

BCM1480 Memory Aliasing and ccNUMA Initialization

by
Ali Ghorashi
April 28, 2005



Contents

1	Purpose	3
2	Overview	3
3	HyperTransport	4
3.1	Bridge Configuration	5
3.2	PCI Device Numbering	6
3.3	Routing Through the Hypertransport Subsystem	6
3.3.1	Address Range Register Forwarding	6
3.3.2	Node Routing	7
3.3.3	Routing Algorithm	7
4	ccNUMA	8
4.1	Cache Coherency	8
4.2	BCM1480 ccNUMA Implementation	9
4.3	1480 Remote Line Directory	9
5	Memory Aliasing Initialization Driver Usage	10
5.1	Driver Parameters	10
6	Notes on Caching Mechanisms	10
6.1	Cache Mapping and Associativity	10
6.2	Comparison of Cache Mapping Techniques	11

List of Figures

1	Structure of the BCM1480 Memory	3
2	Multi-Chip Connections	4
3	Internal Bridge Structure	5
4	Bridge Routing Rules	5
5	Bridge Connectivity in a Motherboard-Daughter Card configuration	6
6	In-Line Topology	6
7	Bridge Configuration in a 4-Node In-Line Topology	8
8	Transaction Data Path	8
9	RLD Entry	9
10	RLD Read Operation Example	10

Abstract

The purpose of this document is to provide the background information necessary to understand the Memory Aliasing Initialization Driver code.

1 Purpose

The purpose of the Memory Aliasing Initialization Driver is to initialize the memory aliasing and ccNUMA systems.

2 Overview

The BCM1480 has a built in system for sharing memory between multiple chips [1, p 33]. The memory space of each 1480 is split into several regions which are used to access the memory of another chip. Figure 1 shows the memory structure of the 1480. The address space between 0x40_0000_0000 and 0xF0_0000_0000 is referred to as the aliased region. Each chip is assigned a unique *node id* with a value between 0x4 and 0xE. The *Memory Aliasing Initialization Driver* sets the value of the *node id* by configuring the appropriate bits in the *System Configuration* register.

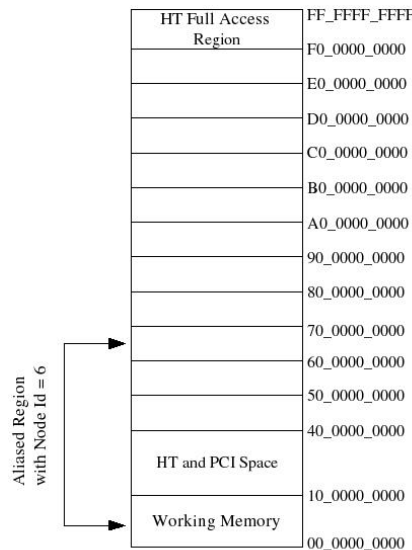


Figure 1: Structure of the BCM1480 Memory

The working memory region (0x00_0000_0000:0x10_0000_0000-1) on each chip corresponds to an *alias* memory region located between 0x[N]0_0000_0000 and 0x[N+1]0_0000_0000-1 where N is the node id of that chip. Reads and writes to the alias region have the same effect as accessing the working region directly. That is, a read from 0x01_0000_0000 is equivalent to a read from 0xN1_0000_0000. Cache coherency between the working and alias memory is accomplished through a system called *ccNUMA*. See section 4 for more details.

In the following discussion, it is assumed that 3 BCM1480 chips are connected in the configuration shown in figure 2.

