AFFECT OF PARALLEL COMPUTING ON MULTICORE PROCESSORS

V R Rao Konkimalla[1] and Niraj Upadhayaya[2]

[1]Department of CSE, JB Institute of Engineering Technology, Hyderabad
Andhra Pradesh - 500075, INDIA
kvrrao@jbiet.edu.in
[2]Dean Academics, JB Institute of Engineering Technology, Hyderabad
Andhra Pradesh - 500075, INDIA
niraj@jbiet.edu.in

ABSTRACT

Our main aim of research is to find the limit of Amdahl's Law for multicore processors, to make number of cores giving more efficiency to overall architecture of the CMP(Chip Multi Processor a.k.a. Multicore Processor). As it is expected this limit will be in the architecture of Multicore Processor, or in the programming. We surveyed the architecture of the Multicore processors of various chip manufacturers namely INTEL™, AMD™, IBM™ etc., and the various techniques there followed in, for improving the performance of the Multicore Processors.

We conducted cluster experiments to find this limit. In this paper we propose an alternate design of Multicore processor based on the results of our cluster experiment.

KEYWORDS

Amdahl's Law, Architecture, Chip Multicore Processor, Performance and Speedup

1. INTRODUCTION: MULTICORE PROCESSOR ARCHITECTURES

As we know a processor (CPU) will have 1) The front end for instruction issue and 2) The Rear end as execution engine, a Multicore processor is a one that is having one Front end to issue instructions and a number of Execution engines or cores for execution.

The Multicore architectures of both *INTEL*™ [1] and AMD™ [2] look almost alike except that AMD™ has an additional cache attached to the core itself as shown in the Figures (Fig.1 and Fig.2) below. A Multicore processor will have at least two levels of cache hierarchy, as per the placing of theL_2 (or L_3) cache.

Jan Zizka (Eds) : CCSIT, SIPP, AISC, PDCTA - 2013
pp. 41–54, 2013. © CS & IT-CSCP 2013 DOI : 10.5121/csit.2013.3605

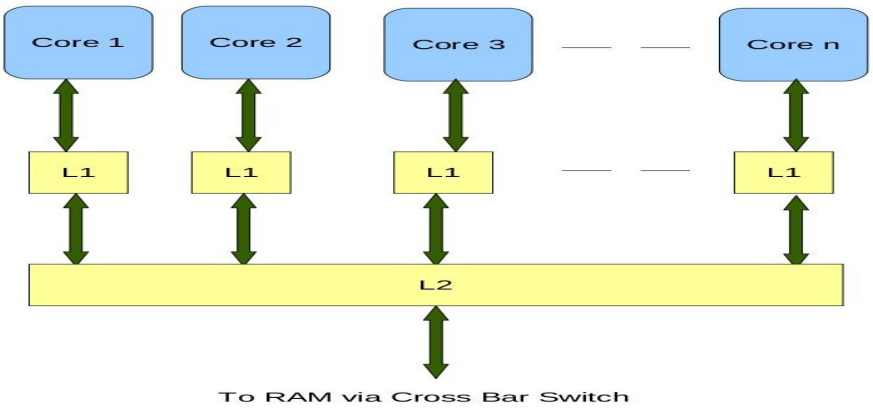

Figure 1 INTEL™ Multicore Architecture

In this paper we present a detailed discussion on the various factors affecting the performance of a Multicore processor. Next section (Background) will talk on the various techniques followed for improving the performance of a CMP. In section 3 we detail the various factors affecting CMP. In section 4 problems in a CMP along with analysis of our cluster experiment results. In section 5 we conclude with proposal of a Multicore Design for better performance.

