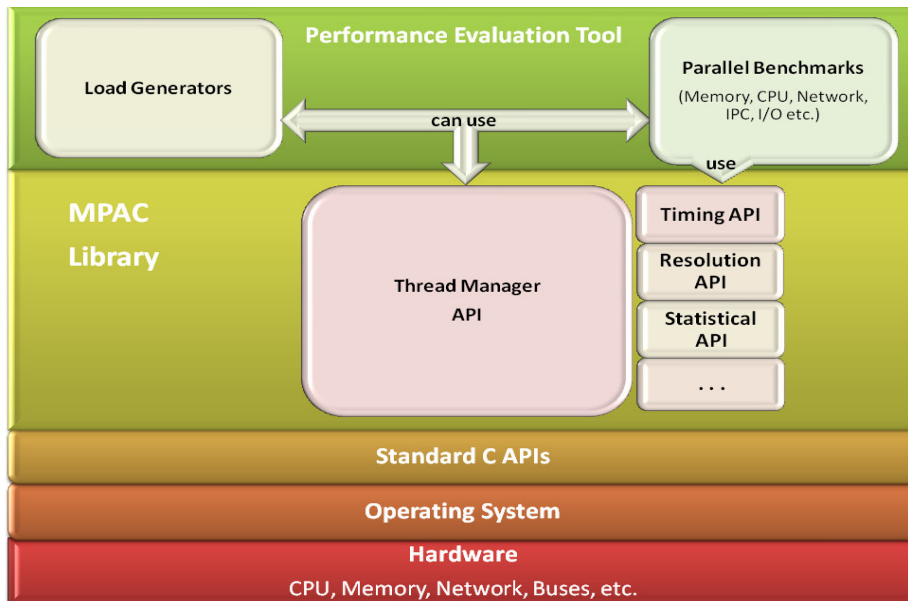

Figure 1: A high-level architecture of MPAC Library's extensible benchmarking infrastructure.

MPAC architecture allows a user to write a specification for the workload without any parallelism. MPAC library can allow the user to determine suitable experimental control and allows the same workload to be replicated across multiple processors or cores using a fork-and-join parallelism. Figure 2 shows an overview of MPAC fork-and-join infrastructure.
In the following subsections, we provide details about various MPAC modules that can be used through its API.

[2] Available online at: http://www.kics.edu.pk/hpcnl/download.php

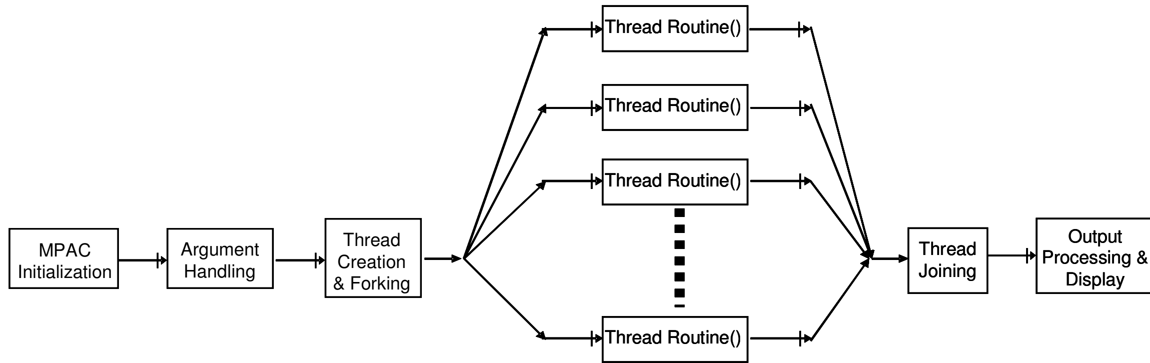

Figure 2. An overview of MPAC Benchmark fork and join infrastructure.

3.1 MPAC Initialization

For accurate and reliable performance measurements, every benchmark needs to account for various measurement overheads. MPAC library provides an initialization function that measures timing overheads, loop overheads, clock resolutions, minimum time duration of a task that can be measured and the number of cores of the SUT. These estimates can be used to remove the effect of overheads from the user measured values to increase the accuracy and precision of the developed benchmark. The number of cores helps the user to decide the number of threads the user wants to create for benchmarking the SUT. Table 1 shows a list of functions available for MPAC initialization.

Table 1. List of MPAC initialization functions.

Function Name	Functionality
`mpac_init (args);`	Provides system and OS specific information
`mpac_resolution_swrtc();`	Determines clock resolution
`mpac_time_loop_overhead();`	Determines Loop overhead
`mpac_time_min_measurable();`	Determines minimum measurable duration of a task

3.2 Thread Manager

Developing multithreaded benchmarks require thread creation, execution control, and termination. The types of thread vary for different tasks. A user may require a thread to terminate after it has completed its task or wait for other threads to complete their tasks and terminate together. MPAC library provides a Thread Manager (TM), which facilitates handling thread related activities transparently from the end user.

