# Memory Hierarchies for Multi-Core Processors

Ricardo Orozco, Hai-Yue Han, Richard Schafer, Taylor Johnson

ECE 570: High Performance Computer Architecture Winter 2010

## Table of Contents

# Introduction

## Project Description

This project intends to do three things: (1) give a brief introduction to some of the issues that make multi-core memory hierarchies different from single-core memory hierarchies, and from one another; (2) examine the memory hierarchy of some processors that are currently on the market from Intel, AMD, and others; (3) highlight some examples of ongoing memory hierarchy research. First, however, an introduction to multi-cores and memory hierarchy is needed.

During the last decade, the computer industry witnessed the rise of the multi-core processor: the general-purpose processor industry began placing multiple processing cores on a single die, forming what is now commonly referred to as a multi-core processor. The industry made this switch for several reasons, the main reason being that their continued efforts to design single-core processors that efficiently exploit instruction-level parallelism (ILP) was leading to increased design and verification times, and yielding only marginal performance improvements, at the cost of ever-climbing levels of power consumption.

To address this issue, the industry decided to begin producing processors with multiple identical cores on a single die, with the intention of gaining performance increases through exploitation of thread-level parallelism (TLP) in addition to ILP.

Memory hierarchy plays an important role in the computer architecture landscape. The time it takes for a processor to retrieve a needed piece of data or an instruction from main memory is quite large relative to the cycle time of the processor. Traditionally, if there is no memory hierarchy in place (meaning the processor's request is only fulfilled by the main memory), then the processor must either stall or work on another task until its request has been serviced by the much slower main memory. By putting one or more levels of cache in between the processor and main memory, a memory hierarchy is established, and the average time it takes for a processor's read/write request to be serviced decreases dramatically. This technique has been shown to be effective time and time again for single core processor systems, and is even more effective and important in multi-core systems, where multiple cores may all be issuing read and write requests (which the memory hierarchy must be able handle).

## Memory Hierarchy Issues

There are several features of multi-core memory hierarchies that are referenced in the later sections about industry practice and ongoing research. As such, it is prudent to first give a brief introduction to some of these issues.

## Private vs. Shared

Caches are generally arranged into either a private or a shared configuration. A private cache is defined as a cache that one core has exclusive access to, whereas a shared cache is a cache that multiple cores can access. Private caches tend to be near the core they interact with, and consequently have consistent access times. Shared caches are often distributed, and access times vary depending on where within the cache the data is, and what processor core is accessing the data. As shown in Figure 1, processor P1 can have a larger access time for L1 in the shared case, even though the data is no further away than it is in the private case.

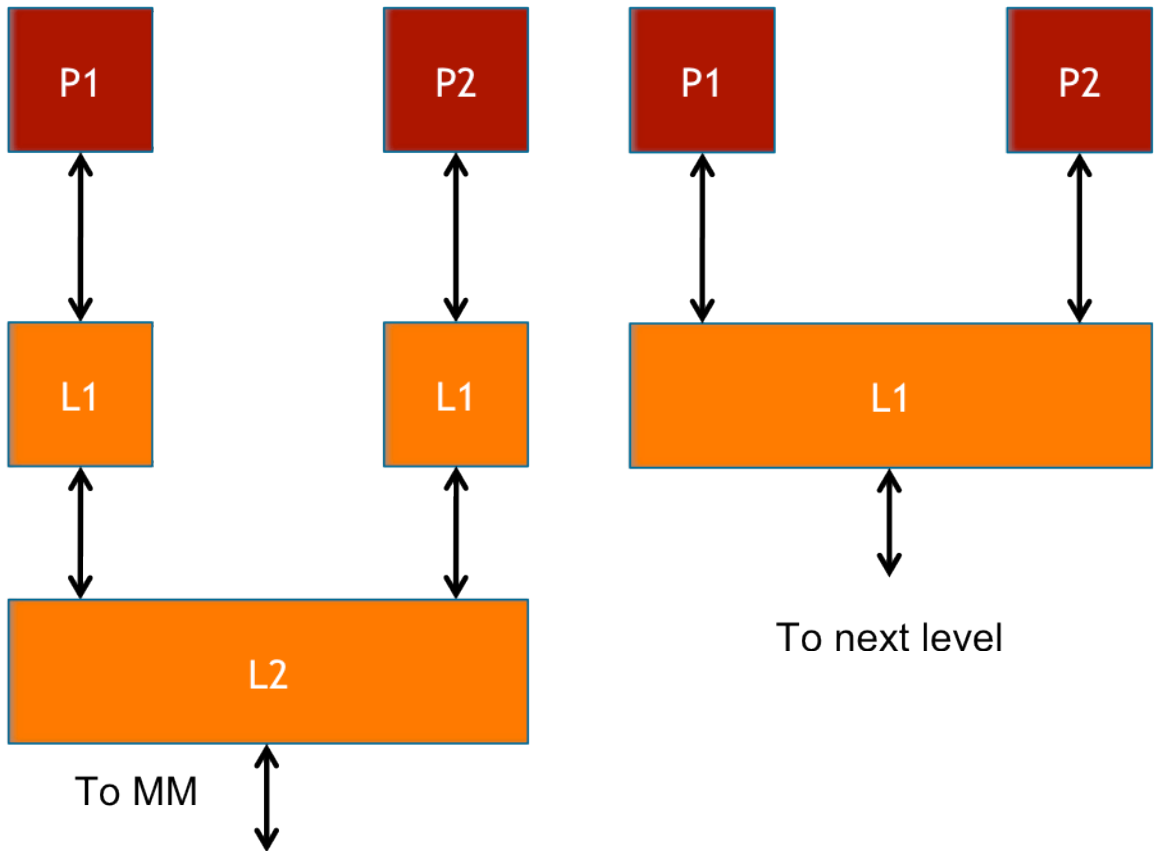


**Figure 1 - Private (left) and shared (right) cache arrangements**

Cache coherency (described in more detail later) is easily implemented in the shared scheme. On average, private caches tend to have lower hit latencies, while shared caches tend to have lower miss rates.

## Unified vs. Non-Unified

In each cache level in the memory hierarchy, caches are either unified or non-unified. A unified cache level makes no distinction between data and instructions, and therefore is able to store both in the same single cache. A non-unified cache level keeps data and instructions separate, storing instructions in one cache and storing data in another separate cache.

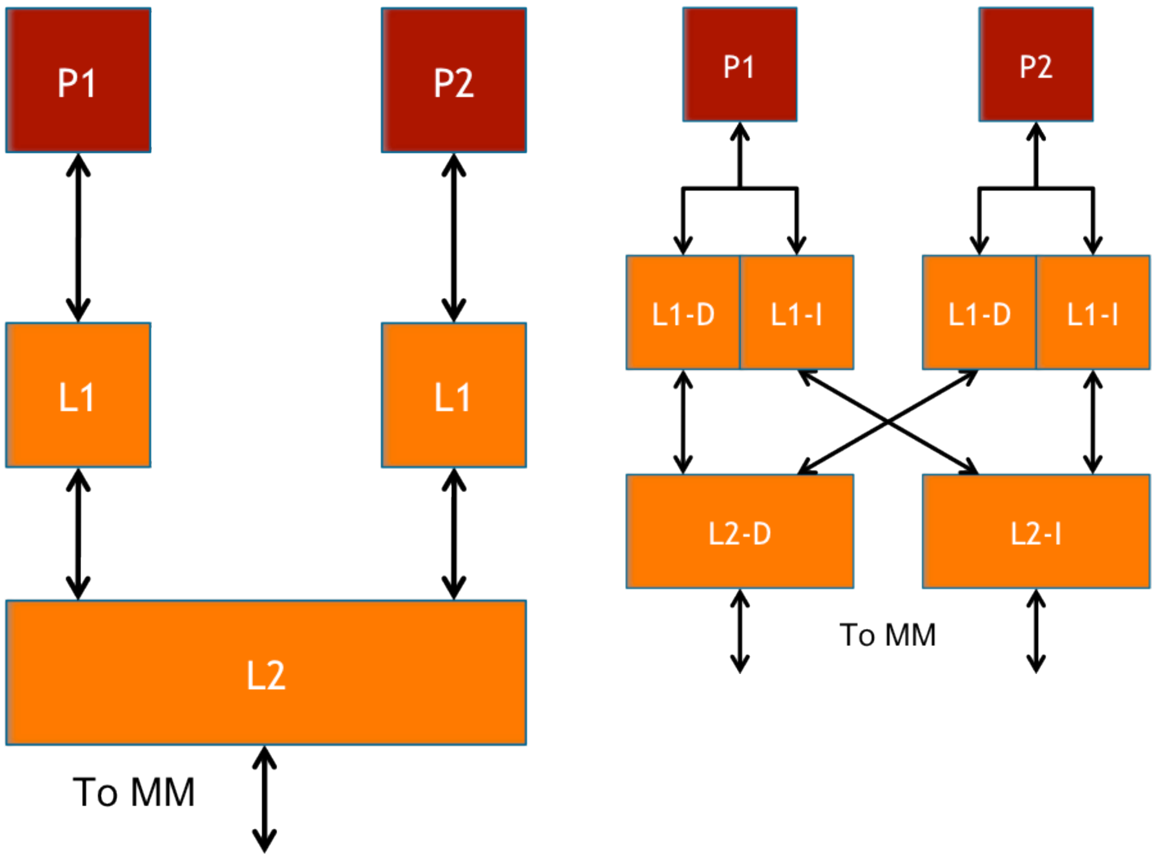


**Figure 2 - Unified (left) and non-unified (right) cache arrangements**

One reason to have a unified cache level is that it is simple to implement: no distinction is made between instructions and data, and therefore no complex logic is needed to tell one from another. One advantage to having a non-unified cache level is that instructions may be a different size (bit width) than the data, and therefore entries in the instruction cache may be one size and entries in the data cache may be another size. Having separate caches for data and instructions in this scenario saves space by not making the smaller entities take up a full slot the size of the bigger entity.

A common cache hierarchy configuration is to have the top levels (L1, and L2 if there is an L3) of the cache hierarchy be non-unified and have the last level of the cache hierarchy (L2, or L3 if it exists) be unified. This can be seen in some of the current industry designs discussed later in this report.

## Non-Uniform Cache Architectures

Non-uniform cache architectures (NUCAs) are a way to organize large last-level on-chip caches with the purpose of providing low access latencies in wire-delaydominated environments. The cache is partitioned into sub-banks and the resulting access latency is a function of the physical position of the requested data. Typically, NUCA caches connect the different sub-banks and the cache controller by means of a switched network, made up of links and routers.