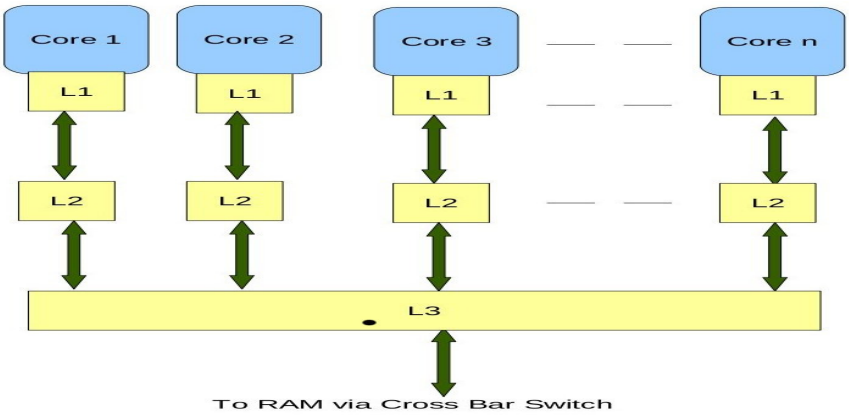

Figure 2 AMD™ Multicore Architecture

2. BACKGROUND: SPEED-UP IN MULTICORE PROCESSORS

In the present day scenario processor performance is of the highest interest to the end users, as well all the chip manufacturers are coming out with Multicore processors also known as Chip Multi Processors (CMPs), which we call the Next generation Processors. These Multicore processors are the present day state-of-the-art processors and all the Chip manufacturers concentrate on how to improve the number of cores on the die with the increased number of cores year after year i.e. Dual core, Quad core, Oct core etc. At the same time it needs to exploit the techniques for efficient use of these cores to achieve more parallelisation or to gain speed-up.
To gain speed-up on these Multicore processors several techniques were identified by many researchers that are listed here under:

1. Cascaded Execution,
2. Execution Migration,

3. Speculative Pre-computation,
4. Dynamic Speculative Pre-computation,
5. Dynamic Prefetching thread,
6. Core spilling.

2.1. Cascaded Execution

Due to the presence of inherently sequential code and because of the certain limitations of paralleling compilers, many application programs suffer with speed-up on SMP processors. Cascaded execution is a technique followed on SMP where the non-parallel loops will be distributed across multiple processors for their sequential part, but only a single processor executes the loop body.

In cascaded execution the processors that are idle will execute the helper thread to optimize their memory state [3]. In the helper phase a processor predicts and loads the data that is needed immediately in near future onto the shared memory and all the processors alternatively switch between helper and execution phases, but only one processor will be in execution phase and all other processors are in their helper phases of the non-parallel loop body.

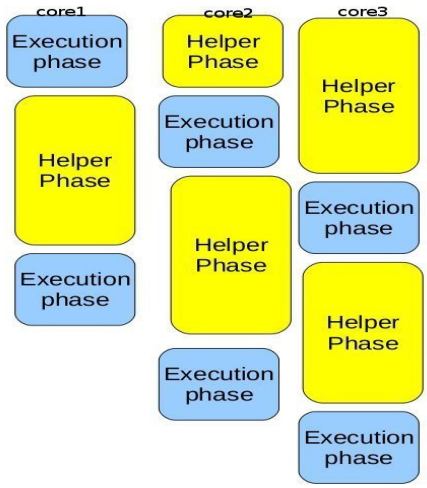

Figure 3 Cascaded Execution

In general a helper phase is responsible either for Prefetching or for sequential buffer data restructuring.

2.2. Execution Migration

In this technique a sequential program can migrate from one core to another automatically during execution. This is made possible, by making use of the different levels of caches available on a Multi core processor, by bringing a lot of cache capacity closer to the execution core and hence reducing the average memory access latency. When a single threaded process (or a sequential task) is executing on a multi core processor, the processor uses only one core and the cache capacity on the other core is wasted.

The Execution migration technique uses one characteristic called split-ability of the working sets to identify the cache misses for migration using the affinity algorithm [4].

Execution migration will take place from one core to another basing on the principle "If we cannot bring the data to the processor as fast as we would like, we could instead take the processor to the data" as proposed in [5].

In general if the working set is not cache-able on to a single L_2 it is possible to fit in the overall L_2 cache, so assign the single sequential task to the complete processor and migrate the execution with some frequency in a random access policy. For better results it was suggested by [6] to go for circular accesses where the frequency of migration will be around ½.

The performance loss due to this migration on a Multicore with private L_1 and a shared L_2 can be minimized if: (a) a migrating thread continues its execution on a core that was previously visited by the thread, and (b) cores remember their predictor state for the previous activation. Dean. M. Tullsen, Susan. J. Eggers and others in [6] presented an architecture for simultaneous multi-threading a.k.a. Hyper threading, that is best suited for (a) minimizing the architecture of the conventional super scalar design, (b) providing higher throughput with multi threaded execution and is also best suited for single threaded execution (when executing alone).