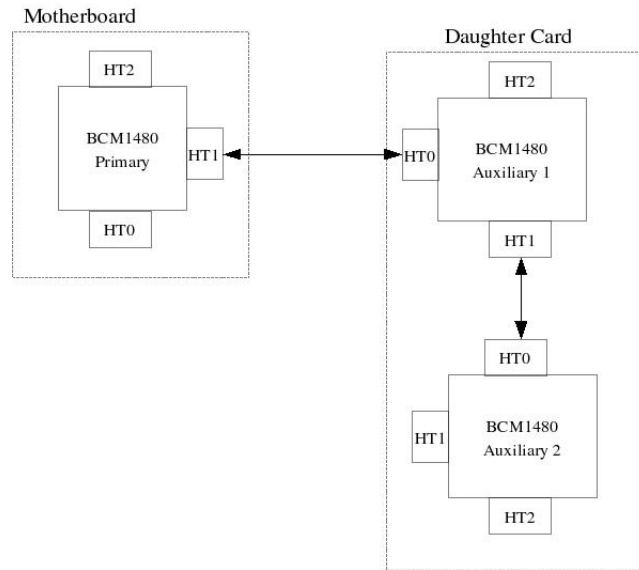


Figure 2: Multi-Chip Connections

3 HyperTransport

[2] Exchange of information between the chips is accomplished through the HyperTransport ports. Each BCM1480 is equipped with three HyperTransport ports. [1, p 197] Each HyperTransport port is connected to a PCI to HyperTransport bridge which connects the internal "virtual" PCI bus to the outside world (see figure 3). These HT-PCI bridges can be configured to operate in either one of three modes: *Disabled*, *Primary* and *Secondary*. A bridge in primary mode acts as an endpoint HT device (referred to as a *cave* in HT). A given system can have 0 to 3 bridges in Secondary mode but only 1 bridge can be configured to operate in Primary mode. Each bridge is configured with an address window defined by a set of base-limit registers. A transaction with an address that falls inside this window is considered to be a matching address on that bus. The mode of the bridge determines how the Hypertransport side is viewed by the bridge¹. Figure 4 shows the routing rules in each type of bridge. The forwarding rules can be summarized as follows.

- Matching addresses on the primary bus are forwarded to the secondary bus; they are ignored if generated on the secondary bus.
- Non-Matching addresses on the primary bus are not forwarded to the secondary bus.
- Matching addresses on the secondary bus are not forwarded to the primary bus.
- Non-Matching addresses on the secondary bus are forwarded to the primary bus.

The ZB-bus² is connected to the virtual PCI bus through a *Host Bridge* (see figure 3). This PCI-ZB bridge can be configured to allow full access to the local aliased memory region (i.e N0.0000.0000 with N=Node Id) through a pair of PCI configuration registers called *Full Access BARs*. These registers allow the Host bridge to open up a 64G byte window to the chip's ZB-bus. PCI transactions on the virtual bus with addresses that match the value programmed in these registers are placed on the ZB-Bus with their address bits 36:39 changed to the Node Id. Therefore, all accesses from the virtual PCI bus will always be mapped to the local aliased region.

¹i.e. a bridge in primary mode considers the HT side the primary bus and a bridge in secondary mode considers the HT side the secondary bus

²The internal bus connecting, among other things, CPU cores and memory.

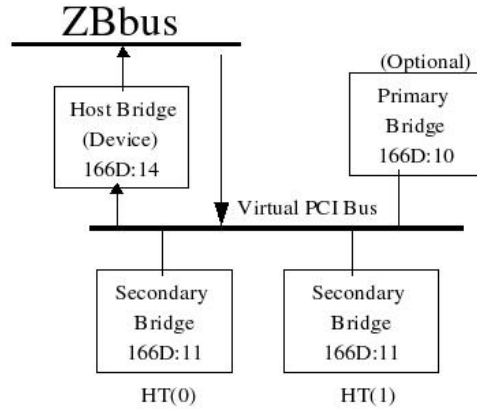


Figure 3: Internal Bridge Structure

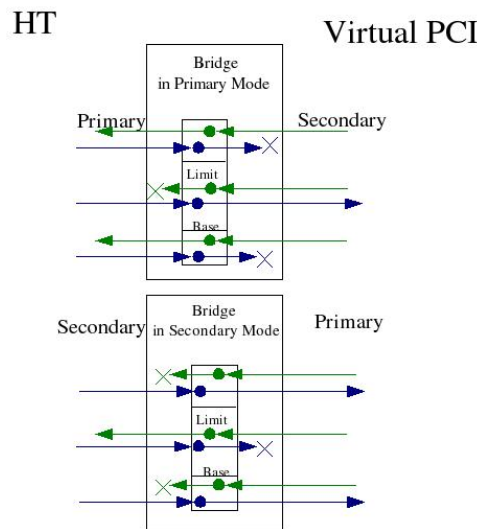


Figure 4: Bridge Routing Rules

3.1 Bridge Configuration

Figure 5 shows the bridge connectivity in a Motherboard-Daughter Card configuration. In order for memory access requests to be forwarded to the correct chip, the HT-PCI and the PCI-ZB bridges need to be configured properly. In addition, if more than two BCM1480 are connected, each device needs to be programmed with a set of routing rules. For the purpose of this design, it is assumed that an in-line configuration shown in figure 6 is used. However, this design can be enhanced to incorporate different connection topologies (e.g. square and mesh). The in-line configuration is the simplest topology and is inherently free of cyclical connections which can cause routing dead-locks ³.

