ECE 4750 Computer Architecture

Topic 3: Memory Concepts

http://www.csl.cornell.edu/courses/ece4750
School of Electrical and Computer Engineering
Cornell University

revision: 2022-11-14-10-41

List of Problems

1 Short Answer **2**

1.A Qualitatively Analyzing Locality . . . 3

1.B Linear Page Tables and TLBs . . . 4

1.C Linear Page Tables, Two-Level Page Tables, and TLBs . . . 6

2 Augmenting L1 Cache with a L1.5 Cache **7**

2.A Performance with L1 Cache (Weight: ×2) . . . 9

2.B Performance with L1 Cache and L1.5 Cache (Weight: ×2) . . . 10

3 TinyRV1 Instruction Cache **11**

3.A Categorizing Cache Misses . . . 11

3.B Average Memory Access Latency . . . 11

3.C Set-Associativity . . . 11

4 Page-Based Memory Translation **12**

4.A Two-Level Page Tables . . . 13

4.B Translation-Lookaside Buffer . . . 14

Problem 1. Short Answer

Part 1.A Qualitatively Analyzing Locality

Consider the following C function which uses a lookup table (`lut`) to convert an array of integers (`src`) into an array of characters (`dest`). The function assumes that the `src` array only contains numbers in the range 0–9. Assume the numbers in the `src` are uniform randomly distributed, and that the size of the `src` and `dest` arrays is specified with `size` argument. Note that each element in the look-up table is a `char` which requires a single byte of storage. *For this problem, assume that the source and destination arrays contain on the order of hundreds of thousands of elements..*

```
1  void num2char( char* dest, int* src, int n )
2  {
3    char lut[10] = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9' };
4    for ( int i = 0; i < n; i++ )
5      dest[i] = lut[ src[i] ];
6  }
```

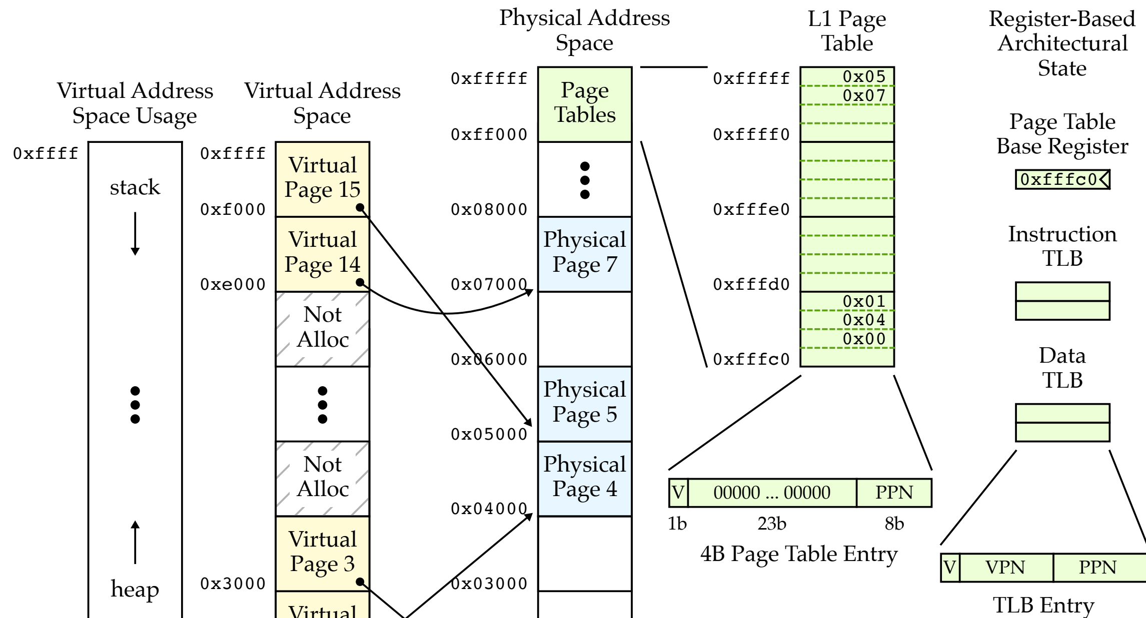
In the table below, circle a number suggesting how much temporal/spatial locality is present in the given memory access stream. Use 0 to indicate no locality and 5 to indicate very significant locality. Provide a brief explanation of your answers.