2.3. Speculative Pre-computation

Day by day processor frequencies are increasing compared to memory access times and hence memory latency dominates the processor performance. One way to over come this is to use the technique of SMT. But this will not improve the performance of a sequential program (single-threaded execution). H.Wang, J. P. Shen., et, al; propose a technique called Speculative Pre computation (SP) on a SMT [7] or a CMP [8] processor, which uses the idle hard ware thread context to execute Speculative threads. SP is a special Prefetch mechanism that identifies load instructions that are difficult to Prefetch such as chains of loads. SP concentrates on delinquent loads, as they cause more than 80% of L_1 cache misses. The addresses accessed by the delinquent loads are extracted by a set of dependent instructions known as Pre computation slices (p-slices). These P-slices were executed by speculative threads, i.e. when a speculative thread is spawned, it pre-computes the address accessed by a future delinquent load and Prefetch the data. This speculative thread spawning happens in two ways.

- Basic trigger: - when an instruction in the non-speculative thread reaches the commit stage initiates the speculative thread spawn it is called basic trigger.
- Chaining trigger:- The speculative thread explicitly spawning another speculative thread.

The process of SP requires the following tasks:

1. Identification of the delinquent loads:-These are identified by memory access profiling, which is done by either the Compiler or by a memory access simulator. The load instructions that have major impact on performance are selected as delinquent loads (using L_1 misses major, some times L_2 or L_3 or memory misses may also be considered).
2. Construction of Pre computation slices (p-slices):-At the time of delinquent load execution, the instructions that were executed around 128 instructions prior to are identified as a basic trigger, if the same delinquent load is executed for each time this instruction is executed, the trigger is confirmed and the p-slice is constructed using the load instruction. These constructed p-slices were subjected to optimization by eliminating the redundant triggers and useless triggers, and
3. Establishment of the triggers (Basic and Chaining):- For this the p-slices will be added to the object code.

As the Prefetching threads run independently from the main thread, and due to the reason the addresses are calculated using the code executed from the main thread the technique of SP are more efficient than regular Prefetching techniques. The SP spawns the speculative thread either at the rename stage of the trigger or at the commit stage. In both the cases there is a need for the H/W support so as to copy the Live-in values to the child thread's context from the main thread's context. It is observed that on a simulation higher performance is possible with more thread contexts as it provides more possibilities for more speculation, but in a realistic environment there is a possible degrade of performance due to the overhead of speculation, when using basic triggers, as there is a possibility for the main thread to stall as each speculative thread will occupy a thread's context for a longer time period. These problems were overcome by chaining triggers.

The p-slices for the chaining trigger will have 3 parts :

1. A prologue,
2. A Spawn instruction for spawning another copy of p-slice and
3. An epilogue.

The prologue contains the loop induction variables and the epilogue contains the actual instruction to produce the address of the delinquent load. Spawning a thread via a chaining trigger will have less spawning overhead as trigger has no action from the main thread, and the speculative thread directly stores the live-in values on to the Live-in buffer and spawns the child thread. In case of a processor with a few number of thread contexts it is better to have a pending slice queue (PSQ) for maintaining pending p-slices when all thread contexts are busy and the chaining trigger spawns speculative threads.

As the Chaining trigger independently spawns the child threads there is a need to verify that

- The speculation is not too aggressive i.e. the data required by the main thread is not evicted as well,
- The spawning should not continue after the main thread was exited.

2.4. Dynamic Speculative Pre-computation

J.D. Collins, H.Wang and D.M.Tullsen, et, al, proposed a technique called Dynamic Speculative Pre computation (DSP)[9], which performs all necessary instruction analysis, extraction, and optimization with the help of back-end instruction analysis hardware, located off the processor's critical path. DSP is hard ware controlled thread based model of the software based manually constructed SP of the previous subsection.

This architecture uses:-

a) Delinquent Load Identification Table for identifying delinquent loads,
b) Retired Instruction Buffer for construction of pre computation slices and
c) Spawn Instruction Table for spawning instructions.

2.5. Dynamic Prefetching thread

To better utilize the resources of the CMP is becoming a difficult task for the sequential programs. Hou Rui, Long bing Zhang, Weiwu Hu in their paper [10] identify a hardware generated Prefetching helper threads to accelerate sequential programs known as Dynamic Prefetching Thread (DPT), with two aggressive thread construction policies, "Self-Loop" and "Fork-on-recursive-call".

In the basic policy The Dynamic Prefetching Thread (DPT) is constructed automatically with the delinquent loads which are the main cause for the memory stall cycles, where in the DPT is run on the idle core of a CMP which gets its context from the “Shadow register”.

The ‘Shadow Register’ is a memory component added to the hardware in the design, between the two cores of the dual core processor, and maintains the same data with the registers of the main core (the core executing the main thread). For the shadow register, the main core running the actual program will have write permission and the other core running the helper thread will have only read permissions. With this basic thread construction policy the speedup is very low due to limited Prefetching coverage.

