Performance analysis on Multicore Processors

Naresh Kasturi, Saravana Kumar Gajendran
Department of Computer Science
San Jose State University
San Jose, CA 95112
e-mail: {naresh.kasturi, saravana.kumar}@sjsu.edu

ABSTRACT

With advance in prevalence personal computers, the end - user needs faster and more capable systems. This can be achieved by increasing the clock speed or adding multiple processing cores to the same chip. But this is an old trend, so manufacturers are focusing multicore processors. The use of low cost multicore processors with small-scale parallelism of several or several processing units has been spread to general purpose personal computers. In this paper, we focus on implementing multicore processor architecture to evolutionary computation. With large use of multicore processors, we focus on benchmarking these systems at operating system level. So we introduce multicore processor architecture and communication (MPAC). We use these benchmark techniques to validate MPAC based performance analysis on Intel, AMD multicore based platforms.

1. INTRODUCTION

Since the early 1990s, research on methods for boosting up evolutionary computation through implementations on massively parallel computers has been quite active. Besides, the use of multicore processors has recently been expanding even in general purpose personal computers. In this paper, we present the description that all CPUs in a multi-core processor be able to directly reference the local memory of each core without having to go through main memory. So we target the improvement in the execution performance of evolutionary computation and to reduce the energy (power) consumption. Performance benchmarking depends on development methods and specialized knowledge which lead to the problems: portable and accurate time measurement, execution control and repetitions, experimental design, statistical analyses of measurements and presentation of results. So design and development organizations need micro-benchmarks to fully understand the performance impact of state-of-the-art processors based computing platforms to host their new products. Present benchmarking practice depend on two contradictory methodologies: using well-known industry standard benchmarks or developing customized benchmarks. Industry standard benchmarks provide baseline performance for a system or a platform. Customized benchmarks are for evolving processor, memory, network and storage architecture. Such benchmarks implement customized workload specifications that are significant to the prototype, they may not be reused for any other platform or application performance. Thus these both technologies do not serve the need of rapidly evolving computer sub-systems, including multi-core processors, complex memory subsystems and high-performance interconnects. So we use specification based benchmarking as an alternative to the existing benchmarking techniques. 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. This approach lets vendors to write an optimized code with their own choices of language, compiler, and run-time system for their target architecture. So develop a multi-core processor architecture and communication (MPAC) framework, which is an open source C-based, POSIX-complaint, benchmarking library and is freely available. MPAC library is portable across hardware platforms and hence we present details of this benchmarking framework.

2. MULTICORE PROCESSOR

A multicore processor is a computing component with more than one central processing units or cores. A core is a unit that reads and executes instruction, but a multicore processor can run multiple instruction at the same time, thus increasing the speed and performance of the system. This improvement in performance may be gained by the software algorithm used in it and its implementation.

2.1 A Brief History of Microprocessors

Intel manufactured the first microprocessor, the 4-bit 4004, in the early 1970s which was basically just a number-crunching machine. Shortly afterwards they developed the 8008 and 8080, both 8-bit, and Motorola followed suit with their 6800 which was equivalent to Intel's 8080. The companies then fabricated 16-bit microprocessors, Motorola had their 68000 and Intel the 8086 and 8088; the former would be the basis for Intel's 80386 32-bit and later their popular Pentium lineup which were in the first consumer-based PCs. Each generation of processors grew smaller, faster, dissipated more heat, and consumed more power.

2.2 Moore's Law

One of the guiding principles of computer architecture is known as Moore's Law. In 1965 Gordon Moore stated that the number of transistors on a chip will roughly double each year (he later refined this, in 1975, to every two years). What is often quoted as Moore's Law is Dave House’s revision that computer performance will double every 18 months.

The graph in fig.1 plots initial processors with number of transistors per chip. The number of transistors has roughly doubled every 2 years. Moore's law continues to reign; for example, if the current trend continues to 2020, the number of

transistors would reach 32 billion. House's prediction, however, needs another correction. Throughout the 1990's and the earlier part of this decade microprocessor frequency was synonymous with performance; higher frequency meant a faster, more capable computer. Since processor frequency has reached a plateau, we must now consider other aspects of the overall performance of a system: power consumption, temperature dissipation, frequency, and number of cores. Multicore processors are often run at slower frequencies, but have much better performance than a single-core processor.