³If a topology with cyclical connections is used, the route tables must be configured carefully in order to avoid routing dead locks.

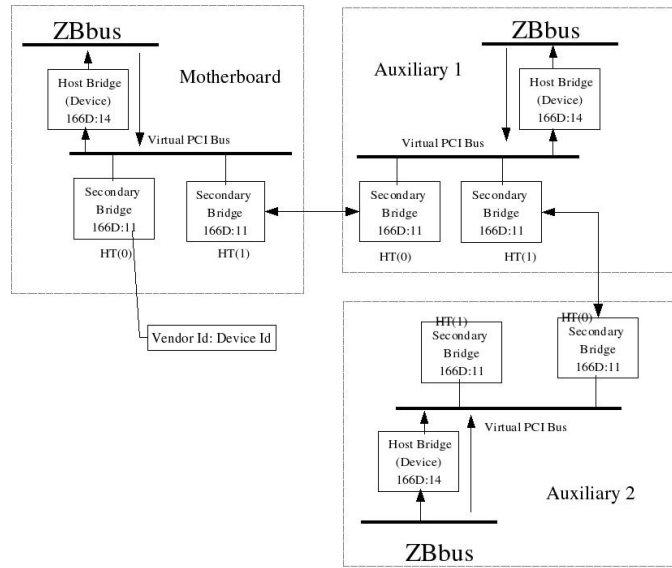


Figure 5: Bridge Connectivity in a Motherboard-Daughter Card configuration

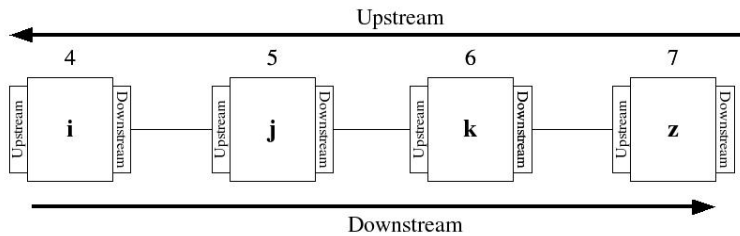


Figure 6: In-Line Topology

3.2 PCI Device Numbering

Devices on the internal virtual PCI bus are assigned device numbers based on the following rules:

- Ports in "primary" mode are assigned the bus number and Unit Id chosen by the enumeration root.
- Ports in "secondary" mode have a fixed Device Id on the internal bus. Physical ports 0, 1, and 2 are always assigned to Device Ids 0, 1 and 2.
- The host bridge is always assigned to Device Id 4.

The Memory Aliasing Initialization Driver uses the above rules to determine which bridge to configure for an upstream/downstream port.

3.3 Routing Through the Hypertransport Subsystem

There are two ways to route data through the chips: Address Range Register (ARR) Forwarding and Node Routing.

3.3.1 Address Range Register Forwarding

In ARR, Secondary bridge Base Address Registers (BAR) are configured to pass data down/upstream by inspecting the address of the transaction [1, p 212]. In this implementation, the upstream bridge is setup



to accept all downstream addresses including its own aliased memory region. The downstream bridge is configured to accept all upstream addresses including its own aliased memory region. This method is a little more difficult to setup and manage. The Memory Aliasing Initialization Driver uses another BCM1480 specific routing mechanism called *Node Routing* described below.

3.3.2 Node Routing

In addition to the standard HT routing mechanism, the BCM1480 implementation supports a special kind of routing called *Node Routing* [1, p 216]. This method is designed to simplify Multi-chip routing. Each bridge, either in Primary or Secondary mode, has a 16 entry⁴ route table. Each entry in this table is 4-bits wide. The upper 4 bits (36:39 i.e. the node id) of a transaction address are used to index into this routing table in order to look up the appropriate action. Each bit in the route table entry is defined as follows:

IsOnSecForIO If this bit is set, the transaction should be placed on the secondary side. If this bit is clear the transaction should be on the primary side.

