1 Memory Microarchitectural Design Patterns 3
  1.1. Transactions and Steps ........................................ 3
  1.2. Microarchitecture Overview .................................. 4

2 FSM Cache 5
  2.1. High-Level Idea for FSM Cache ............................... 6
  2.2. FSM Cache Datapath ............................................ 7
  2.3. FSM Cache Control Unit ....................................... 9
  2.4. Analyzing Performance ....................................... 10

3 Pipelined Cache 12
  3.1. High-Level Idea for Pipelined Cache ........................ 13
  3.2. Pipelined Cache Datapath and Control Unit ................ 14
  3.3. Analyzing Performance ....................................... 20
  3.4. Pipelined Cache with TLB .................................... 22

4 Cache Microarchitecture Optimizations 25
  4.1. Reduce Hit Time ............................................... 25
  4.2. Reduce Miss Rate ............................................. 26
1. Memory Microarchitectural Design Patterns

\[
\text{Time} \quad \frac{\text{Mem Access Sequence}}{\text{Mem Access Sequence}} = \frac{\text{Mem Accesses}}{\text{SEQ}} \times \frac{\text{Avg Cycles}}{\text{Mem Access}} \times \frac{\text{Time}}{\text{Cycle}}
\]

\[
\frac{\text{Avg Cycles}}{\text{Mem Access}} = \frac{\text{Avg Cycles}}{\text{Hit}} + \left( \frac{\text{Num Misses}}{\text{Num Accesses}} \times \frac{\text{Avg Extra Cycles}}{\text{Miss}} \right)
\]

<table>
<thead>
<tr>
<th>Microarchitecture</th>
<th>Hit Latency</th>
<th>Extra Accesses for Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSM Cache</td>
<td>&gt;1</td>
<td>1+</td>
</tr>
<tr>
<td>Pipelined Cache</td>
<td>≈1</td>
<td>1+</td>
</tr>
<tr>
<td>Pipelined Cache + TLB</td>
<td>≈1</td>
<td>≈0</td>
</tr>
</tbody>
</table>

1.1. Transactions and Steps

- We can think of each memory access as a transaction
- Executing a memory access involves a sequence of steps
  - Check Tag : Check one or more tags in cache
  - Select Victim : Select victim line from cache using replacement policy
  - Evict Victim : Evict victim line from cache and write victim to memory
  - Refill : Refill requested line by reading line from memory
  - Write Mem : Write requested word to memory
  - Access Data : Read or write requested word in cache
1. Memory Microarchitectural Design Patterns

1.1. Transactions and Steps

Steps for Write-Through with No Write Allocate

1.2. Microarchitecture Overview

Steps for Write-Back with Write Allocate
2. FSM Cache

\[
\frac{\text{Time}}{\text{Mem Access Sequence}} = \frac{\text{Mem Accesses}}{\text{Sequence}} \times \frac{\text{Avg Cycles}}{\text{Mem Access}} \times \frac{\text{Time}}{\text{Cycle}}
\]

\[
\frac{\text{Avg Cycles}}{\text{Mem Access}} = \frac{\text{Avg Cycles}}{\text{Hit}} + \left( \frac{\text{Num Misses}}{\text{Num Accesses}} \times \frac{\text{Avg Extra Cycles}}{\text{Miss}} \right)
\]

<table>
<thead>
<tr>
<th>Microarchitecture</th>
<th>Hit Latency</th>
<th>Extra Accesses for Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSM Cache</td>
<td>&gt;1</td>
<td>1+</td>
</tr>
<tr>
<td>Pipelined Cache</td>
<td>≈1</td>
<td>1+</td>
</tr>
<tr>
<td>Pipelined Cache + TLB</td>
<td>≈1</td>
<td>≈0</td>
</tr>
</tbody>
</table>

**Assumptions**

- Page-based translation, no TLB, physically addr cache
- Single-ported combinational SRAMs for tag, data storage
- Unrealistic combinational main memory
- Cache requests are 4 B

**Configuration**

- Four 16 B cache lines
- Two-way set-associative
- Replacement policy: LRU
- Write policy: write-through, no write allocate
2.1. High-Level Idea for FSM Cache