- Self-Loop policy: - In the basic policy the delinquent load is dispatched to the idle core where as in the 'Self-loop' policy, this delinquent load is dispatched to the idle core to Prefetch the next instance of the same delinquent load, i.e. the next instance of the same delinquent load is Prefetched into the same core by the same DPT, so that more instances of the same delinquent load is available on the same core to reduce the memory contention of the shadow register.
- Fork-on-recursive-call: - The basic policy as well as the "Self-loop" policy cannot improve the performance of the sequential programs that use tree or graph like linked data structures. Both these data structures 'will have nodes connected to adjacent nodes, which can be felt like a node of the graph or tree connected to another sub-graph or sub-tree, accordingly the data structures can be invoked by a recursive function call, and these recursive function calls are identified as a Prefetching thread that can be dispatched to an idle core. These two policies were considered together in the thread construction policy with higher priority for “Fork-on-recursive-call” than the “Self Loop” policy to achieve higher speed-up.

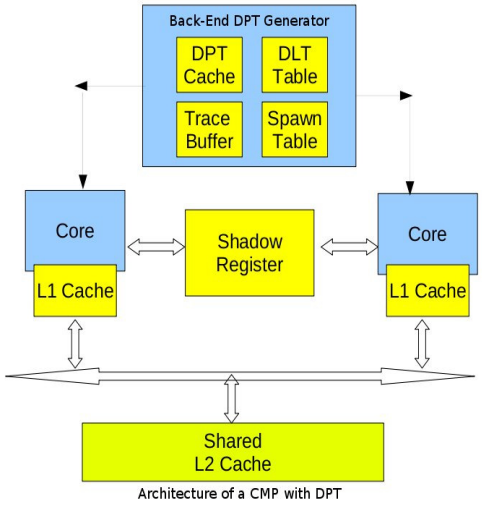

Figure 4 Architecture of Dual core with DPT

2.6. Core spilling

Logically CMP should give speed-up on sequential applications also, due to higher transistor density available. One way to create CMP is to emulate a cluster with shared common fetch and dispatch units. In [11] Cong, J. Han Guoling Jagannathan, et,al., Went orthogonal to this approach and proposed the scheme for statically partitioned cores (Which may be clustered subsequently).

One bottleneck with clustering approach was centralized cluster scheduling unit, limiting scalability. Core Spilling method introduced by them, claims to overcome this giving higher scalability.

Core spilling technique presented for single threaded application includes following functional parts:

1. An application can continue the speculative execution, even after all available resources like Physical register and Scheduling Window.
2. A Core Spill occurs when either the Instruction Window or the Scheduling Window fills up on a core.

For any given spill two cores are involved.

a) **Parent** which is executing programme before the spill and
b) **Child** that receives the spill from the Parent.

Spilling can be implemented by augmenting each core with two additional registers:

1. Spilling Register: It is single bit register to show the status of Core Spill (Happened / Not Happened).
2. Descendent Register: This keeps core ID of the Child of given core.

Along with one Register File, Store Buffer and In-Flight Store need to be created. Core spilling performance can be improved by

1. Prefetching,
2. Locality based Filtering.

2.7. Comparison of the different techniques

The different techniques discussed above claim the following speed-up:

- Speculative Pre computation on Chip Multi Processors - 10 to 12%
- Dynamic Speculative Pre computation (simple p-slices) – 14%
- Dynamic Speculative Pre computation (aggressive optimizations) -33%
- Dynamic Prefetching Thread (basic thread) - 3.8%
- Dynamic Prefetching Thread (aggressive thread construction policies) - 29.6%
- Core Spilling – 40%.