Spatial locality in instruction stream	0	1	2	3	4	5
Temporal locality in instruction stream	0	1	2	3	4	5
Spatial locality in accesses to array `src`	0	1	2	3	4	5
Temporal locality in accesses to array `src`	0	1	2	3	4	5
Spatial locality in accesses to array `dest`	0	1	2	3	4	5
Temporal locality in accesses to array `dest`	0	1	2	3	4	5
Spatial locality in accesses to array `lut`	0	1	2	3	4	5
Temporal locality in accesses to array `lut`	0	1	2	3	4	5

Part 1.B Linear Page Tables and TLBs

Consider the small-scale page-based memory translation system shown below. The memory system uses byte addresses, 16-bit virtual addresses (i.e., we have 64KB of virtual memory), 20-bit physical addresses (i.e., we have 1MB of physical memory), 4 KB pages, and two-entry fully associative translation-lookaside buffers (TLBs) with LRU replacement.

Assume program A was running for some amount of time, was context swapped by the operating system so that program B could run, and is now being context swapped back onto the processor. This means that some amount of physical memory has already been allocated to program A and the page table is already initialized and stored in physical memory. However, since a context swap flushes the TLBs, all entries in both TLBs are now invalidated. The figure shows the state of the system when we restart execution of program A. The page table starts at address `0xfffc0`. All page-table entries (PTEs) are assumed to be four bytes: one valid bit, 11 bits that are always zero, and 20 bits for a physical address. A page-table base register is already initialized to point to the base of the page table.



Assume the following instructions are the first instructions executed after program A is context swapped back onto the processor. Assume `x2` is initially `0xe100`, `x4` is initially `0xe200`, and `x6` is initially `0xf100`.

```
1  0x1100 lw  x1, 0(x2)   # assume x2 is initially 0xe100
2  0x1104 lw  x3, 0(x4)   # assume x4 is initially 0xe200
3  0x1108 add x5, x1, x3
4  0x110c sw  x5, 0(x6)   # assume x6 is initially 0xf100
```

Fill in the following two tables to show the state of both the instruction and data TLB during the given instruction sequence. We use a dash (–) to indicate an invalid TLB entry (recall that all TLB entries are initially invalid). Fill in the VPN and page offset for each memory transaction before updating the VPN and PPN of each TLB entry after each memory transaction. Indicate which accesses result in a TLB miss or hit. Indicate the total number of memory accesses for each memory transaction (i.e., include any accesses to the page tables and the actual access corresponding to the memory transaction). *You only need to fill in elements in the table when the value changes! Remember that the TLB entries should reflect the state of the TLB before the corresponding transaction on that row executes!*

Instruction TLB

Virtual Transaction Address	VPN	Page Offset	m/h	Total Num Mem Accesses	Instruction TLB: TLB Way 0 VPN	TLB Way 0 PPN	TLB Way 1 VPN	TLB Way 1 PPN
					–	–	–	–

Data TLB

Virtual Transaction Address	VPN	Page Offset	m/h	Total Num Mem Accesses	Data TLB: TLB Way 0 VPN	TLB Way 0 PPN	TLB Way 1 VPN	TLB Way 1 PPN
					–	–	–	–

Part 1.C Linear Page Tables, Two-Level Page Tables, and TLBs

Consider three different page-based memory management units (MMUs): the MMU-1L design uses a linear page table with no TLB; the MMU-2L design uses a two-level page table with no TLB; and the MMU-TLB design uses a TLB to cache translations from a two-level page table. In all three designs, the root of the current page table is stored in a page-table base register, while the rest of the page table is stored in physical memory. For this problem, *assume that there are no TLB misses or virtual memory page faults.*

Calculate the total number of memory accesses required to fetch and execute a single `lw` instruction using each of the three MMU designs. *Remember that you must account for memory accesses due to both the instruction fetch and the actual data memory access.* **Show your work, explain your answers, and answer the final comparison question show below.**

- ▷ How many memory accesses are required with the MMU-1L design?
- ▷ How many memory accesses are required with the MMU-2L design?
- ▷ How many memory accesses are required with the MMU-TLB design?
- ▷ Briefly compare the MMU-1L and MMU-2L designs in terms of the number of memory accesses and the required space overhead.