The diagram illustrates the high-level idea of FSM Cache, which involves:

- **Check Tag**
- **Access Data**
- **Select Victim**
- **Refill**
- **Write Mem**

The process flows as follows:

1. **Check**
2. **Tag**
3. **Access**
4. **Data**
5. **Check**
6. **Tag**
7. **Access**
8. **Data**
9. **Check**
10. **Tag**
11. **Access**
12. **Data**
13. **Check**
14. **Tag**
15. **Access**
16. **Data**

- **read hit**: Path from Check to Access Data
- **read miss**: Path from Check to Select Victim
- **write**: Path from Select Victim to Refill
- **write hit**: Path from Refill to Write Mem

In the FSM, the states are marked with **Check Tag**, **Access Data**, **Select Victim**, and **Refill**. The transitions are labeled as **read hit**, **read miss**, **write**, and **write hit**.
2.2. FSM Cache Datapath

As with processors, we design our cache datapath by incrementally adding support for each transaction and resolving conflicts with muxes.

**Implementing READ transactions that hit**

MT: Check tag
MRD: Read data array, return cacheresp

**Implementing READ transactions that miss**

MT: Check tag
MRD: Read data array, return cacheresp
R0: Send refill memreq, get memresp
R1: Write data array with refill cache line
Implementing WRITE transactions that miss

MT: Check tag
MRD: Read data array, return cacheresp
R0: Send refill memreq, get memresp
R1: Write data array with refill cache line
MWD: Send write memreq, write data array

Implementing WRITE transactions that hit

MT: Check tag
MRD: Read data array, return cacheresp
R0: Send refill memreq, get memresp
R1: Write data array with refill cache line
MWD: Send write memreq, write data array
2.3. FSM Cache Control Unit

We will need to keep valid bits in the control unit, with one valid bit for every cache line. We will also need to keep use bits which are updated on every access to indicate which was the last “used” line. Assume we create the following two control signals generated by the FSM control unit.

\[
\text{hit} = ( \text{tarray0\_match} \land\land \text{valid0}[\text{idx}] ) \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad || ( \text{tarray1\_match} \land\land \text{valid1}[\text{idx}] )
\]

\[
\text{victim} = \lnot \text{use}[\text{idx}]
\]

<table>
<thead>
<tr>
<th></th>
<th>tarray0</th>
<th>tarray1</th>
<th>darray</th>
<th>darray</th>
<th>worden</th>
<th>z4b</th>
<th>memreq</th>
<th>cachresp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>en</td>
<td>en</td>
<td>en</td>
<td>en</td>
<td>sel</td>
<td>sel</td>
<td>sel</td>
<td>val</td>
</tr>
<tr>
<td>MT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MRD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MWD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.4. Analyzing Performance

\[
\text{Time} = \frac{\text{Mem Access Sequence}}{\text{Mem Accesses Sequence}} \times \frac{\text{Avg Cycles}}{\text{Mem Access}} \times \frac{\text{Time}}{\text{Cycle}}
\]

\[
\frac{\text{Avg Cycles}}{\text{Mem Access}} = \frac{\text{Avg Cycles}}{\text{Hit}} + \left( \frac{\text{Num Misses}}{\text{Num Accesses}} \times \frac{\text{Avg Extra Cycles}}{\text{Miss}} \right)
\]

Estimating cycle time

- register read/write = 1\(\tau\)
- tag array read/write = 10\(\tau\)
- data array read/write = 10\(\tau\)
- mem read/write = 20\(\tau\)
- decoder = 3\(\tau\)
- comparator = 10\(\tau\)
- mux = 3\(\tau\)
- repl unit = 0\(\tau\)
- z4b = 0\(\tau\)
Estimating AMAL

Consider the following sequence of memory accesses which might correspond to copying 4 B elements from a source array to a destination array. Each array contains 64 elements. What is the AMAL?

\[
\begin{align*}
\text{rd} & 0x1000 \\
\text{wr} & 0x2000 \\
\text{rd} & 0x1004 \\
\text{wr} & 0x2004 \\
\text{rd} & 0x1008 \\
\text{wr} & 0x2008 \\
& \ldots \\
\text{rd} & 0x1040 \\
\text{wr} & 0x2040 \\
\end{align*}
\]