It offers high level functions to manage the life cycle of user-specified thread pool of non-interacting workers. It is based on a fork-and-join threading model for concurrent execution of same workload on all processor cores. Thread manager functions are described in the following subsections. Table 2 shows a list of functions available for MPAC under TM.

Table 2. List of MPAC TM functions.

Function Name	Functionality
`mpac_thread_manager_init( args);`	Initializes MPAC Thread Manager Object
`mpac_thread_manager_free(args);`	Frees thread resources
`mpac_thread_manager_end(args);`	Kills the threads
`mpac_thread_manager_wait(args);`	Encapsulates Join operation for multiple threads

`mpac_thread_manager_isolate(args);`	Encapsulates Detach operation for multiple threads
`mpac_thread_manager_sendsig(args);`	Encapsulates Kill operation for multiple threads
`mpac_thread_manager_No_of_cores();`	Determines number of cores in the system

3.2.1 Thread Creation

As thread creation and termination is integral part of multithreaded applications, the TM provides two functions for thread creation depending of the user specification of joinable or detachable threads. The TM releases the burden of the user by providing a single function call that initializes creates, joins/detaches, and frees resources of a thread pool. Table 3 shows a list of functions available for MPAC thread creation.

Table 3. List of MPAC TM thread creation functions.

Function Name	Functionality
`mpac_thread_manager_startj(args);`	Creates user defined number of joinable threads
`mpac_thread_manager_startd(args);`	Creates user defined number of detachable threads
`mpac_thread_manager_fork(args);`	Starts created threads

3.2.2 Thread locking

Dealing with threads can sometimes be a cumbersome task that includes ordering of tasks, waiting for certain conditions to be met before starting a task, synchronizing threads, and so on. The TM provides user-friendly wrapper functions to incorporate thread locking. Sometimes, user specification requires synchronizing the threads to start or end their execution for timing purpose. The TM implements a barrier synchronization mechanism. Table 4 shows a list of functions available for MPAC thread locking and synchronization.

Table 4. List of MPAC TM thread locking functions.

Function Name	Functionality
`mpac_thread_manager_initlock(args);`	Initializes lock for threads
`mpac_thread_manager_destroylock(args);`	Destroys locks for threads
`mpac_thread_manager_barrier(args);`	Synchronizes threads

3.2.3 Thread Affinity

A user may require a task to execute on a specific processor core. 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 [23]. The TM provides two methods of implementing thread affinity. Binding threads to cores in a round robin fashion at initialization phase or according to user specification. Table 5 shows a list of functions available for MPAC thread affinity.

Table 5. List of MPAC TM thread affinity functions.

Function Name	Functionality
`mpac_thread_manager_core_affinity();`	Sets affinity for single threads if flag is set
`mpac_thread_manager_affinity_setter();`	Sets affinity for all threads if flag is set

3.3 Time Measurement

The most common task in benchmarking is the time measurements. The MPAC Library provides the functionality for measuring the execution time of a task as well as to execute a task for a desired duration. User specifications are executed repeatedly during this interval. It is essential to estimate the loop overhead as well as timer resolution for accurate benchmarking.

Figure 3 and Figure 4 provide pseudo code for determining loop overhead (inspired from LMbench) and clock resolution. Our sample benchmarks subtract these values from the time measured, for precision.

```
N  ← very large number

CALL Get_Clk RETURNING starttime
   FOR each iteration i out of N
       Register_Operation()
   END FOR

CALL Get_Clk RETURNING endtime
T1 ← endtime -starttime

CALL Get_Clk RETURNING starttime
   FOR each iteration i out of N
       Register_Operation()
       Register_Operation()
   END FOR

CALL Get_Clk RETURNING endtime
T2 ← endtime -starttime

RETURN (2*T1 - T2)/N
```

Figure 3. Pseudo code to determine loop overhead.

```
M ← No. of Iterations

CALL Get_Clk RETURNING starttime
 WHILE (Iterations complete)
  Increment count;
  CALL Get_Clk RETURNING currenttime
  diffs[iteration] ←  currenttime - starttime;

  IF diffs[iteration] > 1.0E-9
       Increment iteration;
       starttime ← currenttime;
  END IF
 END WHILE

FOR each iteration j out of M
      sum all diffs[j];
END FOR
       Avr ← total/M;
RETURN Avr;
```

Figure 4. Pseudo code to estimate clock resolution.