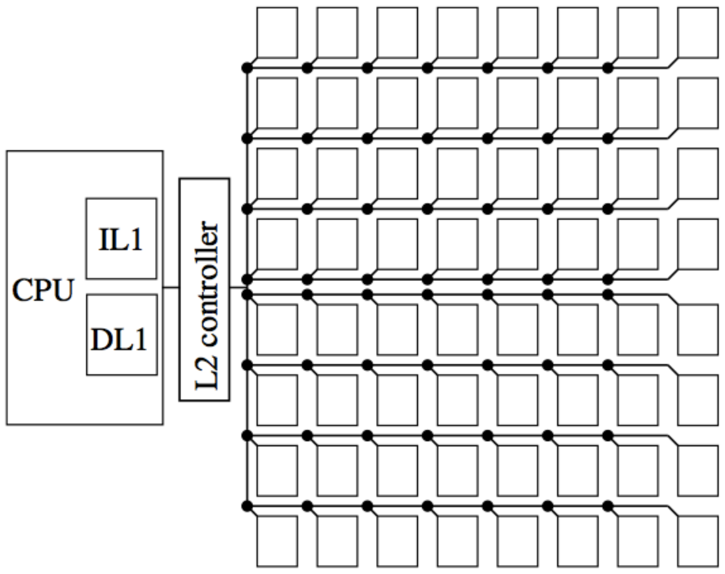


**Figure 3 - NUCA example with a 64-bank L2 cache, connected via routers**

## Bus vs. Interconnect

There are two main ways of connecting a memory hierarchy to the (potentially) many cores of a multi-core processor. The first, more traditional approach is to use a shared bus – in this approach, all of the processors access the memory hierarchy through the same set of wires, and their use of the shared bus is controlled by some arbitration logic. This traditional approach has worked well when there are only a few elements sharing the bus, but it is not very scalable – arbitrating 100+ elements on a single bus is costly and inefficient.

This problem of scalability is the inspiration for the second approach, interconnections, which provide a point-to-point connection between elements.

## Cache Coherency

Cache coherency is one of the most important issues for multi-core memory hierarchies. Cache coherency refers to maintaining the integrity of a shared resource – in this case, memory and the data/instructions contained within.

The basic idea relates to when multiple cores are sharing a piece of data. If that piece of data, which is stored in more than one level or location within the memory hierarchy, is altered in one location but not the other, then there is a potential for a discrepancy in the data – one or more cores will not have access to the newest state of the data, but they will think that they actually do. Consider the scenario depicted in Figure 4, below:

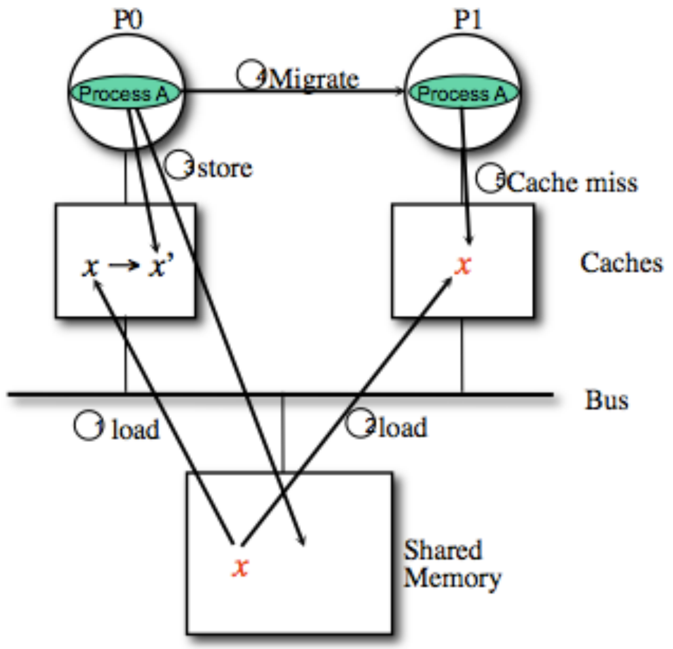


**Figure 4 - Cache coherency needed**

1. X is loaded from the shared memory into the cache of processor P0.
2. X is loaded from the shared memory into the cache of processor P1.
3. X is modified to X' by process A, which is currently running on processor P0.
4. Process A migrates from processor P0 to processor P1.
5. Process A reads what it thinks is X' from the cache of processor P1, but the value it actually reads is X. **Without cache coherency, Process A will not be protected from this problem!**

There are several ways to address the issue of cache coherency. The following sub-sections detail a few of the finer details of implementing cache coherency:

Snooping

Snooping is the process where the individual caches monitor or snoop a shared bus between different processor's caches, "listening" for address lines or accesses to memory locations that they have stored in their cached. Typically, when another processor writes to the memory location that they have cached, the processor not doing the writing will invalidate its copy to prevent the use of "bad" data.

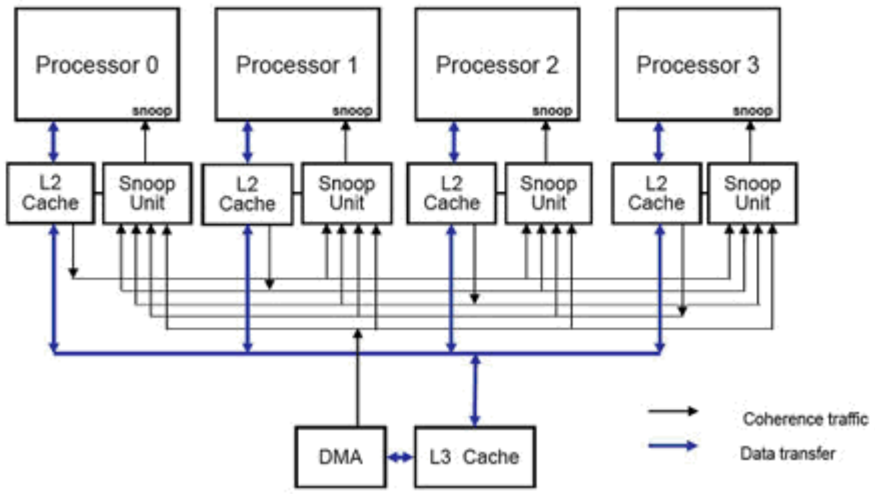


**Figure 5 - A typical snooping setup**

Snooping puts a strain on the bus and does not scale well to many cores. This is due to the bus having not only to support memory traffic but also cache coherency traffic, as shown in Figure 5 (the bus supports both regular and coherence traffic).

<u>Directory-based Protocol</u>

The directory process employs a directory as an intermediate in-between processor caches on the bus. A single location or directory keeps track of the sharing status of a block of memory and acts as a filter through which the processor must ask permission to load an entry from the primary memory to its cache as shown in Figure 6 below.

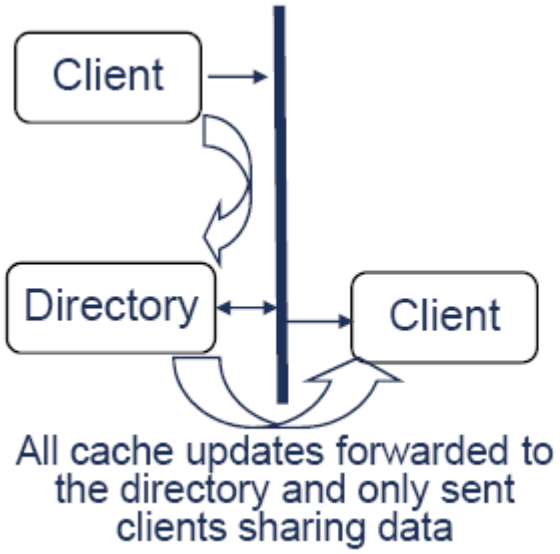

Client
Directory
Client
All cache updates forwarded to the directory and only sent clients sharing data

**Figure 6 - Directory-based cache coherency**

This lowers bus traffic, but is slower for lower number of processors. This is method is more easily scaled to a large amount of processors than snooping since only caches holding shared data are "snooped" but the addition of the directory increases the latency of transferring data.

Snarfing

Snarfing is very similar to snooping, with the difference being that the individual caches also monitor data moving along the bus. The intent is to grab the data immediately when the address on the bus matches the address in the processors cache.

## Current Industry Practices

This section gives a brief overview of some of the memory hierarchy-related features of some current (as in, currently or soon to be on the market) processors.