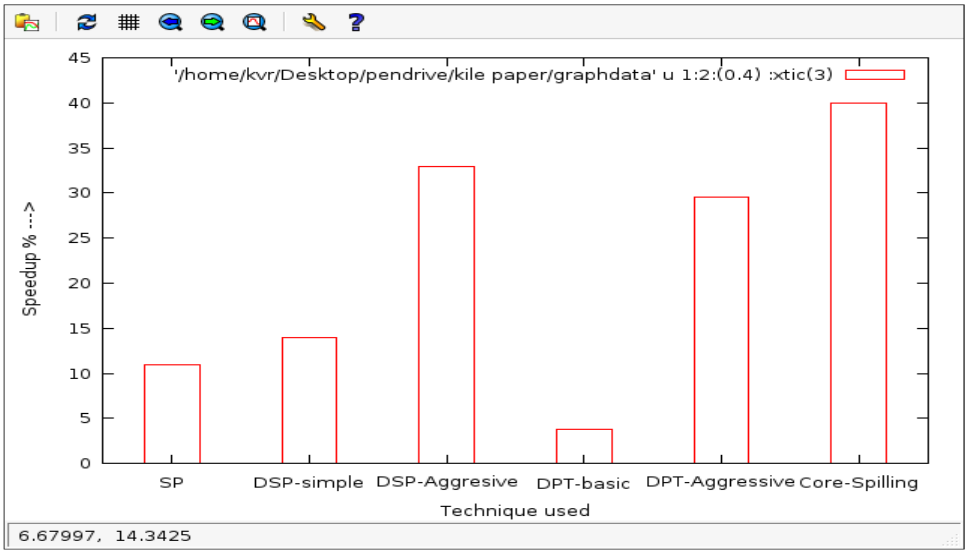

Figure 5 Comparison of the different techniques for speedup on a Multicore

3. FACTORS AFFECTING PERFORMANCE

The CMPs are not mainly designed for performance improvement but to reduce the heat produced. As per Murphy's Law something that is evolved for a particular characteristic / reason can not yield something else, of course the improvement in performance of a multicore is a by-product due to the multiprocessing capabilities of the CMP.

Various Factors: *There are many factors that affect the Performance of the CMP, they are:*

- Number of Cores,
- Effective cache,
- Core Speed,
- Density of element and
- Amdahl's Law.

For this discussion we are considering the Multicore Architectures of two competing processor manufacturers namely INTEL™ & AMD™ that produce two different designs of the CMP. AMD™ having L_2 & L_3 where as INTEL™ is having simply L_2 (refer to Figure 1 & 2).

3.1. Number of Cores

Performance of a CMP is linearly proportional to the no of cores where by a processor can accommodate a number of process contexts. As

- Higher number of Cores can go for higher number of Processes depending on the availability,
- In case of non-availability of Processes other Cores can go for other tasks like Prefetching and Profiling to increase the overall throughput,

- As well in such a situation some of the cores will be involved in overall activity and fault tolerant processing

Performance α C

Accordingly Multicore processors' performance is always better than a single-core processor and this can deteriorate only on one condition that there is a single process for execution, as well the process and data is sequentially sharp and is having lot of jumps where by profiling and/or Prefetching is not possible to increase the speedup or throughput.

The constant of proportionality in the above equation depends on several factors like:

1. The number of parallelisable processes,
2. The number of cores involved in supporting process like Prefetching and profiling,
3. Parallelism of the Job (Inherent parallelism) and
4. Amdahl's Law.

3.2. Effective cache

The degree of multiprocessing depends on the total number of process contexts available in the memory, Which in turn depends on the total available cache to a core, this is in-line with the 'Stored Program Concept', Which states that a task will get done only when its context is available in the memory. In general the size of cache controls the degree of multiprocessing as it accommodates the process contexts. So the degree of multiprocessing will mainly depend on the total available cache (L_2 or L_3) per the core but not on the entire Chip.

Effective Cache = {Cache / Core} α L_2 / core + L_3
(in case of AMD™)

Effective Cache = {Cache / Core} α L2
(in case of INTEL™)

The processing performance depends on 1). Seek time and 2). Fault frequency, of the two performance depends linearly on the seek time but exponentially on fault time, when a fault occurs the performance goes down, if more processes are there more faults will occur to degrade the performance in AMD™ design, in spite of its superior design. One major reason for this is that AMD™ is still at 45 nm technology where as INTEL™ is at 32 nm technology, so that INTEL™ can afford to install more L_2; as memory requires more density of element. Performance of a CMP is linearly proportional to the seek time and the size of Effective cache.

$$\text{Performance} \alpha T_s * L$$

The performance is inversely proportional to the memory fault at exponential order to the Effective cache size.

$$\text{Performance} \alpha T_f / \log L$$

Where T_s is the seek time, T_f is the fault time and L is the Effective cache

Having L_2 & L_3 is not a technical advantage of AMD™ but they are countering INTEL™ by reducing the cost of the processor. Until and unless seek time reaches the fault time AMD™ can not demolish the drawback of lower performance but is never possible as when a fault occurs data and/or instruction are to be fetched from a slow speed memory.

3.3. Core Speed

The performance also proportional to the core speed, as in a CMP there are a number of cores working at the same frequency, Performance is proportional exponentially to the core speed at the order of the number of cores. When the Core speed is high the number of mnemonics executed is high. Mnemonics executed depends on T cycle, if the Core Speed is high, more T cycles are executed in a single unit of time and hence more number of instructions gets executed in time also increase, accordingly throughput per core.

$$\text{Performance} \alpha C_s^{\,n}$$

Where C_s is the core speed and 'n' is a constant.