3.4 Statistics Measurement

The MPAC library provides the Statistics Interface with common statistics functions such as mean, mode, median, minimum, maximum, variance, standard deviation, and confidence interval. The users can extend this interface along similar terms, according to their requirement. Table 6 shows a list of functions available for MPAC statistics library.

Table 6. List of MPAC statistics library functions.

Function Name	Functionality
mpac_stats_mode(args);	Returns mode of data array of size n
mpac_stats_mean(args);	Returns mean of data array of size n
mpac_stats_median(args);	Returns median of data array of size n
mpac_stats_min(args);	Returns minimum of data array of size n
mpac_stats_max(args);	Returns maximum of data array of size n
mpac_stats_var(args);	Returns variance of data array of size n
mpac_stats_std(args);	Returns standard deviation of data array of size n
mpac_stats_conf_intr(args);	Stores CI of data array, of size n, in (ci[0],ci[1])
mpac_stats_cmp(args);	Compares two strings

3.5 I/O Interface

Performance measurements targeting communication among processes, storage devices, and networks require many small but tedious Input/Output functions. The MPAC library provides an Input/Output interface, which includes commonly used file and network I/O functions for file handling, reporting, logging, data storage, communication initialization, communication tear-down, etc. This interface facilitates the development of benchmarks by not repeatedly having to write these functions. Table 7 shows a list of functions available for MPAC I/O interface.

Table 7. List of MPAC I/O interface functions.

Function Name	Functionality
mpac_io_file_length(args);	Returns the size of file
mpac_io_file_read (args);	Stores the contents of a file to a memory buffer
mpac_io_file_readln(args);	Reads a single line of file, stores to buff
mpac_io_net_init(args);	Initializes network handler object
mpac_io_net_getipaddr(args);	Get IP address
mpac_io_net_getportno(args);	Get port number
mpac_io_net_getprotono(args);	Get protocol number

3.6 Benchmark Development

A four step generic procedure is required to develop any benchmark using MPAC library: (1) declarations; (2) thread routine; (3) thread creation; and (4) optional final calculations and garbage collection.

The declaration step initializes user input structure and thread data structure variables. The thread routine requires the writing of benchmark specification, which is to be executed by threads. Thread creation phase creates a joinable or detachable thread pool according to user requirement using thread manager. The Optional calculations and garbage collection step, in case of joinable threads, performs the final calculations, displaying output and releasing the resources acquired.

4. Reference MPAC Benchmarks

In order to validate the MPAC library, we take the specification of several well-known micro-benchmarks and implement them through MPAC library. We compare the measurements of

these existing benchmarks with the benchmarks developed through MPAC library on various x86 and MIPS64 architectures. In this section, we describe the design details of reference MPAC benchmark. We have developed bogomips, memory, end-to-end network and pthread micro-benchmarks using MPAC library, which implement the specification of bogomips [24], STREAM benchmark [12], netperf benchmark [5] and pthreadbench [16], respectively. The specifications of three SUTs used for validation are shown in Table 8.

Table 8: Specifications of Systems under Test (SUTs).

Platform Attributes	Systems under Test		
Processor	Quad Core Intel® Xeon® E5405, 1333MHz FSB	Dual Core AMD Opteron Proc 2212HE	Cavium Octeon CN3860
Physical CPU chips	2	2	1
No. of Cores	2 x 4 = 8	2 x 2 = 4	16
CPU Speed	2.0 GHz	2.0 GHz	500 MHz
L1 D Cache	32 KB	64 KB	8 KB
L1 I Cache	32 KB	64 KB	32 KB
L2 Cache	2 x (2 x 6) = 24 MB	2x(2x1) = 4 MB	1 MB shared
DRAM Size	8 GB	8 GB	4 GB
OS Version	2.6.23.1-42, Fedora core 8	2.6.23.1-42, Fedora core 8	Debian 2.6.16.26
Compiler	gcc 4.1.2, -O3	gcc 4.1.2, -O3	gcc 4.1.2, -O3

4.1 Bogomips Benchmark

The pseudo code for bogomips benchmark is shown in Figure 5. These specifications are determined from examining the code of bogomips benchmark [24]. The MPAC reference implementation of bogomips can enhance its usage with multiple threads to exercise multiple processor cores. Bogomips measures number of empty loop iterations (in millions) that underlying processor can execute in one second.

```
Main Thread:
N ← No of Threads

CALL mpac_int();
CALL mpac_bogo_arg_handler();
arg ← user_input;

CALL mpac_thread_manager_startj(N,arg,Routine);

Process Output;
CALL mpac_bogo_output();

Thread_local_processing (Routine):
Loops ← (1<<12);
WHILE incrementing Loops
       ticks ← clock();
              CALL delay(Loops);
       ticks ← clock() - ticks;
       IF ticks
          break;
       END IF
END WHILE
RETURN Loops
```

Figure 5. Pseudo code of bogomips benchmark.