### Intel

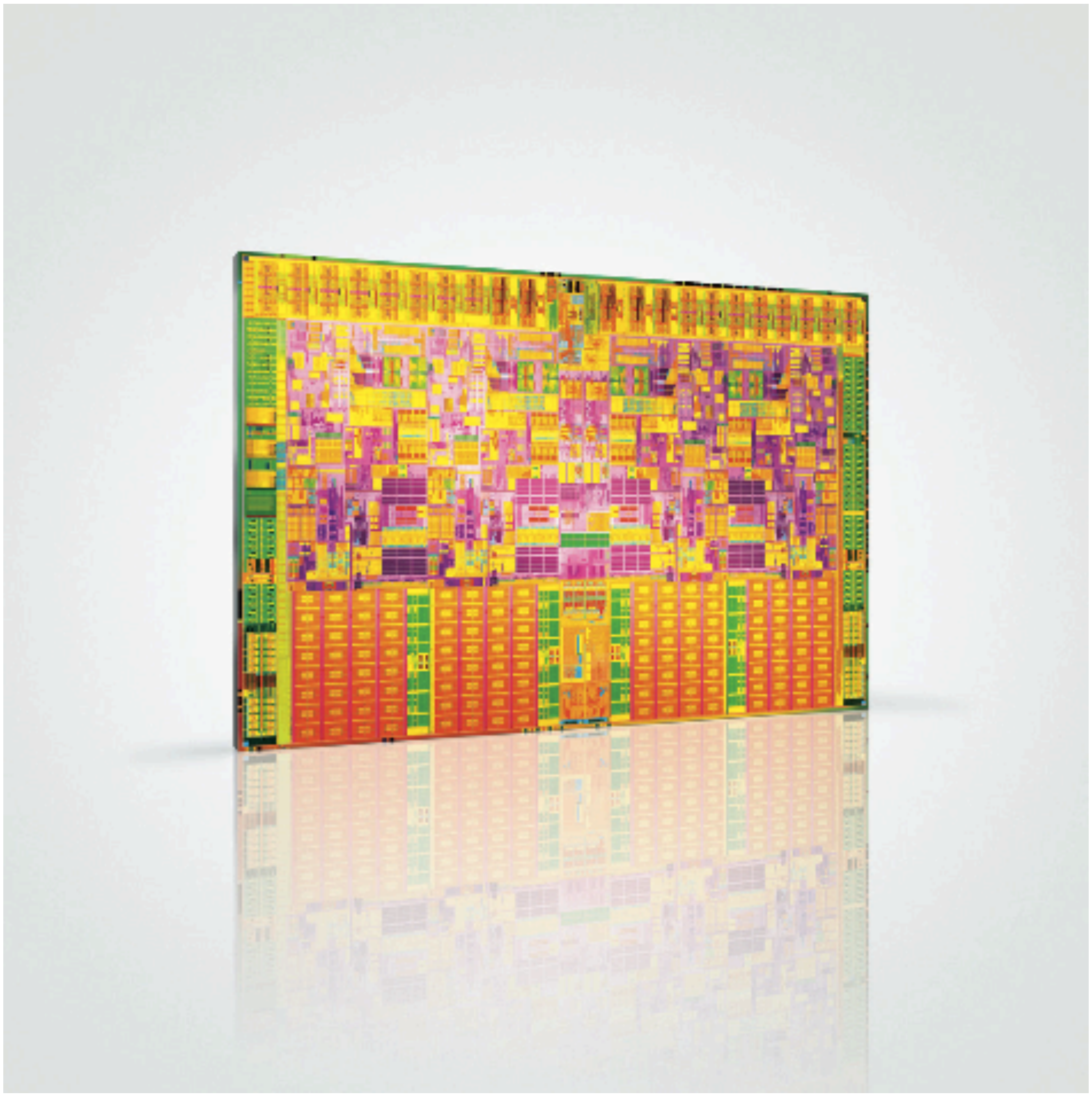


**Figure 7 - A Nehalem die shot**

## Introduction

Intel's latest microprocessor, code named "Nehalem", is a successor to the Core Duo family. The Nehalem is a multi-core processor that communicates faster and more efficiently between cores, through better memory management and cache organization.

## Cache Hierarchy

The Nehalem architecture has three levels of cache: L1, L2, and L3. Level 1 cache is non-unified with 32KB of instruction cache and 32KB of data cache. Level 2 is 256 KB and it is for both data and instructions. Level 3 is the latest enhancement to the Core Duo family: it is 256KB in size and shared between all the cores. Figure 8 shows the cache organization within the processor:

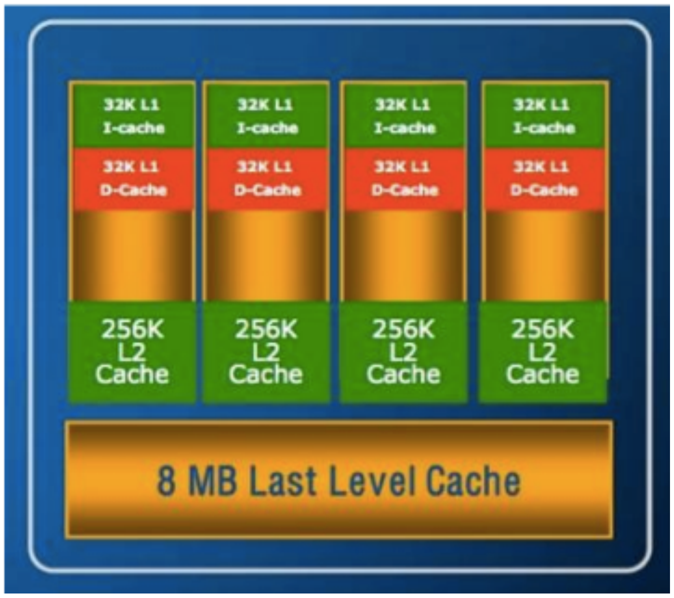


**Figure 8 - Nehalem cache configuration**

In the Nehalem architecture, each core has its own L2 cache, and even though the L2 has been reduced in size from previous generations, its lower latency has increased the overall L2 cache performance.

Nehalem does share its last level cache memory. The L3 cache is relatively large and it is inclusive, meaning it contains duplicated data from all the cores in the processor. The duplication adds to the core-to-core communication efficiency because any given core does not have to locate data in any cache core's L1 or L2 cache.  This reduces the "snooping" coherency traffic between the cores, thus reducing bandwidth and improving efficiency.

The benefit of sharing L2 memory is important because of increased performance it can provide. That is, if a core is not utilizing its memory L2 memory, other cores can take advantage of that and use the extra available L2, thus reducing hit time and increasing performance. This is illustrated below in Figure 9.

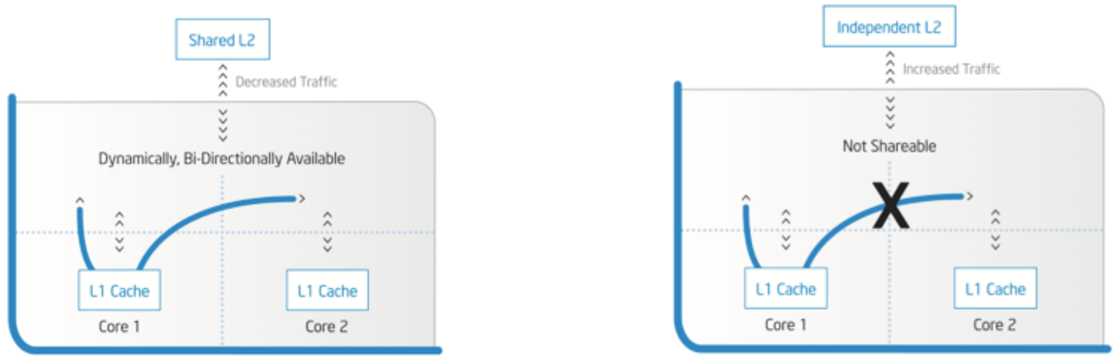


**Figure 9 - The difference between sharing (left) and not sharing (right)**

## Coherency (MESIF)

In order to make sure that making modifications to the Core Duo architecture was going to lead to improvements, Intel modified the MESI protocol to adhere and maximize the resources available. The new cache organization is known as MESIF (Modified, Exclusive, Shared, Invalid, Forward). This protocol is the same as MESI with the exception of the additional Forwarding (F) state. With this protocol only the cache line in the Forwarding state may be duplicated. Other cache lines may hold the data but cannot be copied, thus shared state's role changed to "silent" whenever there is a request with a matching cache line. Forwarding reduces traffic on the bus by not having multiple cache locations respond to a cache line request. Figure 10 shows how the state flows, and Figure 11 shows what each state means.

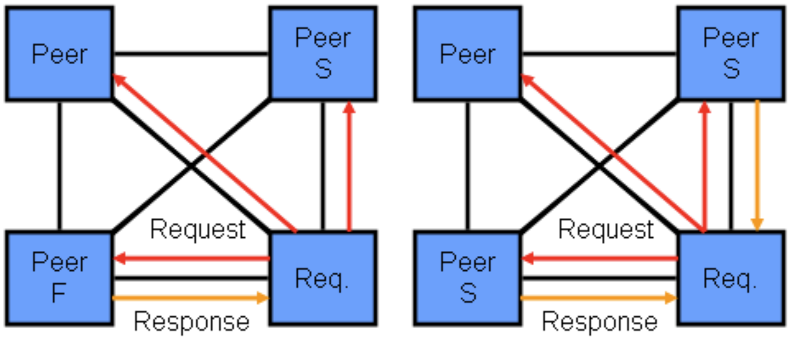
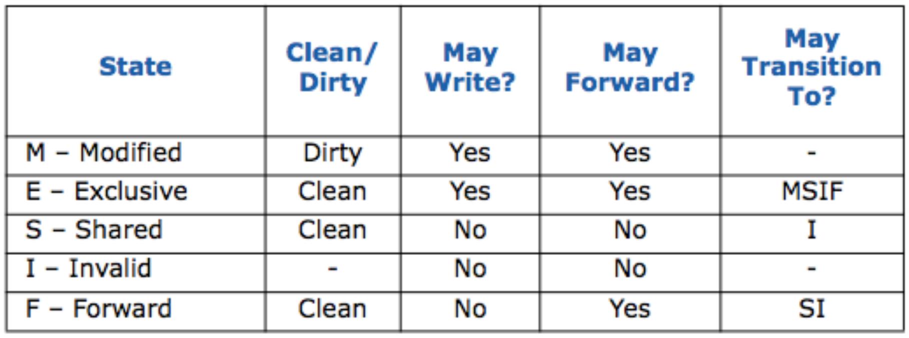


**Figure 10 - State diagram**



| State | Clean/ Dirty | May Write? | May Forward? | May Transition To? |
|---|---|---|---|---|
| M – Modified | Dirty | Yes | Yes | - |
| E – Exclusive | Clean | Yes | Yes | MSIF |
| S – Shared | Clean | No | No | I |
| I – Invalid | - | No | No | - |
| F – Forward | Clean | No | Yes | SI |

**Figure 11 - MESIF cache states**

## Intel Smart Cache

One of the key features of the Core Due family is memory disambiguation. This feature improves system performance by optimizing the out-of-order processing with a built-in intelligence that eliminates false store dependencies. Figure 12 illustrates the idea behind memory disambiguation:

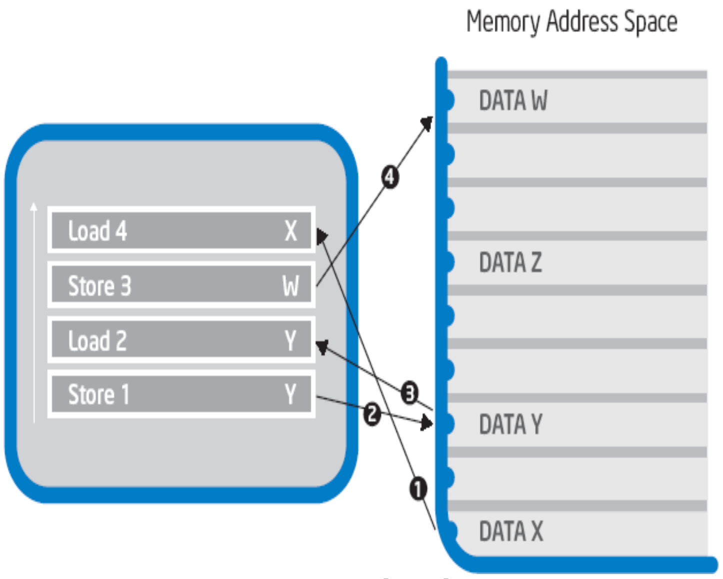


**Figure 12 - Memory disambiguation**

Figure 12 shows the execution order of a program on the left and the chronological order on the right. The 'Store 1' and 'Load 2' instruction are dependent; that is, 'Load 2' cannot execute ahead of 'Store 1' if the program is to retain is logical sense. However 'Store 3' and 'Load4' could go ahead of 'Store 1' and 'Load 2' since they are purely independent. In previous generations, this dependency was not addressed thus potentially reducing the amount of instructions that could be executed per cycle. With the Nehalem architecture, Intel addresses this 'false' dependency and improves efficiency by predicting dependency between loads and stores for out-of-order execution. It is implemented through a built-in intelligence to speculatively load data for instructions that are about to execute.