OverRideForIO If this bit is set, node routing for specified node id is enabled. If this bit is clear, ARR routing is used.

IsOnSecForCC Same as IsOnSecForIO except it is used for Cache Coherency transactions.

OverRideForCC Same as OverRideForIO except it is used for Cache Coherency transactions.

3.3.3 Routing Algorithm

The Memory Aliasing Initialization Driver uses the following assumptions and conventions to setup the Node Routing mechanisms in each bridge:

- Each BCM1480 will be assigned a unique sequential Node Id by the module loading framework.
- The left most (see figure 6) device shall have the lowest Node Id.
- The HT port and the corresponding bridge connected to a device with a lower Node Id are called *Upstream Port* and *Upstream Bridge* respectively.
- The HT port and the corresponding bridge connected to a device with a higher Node Id are called *Downstream Port* and *Downstream Bridge* respectively.

Given the above assumptions, we can define a bridge configuration algorithm as shown in listing 1. Figure 7 illustrates the bridge configuration for a four node system. Even though the Motherboard-Daughter card configuration only uses 3 BCM1480s, this example shows how this algorithm can support up to 11 devices (maximum number of node ids).

Algorithm 1 Algorithm For Configuring Upstream and Downstream bridge

Require: All routing entries are cleared on both the upstream and downstream bridges.

- 1: Enable *OverRideForCC* for all entries
 - 2: Enable *OverRideForIO* for all entries
 - 3: **for all** Upstream Ids **do**
 - 4: Set *IsOnSecForIO* and *IsOnSecForCC* on the Upstream Bridge
 - 5: **end for**
 - 6: **for all** Downstream Ids **do**
 - 7: Set *IsOnSecForIO* and *IsOnSecForCC* on the Downstream Bridge
 - 8: **end for**
-

With the route tables shown in figure 7, memory can be shared between multiple chips. Figure 8 shows the path used by a transaction from the chip on the motherboard to the second auxiliary BCM1480 on the daughter card.

⁴Even though the table has 16 entries, only rows 4-14 are valid. Writing to other entries can have undefined effects.

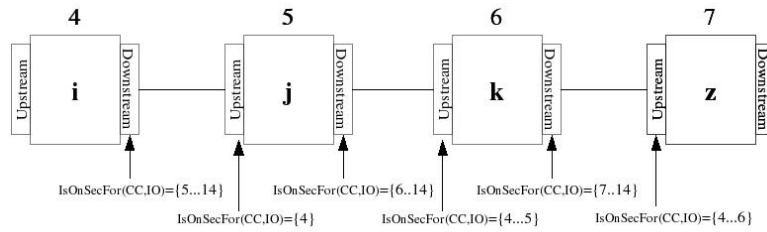


Figure 7: Bridge Configuration in a 4-Node In-Line Topology

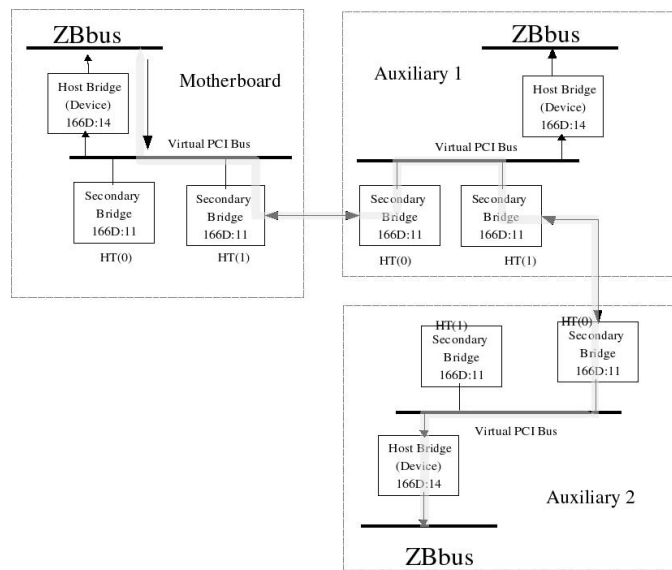


Figure 8: Transaction Data Path

4 ccNUMA

Cache Coherent Non-Uniform Memory Access (*ccNUMA*) is a scheme used to maintain cache coherency between multiple processors sharing the same memory. Since data can be lumped at a particular node and memory access latency can vary greatly between each node due to network topology, this scheme is referred to as "non-uniform".