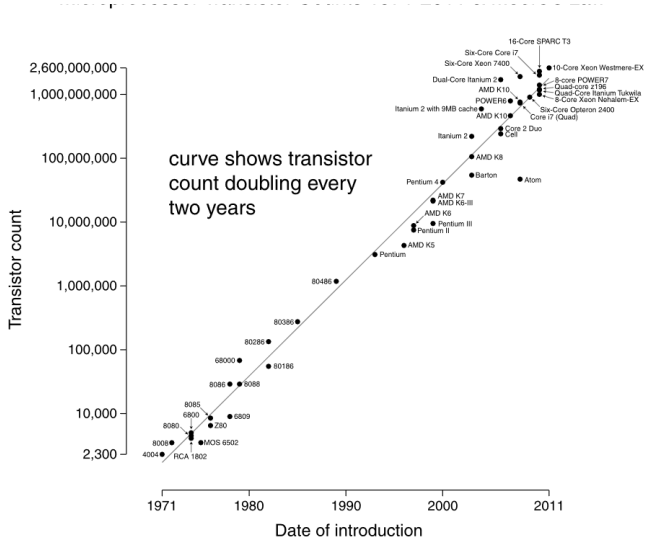

Figure 1. Microprocessor Transistor Counts & Moore's Law

2.3 Configuration of Multicore processors

Multi-core processors have come to be installed in many types of computing equipment in recent years. In fact, multi-core processors equipped with multiple general purpose cores of the same type (homogeneous multi-core processors) are coming to be installed even in PCs. In the following, "multi-core processor" refers to a homogeneous multi-core processor. Obtaining better performance through the use of multi-core processors is not only a matter of integrating multiple cores, but also high memory-access performance that matches CPU operational ability is necessary. Memory-access performance has traditionally been improved by increasing the capacity of cache memory, but such an approach increases the area of the processor and drives up energy consumption. In addition, increasing the number of cores means more control overhead for maintaining coherence between caches, which can lead to a drop in performance. In response to these problems, the use of local memory called scratchpad memory (SPM) inside a core has been attracting attention since this kind of memory can achieve the same access performance as cache memory while having an energy-saving effect. The basic configuration of a typical multi-core processor equipped with local memory is shown in Figure 2. In this configuration, multiple cores, each consisting of a CPU and local memory, connect to main memory via a system bus. Here, it is generally specified that local memory in each core be several KB to several hundred KB in capacity in accordance with chip area and that read/write operations be limited to that core[12]. Thus, if one core needs to reference the data stored in the local memory of another core, it can only do so after that data has been moved to main memory. Typical data read/write speeds (number of clock cycles) among the CPU, local memory, and cache memory that configure each core are shown in Figure 3. Compared to data read/write speeds between the CPU and cache/local memories, those between main memory and cache/local memories are as much as 100 times slower. In addition, while transfers between main memory and cache memory are controlled by hardware, those between main memory and local memory are all controlled by software. Thus, if transfer control between main memory and local memory has to be performed frequently, transfer overhead will increase leading to a drop in performance. It is essential that this overhead in transfer control between main memory and local memory be decreased to make effective use of local memory.

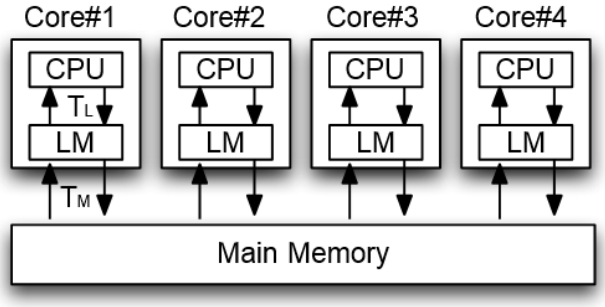

Figure 2. Basic Configuration of multi-core processor

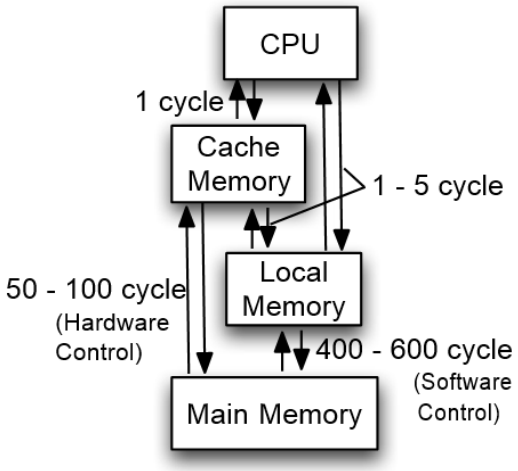

Figure 3. Core configuration and data Read/Write speeds

3. SOFTWARE ARCHITECTURE OF MPAC

MPAC library provides a common benchmarking infrastructure that eases the development of specification-based micro-benchmarks, application benchmarks, and network traffic load generators for state-of-the-art multi-core processors based computing and networking platforms by leveraging hardware and operating system resources. MPAC Library uses multiple threads in a fork-and-join approach that helps simultaneously exercise multiple processor cores of a system under test (SUT) according to user specified workload.