The value of 'n' depends on several factors like Limit of Amdahl's Law, Inter communication between the cores, losses due to cache conflicts and inherent parallelism of code, and etc.

3.4. Density of element

If density is higher it will affect the performance, because more memory can be put in the die which will reduce the number of faults as there is more space on the die, INTEL™ can afford for more L_2 and / or more cores. Accordingly there will be an automatic improvement in performance and so is the case of INTEL™ design. At Higher density one can afford for more core or element which automatically means more functional units and /or more threads per core and hence improvement in performance is exponential.

$$\text{Performance} \alpha D^m$$

Where D is the Density and 'm' is a constant.

The value of 'm' depends on other factors like number of interconnects. INTEL™ processors are performing better due to an additional reason that they have removed interconnects or having pin less design, and they are in the die, as at 32 nm technology it is very difficult to put the pin. Interconnects are higher implies impedance is more, between each interconnects there is a distance which is inversely proportional to capacitance. Which means more impedance producing more heat to counter this there is no other go other than reducing the frequency that consume lesser wattage.

So higher density will automatically necessitate more core running at lesser frequency and more core with more memory will provide better performance is the secret of INTEL™. INTEL™ is going for Multicore not because it wants more core but it has reached to the extent that it should go for Multicore where as AMD™ went to Multicore just to demonstrate Multicore, that is the reason Hex core of AMD™ came to the market much earlier than Hex Core INTEL™. INTEL™ Multicore can be over clocked where as AMD™ can not be. That is very clearly evident as INTEL™ is using technology only when it is needed and Core 2 Duo or Dual Core are very popular, AMD™ is trying to increase the number of cores to increase the degree of multi processing with 45 nm technology and the Effective cache per Core (per Core L_2 + total L_3) is not increased, so with lesser sized effective cache when more process contexts are available the total number of faults are also more which degrades the performance to a higher level.

3.5. Amdahl's Law

The Amdahl's law for parallel speed up is given by the formula

$$\text{Speedup} = T(1) / T(N)$$

Where T (N) is the time it takes to execute the program when using N number of processors [12, 13]. In case of parallelism this law states that if P is the proportion of a task that can gain benefit from parallelisation and **(1 - P)** is the part that cannot go parallel, remains serial, then the maximum speed up that can be achieved by using N processors is given by **1/(1-P+(P/N))**. When N is large i.e. under the limit N tends to Infinite, the speed up is given by **1/ (1-P)** which is independent of N, the number of processors (cores) and there is no improvement with increasing the number of processing elements. This means there is an upper limit of parallelisation for the task accordingly the number of cores in multi core processor.

4. PROBLEMS IN A MULTICORE PROCESSOR

A Multicore Processor is bundled with a few Problems:-

- In a Multicore Processor not only the rate of data transfer but also the amount of data transfer will have its impact on the throughput or speedup as it has to go for a number of tasks (processes) in parallel, accordingly Not only the Core Speed but also the size of cache (Effective cache) is of major concern and
- As per Amdahl's Law the number of cores for parallel execution is also another factor that limits the performance.

We categorized these problems of the Multicore into:-

Performance: - Performance of the Multicore processor should increase with enhancement of the number of cores.

Reliability: - Reliability of the Multicore processor should improve with improvement in the number of cores.

Backward Compatibility: - The Multicore processor should be backward compatible with all the software already in use, or other wise due to the increase in number of cores what ever the software technique already in use should not become obsolete.

Localized Processing: - The Multicore Processor should localize processing at each core as much as possible; the best example here is the case of a Neural Network where in a lot number of neurons working in parallel achieving highest performance with reliability.

4.1. How to improve the Multicore Processor's Performance

There are few things to be considered at this point

Collaboration among the Cores: - The Major contribution of the Multicore Processor Performance is due to the number of Cores, that does parallel execution of the task. So performance definitely depends on the arrangement of these cores and their inter connecting bus for improved collaboration among the cores. Accordingly the design should provide a mechanism for enhanced collaboration among the cores with the help of a communication channel among the cores.

Limit of Collaboration: -In case highest collaboration is provided there will be more communication between the cores that becomes a overhead for the communication channel that is provided.

What is the Limit of Amdahl's Law: - To identify this limit of parallelization, we conducted an experiment that is described in [14].