4.1 Cache Coherency

In order to reduce latencies associated with memory access, most modern processors use a data caching scheme. In a multiprocessor system, it is necessary to keep the cache memory coherent across all processors. This is necessary since processor caches have multiple copies of the same data. When one processor modifies data, it does so in its cache, on a cache line basis. In a cache coherent system, the process of modifying memory must also include the step of notifying other processors sharing the same data. The next time the cache line is referenced, the home processor (the one that modified the data) is forced to write the data back to memory and then cause it to be reloaded into the consuming processor's cache[3].

There are two approaches of notifying other processors in cache coherent system: "snoopy" schemes and Directory-based schemes. In the "snoopy" approach each node "listens" to the memory bus and

watches for memory addresses that it may have encached. This method does not scale very well and can be only be used in small CPU clusters. The BCM1480 uses a snooping mechanism to maintain cache coherency within the chip among the four Sibyte cores.

In a directory-based coherency mechanism the memory state is cataloged in a memory based table (directory). The directory is used to notify any requesting processor of the state of a given cache line (clean and unmodified, or modified and dirty). It also provides information as to which cache owns the cache line.

4.2 BCM1480 ccNUMA Implementation

The BCM1480 implements a directory based cache coherency mechanism [1, p 291]. The directory catalog in the BCM1480 is referred to as **Remote Line Directory (RLD)**. This directory behaves like a cache for the L2 cache. The RLD contains 16K entries organized into an 8-way associative table (see 6.1) with a random placement algorithm. Since the RLD does not have enough room to contain all the cache lines in a system, it can only contain a subset of all coherent memory lines. When the RLD runs out of room, it will evict an old entry and issues a notification to all the concerned nodes.

4.3 1480 Remote Line Directory

[4][5] For each cache line, the node controller maintains a directory which describes the state of each cached block [1, 295]. Thus, the system keeps track of cache inconsistencies and can take remedial action when necessary. The node whose memory controller space contains a given cache block is referred to as the *home node*. During remote access or write operations, the home node directory is consulted to resolve cache inconsistencies. Figure 9 shows the basic structure of the BCM1480 RLD table entry. The main components of each entry are:

Node Vector A bit field used to keep track of which node owns a shared copy of a particular entry. If a node contains a shared copy of the cache line, the bit corresponding to that node is set to '1'.

State The state of the entry. It can be one of the following:

- Shared
- Modified (dirty)
- Invalid

Owner Node ID The ID of the node that owns the cache entry.

ECC Bits Bits used to preform **Error Checking and Correction (ECC)**

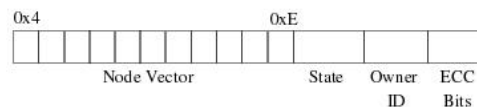


Figure 9: RLD Entry

Let's consider a simple example (see figure 10): Assume node 0x4 owns location A and assigns it a value of 1. At a later time, node 0x5 tries to write a value of 0x2 to location A. In order to do this, node 0x5 sends a message to node 0x4 telling it to mark A as dirty. This message forces node 0x4 to issue invalidation (KILL) requests to all the outstanding shared copies, as indicated by the node vector. Node 0x4 does not allow any more access requests to location A until all shared processors have responded (ACK) to the invalidate message. When node 0x4 receives all the acknowledgments, it changes the *owner node id* to node 0x5. Now, let's assume node 0x6 tries to read A. The home node of A, node 0x4, looks up the owner of the block (node 0x5) and forwards the request to it. After receiving the request, node 0x5 responds with data and in effect "short circuits" node 0x4. Then, node 0x4 writes a '1' to the slot corresponding to node 0x6 in the node vector of block A to indicate that node 0x6 also has a cached copy of the data.

There are multiple variation on the above example involving exclusive access and modification operations. See [5] for an in depth discussion.