The flexibility of MPAC software architecture allows a user to generate specification driven workload for micro-benchmarking without any parallelism. MPAC library allows the user to implement suitable experimental control and allows the same workload to be replicated across multiple processors or cores using a fork and join parallelism. Hence, user can focus on specifying the measurement-based experiment and evaluating the results instead of implementing common benchmarking tasks.

MPAC library is an open source C-based, POSIX complaint, library, which is freely available under FreeBSD style licensing model. 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 also includes sample reference benchmarks using this library.

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 also includes sample reference benchmarks using this library.

Fig. 4 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.

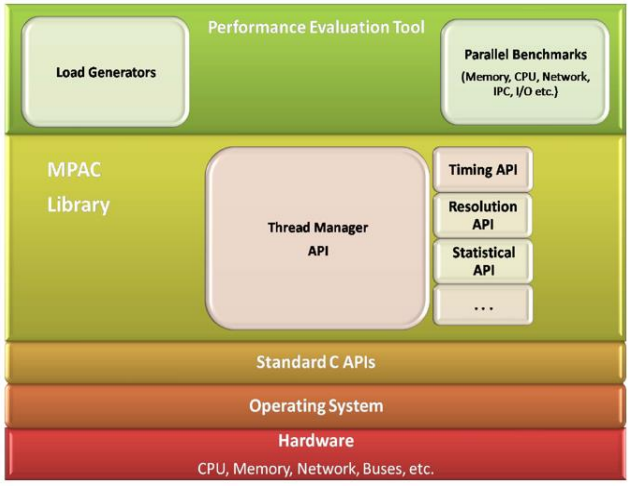

Figure 4. A high-level architecture of MPAC Library's extensible benchmarking infrastructure.

Figure 5 shows an overview of MPAC fork-and-join based execution model. In the following subsections, we provide details about various MPAC modules that can be used through its API.

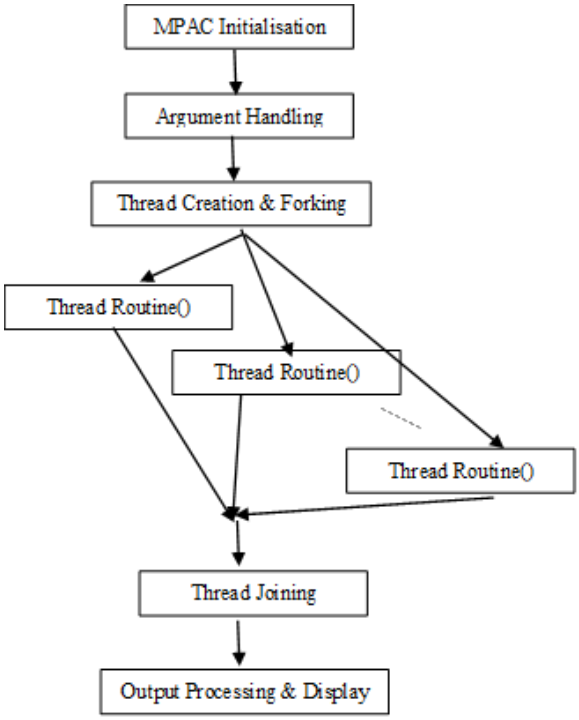

Figure 5. Overview of MPAC Benchmark fork and join infrastructure

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 determine the number of threads that user will create for benchmarking the SUT.

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.

3.2.1 Thread Creation:

As thread creation and termination is an integral part of multithreaded applications, the TM provides two functions for thread creation depending on the user specification of joinable or detachable threads. The TM facilitates the user by providing a single function call that initializes creates, joins/detaches, and frees resources of a thread pool.

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.

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. 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.

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 as well as timing system call overhead for accurate benchmarking. Our sample benchmarks subtract these values from the measured execution time, for precision.

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

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.