Consider the following sequence of memory accesses which might correspond to incrementing 4 B elements in an array. The array contains 64 elements. What is the AMAL?

\[
\begin{align*}
\text{rd} & 0x1000 \\
\text{wr} & 0x1000 \\
\text{rd} & 0x1004 \\
\text{wr} & 0x1004 \\
\text{rd} & 0x1008 \\
\text{wr} & 0x1008 \\
& \ldots \\
\text{rd} & 0x1040 \\
\text{wr} & 0x1040 \\
\end{align*}
\]
3. Pipelined Cache

Time
\[
\text{Mem Access Sequence} = \frac{\text{Mem Accesses}}{\text{Sequence}} \times \frac{\text{Avg Cycles}}{\text{Mem Access}} \times \frac{\text{Time}}{\text{Cycle}}
\]

Avg Cycles
\[
\frac{\text{Avg Cycles}}{\text{Mem Access}} = \frac{\text{Avg Cycles}}{\text{Hit}} + \left( \frac{\text{Num Misses}}{\text{Num Accesses}} \times \frac{\text{Avg Extra Cycles}}{\text{Miss}} \right)
\]

<table>
<thead>
<tr>
<th>Microarchitecture</th>
<th>Hit Latency</th>
<th>Extra Accesses for Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSM Cache</td>
<td>&gt;1</td>
<td>1+</td>
</tr>
<tr>
<td>Pipelined Cache</td>
<td>≈1</td>
<td>1+</td>
</tr>
<tr>
<td>Pipelined Cache + TLB</td>
<td>≈1</td>
<td>≈0</td>
</tr>
</tbody>
</table>

Assumptions

- Page-based translation, no TLB, physically addr cache
- Single-ported combinational SRAMs for tag, data storage
- Unrealistic combinational main memory
- Cache requests are 4 B

Configuration

- Four 16 B cache lines
- Direct-mapped
- Replacement policy: LRU
- Write policy: write-through, no write allocate
3. Pipelined Cache

3.1. High-Level Idea for Pipelined Cache

![Diagram of pipelined cache]

- **Check Tag**
- **Access Data**
- **Select Victim**
- **Write Mem**
- **Refill**

FSM:
- **read hit**
- **write hit**
- **read miss**

Pipelined:
3.2. Pipelined Cache Datapath and Control Unit

As with processors, we incrementally adding support for each transaction and resolving conflicts using muxes.

Implementing READ transactions that hit

Implementing WRITE transactions that hit
Implementing transactions that miss

- Hybrid pipeline/FSM design pattern
- Hit path is pipelined with two-cycle hit latency
- Miss path stalls in M0 stage to refill cache line

**Pipeline diagram for pipelined cache with 2-cycle hit latency**

| RD (hit) | WR (hit) | RD (miss) | RD (miss) | WR (miss) | RD (hit) |
Parallel read with pipelined write datapath

Pipeline diagram for parallel read with pipelined write

rd (hit)
rd (hit)
rd (hit)
wr (hit)
wr (hit)
rd (hit)

- Achieves single-cycle hit latency for reads
- Two-cycle hit latency for writes, but is this latency observable?
- With write acks, send write-ack back in M0 stage
- How do we resolve structural hazards?
Resolving structural hazard by exposing in ISA

```
rd (hit)
wr (hit)
nop
rd (hit)
```

Resolving structural hazard with hardware stalling

```
rd (hit)
wr (hit)
rd (hit)
```

\[
\text{ostall}_M0 = \text{val}_M0 && (\text{type}_M0 == \text{RD}) \\
&& \text{val}_M1 && (\text{type}_M1 == \text{WR})
\]

Resolving structural hazard with hardware duplication

```
rd (hit)
wr (hit)
rd (hit)
```

Resolving RAW data hazard with software scheduling

Software scheduling: hazard depends on memory address, so difficult to know at compile time!