On average, 38% of x86 code involves memory references, and about half of those are stores. Thus, by successfully implementing this idea, in principle performance should increase since the out-of-order instruction execution goes up.

### Intel QuickPath Interconnect

For multiprocessors systems such as servers, there is often a need for faster communication between the processors. Before the Nehalem architecture, the Front-Side-Bus presented the bottleneck for data communication due to the difference in speed of the FSB and the processors. The QuickPath Interconnect bridges that gap by providing efficient point-to-point communication path between processors by facilitating high-speed non-uniform memory accesses (NUMA). This also increases the bandwidth communication between the processor and high-speed hardware like PCI Express. Figure 11 shows what this new capability looks like:

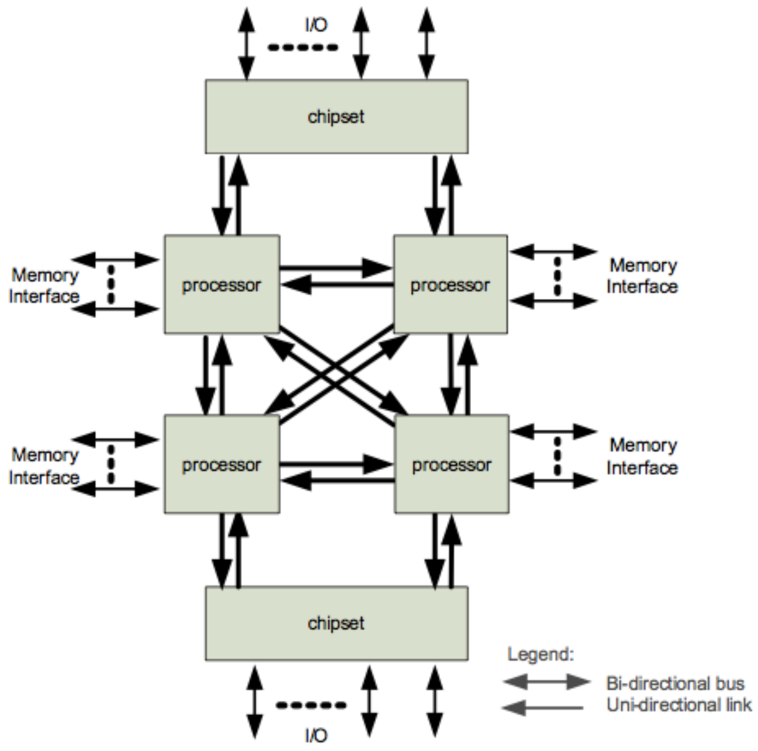


**Figure 13 - QuickPath Interconnect**

## AMD

### Cache Hierarchy

AMD's microarchitecture is similar in many ways to Intel's microarchitecture. The Opteron architecture also has three memory levels: L1, L2 and L3. Figure 14 shows the layout of the AMD Opteron, with dedicated L1 and L2 caches and shared L3 chase. The cache is non-inclusive, meaning that the cache line will either reside in L1, L2 or L3.

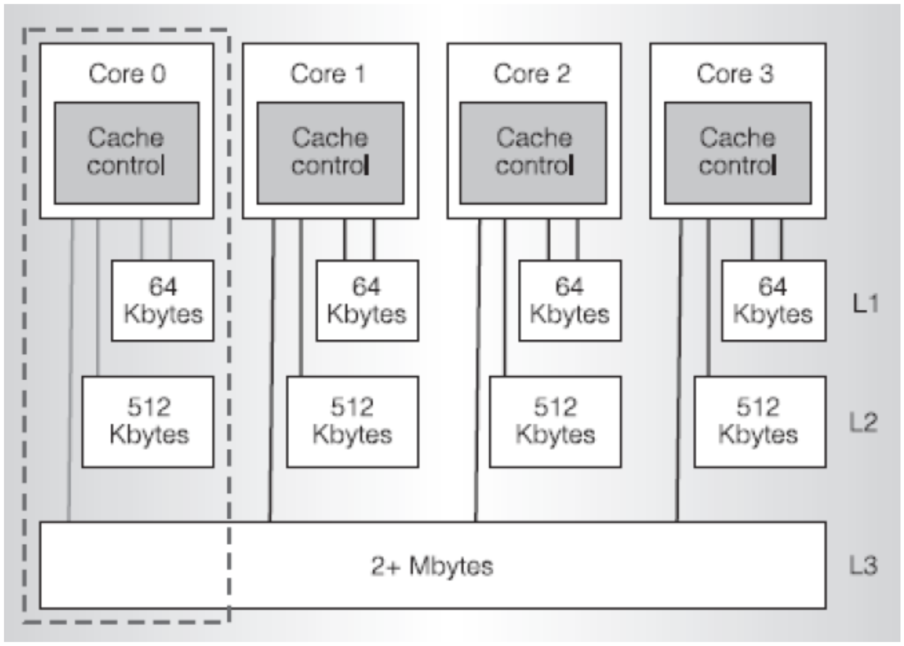


**Figure 14: AMD Opteron Memory Hierarchy**

## Coherency (MOESI)

AMD also uses a more elaborate version of the MESI cache coherence protocol: the MOESI protocol. With the additional state of "owner", the MOESI protocol avoids the need to write back modifications to the main memory when another core reads it. Instead, the owner state allows data to be supplied directly from one processor to another. Figure 15 shows the state machine for the MOESI protocol.

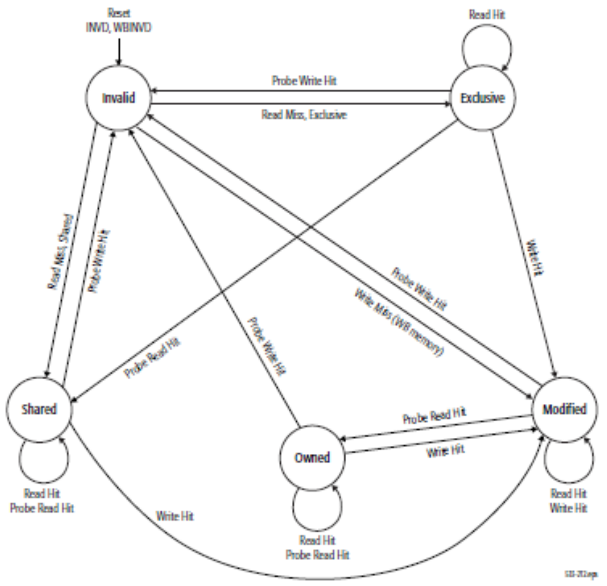


**Figure 15: The AMD MOESI protocol**

With the owner state, the same cache line can be stale in all other processors as well as the main memory. This is beneficial when the bandwidth between processors is significantly higher than the bandwidth between processors and main memory, as is the case in most modern microprocessors.

## HyperTransport

The latest AMD microarchitecture integrates the main memory controller on-die, thus eliminating the bottleneck of the North Bridge. Figure 16 illustrates the new AMD Barcelona micro-architecture.

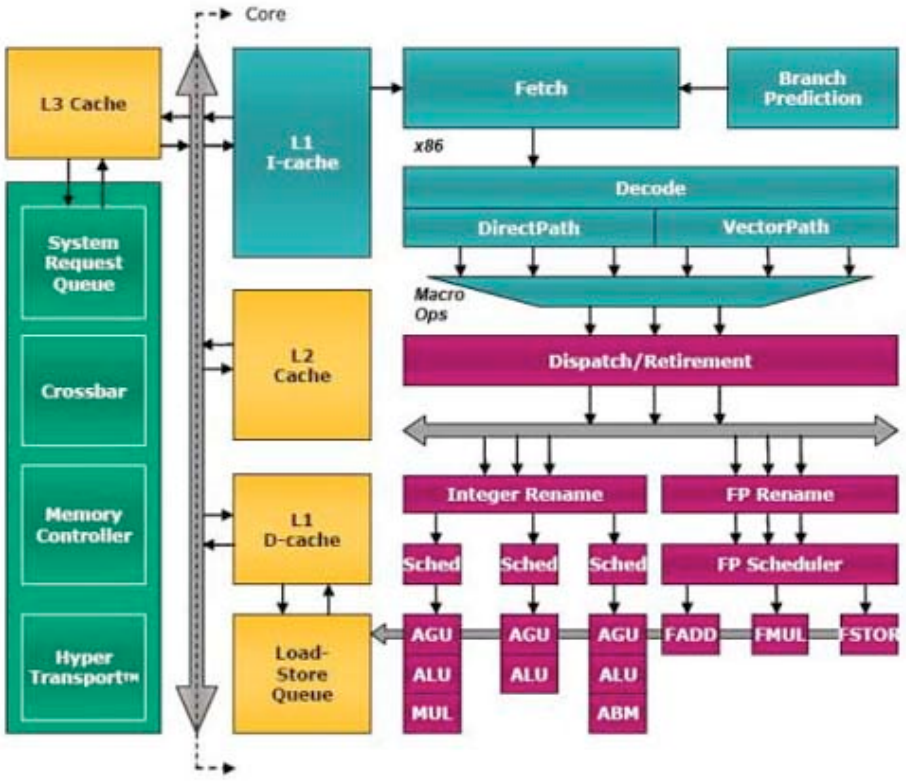


**Figure 16: AMD's Barcelona Microarchitecture**

The Barcelona microarchitecture is implemented on the latest Phenom and Phenom II processors and it has the same dedicated L1 and L2 cache with shared L3. However, a new HyperTransport protocol is implemented to enhance communications between the processor and other devices. HyperTransport enhances noise immunity and transmission congestion by using a similar electrical protocol as 1.2V LVDS (Low Voltage Differential Signal) and implementing a high speed, packet based protocol.

## Sun

### Introduction

The Sun Ultrasparc t2(plus) is an example of a multi-core processor designed for network infrastructure servers and commercial workloads that exhibit a large amount of thread level parallelism. Figure 17 shows the physical layout of the processor with the cores and L2 cache dominating the chip area.