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 TM. The optional calculations and garbage collection step, in case of joinable threads, performs the final calculations, displaying output and releasing the resources acquired.

4. MPAC BENCHMARKS

To evaluate MPAC benchmarks, we consider the specifications of well-known processors. We compare the measurements of these existing benchmarks with the benchmarks developed through MPAC library on various x86 and MIPS64 architectures for single thread. The Specifications for these processors is given in table 1.

4.1 CPU Benchmark

We develop an MPAC based CPU benchmark, that exercises the floating point, integer and logic unit of the processor, to measure the CPU scaling with number of cores. In absence of any memory accesses, we expect a linear scale-up of CPU benchmark throughput (as operations per second) with number of cores. We can use this criterion for validating the CPU benchmark.

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. The magnitudes of CPU benchmark throughputs are different across these platforms due to differences in micro-architectures of three multi-core processors: Intel quad-core Xeon, AMD dual-core Opteron, and Cavium 16-core Octeon. Linear scalability for CPU operations as well as across platforms validates benchmark.

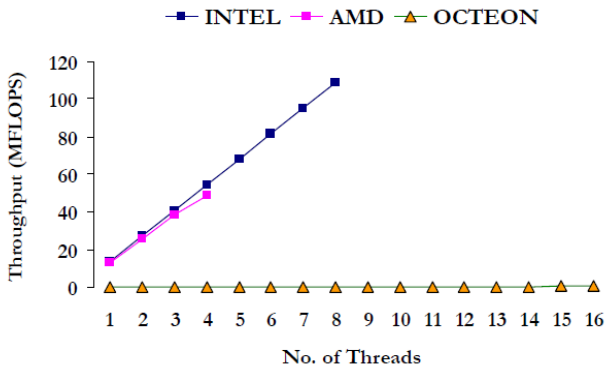

Figure 5.1 sin

Table 1. Specifications of Systetms Under Tests

Platform Attributes	Systems under Test		
Processor	Quad Core Intel® Xeon® E5405	Dual Core AMD Opteron Proc 2212HE	Cavium Octeon CN3860
CPU-Memory Bus Speed	1333 MHz FSB	1000 MHz Hyperport Bus	333 MHz
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

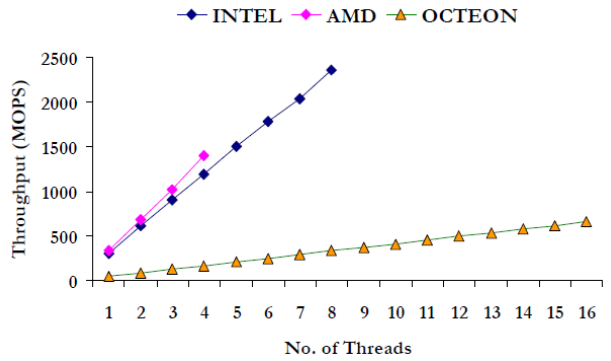

Figure 5.2 Summation

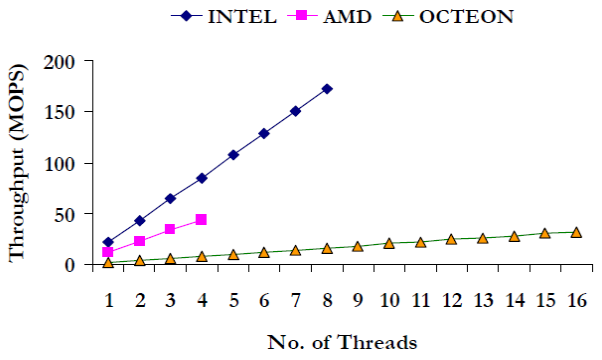

Figure 5.3 String Operation

4.2 Memory Benchmark

The MPAC memory benchmark takes the number of threads, data size, data type, affinity flag, and number of repetitions as input from user. To validate the results of MPAC, we compare it with steam benchmark's default results on the SUTS which is shown in table 2. The percentage of deviation is from 2 – 5 %, which is relatively small, thus validating the results of MPAC benchmarking. Fig. 6 shows memory throughput versus number of threads of MPAC memory 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 Fig. 6 (a), (b) and (c) that the throughput scales linearly[11]. Fig. 6 (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 of 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. 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 for a similar dual quad-core based system. On the other hand, throughput scales linearly for AMD and Cavium based SUT, for 16 MB of data size, due to their more efficient low-latency memory controllers instead of a shared system bus.