Problem 2. Augmenting L1 Cache with a L1.5 Cache

In this problem, you will explore the two memory systems shown in Figures 1 and 2. The baseline design in Figure 1 uses a standard direct-mapped cache, while the alternative design in Figure 2 adds an L1.5 cache in between the L1 cache and main memory. For this problem we will only focus on read transactions. In both designs, the L1 cache uses the following configuration:

- L1 Total Capacity : 256B cache
- L1 Cache Line Size : 16B cache lines
- L1 Num Cache Lines : 16
- L1 Hit Latency : 2 cycles
- L1 Style : direct mapped

As illustrated in Figure 1, an L1 miss in the baseline design results in a miss penalty of 10 cycles. The L1.5 cache is a small fully associative cache with the following configuration:

- L1.5 Total Capacity : 32B cache
- L1.5 Cache Line Size : 16B cache lines
- L1.5 Num Cache Lines : 2
- L1.5 Hit Latency : 1 cycle
- L1.5 Style : fully associative
- L1.5 Replacement Policy : LRU

On an L1 miss in the alternative design, we will first check to see if there is a hit in the L1.5 cache before accessing main memory. If there is a hit in the L1.5 cache then we do not need to access main memory and we can directly bring the desired cache line into the L1 cache. If we miss in both the L1 and L1.5 caches, then we bring the cache line into *both* the L1.5 cache *and* the L1 cache. As illustrated in Figure 2, the L1 miss penalty in the alternative design depends on whether or not this L1 miss hits in the L1.5 cache or also misses in the L1.5 cache. If the L1 miss hits in the L1.5 cache, then the L1 miss penalty is simply the L1.5 hit latency (i.e., one cycle). If the L1 miss also misses in the L1.5 cache, then the L1 miss penalty increases to 11 cycles.

We will examine these two memory systems as they execute the following transaction sequence. All caches initially start with every cache line invalidated, and each access reads four bytes at the given address. There are no write accesses, so there are no evictions. Assume that the transaction pattern repeats such that the entire transaction sequence includes thousands of transactions.

```
1  0x1010, 0x2024, 0x3010, 0x2024,
2  0x1010, 0x2024, 0x3010, 0x2024,
3  0x1010, 0x2024, 0x3010, 0x2024, ...
```

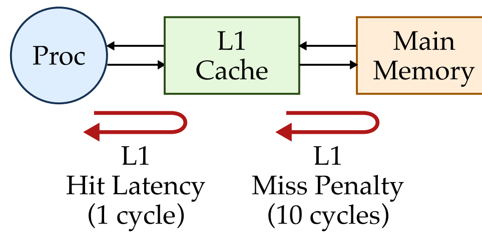


Figure 1: Baseline Design with L1 Cache

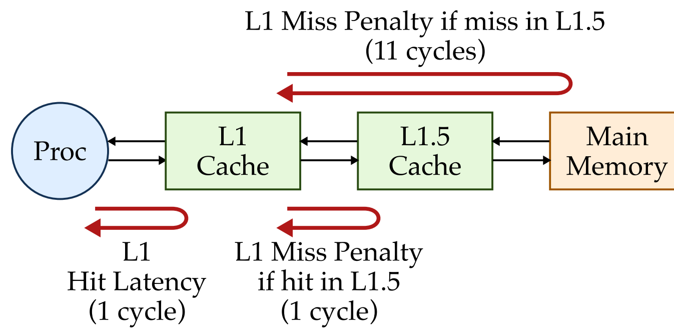


Figure 2: Alternative Design with L1 and L1.5 Caches

Part	Microarchitecture	L1 Hit Latency (cycles)	L1 Miss Rate	L1.5 Miss Rate	L1 Miss Penalty (cycles)	AMAL (cycles)
3.A	**L1**	2		n/a		
3.B	**L1 + L1.5**	2				

Figure 3: Memory Performance for Entire Sequence of Thousands of Transactions on Two Different Microarchitectures

Part 2.A Performance with L1 Cache (Weight: ×2)

In this part, we will estimate the average memory access latency in cycles for the transaction sequence shown above executing on the baseline design with just an L1 cache. **Fill in the following table to illustrate how the state of the L1 cache changes over time.** Since there are 16 lines in the L1 cache, you must specify the appropriate line index at the top of each column in the given boxes. The latency column should indicate the total number of cycles required for that transaction. Use a dash symbol (–) to indicate if the corresponding state is invalid. If the state of a cache line does not change, then you can leave the corresponding entry blank. All addresses should be in hex. *All state should reflect the state before the transaction on that row executes!* **Fill in the appropriate row of the table in Figure 3 considering the entire transaction sequence which includes thousands of transactions. Please clearly explain how you calculate the miss rate and the miss penalty. You must show your work and state any assumptions.** The L1 miss rate is for the L1 cache in isolation (ignoring what happens in the L1.5 cache).

Transaction			L1 Cache					latency
				Set Index				
Address	tag	idx	m/h	0				(cycles)
0x1010								
0x2024								
0x3010								
0x2024								
0x1010								
0x2024								
0x3010								
0x2024								
0x1010								
0x2024								
0x3010								
0x2024								
...								