4.2. Finding the Limit of Amdahl's Law

With the proposed design it is possible to combat with the architectural problems in Multicore Processors. It is required, now to identify the Limit of Amdahl's Law for Multicore Processors. For this We at J.B. Institute of Engineering & Technology - Hyderabad tested the performance of the cluster by adding a number of nodes one after another and plotted the G Flops against Cluster size with HPL Benchmark [15] on PelicanHPC [16] Cluster. For more details one can refer our paper [14].

4.3. Important Observations

The results of our experiment show that there is no improvement in speed-up after 8 cores, meaning that they are in line with the Amdahl's law that predicts the maximum number of parallelisable Cores. This state of the processor is termed as 'The Multicore Bottleneck'.

4.4. Reasons

The problem is due to the communication bottleneck with the bus that connects the last level cache to the RAM via Cross Bar Switch, whose bandwidth is not equal to the sum of the bandwidths of the individual cores that connects first level caches to that off last level. Accordingly not all the cores can receive or send Data/Instructions simultaneously. This situation restricts to keep majority of the cores in idle state, in case the number of cores are increased (The value of N in the Amdahl's law) in a processor makes more cores to be idle and no gain in speed-up. In 2010 itself the amount of speed-up on average is 0.82 with N =6 cores and is expected to be 0.87 in 2013 with N= 32 cores and to have the remaining 12% (for 0.99) with N=infinite or around some thousands of cores! Is it reasonable to go for the thousand core processors, to gain this small amount of improvement in efficiency or speedup?

5. OUR PROPOSED DESIGN OF THE MULTICORE PROCESSOR

5.1. Design Objectives

On the basis of our observations, we propose that processors will perform better when designed keeping the following points in consideration:

a) A number of Cores with in the limits of Amdahl's law,
b) Possibly largest size of Effective cache, or to make all the cache as Effective cache for all the Cores,
c) L_2 cache to be designed as shown in Fig. 18 so that each L_2serves a number of Cores,
d) All the L_2caches are set-associative with other L_2caches, for improved collaboration among the cores
e) Design at higher density of the element to facilitate more core and large sized L_3cache,

f) Running at maximum possible Core Speed (with in the limits of power budgeting) so as to keep T cycle low. So as to execute a number of mnemonics in a given slot of time.

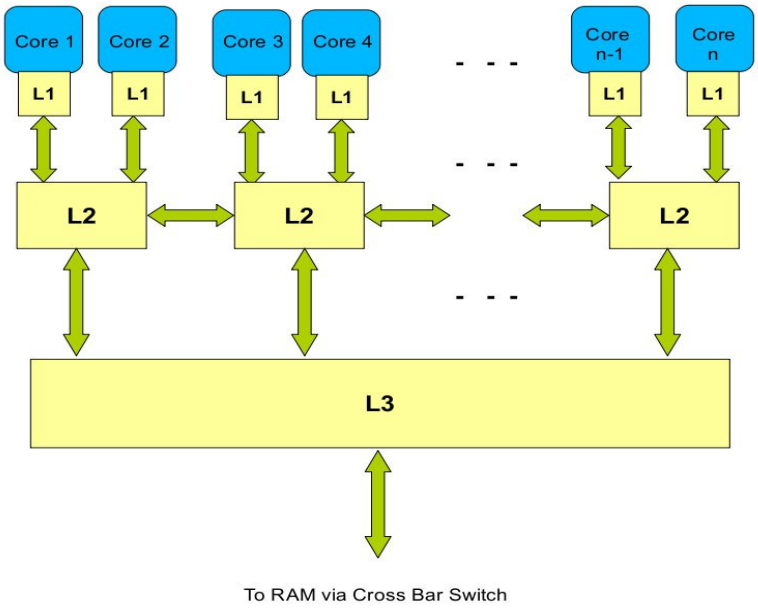

Figure 6 Our Proposed Multicore Processor Architecture

5.2. Advantages with the proposed design

Our proposed design is a hybrid model that will take into consideration the advantages of both INTEL™ and AMD™

- ✓ Our design proposes the L_1 cache as in AMD™ to improve localized processing at each core,
- ✓ All the L_2 caches are set-associative with other L_2caches so that all the Cores can go for cooperative processing in the absence of multiple tasks,
- ✓ When limited number of tasks are present, the second core attached to each L_2 can be used for running the helper thread under Cascaded Execution policy,
- ✓ The L_2 cache can be used as the Shadow Register for Dynamic Prefetching Thread Execution policy,
- ✓ When Executing a single sequential task all the L_2 and L_3can be used by any core to accommodate a large amount of data and / or instructions in memory for the sequential execution performance improvement,
- ✓ Our design proposes a Higher Density of element as in INTEL™ so as to make room for all three Levels of cache in the chip.