In order to validate MPAC-bogomips benchmark, we compare with existing standalone bogomips benchmark results on three different SUTs. We use a single thread based execution to measure the bogomips value to mimic the bogomips benchmark approach. Table 9 shows bogomips values of the bogomips benchmark and MPAC-bogomips benchmark. The similarity in results of both benchmarks validates our benchmark.

Table 9. The bogomips values on different SUTs.

SUT	BogoMIPS Benchmark	MPAC Benchmark
intel	535.75	563.60
amd	565.24	514.45
cavium	116.73	110.59

We want to know the CPU scaling across different cores, which is not possible using bogomips benchmark specifications because clock() is a system call, which is handled at the kernel level thread and cannot be controlled from user level thread. We develop a different MPAC CPU benchmark that exercises the floating point, integer and logic unit of the processor. The pseudo code of the CPU benchmark is shown in Figure 6. Figure 7 show the throughput of different arithmetic and logical operation across number of threads for different SUTs. It is observed that the throughput scales linearly across number of threads as expected.

```
Main Thread:

N ← No of Threads

CALL mpac_int();
CALL mpac_cpu_arg_handler();
arg ← user_input;

CALL mpac_thread_manager_startj(N,arg,Routine);

Process Output;

CALL mpac_cpu_output();

Thread_local_processing (Routine):

M ← Very Large Number

barrier(starttime);
  FOR each iteration j out of M
           maths_operation(j);
  END FOR
barrier(endtime);

RETURN endtime - starttime
```

Figure 6. Pseudo code of MPAC CPU benchmark.

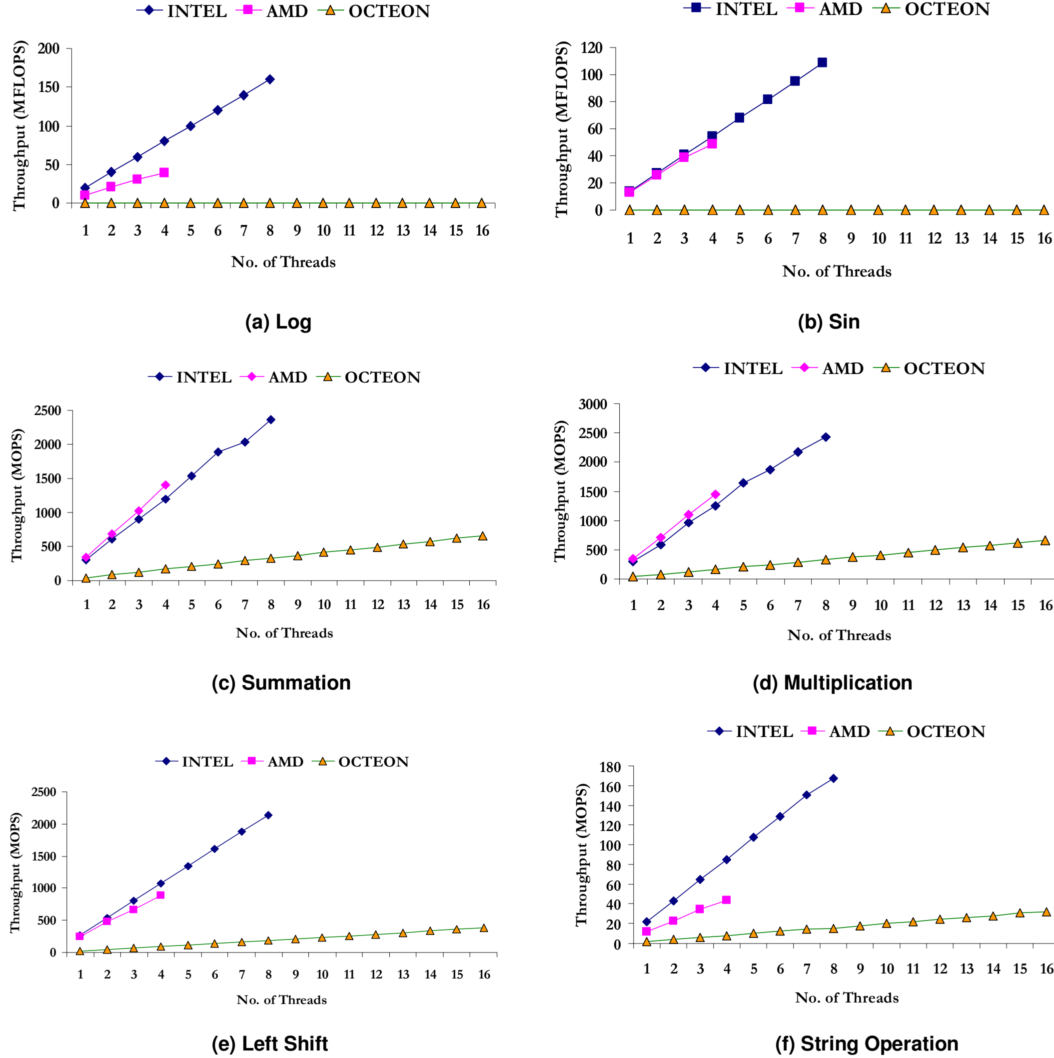

Figure 7. CPU throughput in MOPS across number of threads for different arithmetic and logical functions for different SUTs