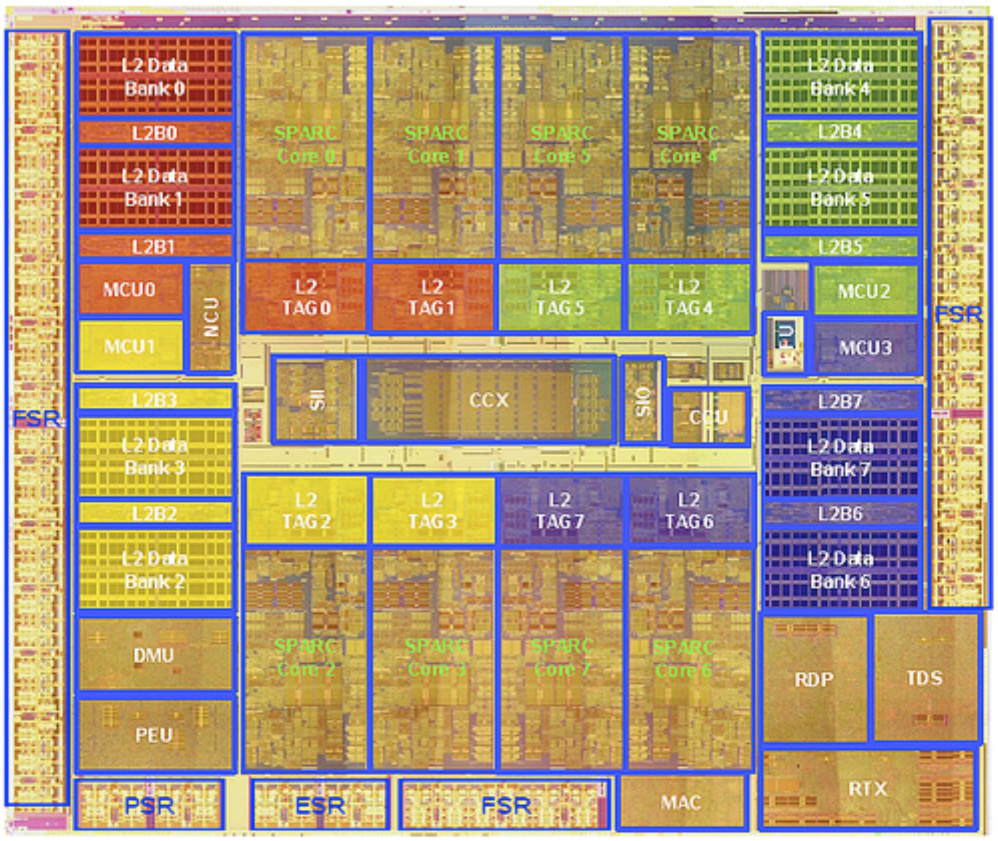


**Figure 17 – Sun UltraSparc t2**

### Cache Hierarchy

As seen in Figure 18, the processor features 8 cores, running at 1.4Ghz, each supporting 8 threads. The memory hierarchy is organized with L1 cache, L2, cache and four memory controllers each connecting to 2 FBDIMMs. The L1 cache is private and non-unified which is split into a 8KB = 4-way set associative data cache. The L1 instruction cache is 16KB and 8-way set associative. These cache sizes are very small when compared to desktop processors, implying that access time is of primary concern to AMD.

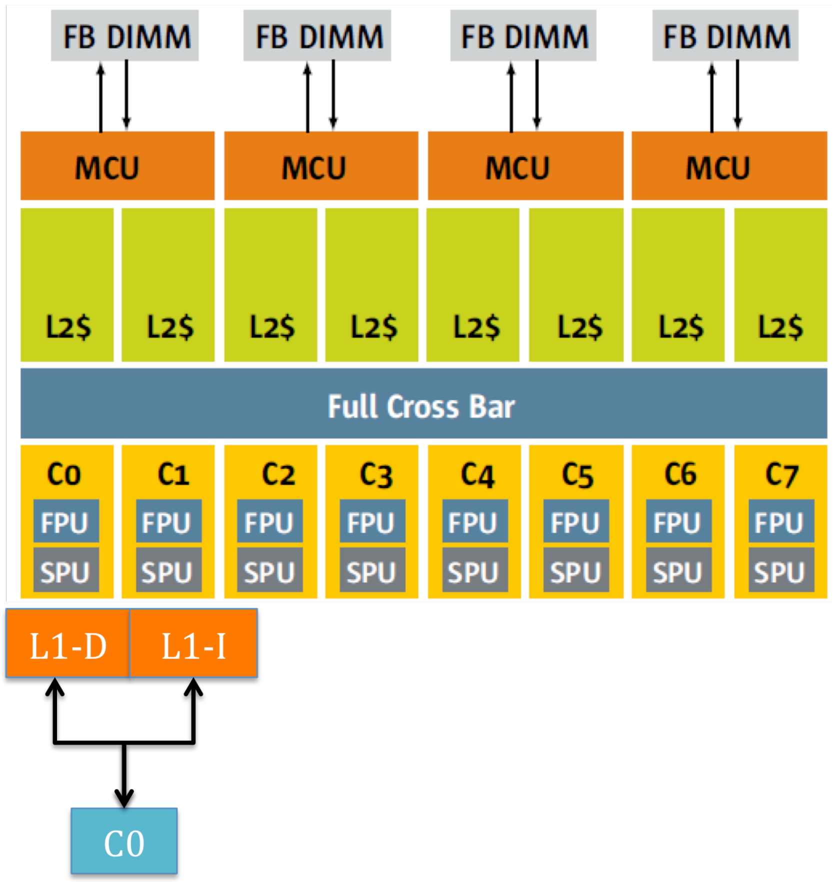


**Figure 18 - Sun processor organization**

The L2 cache is 4MB and 16-way set associative. The unique aspect of the L1/L2 arrangement is the full 8x9 crossbar interconnect, which connects every core to every L2 bank, and vice-versa. The L2 is divided into 8 banks with each L1 cache connected to each bank of L2 cache. The additional connection is for I/O devices.

## ARM

### Introduction

ARM Cortex-A9 MPCore is ARM's newest multi-core processor. It is an embedded processor and consequently its designed to be flexible, comes in different configurations depending on need. Mobile applications require very lower power (~250mW) – this is extremely low when compared to typical desktop and server power levels.

### Cache Hierarchy

The diagram of the processor is shown in Figure 19:

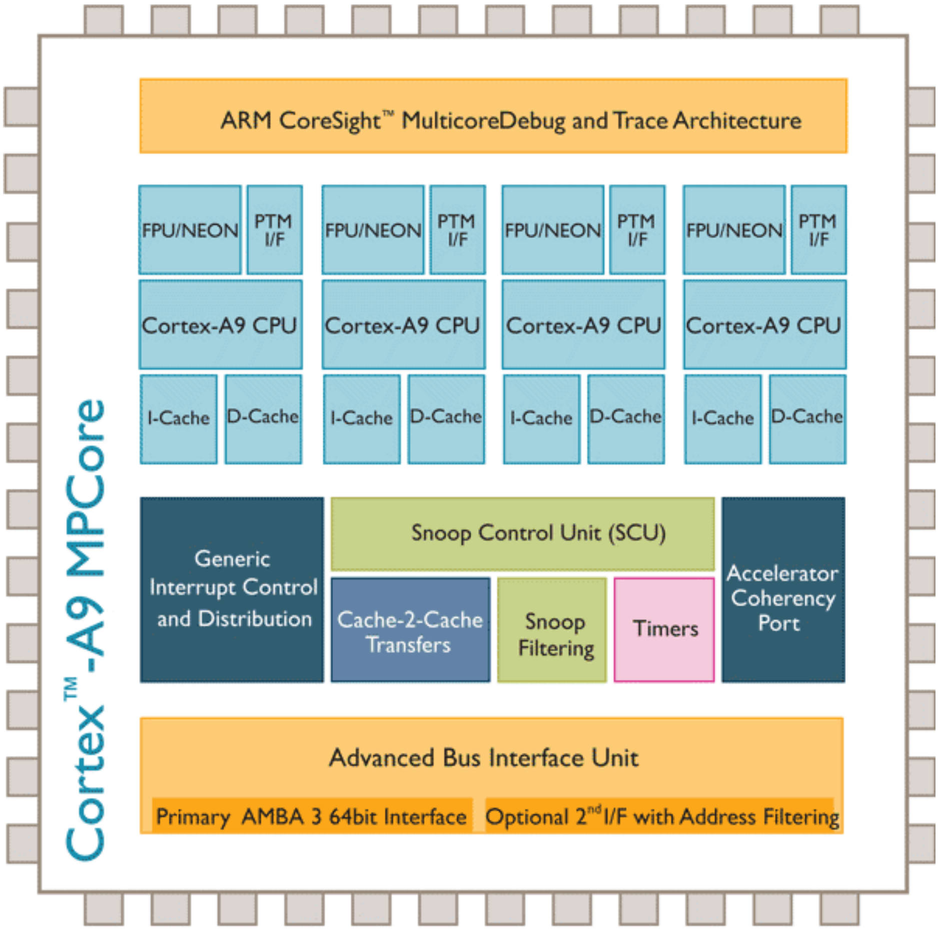

ARM CoreSight™ MulticoreDebug and Trace Architecture
FPU/NEON
PTM I/F
Cortex-A9 CPU
I-Cache
D-Cache
Generic Interrupt Control and Distribution
Snoop Control Unit (SCU)
Cache-2-Cache Transfers
Snoop Filtering
Timers
Accelerator Coherency Port
Advanced Bus Interface Unit
Primary AMBA 3 64bit Interface
Optional 2nd I/F with Address Filtering
Cortex™-A9 MPCore

**Figure 19 - ARM processor organization**

The processor features 4 cores running at 2 speeds from 800MHz – 2GHz these are the low power version and performance version, respectively. They consume 0.5W to 1.5W depending on the operating speed. The L1 cache is private and non unified and can be implemented 16, 32 or 64KB sizes all which are four way associative. A snoop control unit implements an enhanced MESI Protocol for the cache coherency between cores. L2 is not shown and is optional. It is external to the chip, and can be provided by an optional Level 2 Cache controller that supports 128KB to 8MB.