Resolving RAW data hazard with hardware stalling

```
\text{ostall}_M0 = \ldots
```
Resolving RAW data hazard with hardware bypassing

We could use the previous stall signal as our bypass signal, but we will also need a new bypass path in our datapath. Draw this new bypass path on the following datapath diagram.
Parallel read and pipelined write in set associative caches

To implement parallel read in set-associative caches, we must speculatively read a line from each way in parallel with tag check. This can be expensive in terms of latency, motivating two-cycle hit latencies for highly associative caches.
3.3. Analyzing Performance

\[
\text{Time} = \frac{\text{Mem Access Sequence}}{\text{Mem Accesses Sequence}} \times \frac{\text{Avg Cycles}}{\text{Mem Access}} \times \frac{\text{Time}}{\text{Cycle}}
\]

\[
\frac{\text{Avg Cycles}}{\text{Mem Access}} = \frac{\text{Avg Cycles}}{\text{Hit}} + \left( \frac{\text{Num Misses}}{\text{Num Accesses}} \times \frac{\text{Avg Extra Cycles}}{\text{Miss}} \right)
\]

Estimating cycle time

- register read/write = 1\(\tau\)
- tag array read/write = 10\(\tau\)
- data array read/write = 10\(\tau\)
- mem read/write = 20\(\tau\)
- decoder = 3\(\tau\)
- comparator = 10\(\tau\)
- mux = 3\(\tau\)
- repl unit = 0\(\tau\)
- z4b = 0\(\tau\)
Estimating AMAL

Assume a parallel-read/pipelined-write microarchitecture with hardware duplication to resolve the structural hazard and stalling to resolve RAW hazards. Consider the following sequence of memory accesses which might correspond to copying 4 B elements from a source array to a destination array. Each array contains 64 elements. What is the AMAL?

rd 0x1000
wr 0x2000
rd 0x1004
wr 0x2004
rd 0x1008
wr 0x2008
rd 0x100c
wr 0x200c
wr 0x1010
3.4. Pipelined Cache with TLB

How should we integrate a MMU (TLB) into a pipelined cache?

Physically Addressed Caches

Perform memory translation before cache access

- Advantages:
  - Physical addresses are unique, so cache entries are unique
  - Updating memory translation simply requires changing TLB

- Disadvantages:
  - Increases hit latency

Virtually Addressed Caches

Perform memory translation after or in parallel with cache access
• Advantages:
  – Simple one-step process for hits

• Disadvantages:
  – Intra-program protection (store protection bits in cache?)
  – I/O uses physical addr (map into virtual addr space?)
  – Virtual address homonyms
  – Virtual address synonyms (aliases)

Virtual Address Homonyms

Single virtual address points to two physical address.

![Virtual Address Homonyms Diagram]

• Example scenario
  – Program 1 brings VA 0x3000 into cache
  – Program 1 is context swapped for program 2
  – Program 2 hits in cache, but gets incorrect data!

• Potential solutions
  – Flush cache on context swap
  – Store program ids (address space IDS) in cache
Virtual Address Synonyms (Aliases)

- **Example scenarios**
  - User and OS can potentially have different VAs point to same PA
  - Memory map the same file (via `mmap`) in two different programs

- **Potential solutions**
  - Hardware checks all ways (and potentially different sets) on a miss to ensure that a given physical address can only live in one location in cache
  - Software forces aliases to share some address bits (page coloring) reducing the number of sets we need to check on a miss (or reducing the need to check any other locations for a direct mapped cache)
Virtually Indexed and Physically Tagged Caches

- Page offset is the same in VA and PA
- Up to $n$ bits of physical address available without translation
- If index bits + cache offset ($n+b$) < page offset bits ($m$), can do translation in parallel with reading out the physical tag and data
- Complete (physical) tag check once tag/data access complete
- With 4 KB pages, direct-mapped cache must be $\leq$ 4 KB
- Larger page sizes (decrease $k$) or higher associativity (decrease $n$) enable larger virtually indexed, physically tagged caches
4. Cache Microarchitecture Optimizations

4.1. Reduce Hit Time

Cache Microarchitecture Optimizations

AMAL = Hit Latency + (Miss Rate × Miss Penalty)

- Reduce hit time
  - Small and simple caches

- Reduce miss penalty
  - Multi-level cache hierarchy
  - Prioritize reads

- Reduce miss rate
  - Large block size
  - Large cache size
  - High associativity
  - Hardware prefetching
  - Compiler optimizations

Second Optimization: Way Prediction to Reduce Hit Time

Another approach reduces conflict misses and yet maintains the hit speed of direct-mapped cache. In way prediction, extra bits are kept in the cache to predict the way, or block within the set of the next cache access. This prediction means the multiplexor is set early to select the desired block, and only a single tag comparison is performed that clock cycle in parallel with reading the cache data. A miss results in checking the other blocks for matches in the next clock cycle.