4.2 Memory Benchmark

Our memory benchmark is inspired from the STREAM benchmark[12], which uses sequential task-based timing measurements. In fact, we use the STREAM benchmark results as a reference for single thread-based usage of MPAC memory benchmark for validation.

The pseudo code for memory benchmark is shown is Figure 8. The STREAM benchmark performs memory-to-memory data transfer operation on floating point data of matrix size one million (16 MBytes) using unit stride. Our benchmark uses the same default parameters as that of stream benchmarks The MPAC memory benchmark takes the number of threads, data size, data type, affinity flag and number of repetitions as input from user. Thread local and global timing technique for precise time measurements are employed [7].

Table 10 compares memory-to-memory copy throughputs of the STREAM benchmark and MPAC memory benchmark. The similarity validates the results based on our benchmark. The STREAM benchmark measures slightly higher throughput because it initializes the array elements to constant values (zeros) while our benchmark 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.

```
Main Thread:
N ← No of Threads

CALL mpac_int();
CALL mpac_mem_arg_handler();
arg ← user_input;

CALL mpac_thread_manager_startj(N,arg,Routine);

Process Output;

CALL mpac_mem_output();

Thread_local_processing (Routine):

Source Memory       → A
Destination Memory  → B

  CALL Barrier
       CALL Get_Clk RETURNING starttime
              B = A;
       CALL Get_Clk RETURNING endtime

RETURN endtime - starttime;
```

Figure 8. Pseudo code of memory copy benchmark.

Figure 9 shows memory throughput versus number of threads of our benchmark using floating point data for various data sizes for three SUTs. With data sizes of 4 KB, 16 KB and 1 MB, most of the memory accesses should hit L2 caches rather than the main memory. It is observed in Figure 9 (a), (b) and (c) that the throughput scales linearly.
Figure 9 (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 in case of Intel based SUT. In the case on Intel based SUT, memory copy throughput does not scale linearly with the number of threads. In contrast to data sizes of 16 KB, and 1 MB, which can fit in L2 caches, copying 16 MB require extensive memory accesses through shared bus. 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 [23] for a similar dual quad-core based system.

Table 10. Throughput in Mbps of memory-to-memory copy of 16 MB floating point data on different SUTs.

SUT	**Stream Benchmark**	**MPAC Benchmark**
intel	27905	26434
amd	16172	15744
cavium	6.55	5.63