Part 2.B Performance with L1 Cache and L1.5 Cache (Weight: ×2)

In this part, we will estimate the average memory access latency in cycles for the transaction sequence shown above executing on the alternative design with an L1 cache and L1.5 cache. **Fill in the following table to illustrate how the state of the L1 and L1.5 caches change over time.** Since there are 16 lines in the L1 cache, you must specify the appropriate line index at the top of each column in the given boxes. The latency column should indicate the total number of cycles required for that transaction. Use a dash symbol (–) to indicate if the corresponding state is invalid. If the state of a cache line does not change, then you can leave the corresponding entry blank. All addresses should be in hex. *All state should reflect the state before the transaction on that row executes!* **Fill in the appropriate row of the table in Figure 3 considering the thousands of transactions. Please clearly explain how you calculate the miss rate and the miss penalty. You must show your work and state any assumptions.**

			L1 Cache					L1.5 Cache			
Transaction				Set Index					Way Number		**latency**
Address	**tag**	**idx**	**m/h**	**0**				**m/h**	**0**	**1**	**(cycles)**
0x1010											
0x2024											
0x3010											
0x2024											
0x1010											
0x2024											
0x3010											
0x2024											
0x1010											
0x2024											
0x3010											
0x2024											
...											

Problem 3. TinyRV1 Instruction Cache

In this problem, we will be exploring adding an instruction cache to an TinyRV1 FSM processor. We will be using the TinyRV1 assembly program shown in Figure 4. The first column shows the instruction address for each instruction. Note that these addresses are byte addresses. The value of `x1` is initially 64, meaning that there are 64 iterations in the loop. In this problem, we will be considering the execution of this loop with a direct-mapped instruction cache microarchitecture with eight 16 B cache lines. This means each cache line can hold four instructions and the bottom four bits of an instruction address are the block offset. *Hint: The first instruction in Figure 4 (i.e.,* `addi x1, x1, -1`*), is in the middle of a cache line.*

Part 3.A Categorizing Cache Misses

Create a table like the one shown in Figure 4. In the appropriate column, write *compulsory*, *conflict*, or *capacity* next to each instruction which misses in the instruction cache to indicate the type of instruction cache misses that occur in the first and second iteration of the loop. Assume that the instruction cache is initially completely empty.

Part 3.B Average Memory Access Latency

Calculate the instruction cache miss rate for 64 iterations of the loop. Calculate the average instruction cache memory access latency in cycles for 64 iterations of the loop. Assume the hit time is one cycle and that the miss penalty is 15 cycles. You must show your work, especially the various components of the average memory access latency. **Remark on which kind of miss is dominating the average memory access latency.**

Part 3.C Set-Associativity

Qualitatively, predict how the cache performance would change if we replace the eight-entry, direct-mapped cache with an eight-entry, two-way, set-associative cache. Both caches have a one-cycle hit latency. **What kind of misses would be present with this kind of cache microarchitecture?**

Addr	Instruction	Iteration 1	Iteration 2
	`loop:`		
0x108	`addi x1, x1, -1`		
0x10c	`addi x2, x2, 1`		
0x110	`jal x0, foo`		
	`...`		
	`foo:`		
0x218	`addi x6, x6, 1`		
0x21c	`bne x1, x0, loop`		

Figure 4: Example TinyRV1 Assembly Loop

Problem 4. Page-Based Memory Translation

In this problem, we will be exploring a small-scale page-based memory translation system that uses 4 KB pages, a two-level page-table, and a two-entry translation-lookaside buffer (TLB). For all parts, we will assume that all addresses are byte addresses, virtual addresses are 16 bits (i.e., we have 64KB of virtual memory), and physical addresses are 20 bits (i.e., we have 1MB of physical memory). We have more physical memory than virtual memory to enable multiple programs to be resident in physical memory at the same time. While these small memory spaces are not realistic, they will help simplify the problem.