## Current Research

### Bank-Aware Dynamic Cache Partitioning & Integrated Partitioning

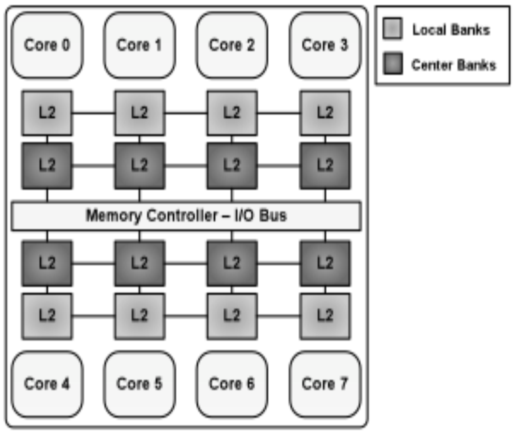
#### Purpose

Chip Multiprocessors (CMPs) have continuously reduced their die size as technology continues to decrease transistor size. As more space becomes available and more components are integrated onto a single chip, sharing and managing the computation resources becomes crucial for the overall efficiency of the CMPs. Potential bottlenecks can exist, possibly due to main memory bandwidth, main memory capacity, cache capacity, cache bandwidth, memory subsystem interconnection bandwidth, and system power. In addition to bottlenecks imposed by shared resources, wire delay has increasingly become a problem.

## Solution #1: Bank-Aware Dynamic Cache Partitioning for Multi-core architectures

As new CMP designs include more cores and cache capacity, a banked last level cache memory design is a promising solution to alleviate the unceasingly number of cores on a processor and mitigate wire delays.  This research explores an alternative for managing the last-level cache in a CMP.

The baseline used for this work included a 16MB, 16-way banked set associative L2 cache memory.  The system includes an 8-core processor running at 4 GHz.  Below, in Figure 20, the full details and a partitioned L2 cache memory.



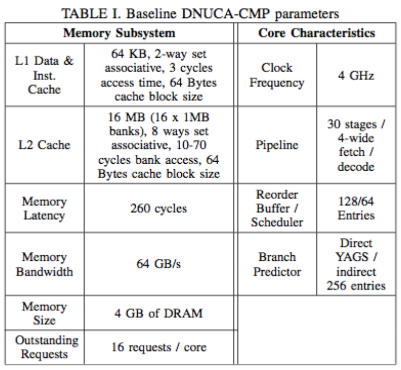


TABLE I. Baseline DNUCA-CMP parameters

| Memory Subsystem | | Core Characteristics | |
|---|---|---|---|
| L1 Data & Inst. Cache | 64 KB, 2-way set associative, 3 cycles access time, 64 Bytes cache block size | Clock Frequency | 4 GHz |
| L2 Cache | 16 MB (16 x 1MB banks), 8 ways set associative, 10-70 cycles bank access, 64 Bytes cache block size | Pipeline | 30 stages / 4-wide fetch / decode |
| Memory Latency | 260 cycles | Reorder Buffer / Scheduler | 128/64 Entries |
| Memory Bandwidth | 64 GB/s | Branch Predictor | Direct YAGS / indirect 256 entries |
| Memory Size | 4 GB of DRAM | | |
| Outstanding Requests | 16 requests / core | | |

**Figure 20 - Last-level partitioning (left) and baseline specs (right)**

The partitioning of the last level cache memory is implemented using Mattson's Stack Algorithm (MSA).  MSA is an algorithm designed to reduce simulation time of trace-driven caches for each core by determining the miss ratios of all possible cache sizes with a single pass through the trace.

This work uses the SPEC CPU2000 benchmark consisting of 26 workloads with various dynamic behaviors, which forces the utilization of various amount of L2 cache.  Choosing three random workloads out of the twenty-six available, the research team found out the following:

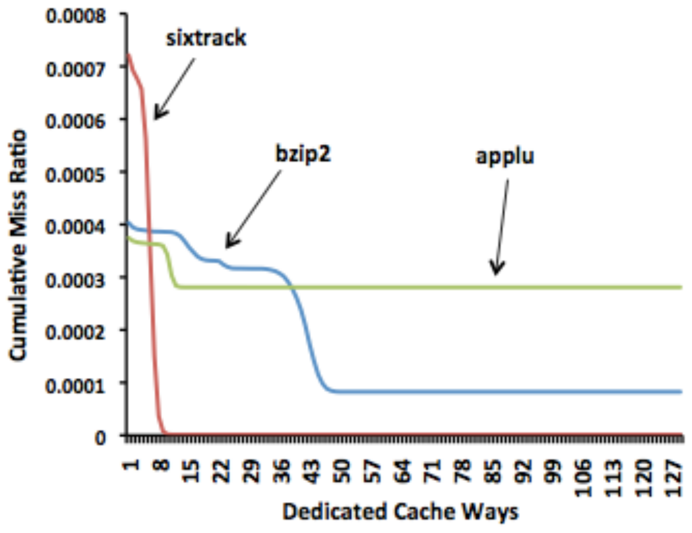


**Figure 21 - Various miss ratios**

## Results #1

The figure above shows the miss ratio of three workloads and their performance as a function of dedicated cache ways. Based from the results shown from the graph, MSA-based profiling allows monitoring of cache capacity of each core dynamically of an application.

From the results obtained it is clear that miss ratio reduces as the number of dedicated cache ways increases. This proves that MSA-based profiling and dynamic-bank aware partitioning can potentially reduce miss ratio and increase performance for a given workload. Figure 22 shows a possible portioning scheme allocated dynamically:

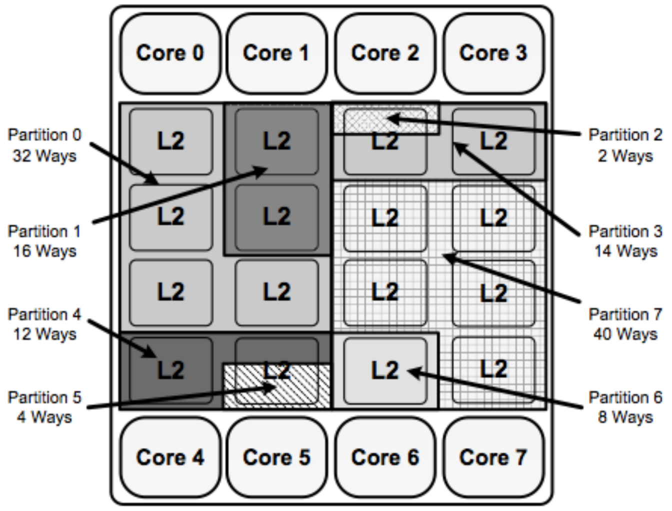


**Figure 22 - Typical CMP cache partitioning example**

The research team conducted a performance comparison with ideal partitioning scheme for a given core and given workload. This comparison (Unrestricted partitioning) is based on the idea of having an ideal case where a core could have an ideal dynamic partitioned cache size and see how that would compare to MSA bank aware partitioning. Figure 23 shows a graphical comparison:

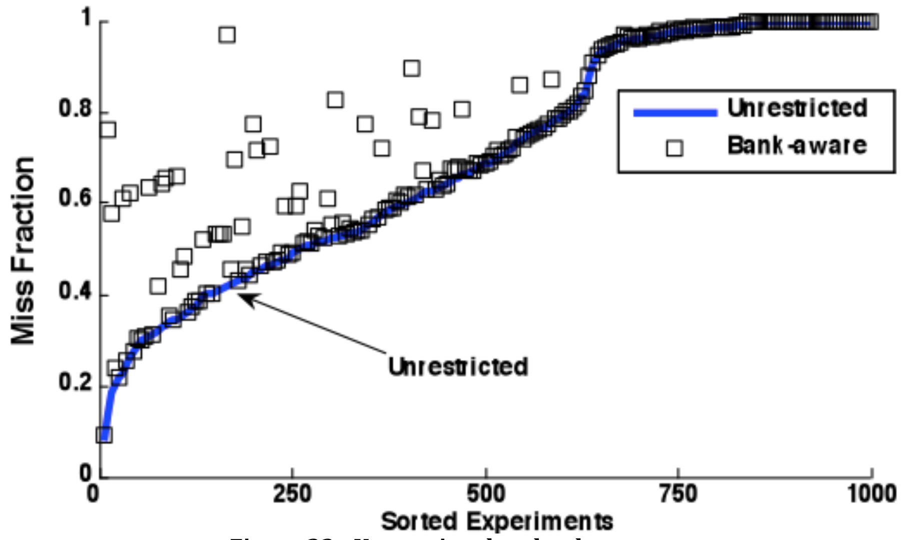


Figure 23 - Unrestricted vs. bank-aware

The results show that over all, the MSA bank-aware partitioning underperforms the unrestricted partitioning by about 5%. Ideally the isolated dots from the graph from bank-aware should fall on the Unrestricted line which is in general true except some outliners that achieved small miss rates reductions than Unrestricted.

The overall all performance of MSA-Bank Aware Partitioning is illustrated below. Comparison of even partitioning and no partitioning at all is plotted for easy comparison in miss ratio and CPI performance. Overall, bank-aware partitioning shows a 25% miss rate reduction compared to even partitioning. Additionally, bank-aware shows a 70% and 43% reduction in misses and CPI over No-partitions respectively.

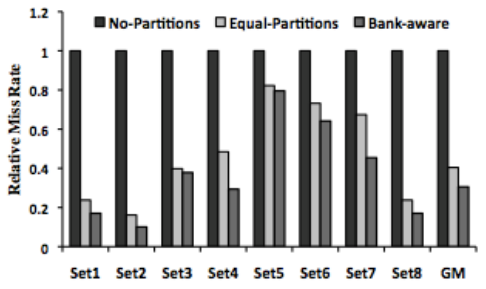


Fig. 8. Relative miss rate of 8-core sets over the no-partitioning scheme

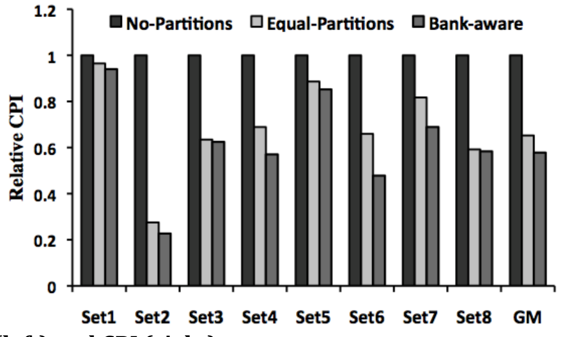


Figure 24 - Miss rate (left) and CPI (right)

## Solution #2: Integrated Partitioning

Another ongoing research topic in the area of cache partitioning is integrated partitioning. There are two types of partitioning that typically occur in today's modern multi-core processor. First, there is the partitioning of processes that are currently running – this is also known as scheduling, and is performed by the operating system. This type of partitioning determines which processes are running on which cores at which time. Second, there is the partitioning of the cache, specifically into segments that correspond to portions of the shared, last level cache. This type of partitioning is performed by the microarchitecture.