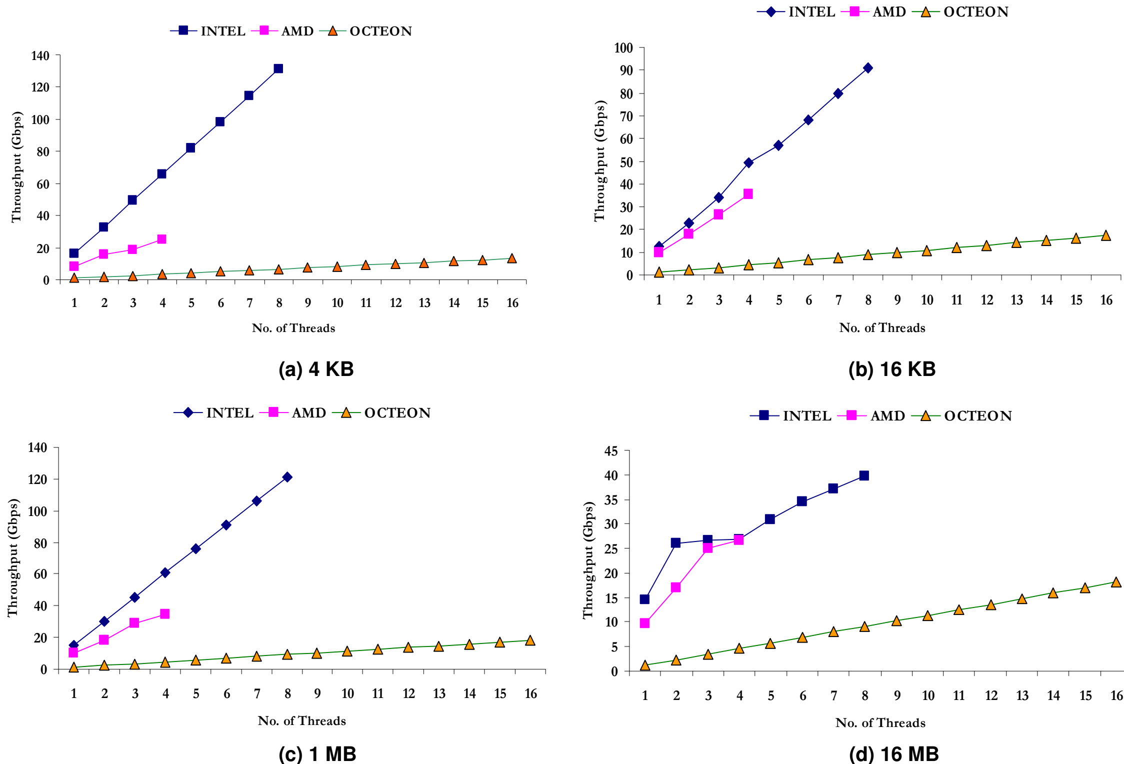

Figure 9. Memory throughput in Gbps across number of threads for data type double for different sizes of data for different SUTs.

On the other hand the throughput scales linearly for AMD and Cavium based SUT for 16 MB of data size, because their architecture do not have system bus that becomes a bottleneck as in Intel based SUT case, but have more efficient memory controllers for low-latency data transfer to main memory.

4.3 Network Benchmark

Our network benchmark is inspired from Netperf benchmark and is implemented using its specification [5]. Figure 10 provides the pseudo code of MPAC end-to-end network benchmark.

In order to validate our network benchmark, we compare the end-to-end network data transfer throughput measured by MPAC network benchmark with Netperf benchmark results on three different SUTs. We use a single thread based execution to measure the end-to-end network data transfer throughput of sending messages to mimic the Netperf benchmark approach. Table 11 shows end-to-end network data transfer throughputs of the Netperf benchmark and MPAC network benchmark. The similarity validates the results based on our benchmark.

Table 11. Throughput in Mbps of end-to-end network data transfer on different SUTs.

SUT	Netperf Benchmark	MPAC Benchmark
intel	7986	7203
amd	4483	4200
cavium	2709	2369

```
Main Thread:
N ← No of Threads

CALL mpac_int();
CALL mpac_net_arg_handler();
arg ← user_input;
CALL mpac_io_open_receiver();

FOR each of N Threads
       CALL Mpac_io_accept_receiver();
END FOR
CALL mpac_thread_manager_startj(N,arg,Routine);

Process Output;
CALL mpac_net_output();
CALL mpac_net_cleanup();

Thread_local_processing (Receiver Side):
  WHILE msgcount more than 0
      Receive_message();
      Decrement msgcount;
  END WHILE

Thread_local_processing (Sender Side):
generate_message();
open_connection();

barrier(NULL);
  CALL Get_Clk RETURNING starttime

  WHILE message_count more than 0
        Send_message();
        Decrement message_count;
  END WHILE

  CALL Get_Clk RETURNING starttime
close_connection();
```

Figure 10. Pseudo code of Network benchmark.

Figure 11 show the throughput of end-to-end network data transfer on different SUTs. Linear scalability across threads is observed on all SUTs using MPAC benchmark.

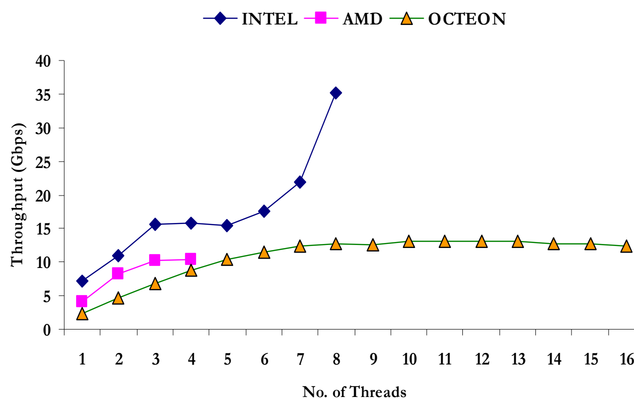

Figure 11. Throughput in MOPS of integer operations across number of threads for different SUTs.