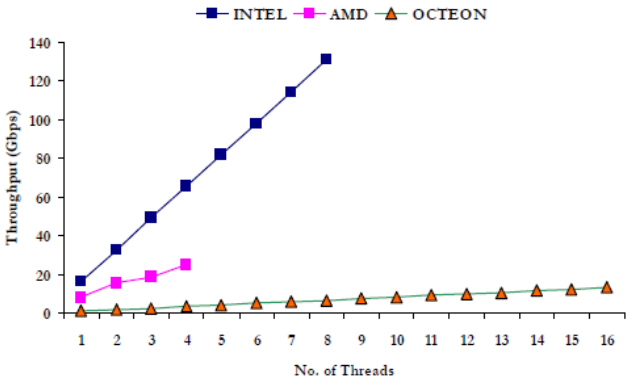

Figure 6.1: 4kb

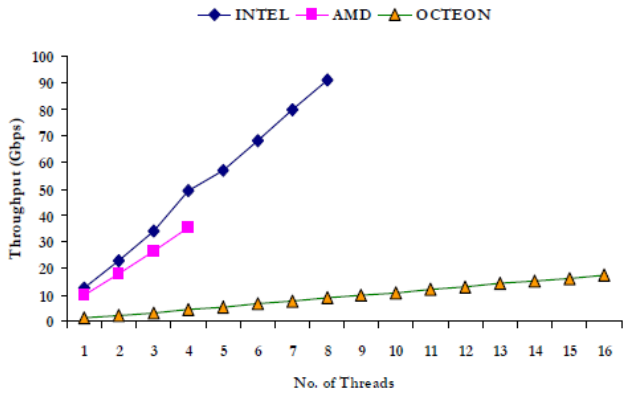

Figure 6.2 16Kb

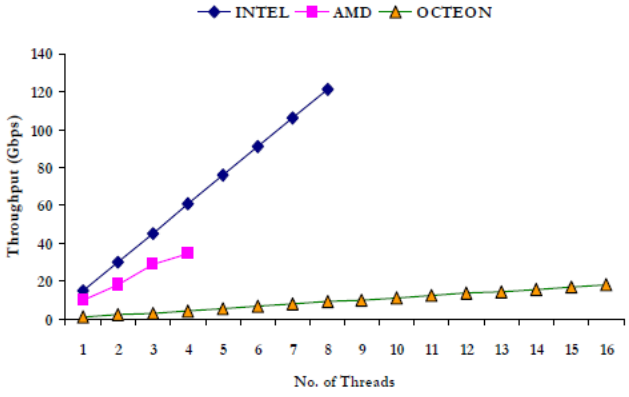

Figure 6.3 1Mb

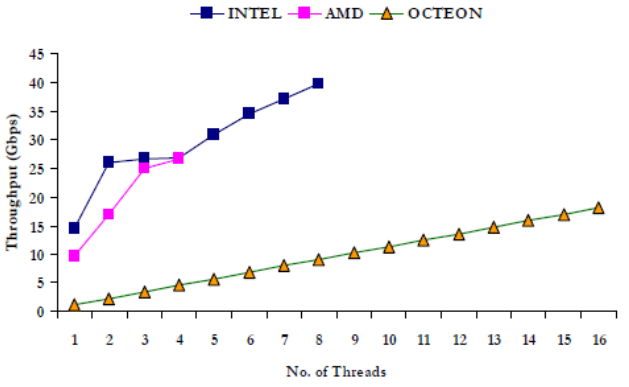

Figure 6.4 16Mb

Table 2. Throughput in Mbps of Memory-To-Memory Copy of 16 Mb Floating Point Data on Different SUTs for N=1

SUT	Stream Benchmark	MPAC Benchmark	% Deviation
Intel	27905	26434	5.3
AMD	16172	15744	2.6
Caviu	6.23	5.89	5.5

4.3 Network Benchmark

To validate the results of MPAC network benchmark, we compare the results with Netperf benchmark results on the SUTs. From the table 3, we can confirm that the deviation between Netperf benchmark results and MPAC network benchmark is too small and hence our results are valid. Fig. 10 presents scalability characteristics of the throughput of end-to-end network data transfer on different SUTs using MPAC network benchmark. TCP client and server threads send and receive message, respectively, using loop-back interface.