REFERENCES

[1] Simcha Gochman, Avi Mendelson, Alon Naveh, Efraim Rotem, Introduction to INTEL™ Core Duo®Processor Architecture INTEL™Technology Journal Volume 10, Issue 02, May 15, 2006 ISSN 1535-864X.

[2] AMD™ 64 Architecture Programmers Manual Volume 1: Application Programming Advanced Micro Devices Publication No.:24592 Revision: 3.14 Date: September 2007 page: 97.

[3] R.E.Anderson, Thu. D.Nguyen, et. al., Cascaded Execution: Speeding up unparallelised execution on shared memory multiprocessors. In proceedings of the 13th International parallel processing symposium.

[4] Pierre Michaud ; Exploiting the cache capacity of a single-chip Multi core processor with execution migration, In proceedings of 10th International symposium on High performance computer Architecture, 2004.
[5] Garcia-Molina, Lipton, Valdes, 'A Massive Memory Machine', IEEE Transactions on Computers, May 1984.
[6] Dean M. Tullsen , Susan J. Eggers , Joel S. Emery , Henry M. Levy , Jack L. Lo , and Rebecca L. Stammy; Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor ; In Proceedings of the 23rd Annual International Symposium on Computer Architecture, Philadelphia, PA, May, 1996.
[7] J.D.Collins, H.Wang, D.M.Tullsen, C.Hughes, Y.Lee, D.Lavery, J.Shen; Speculative Pre computation: Long range Prefetching of delinquent loads; In proceedings of the 28th Annual International symposium on computer Architecture, july 2001.
[8] J.A. Brown, H. Wang, G. Chrysos, P. H. Wang, and J. P. Shen; "Speculative Pre computation on Chip Multiprocessors". In Proceedings of MEAC 6, November 2001.
[9] J.D.Collins, H.Wang, D.M.Tullsen, C.Hughes, Y.Lee, D.Lavery, J.Shen; Dynamic Speculative Pre computation: In Proceedings of the 34th International Symposium on Micro architecture, December, 2001.
[10] Hou Rui, Long bing Zhang, Weiwu Hu; Accelerating sequential applications on Chip Multiprocessors via Dynamic Prefetching thread ;Science Direct journal of microprocessors and micro systems Elsevier 2007.
[11] Cong. J, Han Guoling Jagannathan, A. Reinman. G, Rutkowski. K; Accelerating Sequential Applications on CMPs Using Core Spilling; In proceedings of IEEE transactions on parallel and distributed systems 2007, volume 18, pages 1094-1107
[12] G.M.Amdahl, 'Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities', Proceedings of American Federation of Information Processing Societies Conf., vol. 30 (Atlantic City, N.J., Apr. 18-20)AFIPS Press, 1967, pp. 483-485.
[13] J.L. Gustafson, 'Re evaluating Amdahl's Law,' ACM communications, May 1988, Volume 31, Number 5 pp. 532-533.
[14] K.V.Ranga Rao and Niraj Upadhayaya; The Multi core Bottleneck in International Journal of Multidisciplinary Research and Advances in Engineering (IJMRAE) ISSN 0975-7074 Volume 2, No. II July 2010 pages 149-159.
[15] http://www.netlib.org/benchmark/hpl
[16] http://pareto.uab.es/mcreel/PelicanHPC/download/pelicanhpc-v1.8-64bit.iso.

Authors

K.V Ranga Rao, obtained his Bachelor degree in Computer science & Engineering from the Nagarjuna University, Guntur, Master of Science in Software Systems from BITS (DLPD), Pilani and Master of Technology in Computer Science from JNT University Hyderabad. He is currently working as Associate Professor in the Department of CSE of JB Institute of Engineering & Technology, Hyderabad and has submitted his Ph.D in CSE (Parallel Computing) to JNTUK -Kakinada. His research interests include high-performance computer architectures.

Dr. Niraj Upadhyaya, Ph.D., is an eminent scholar, professor in the CSE Department and Dean of Academics at J.B. Institute of Engineering & Technology, Hyderabad. He has his name established in the field of "Parallel Computing" and has many articles, papers and journals to his name. He had his Ph.D. from the University of West of England, Bristol, UK with the topic of "Memory Management of Cluster HPCs". He has more than 20 years of experience which helps the students to always learn from him.