4.4 Pthread Benchmark

MPAC Pthread benchmark is based on the specifications of pthreadbench [16]. The context switching and thread lifecycle modules of pthreadbench are incorporated in the MPAC benchmark. In order to validate our pthread benchmark, we compare the number of context switching per second and number of thread created per second, measured by MPAC pthread benchmark with pthreadbench benchmark results on three different SUTs. The pthreadbench is enhanced with multithreading to measure the performance of pthreadbench workload on given processor cores. Figure 12 and Figure 13 provide the pseudo code of MPAC pthread benchmark.

```
Main Thread:
N ← No of Threads
D ← Duration

CALL mpac_int();
CALL mpac_pthread_arg_handler();
arg ← user_input;

CALL mpac_signal_settimer(D);
CALL mpac_thread_manager_startj(N,arg,Routine);

Process Output;
CALL mpac_pthread_output();

Thread_local_processing (Routine)
  barrier(NULL);
  WHILE TRUE
        ThreadCreate(Lifecycle);
        ThreadJoin(lifecycle);
  END WHILE

Thread_Functions:
LifeCycle()
{
       Count++;
}
```

Figure 12. Pseudo code of thread lifecycle module of pthread benchmark.

Table 12 and Table 13 show the number of thread created per second and the number of context switching per second and respectively of the pthreadbench benchmark and MPAC network benchmark. The similarity validates the results based on our benchmark.

Table 12. Number of thread lifecycle per second on different SUTs.

SUT	Pthreadbench	MPAC Benchmark
intel	86185	85720
amd	83490	83666
cavium	16956	16917

Table 13. Number of context switching per second on different SUTs.

SUT	Pthreadbench	MPAC Benchmark
intel	210630	290902
amd	330273	358457
cavium	407252	439365

```
Main Thread:
N ← No of Threads
D ← Duration

CALL mpac_int();
CALL mpac_pthread_arg_handler();
arg ← user_input;

CALL mpac_signal_settimer(D);
CALL mpac_thread_manager_startj(N,arg,Routine);

Process Output;
CALL mpac_pthread_output();

Thread_local_processing (Routine)
barrier(NULL);
Create Two Threads
Join Two Threads

Thread_Function 1:
  WHILE(TRUE)
  Mutex_Lock(Lock 1);
       Count++;
  Mutex_UnLock(Lock 2);
  END WHILE

Thread_Function 2:
  WHILE(TRUE)
  Mutex_Lock(Lock 2);
       Count++;
  Mutex_UnLock(Lock 1);
  END WHILE
```

Figure 13. Pseudo code of context switching module of pthread benchmark.

To see the scalability of MPAC pthread benchmark we run thread life cycle module across number of cores on different SUTs. The thread context switching is not considered for scalability test because whenever there is context switching the SMP kernel thread comes into action and the scheduling is out of the control of the user level thread.
Linear scalability is observed across number of threads for different SUTs as shown in Figure 14.

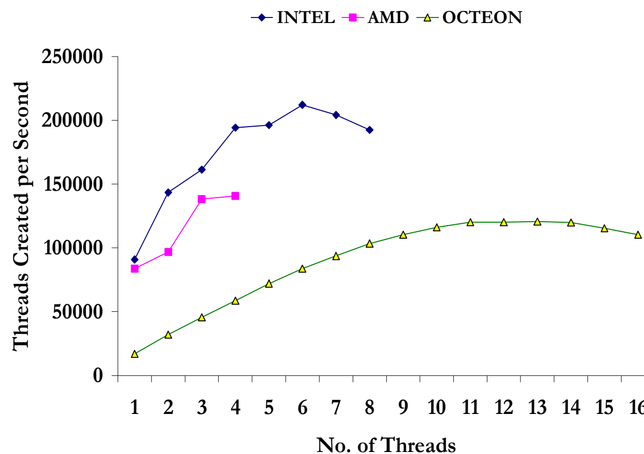

Figure 14. Graph of thread creation per second across number of threads for different SUTs.

5. Conclusion

Existing micro-benchmarks and application benchmarks hardly provide an extensible framework for specification-based benchmarks and load generators for recent multi-core architectures and high-performance networks, which is becoming important due to technology trends in processor and network architectures. MPAC Library provides an Application Programming Interface (API) and implementation of common benchmarking functions, such as accepting experimental factors, precise interval timing, thread management, thread affinity, statistical analysis, and reporting of results. MPAC library makes it easier for end user to focus on benchmarking and performance evaluation instead of writing the benchmarking framework from scratch. Existing benchmarks can be implemented in MPAC based on their specification easily.

MPAC library will enable the future benchmarks to provide a holistic view of system performance. Benchmarks developed using MPAC library are equally useful for system architects as well as end users. MPAC user has full freedom to choose the size of thread pool and other performance related parameters.

Current release of MPAC benchmarking suite includes MPAC Library itself and sample reference benchmarks for cache and memory subsystem, CPU, pthread and high performance networks (using both TCP and UDP messages).

Acknowledgements

We would like to thank National ICT R&D Fund, Ministry of Information Technology, Pakistan, for funding this project.