Added to each block of a cache are block predictor bits. The bits select which of the blocks to try on the next cache access. If the predictor is correct, the cache access latency is the fast hit time. If not, it tries the other block, changes the way predictor, and has a latency of one extra clock cycle. Simulations suggest that set prediction accuracy is in excess of 90% for a two-way set associative cache and 80% for a four-way set associative cache, with better accuracy on I-caches than D-caches. Way prediction yields lower average memory access time for a two-way set associative cache if it is at least 10% faster, which is quite likely. Way prediction was first used in the MIPS R10000 in the mid-1990s. It is popular in processors that use two-way set associativity and is used in the ARM Cortex-A8 with four-way set associative caches. For very fast processors, it may be challenging to implement the one cycle stall that is critical to keeping the way prediction penalty small.
4.2. Reduce Miss Rate

Large Block Size

- Less tag overhead
- Exploit fast burst transfers from DRAM and over wide on-chip busses
- Can waste bandwidth if data is not used
- Fewer blocks → more conflicts

Large Cache Size or High Associativity

If cache size is doubled, miss rate usually drops by about $\sqrt{2}$

Direct-mapped cache of size $N$ has about the same miss rate as a two-way set-associative cache of size $N/2$
Hardware Prefetching

- Previous techniques only help capacity and conflict misses
- Hardware prefetcher looks for patterns in miss address stream
- Attempts to predict what the next miss might be
- Prefetches this next miss into a prefetch buffer
- Very effective in reducing compulsory misses for streaming accesses
Compiler Optimizations

- Restructuring code affects the data block access sequence
  - Group data accesses together to improve spatial locality
  - Re-order data accesses to improve temporal locality

- Prevent data from entering the cache
  - Useful for variables that will only be accessed once before eviction
  - Needs mechanism for software to tell hardware not to cache data
    (“no-allocate” instruction hits or page table bits)

- Kill data that will never be used again
  - Streaming data exploits spatial locality but not temporal locality
  - Replace into dead-cache locations

Loop Interchange and Fusion

What type of locality does each optimization improve?

```c
for(j=0; j < N; j++) {
    for(i=0; i < M; i++) {
        x[i][j] = 2 * x[i][j];
    }
}
```
Matrix Multiply with Naive Code

for(i=0; i < N; i++)
    for(j=0; j < N; j++) {
        r = 0;
        for(k=0; k < N; k++)
            r = r + y[i][k] * z[k][j];
        x[i][j] = r;
    }

Matrix Multiply with Cache Tiling

for(jj=0; jj < N; jj=jj+B)
    for(kk=0; kk < N; kk=kk+B)
        for(i=0; i < N; i++)
            for(j=jj; j < min(jj+B, N); j++) {
                r = 0;
                for(k=kk; k < min(kk+B, N); k++)
                    r = r + y[i][k] * z[k][j];
                x[i][j] = x[i][j] + r;
            }
4.3. Reduce Miss Penalty

Multi-Level Caches

\[ AMAL_{L1} = \text{Hit Latency of L1} + (\text{Miss Rate of L1} \times AMAL_{L2}) \]

\[ AMAL_{L2} = \text{Hit Latency of L2} + (\text{Miss Rate of L2} \times \text{Miss Penalty of L2}) \]

- Local miss rate = misses in cache / accesses to cache
- Global miss rate = misses in cache / processor memory accesses
- Misses per instruction = misses in cache / number of instructions

- Use smaller L1 if there is also a L2
  - Trade increased L1 miss rate for reduced L1 hit time & L1 miss penalty
  - Reduces average access energy

- Use simpler write-through L1 with on-chip L2
  - Write-back L2 cache absorbs write traffic, doesn’t go off-chip
  - Simplifies processor pipeline
  - Simplifies on-chip coherence issues