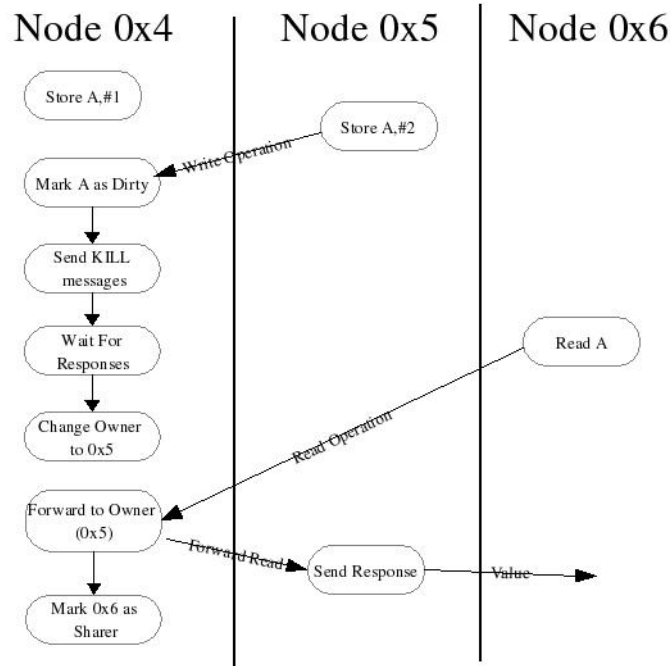


Figure 10: RLD Read Operation Example

5 Memory Aliasing Initialization Driver Usage

`insmod mema.o [node_id = N] [upstream_port = N] [downstream_port = N]`

5.1 Driver Parameters

node_id Node ID of the current chip

upstream_port Hypertransport port used for upstream transactions. If this value is set to 0XF, the driver assumes that the current chip is the first node in the chain.

downstream_port Hypertransport port used for upstream transactions. If this value is set to 0XF, the driver assumes that the current chip is the last node in the chain.

6 Notes on Caching Mechanisms

This section includes notes on different caching mechanisms taken directly from [6].

6.1 Cache Mapping and Associativity

A very important factor in determining the effectiveness of the level 2 cache relates to how the cache is mapped to the system memory. What this means in brief is that there are many different ways to allocate the storage in our cache to the memory addresses it serves. Let's take as an example a system



with 512 KB of L2 cache and 64 MB of main memory. The burning question is: how do we decide how to divvy up the 16,384 address lines in our cache amongst the "huge" 64 MB of memory?

There are three different ways that this mapping can generally be done. The choice of mapping technique is so critical to the design that the cache is often named after this choice:

Direct Mapped Cache The simplest way to allocate the cache to the system memory is to determine how many cache lines there are (16,384 in our example) and just chop the system memory into the same number of chunks. Then each chunk gets the use of one cache line. This is called direct mapping. So if we have 64 MB of main memory addresses, each cache line would be shared by 4,096 memory addresses (64 M divided by 16 K).

Fully Associative Cache Instead of hard-allocating cache lines to particular memory locations, it is possible to design the cache so that any line can store the contents of any memory location. This is called fully associative mapping.

N-Way Set Associative Cache "N" here is a number, typically 2, 4, 8 etc. This is a compromise between the direct mapped and fully associative designs. In this case the cache is broken into sets where each set contains "N" cache lines, let's say 4. Then, each memory address is assigned a set, and can be cached in any one of those 4 locations within the set that it is assigned to. In other words, within each set the cache is associative, and thus the name. This design means that there are "N" possible places that a given memory location may be in the cache. The tradeoff is that there are "N" times as many memory locations competing for the same "N" lines in the set. Let's suppose in our example that we are using a 4-way set associative cache. So instead of a single block of 16,384 lines, we have 4,096 sets with 4 lines in each. Each of these sets is shared by 16,384 memory addresses (64 M divided by 4 K) instead of 4,096 addresses as in the case of the direct mapped cache. So there is more to share (4 lines instead of 1) but more addresses sharing it (16,384 instead of 4,096).