References

[1] D. H. Bailey, et al., "The NAS Parallel Benchmarks", Intl. Journal of Supercomputer Applications, v. 5, no. 3, 1991, pp. 63- 73.

[2] H. Gahvari, M. Hoemmen, J. Demmel, and K. Yelick, “Benchmarking Sparse Matrix-Vector Multiply in Five Minutes,” 2007 SPEC Benchmark Workshop, January, 2007.

[3] S. Gal-On and M. Levy, "Measuring Multi-Core Performance," Computer, Vol. 41, Issue 11, November 2008, pp 99-102.

[4] H. Hanson, K. Rajamani, J. Rubio, S. Ghiasi and F. Rawson, “Benchmarking for Power and Performance,” 2008 SPEC Benchmark Workshop, January, 2008.

[5] Hewlett-Packard Company, Netperf: A Network Performance Benchmark, Feb. 1995. Available on-line from: http://www.netperf.org/netperf/training/Netperf.html

[6] T. Hey, and D. Lancaster, “Parallel Performance Analysis and the Future of Parkbench,” International Journal of High-Performance Computing, 14, 205, 1998.

[7] M. H. Jamal, and A. Waheed, “Precise Measurement of Execution Time of Concurrent, Symmetric, and Short Tasks,” in the Proceedings of CMG’08, Las Vegas, Nevada, 7-12 December, 2008.

[8] F. Jensen, “Working Differently with Parallel Workflows – the New Standard Workstation Benchmark,” 2008 SPEC Benchmark Workshop, January, 2008.

[9] A. M. Joshi, L. Eeckhout, and L. K. John, “The Return of Synthetic Benchmarks,” 2008 SPEC Benchmark Workshop, January, 2008.

[10] Y. Jung, Y. Chiba, D. Kim, Y. Kim, “simCore: an event-driven simulation framework for performance evaluation of computer systems,” in the Proceedings of the 8th

International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2000, pp.274-280.

[11] P. Luszczek, D. Bailey, J. Dongarra, J. Kepner, R. Lucas, R. Rabenseifner, D. Takahashi, "The HPC Challenge (HPCC) Benchmark Suite," SC06 Conference Tutorial, IEEE, Tampa, Florida, November 12, 2006.

[12] J.D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," IEEE Technical Committee on Computer Architecture newsletter, December 1995, pp 1-10

[13] L. McVoy, and C. Staelin, "LMbench: Portable Tools for Performance Analysis," in the Proceedings of the 1996 USENIX Technical Conference, San Diego, CA, January 1996, pp. 279-295.

[14] R. C. Metzger, et al., "The C3I Parallel Benchmark Suite: Introduction and Preliminary Results," in the Proceedings of the 1996 Supercomputing Conference, November, 1996.

[15] P. J. Mucci and K. London, "The cachebench report," Technical Report ut-cs-98-384, University of Tennessee, 1998, pp 15-32

[16] Pthreadbench, Available on-line from: http://www.gelato.unsw.edu.au/patches/pthreadbench/pthreadbench.tar.gz

[17] J. P. Singh, W. D. Weber, and A. Gupta, "SPLASH: Stanford parallel applications for shared-memory," ACM SIGARCH Computer Architecture News, Volume 20, Issue 1, March 1992, pp. 5-44.

[18] L. M. Smith, J. M. Bull, J. Obdrizalek, "Parallel Java Grande Benchmark Suite," Supercomputing, ACM/IEEE 2001 Conference 10-16 November, 2001.

[19] Standard Performance Evaluation Corporation, http://www.spec.org

[20] S. Vadlamani, S. Jenks, "Architectural Considerations for Efficient Software Execution on Parallel Microprocessors," IPDPS 2007, Long Beach, CA, March 2007, pp. 1-10

[21] J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernandez, M. Valero, " Measuring the Performance of Multithreaded Processors," 2007 SPEC Benchmark Workshop, January, 2007.

[22] A. Waheed, J. Ding, "AONBench: A Methodology for Benchmarking XML Based Service Oriented Applications," JNW 2(5), 2007, pp 46-53.

[23] A.Waheed, J. Ding, J. Yao, and L.N. Bhuyan, "Performance Characterization of a Dual Quad-Core Based Application Oriented Networking System," NAS'08, 2008.

[24] D. Wong, "Bogomips Benchmark", Available on-line from: http://djwong.org/programs/bogomips/bogomips-1.4.2.tar.gz

[25] A.B. Yoo, B. R. de Supinski, F. Mueller, and S.A. McKee, "Memory Benchmarks for SMP-Based High Performance Parallel Computers," 29th International Symposium on Computer Architecture, Anchorage, Alaska, May 25-29, 2002.