- Inclusive Multilevel Cache
  - Inner cache holds copy of data in outer cache
  - External coherence is simpler

- Exclusive Multilevel Cache
  - Inner cache may hold data in outer cache
  - Swap lines between inner/outer cache on miss
Prioritize Reads

- Processor not stalled on writes, and read misses can go ahead of writes to main memory
- Write buffer may hold updated value of location needed by read miss
  - On read miss, wait for write buffer to be empty
  - Check write buffer addresses and bypass

### 4.4. Cache Optimization Summary

<table>
<thead>
<tr>
<th>Technique</th>
<th>Hit Lat</th>
<th>Miss Rate</th>
<th>Miss Penalty</th>
<th>BW</th>
<th>HW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smaller caches</td>
<td>−</td>
<td>+</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Avoid TLB before indexing</td>
<td>−</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Large block size</td>
<td>−</td>
<td>+</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Large cache size</td>
<td>+</td>
<td>−</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>High associativity</td>
<td>+</td>
<td>−</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Hardware prefetching</td>
<td>−</td>
<td></td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Compiler optimizations</td>
<td>−</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Multi-level cache</td>
<td>−</td>
<td></td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Prioritize reads</td>
<td>−</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Pipelining</td>
<td>+</td>
<td></td>
<td>+</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
5. Case Study: ARM Cortex A8 and Intel Core i7

5.1. ARM Cortex A8

- **L1 data cache**
  - 32 KB, 64 B cache lines, 4-way set-associative with random replacement
  - $32K/64 = 512$ cache lines, 128 lines per set (set index = 7 bits)
  - Virtually indexed, physically tagged with single-cycle hit latency

- **Memory management unit**
  - TLB with multi-level page tables in physical memory
  - TLB has 32 entries, fully associative, hardware TLB miss handler
  - Variable page size: 4 KB, 16 KB, 64 KB, 1 MB, 16 MB (fig shows 16 KB)
  - TLB tag entries can have wildcards to support multiple page sizes

- **L2 cache**
  - 1 MB, 64 B cache lines, 8-way set-associative
  - $1M/64 = 16K$ cache lines, 2K lines per set (set index = 11 bits)
  - Physically addressed with multi-cycle hit latency
## 5.2. Intel Core i7

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>L1</th>
<th>L2</th>
<th>L3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>32 KB I/32 KB D</td>
<td>256 KB</td>
<td>2 MB per core</td>
</tr>
<tr>
<td>Associativity</td>
<td>4-way I/8-way D</td>
<td>8-way</td>
<td>16-way</td>
</tr>
<tr>
<td>Access latency</td>
<td>4 cycles, pipelined</td>
<td>10 cycles</td>
<td>35 cycles</td>
</tr>
<tr>
<td>Replacement scheme</td>
<td>Pseudo-LRU</td>
<td>Pseudo-LRU</td>
<td>Pseudo-LRU but with an ordered selection algorithm</td>
</tr>
</tbody>
</table>

- Write-back with merging write buffer (more like no write allocate)
- L3 is inclusive of L1/L2
- Hardware prefetching from L2 into L1, from L3 into L2
- Virtually indexed, physically tagged L1 caches
- Physically addressed L2/L3 caches

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Instruction TLB</th>
<th>Data DLB</th>
<th>Second-level TLB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>128</td>
<td>64</td>
<td>512</td>
</tr>
<tr>
<td>Associativity</td>
<td>4-way</td>
<td>4-way</td>
<td>4-way</td>
</tr>
<tr>
<td>Replacement</td>
<td>Pseudo-LRU</td>
<td>Pseudo-LRU</td>
<td>Pseudo-LRU</td>
</tr>
<tr>
<td>Access latency</td>
<td>1 cycle</td>
<td>1 cycle</td>
<td>6 cycles</td>
</tr>
<tr>
<td>Miss</td>
<td>7 cycles</td>
<td>7 cycles</td>
<td>Hundreds of cycles to access page table</td>
</tr>
</tbody>
</table>

- 48 bit virtual addresses and 36 bit physical addresses (36 GB physical mem)
- 4 KB pages except for few large 2–4 MB pages in L1 TLBs