Conceptually, the direct mapped and fully associative caches are just "special cases" of the N-way set associative cache. You can set "N" to 1 to make a "1-way" set associative cache. If you do this, then there is only one line per set, which is the same as a direct mapped cache because each memory address is back to pointing to only one possible cache location. On the other hand, suppose you make "N" really large; say, you set "N" to be equal to the number of lines in the cache (16,384 in our example). If you do this, then you only have one set, containing all of the cache lines, and every memory location points to that huge set. This means that any memory address can be in any line, and you are back to a fully associative cache.

6.2 Comparison of Cache Mapping Techniques

There is a critical tradeoff in cache performance that has led to the creation of the various cache mapping techniques described in the previous section. In order for the cache to have good performance you want to maximize both of the following: Hit Ratio: You want to increase as much as possible the likelihood of the cache containing the memory addresses that the processor wants. Otherwise, you lose much of the benefit of caching because there will be too many misses. Search Speed: You want to be able to determine as quickly as possible if you have scored a hit in the cache. Otherwise, you lose a small amount of time on every access, hit or miss, while you search the cache.

Now let's look at the three cache types and see how they fare:

Direct Mapped Cache The direct mapped cache is the simplest form of cache and the easiest to check for a hit. Since there is only one possible place that any memory location can be cached, there is nothing to search; the line either contains the memory information we are looking for, or it doesn't. Unfortunately, the direct mapped cache also has the worst performance, because again there is only one place that any address can be stored. Let's look again at our 512 KB level 2 cache and 64 MB of system memory. As you recall this cache has 16,384 lines (assuming 32-byte cache lines) and so each one is shared by 4,096 memory addresses. In the absolute worst case, imagine that the processor needs 2 different addresses (call them X and Y) that both map to the same cache line, in alternating sequence (X, Y, X, Y). This could happen in a small loop if you were unlucky. The



processor will load X from memory and store it in cache. Then it will look in the cache for Y, but Y uses the same cache line as X, so it won't be there. So Y is loaded from memory, and stored in the cache for future use. But then the processor requests X, and looks in the cache only to find Y. This conflict repeats over and over. The net result is that the hit ratio here is 0%. This is a worst case scenario, but in general the performance is worst for this type of mapping.

Fully Associative Cache The fully associative cache has the best hit ratio because any line in the cache can hold any address that needs to be cached. This means the problem seen in the direct mapped cache disappears, because there is no dedicated single line that an address must use. However (you knew it was coming), this cache suffers from problems involving searching the cache. If a given address can be stored in any of 16,384 lines, how do you know where it is? Even with specialized hardware to do the searching, a performance penalty is incurred. And this penalty occurs for all accesses to memory, whether a cache hit occurs or not, because it is part of searching the cache to determine a hit. In addition, more logic must be added to determine which of the various lines to use when a new entry must be added (usually some form of a "least recently used" algorithm is employed to decide which cache line to use next). All this overhead adds cost, complexity and execution time.

N-Way Set Associative Cache The set associative cache is a good compromise between the direct mapped and set associative caches. Let's consider the 4-way set associative cache. Here, each address can be cached in any of 4 places. This means that in the example described in the direct mapped cache description above, where we accessed alternately two addresses that map to the same cache line, they would now map to the same cache set instead. This set has 4 lines in it, so one could hold X and another could hold Y. This raises the hit ratio from 0% to near 100%! Again an extreme example, of course. As for searching, since the set only has 4 lines to examine this is not very complicated to deal with, although it does have to do this small search, and it also requires additional circuitry to decide which cache line to use when saving a fresh read from memory. Again, some form of LRU (least recently used) algorithm is typically used.

References

- [1] *BCM1255/BCM1280/BCM1455/BCM1480 User Manual*, Broadcom, 2003.
- [2] J. Trodden and D. Anderson, *Hypertransport System Architecture*. MindShare Inc, 2003.
- [3] (2003) ccnuma overview. Hewlett Packard Corp.
- [4] S. Baden. (2001) System cache. Web. Univerisy of California San Diago. [Online]. Available: <http://www.cse.ucsd.edu/classes/fa00/lectures/Lecture18.html>
- [5] D. L. James Lauden, "The sgi origin: A ccnuma highly scalable server," SGI, Tech. Rep.
- [6] C. M. Kozierok. (2001, April) System cache. Web. Hewlett Packard Corp. [Online]. Available: <http://www.pcguid.com/ref/mbsys/cache/>