This use case exercise memory-to-memory copy throughput with TCP stack level processing within the kernel. However, this does not involve any traffic over physical network, which is limited to 1 Gbps throughput. Thus, using loop-back interface, we avoid the limitation of physical network throughput for running these network benchmark use cases to compare scalability characteristics across three architectures. An increase in throughput is observed across TCP client and server thread pairs when the number of threads increases. With more threads, scheduling overheads due to thread-exclusive TCP message dispatching for each client-server pair prevents hitting the bus throughput limit for Intel, AMD and Cavium SUT.

Table 3. Throughput in Mbps of End-To-End Network Data Transfer on Different SUTs For N=1 Using Loop-Back Interface

SUT	Netperf Benchmark	MPAC Benchmark	% Deviation
Intel	6760	6624	2.0
AMD	4276	4200	1.8
Caviu	2514	2467	1.9

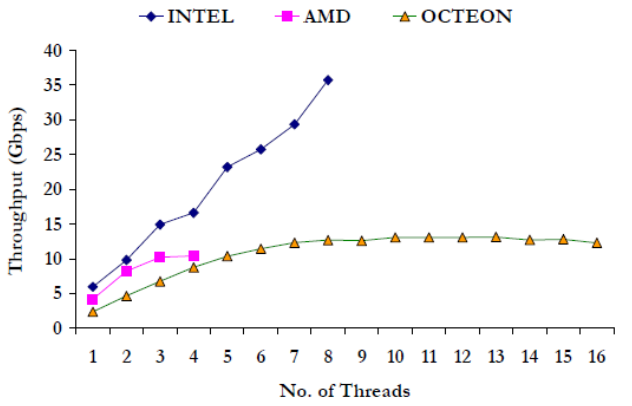

Figure 7. Throughput in Gbps of end-to-end network data transfer across number of threads for different SUTs

5. MULTICORE CHALLENGES

Having multiple cores on a single chip gives rise to some problems and challenges. Power and temperature management are two concerns that can increase exponentially with the addition of multiple cores. Memory/cache coherence is another challenge, since all designs discussed above have distributed L1 and in some cases L2 caches which must be coordinated. And finally, using a multicore processor to its full potential is another issue. If programmers don't write applications that take

advantage of multiple cores there is no gain, and in some cases there is a loss of performance. Application need to be written so that different parts can be run concurrently (without any ties to another part of the application that is being run simultaneously).

5.1 Power and Temperature

If two cores were placed on a single chip without any modification, the chip would, in theory, consume twice as much power and generate a large amount of heat. In the extreme case, if a processor overheats your computer may even combust. To account for this each design above runs the multiple cores at a lower frequency to reduce power consumption.

To combat unnecessary power consumption many designs also incorporate a power control unit that has the authority to shut down unused cores or limit the amount of power. By powering off unused cores and using clock gating the amount of leakage in the chip is reduced.

To lessen the heat generated by multiple cores on a single chip, the chip is architected so that the number of hot spots doesn't grow too large and the heat is spread out across the chip. As seen in Figure 7, the majority of the heat in the CELL processor is dissipated in the Power Processing Element and the rest is spread across the Synergistic Processing Elements. The CELL processor follows a common trend to build temperature monitoring into the system, with its one linear sensor and ten internal digital sensors.

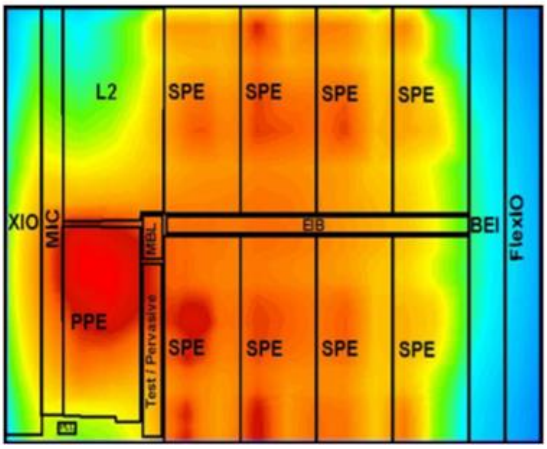

Figure 8 CELL Thermal Diagram

5.2 Cache Coherence

Cache coherence is a concern in a multicore environment because of distributed L1 and L2 cache. Since each core has its own cache, the copy of the data in that cache may not always be the most up-to-date version. For example, imagine a dual-core processor where each core brought a block of memory into its private cache. One core writes a value to a specific location; when the second core attempts to read that value from its cache it won't have the updated copy unless its cache entry is invalidated and a cache miss occurs. This cache miss forces the second core's cache entry to be updated. If this coherence policy wasn't in place garbage data would be read and invalid results would be produced, possibly crashing the program or the entire computer.