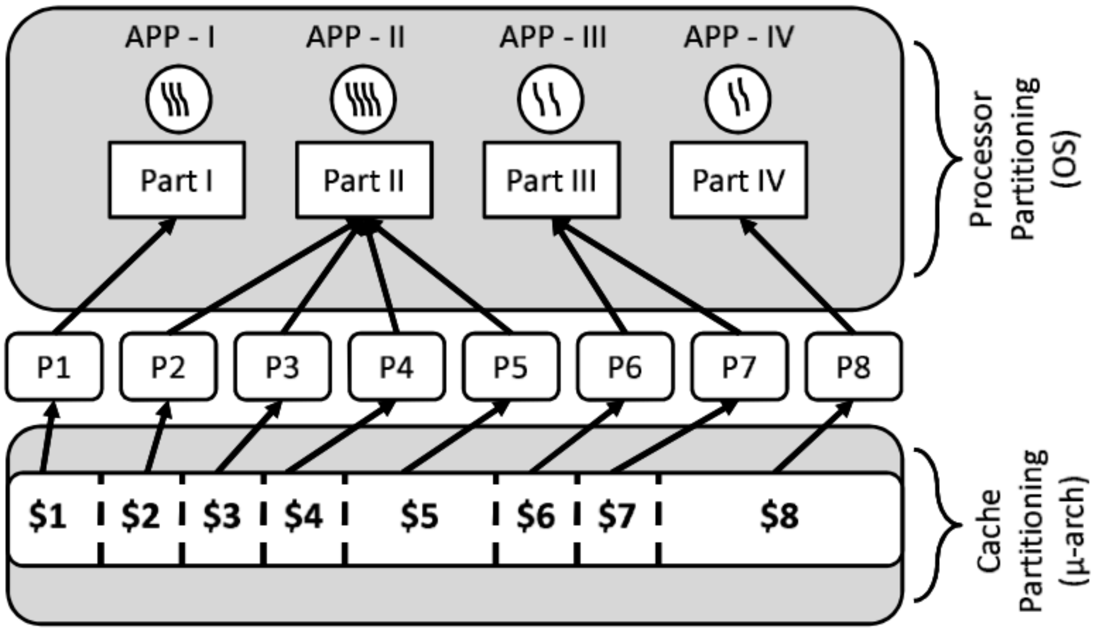


**Figure 25 - Traditional non-integrated partitioning**

The idea behind integrated partitioning is that these two types of partitioning be performed while aware of one another: i.e., when the microarchitecture allocates a specific amount of the shared cache to a particular core, it should know what type of process the operating system has allocated to that core (by what type of process we generally mean how much thread-level parallelism is shown by the process).

## Results #2

Integrated partitioning is expected to increase overall system throughput, and it seems to do just that. According to a fair speedup metric (FS), integrated partitioning performed on average 14.14% better than traditional implicit partitioning.

## Networks-On-a-Chip (NOC)

### Purpose

There are two paradigms for inter-core communication: broadcast based, like processors today, and routed, like having a network on a chip.

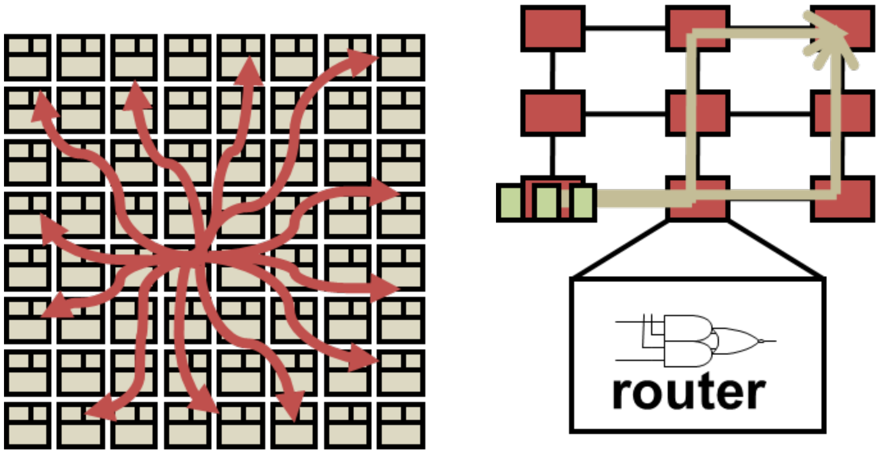


**Figure 26: Broadcast-based communication vs. NOC**

As the number of cores ramps up from four to eight and beyond, having a network on a chip (NOC) topology greatly reduces interconnect complexity and transmission delay by having a router efficiently route data from one processor to another.

However, a problem arises when one core needs to share data amongst a large amount of other cores. Where as in the previous broadcast based shared bus topology, all a core has to do is to snoop the data on the bus, the NOC topology does not allow such operations to happen and as a result, the processor with the data must do multiple unicasts to every single chip, thus having a significant delay and overhead in data transmission.

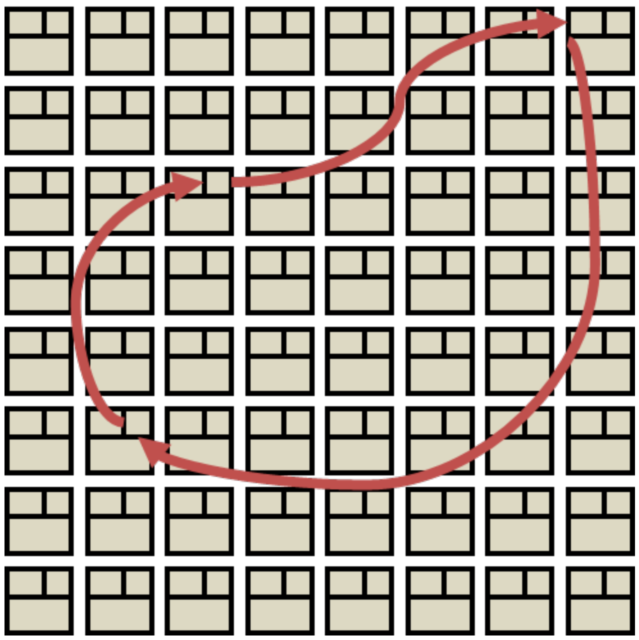


**Figure 27: One core needs to send data to a significant amount of cores**

Researchers at the University of Toronto has come up with an elegant solution called Virtual Tree Multicasting by reusing one of today's most powerful and infamous network protocols: peer-to-peer networking.

## Solution – Virtual Tree Multicast

The solution to the need to broadcast data for NOCs can be found in the peer-to-peer networking protocol. Instead of doing multiple unicasts, as shown in Figure 28, where processor A has to cast data to B, C, D, E and F individually, virtual tree multicasting can be implemented, as shown in Figure 29.

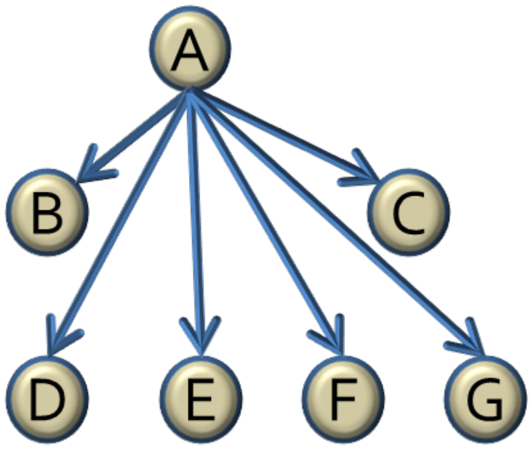


**Figure 28: Multiple Unicasts from Core A to Other Cores**

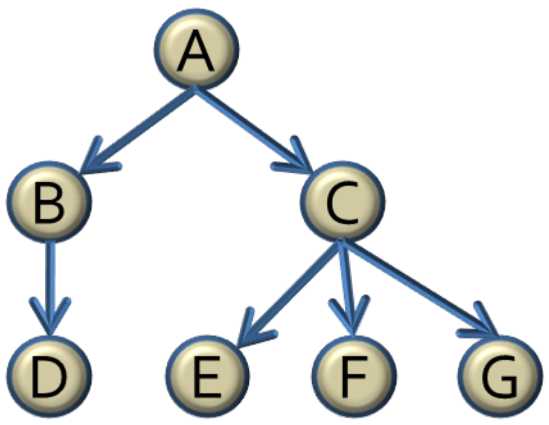


**Figure 29: Virtual Tree Multicasting**

Virtual Tree Multicasting follows a similar idea as peer-to-peer networks. In this example above, processor A is the "seed" and sends data to processors B and C, both of which are spatially close to processor A in this example. Processors B and C then casts to D, E, F and G, which is then spatially close to them. Spatial locality is important since it is one of the most determining factors in latency. By doing a smaller amount of multicasts instead of a large number of unicasts, Virtual Tree Multicasting allows the architectures of tomorrow to communicate more efficiently and effectively.

### Results

University of Toronto has found that by using Virtual Tree Multicasting instead of multiple unicasts, the new architecture can reduce interconnect latency by up to 90 percent and switching activity in the router by up to 53 percent. Power reductions were also found, but values were not given at this time.

### Observations

Virtual Tree Multicasting seems to be starting a new trend in network on a chip design. With multi-cores getting more and more prevalent, it is more important than ever to have fast, efficient ways to keep cache lines coherent and have efficient communication between processors.

## ESP-NUCA

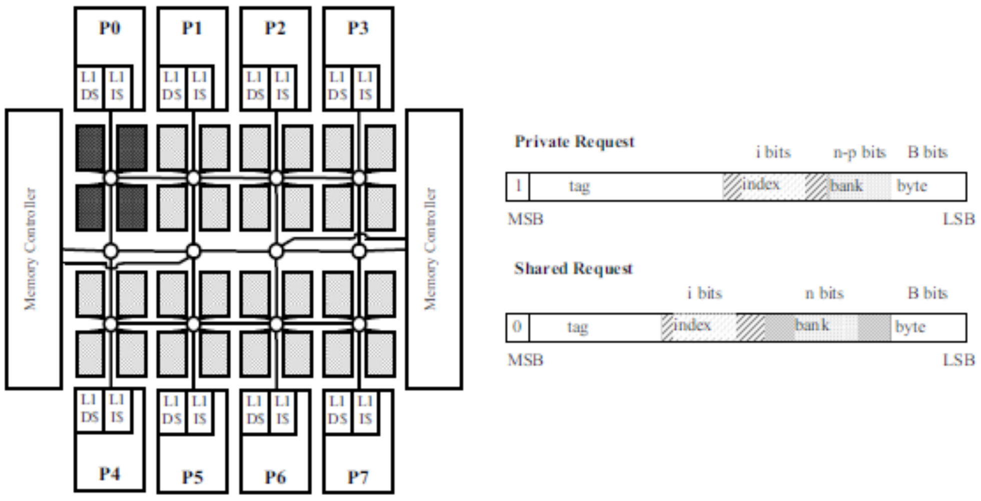
### Purpose

The widening processor-memory performance gap and limited on-chip bandwidth creates a need for improving the performance of on-chip memory hierarchy design and management.  Neither shared nor private configurations provide optimal performance and hybrid schemes attempt to address that issue.