In general there are two schemes for cache coherence, a snooping protocol and a directory-based protocol. The snooping protocol only works with a bus-based system, and uses a number of states to determine whether or not it needs to update cache entries and if it has control over writing to the block. The directory-based protocol can be used on an arbitrary network and is, there- fore, scalable to many processors or cores, in contrast to snooping which isn't scalable. In this scheme a directory is used that holds information about which memory locations are being shared in multiple caches and which are used exclusively by one core's cache. The directory knows when a block needs to be updated or invalidated.

Intel's Core 2 Duo tries to speed up cache coherence by being able to query the second core's L1 cache and the shared L2 cache simultaneously. Having a shared L2 cache also has an added benefit since a coherence protocol doesn't need to be set for this level. AMD's Athlon 64 X2, however, has to monitor cache coherence in both L1 and L2 caches. This is sped up using the HyperTransport connection, but still has more overhead than Intel's model.

5.3 Multithreading

The last, and most important, issue is using multithreading or other parallel processing techniques to get the most performance out of the multicore processor. "With the possible exception of Java, there are no widely used commercial development languages with [multithreaded] ex- tensions." Rebuilding applications to be multithreaded means a complete rework by programmers in most cases. Programmers have to write applications with subroutines able to be run in different cores, meaning that data dependencies will have to be resolved or accounted for (e.g. latency in communication or using a shared cache). Applications should be balanced. If one core is being used much more than another, the programmer is not taking full advantage of the multi- core system. Some companies have heard the call and designed new products with multicore capabilities; Microsoft and Apple's newest operating systems can run on up to 4 cores.

6. OPEN ISSUES

6.1 Improved Memory System

With numerous cores on a single chip there is an enormous need for increased memory. 32-bit processors, such as the Pentium 4, can address up to 4GB of main memory. With cores now using 64-bit addresses the amount of addressable memory is almost infinite. An improved memory system is a necessity; more main memory and larger caches are needed for multithreaded multiprocessors.

6.2 System Bus and Interconnection Networks

Extra memory will be useless if the amount of time required for memory requests doesn't im- prove as well. Redesigning the interconnection network between cores is a major focus of chip manufacturers. A faster network means a lower latency in inter-core communication and memory transactions. Intel is developing their Quickpath interconnect, which is a 20-bit wide bus running between 4.8 and 6.4 GHz; AMD's new HyperTransport 3.0 is a 32-bit wide bus and runs at 5.2 GHz. A different kind of

interconnect is seen in the TILE64's iMesh, which consists of five networks used to fulfill I/O and off-chip memory communication.

Using five mesh networks gives the Tile architecture a per tile (or core) bandwidth of up to 1.28 Tbps (terabits per second). The question remains though, which type of interconnect is best suited for multicore processors? Is a bus-based approach better than an interconnection network? Or is there a hybrid like the mesh network that would work best?

6.3 Parallel Programming

To use multicore, you really have to use multiple threads. If you know how to do it, it's not bad. But the first time you do it there are lots of ways to shoot yourself in the foot. The bugs you introduce with multithreading are so much harder to find.

In May 2007, Intel fellow Shekhar Borkar stated that "The software has to also start following Moore's Law, software has to double the amount of parallelism that it can support every two years." Since the number of cores in a processor is set to double every 18 months, it only makes sense that the software running on these cores takes this into account. Ultimately, programmers need to learn how to write parallel programs that can be split up and run concurrently on multiple cores instead of trying to exploit single-core hardware to increase parallelism of sequential programs.

Developing software for multicore processors brings up some latent concerns. How does a programmer ensure that a high-priority task gets priority across the processor, not just a core? In theory even if a thread had the highest priority within the core on which it is running it might not have a high priority in the system as a whole. Another necessary tool for developers is debugging. However, how do we guarantee that the entire system stops and not just the core on which an application is running?

These issues need to be addressed along with teaching good parallel programming practices for developers. Once programmers have a basic grasp on how to multithread and program in parallel, instead of sequentially, ramping up to follow Moore's law will be easier.

6.4 Starvation

If a program isn't developed correctly for use in a multicore processor one or more of the cores may starve for data. This would be seen if a single-threaded application is run in a multicore system. The thread would simply run in one of the cores while the other cores sat idle. This is an extreme case, but illustrates the problem.