## Solution: Enhanced Shared-Private Non-Uniform Cache Architecture

Enhanced Shared-Private Non-Uniform Cache Architecture begins with a shared NUCA and introduces a low-cost mechanism to dynamically allocate private cache blocks close to their owner processor.

In the S-NUCA (shared nuca) scheme each block is placed in specific bank depending only on address. SP-NUNA introduces a private/shared bit in the MSB position in the cache address as shown in Figure 30 (b). SP-NUCA introduces private/shared blocks and the ability of data to migrate within the cache. When processor P0 requests blocks they are placed near P0 and the accompanying private bit is set in the address. If another processor attempts to access the same block, the bit is reset to “shared”.



(a) Banks (b) Address interpretation

**Figure 30 - NUCA**

A L1 cache miss in S-NUCA requires that L2 be accesses to get the data, as in Figure 31 (a). If L2 misses then processor 2 or MM can respond. The latency for L2 is variable depending on whether or not the data is near the core requesting the data. SP-NUCA adds a private/shared distinction modifying the S-NUCA sequence. An L1 cache miss in SP-NUCA first checks the private L2 cache which reduces latency on L2 hit. If private L2 misses then it remains within the L2 structure and accesses L2 shared. If L2 shared misses another processor, other private L2 caches, or MM can respond.

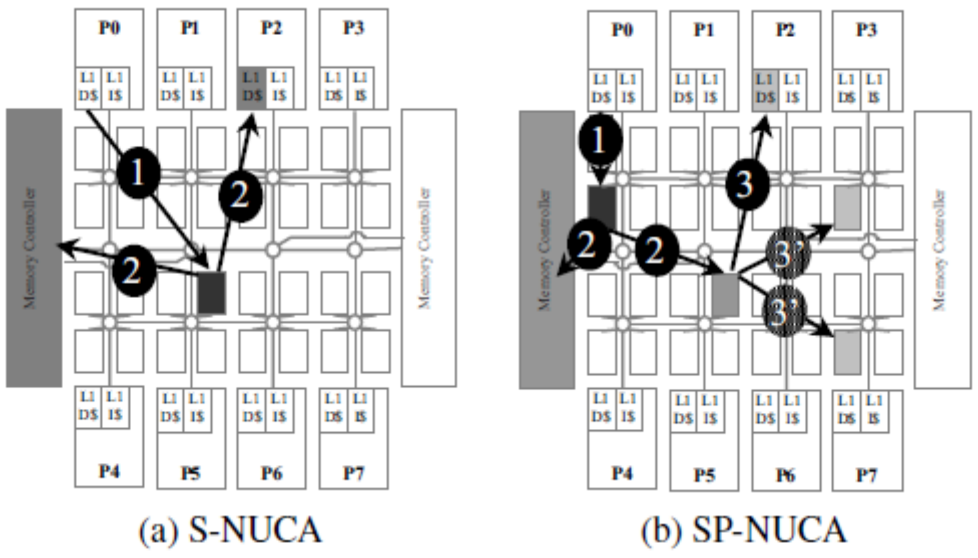


**Figure 31 - S-NUCA (left) and SP-NUCA (right)**

There are still limitations with SP-NUCA such as block replication as seen in private caches and still non-uniform latency as seen in shared caches.

ESP-NUCA attempts to address latency limitations with accessing a remote bank to get a shared block. Two new types of blocks in addition to private/share blocks are introduced called replica and victim blocks. Replica blocks are copies of shared blocks stored in the local partition of the cache and victim blocks are remote private data stored in the shared cache partition. These two block types are grouped as helping blocks, while the former are first-class blocks. Helping blocks should only be kept if the private and shared hit rate does not suffer.

Each set can only contain a limited number of helping blocks. When this threshold is reached, helping blocks are evicted according to LRU policy. When under this limit, the LRU of the whole block is implemented.

## Results

The system configuration used for test is in Figure 32 uses 8 processors, 8MB NUCA 8X4 banks of L2 cache. ESP-NUCA was compared against a private, shared, D-NUCA, ASR, and CC-Avg. The results are summarized in Figure 32.

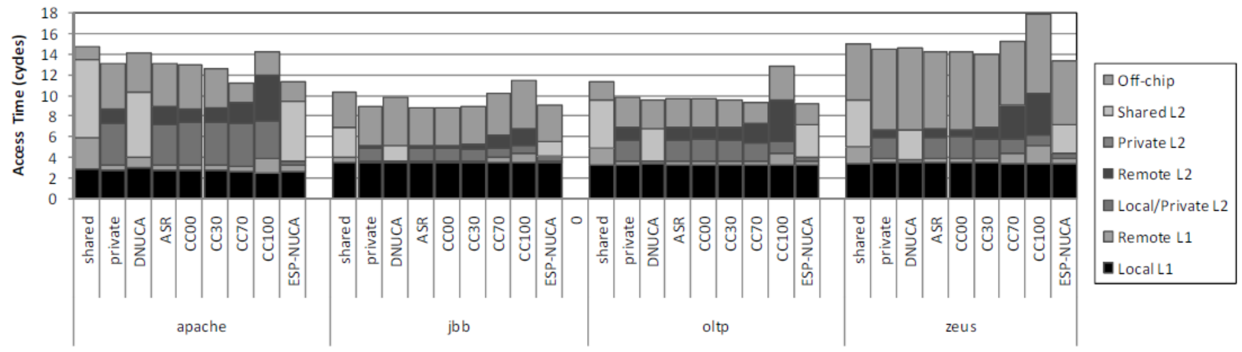


**Figure 32 - Results**

# Conclusion

## Trends & Observations

There are three key observations to be made about the state of memory hierarchy today:

- Memory hierarchy design for multi-core processor systems has many unique considerations that do not apply in single-core situations, not the least of which is cache coherency. Most considerations involve keeping data consistent between cores, and sharing data between cores with maximal efficiency.
- Intel, AMD, and the other companies that design multi-core processors have successfully addressed many of these design considerations in elegant and unique ways.
- There is still a lot of research to be done in the area of memory hierarchy. Specifically, there are two areas of research that show particular promise: cache partitioning, where shared caches are dynamically partitioned into unequal ratios depending on the workload of the processor's cores (or other factors), and networks on a chip, where processors (and their caches) are connected (and therefore share information) via an on-chip network.

It is undeniably clear that the number of cores placed on a single chip will continue to increase indefinitely, and as a result, memory hierarchies for multi-core processors will continue to become more and more important, in the next decade and beyond.

## References

Bailey, Anderson. “Barcelona's Innovative Architecture Is Driven by a New Shared Cache.”
http://developer.amd.com/documentation/articles/Pages/8142007173.aspx#three

Jerger, Natalie D. Enright. “Chip Multiprocessor Coherence and Interconnect System Design.”
http://www.eecg.toronto.edu/~enright/thesis.pdf

Jin, Lei, Hyunjin Lee, and Sangyeun Cho. “A Flexible Data to L2 Cache Mapping Approach for Future Multicore Processors.”
http://www.cs.pitt.edu/cast/papers/jin-mspc06.pdf

Jing, Wen. “Multi Processors, their Memory organizations and Implementations by Intel & AMD.”
http://ece.uic.edu/~wenjing/courses/fa08ECE569/ECE569/w21.pdf

Kaseridis, Dimitris, Jeffrey Stuecheli, and Lizy K. John. “Bank-Aware Dynamic Cache Partitioning for Multicore Architechtures.”
http://lca.ece.utexas.edu/pubs/Kaseridis_ICPP2009.pdf

Merino, Javier, Valentin Puente, and Jose A. Gregorio. “ESP-NUCA: A Low-Cost Adaptive Non-Uniform Cache Architecture.”
www.atc.unican.es/~jmerino/papers/esp-nuca_hpca_10.pdf

Salapura, Valentina.“Taming the cost of coherency for multicore systems.”
http://domino.research.ibm.com/comm/research.nsf/pages/r.arch.coherency.multicore.systems.html

Savage, John E., and Mohammad Zubair. “A Unified Model for Multicore Architectures.”
www.cs.brown.edu/~jes/papers/IFMT08.pdf

Shah, Manish, et al. “UltraSPARC T2: A Highly-Threaded, Power-Efficient, SPARC SOC.”
http://www.opensparc.net/pdf/publications/technical-papers/ultrasparc-t2-a-highly-threaded-power-efficient-sparc-soc.pdf

Srikantaiah, Shekhar, et al. “A Case for Integrated Processor-Cache Partitioning in Chip Multiprocessors.”
http://www.cse.psu.edu/~srikanta/papers/sc2009/integrated/sc09-srikantaiah.pdf

Wechsler, Ofri. "Inside Intel® Core Microarchitecture."
http://download.intel.com/technology/architecture/new_architecture_06.pdf

"AMBA® Level 2 Cache Controller (L2C-310) Technical Reference Manual."
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0246d/index.html

"Cortex-A9 Processor."
http://www.arm.com/products/processors/cortex-a/cortex-a9.php

"Details of a New Cortex Processor Revealed."
http://www.arm.com/files/downloads/Cortex-A9_Devcon-talk_Introduction_FINAL-02.pdf

"EE Times Asia, May 17, 2007."
http://www.eetasia.com/ARTICLES/2007MAY/3/EEOL_2007MAY17_CTRLD_NP_01_10hblock.jpg

"First the Tick, Now the Tock: Next Generation Intel® Microarchitecture (Nehalem)."
http://www.intel.com/technology/architecture-silicon/next-gen/whitepaper.pdf

"HyperTransport Multiprocessor Interconnect."
http://en.wikipedia.org/wiki/HyperTransport#Multiprocessor_interconnect

"Intel® QuickPath Interconnect."
http://www.intel.com/technology/quickpath/whitepaper.pdf

"UltraSPARC® T2 Processor."
http://www.sun.com/processors/UltraSPARC-T2/datasheet.pdf