With a shared cache, for example Intel Core 2 Duo's shared L2 cache, if a proper replacement policy isn't in place one core may starve for cache usage and continually make costly calls out to main memory. The replacement policy should include stipulations for evicting cache entries that other cores have recently loaded. This becomes more difficult with an increased number of cores effectively reducing the amount of evitable cache space without increasing cache misses.

6.5 Homogeneous vs. Heterogeneous Cores

Architects have debated whether the cores in a multicore environment should be homogeneous or heterogeneous, and there is no definitive answer...yet. Homogenous cores are all exactly the same: equivalent frequencies, cache sizes, functions, etc. However, each core in a heterogeneous system may have a different function, frequency, memory model, etc. There is an apparent trade- off between processor complexity and customization. All of the designs discussed above have used homogeneous cores except for the CELL processor, which has one Power Processing Element and eight Synergistic Processing Elements.

Homogeneous cores are easier to produce since the same instruction set is used across all cores and each core contains the same hardware. But are they the most efficient use of multicore technology?

Each core in a heterogeneous environment could have a specific function and run its own specialized instruction set. Building on the CELL example, a heterogeneous model could have a large centralized core built for generic processing and running an OS, a core for graphics, a communications core, an enhanced mathematics core, an audio core, a cryptographic core, and the list goes on. This model is more complex, but may have efficiency, power, and thermal benefits that outweigh its complexity. With major manufacturers on both sides of this issue, this debate will stretch on for years to come; it will be interesting to see which side comes out on top.

7. Conclusion

Before multicore processors the performance increase from generation to generation was easy to see, an increase in frequency. This model broke when the high frequencies caused processors to run at speeds that caused increased power consumption and heat dissipation at detrimental levels. Adding multiple cores within a processor gave the solution of running at lower frequencies, but added interesting new problems.

We presented open-source MPAC benchmarking library that provides a common extensible benchmarking infrastructure. It can be leveraged to ease the development of specification-based micro-benchmarks, application benchmarks, and network traffic load generators for state-of-the-art multi-core processors based platforms. We implemented the specifications of Stream and Netperf micro-benchmarks using MPAC library and validated our MPAC based performance measurements on Intel, AMD, and Cavium based multi-core platforms using these benchmarks for single thread executions.

Multicore processors are architected to adhere to reasonable power consumption, heat dissipation, and cache coherence protocols. However, many issues remain unsolved. In order to use a multicore processor at full capacity the applications run on the system must be multithreaded. There are relatively few applications (and more importantly few programmers with the know-how) written with any level of parallelism. The memory systems and interconnection networks also need improvement. And finally, it is still unclear whether homogeneous or heterogeneous cores are more efficient.

8. REFERENCES

[1] W. Knight, “Two Heads Are Better Than One”, IEEE Review, September 2005

[2] R. Merritt, “CPU Designers Debate Multi-core Future”, EETimes Online, February 2008

[3] P. Frost Gorder, “Multicore Processors for Science and Engineering”, IEEE CS, March/April 2007

[4] D. Geer, “Chip Makers Turn to Multicore Processors”, Computer, IEEE Computer Society, May 2005

[5] L. Peng et al, “Memory Performance and Scalability of Intel‟s and AMD‟s Dual-Core Processors: A Case Study”, IEEE, 2007

[6] D. Pham et al, “The Design and Implementation of a First-Generation CELL Processor”, ISSCC

[7] P. Hofstee and M. Day, “Hardware and Software Architecture for the CELL Processor”, CODES+ISSS ‟05, September 2005

[8] J. Kahle, “The Cell Processor Architecture”, MICRO-38 Keynote, 2005

[9] D. Stasiak et al, “Cell Processor Low-Power Design Methodology”, IEEE MICRO, 2005

[10] D. Pham et al, “Overview of the Architecture, Circuit Design, and Physical Implementation of a First-Generation CeCell Processor”, IEEE Journal of Solid-State Circuits, Vol. 41, No. 1, January 2006

[11] M. Hasan Jamal, Ghulam Mustafa, Abdul Waheed and Waqar Mahmood, An Extensible Infrastructure for Benchmarking Multi-Core Processors based Systems, IEEE SPECTS 2009

[12] Mikiko Sato, Yuji Sato, Member, IEEE and Mitaro Namiki, Member, IEEE, Proposal of a Multi-core Processor from the Viewpoint of Evolutionary Computation, IEEE 2010