Assume program A was running for some amount of time, was context swapped by the operating system so that program B could run, and is now being context swapped back onto the processor. This means that some amount of physical memory has already been allocated to program A and the two-level page table is already initialized and stored in physical memory. However, since a context swap flushes the TLB, all entries in the TLB are now invalidated. Figure 5 shows the state of the system when we restart execution of program A. Note that only five virtual pages have been allocated to program A; the remaining 11 virtual pages are unallocated. The L1 page table and each L2 page table has four entries. The page tables are stored at the very top of the physical memory address space. The L1 page table starts at address `0xffff0` and the L2 page tables are directly below the L1 page table. All page-table entries (PTEs) are assumed to be four bytes: one valid bit,

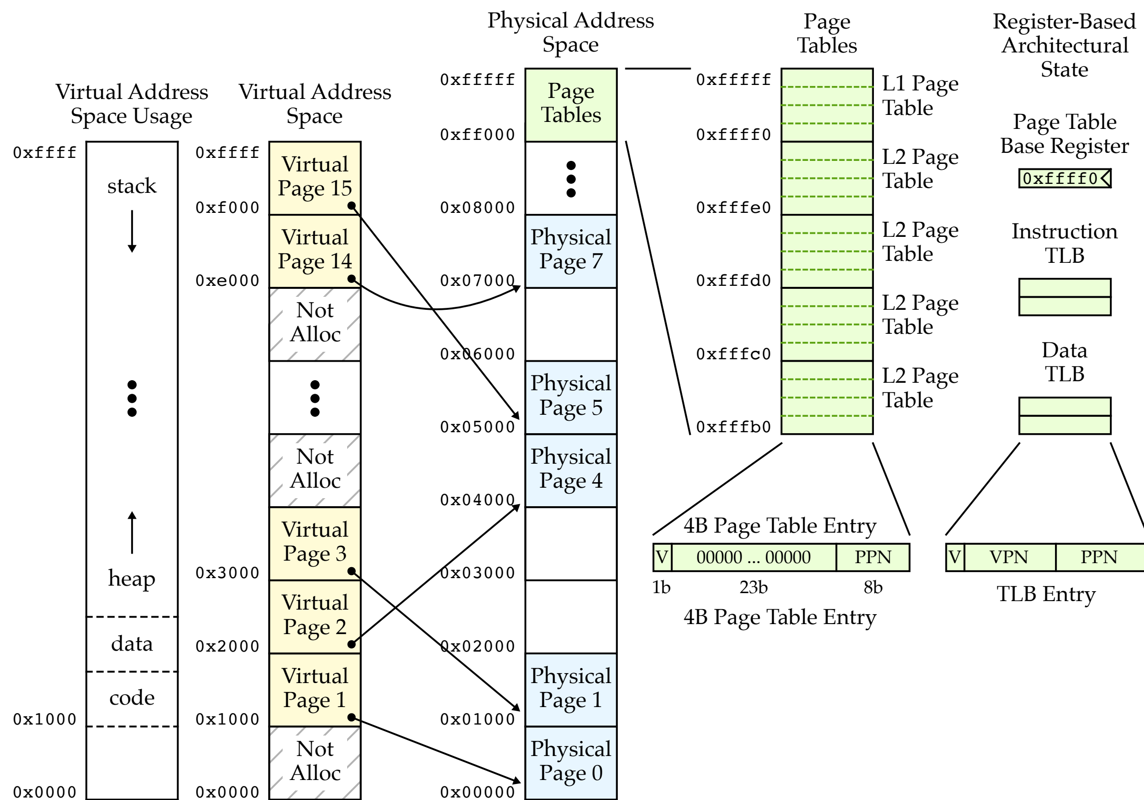


Figure 5: Small-Scale Page-Based Memory Translation System

11 bits that are always zero, and 20 bits for a physical address. A page-table base register is already initialized to point to the base of the L1 page table.

Part 4.A Two-Level Page Tables

A two-level page table is a space-efficient way to translate virtual addresses to physical addresses. The L1 page table entries point to L2 page tables, and the L2 page table entries point to the corresponding page in physical memory. The virtual address is used to "walk" the page table. Some bits of the virtual address are used to index into the L1 page table, different bits of the virtual address are used to index into the L2 page table, and finally the page offset bits are used to index into the physical page. **Clearly indicate which bits of the virtual address are used for: (a) the page offset, (b) the virtual page number, (c) indexing into the L1 page table, and (d) indexing into the L2 page table.**

The L1 page table has four PTEs, and there are four L2 page tables each with four PTEs for a total of 20 PTEs. As discussed in lecture, these page tables are stored in physical memory. **Create a table similar to the one shown in Figure 6 which shows the contents of physical memory where the page tables for program A reside.** We have provided one page-table entry for the L1 page-table to get you started.

As an aside, we probably should not have pre-allocated all five page tables! As you will see, only a subset of these page tables actually need to be allocated, so by pre-allocating all five pages we have mitigated the key advantage of a two-level page table compared to a one-level page table. Please note that if all of the PTEs in a L2 page table are invalid then then there should *not* be a valid PTE entry in the L1 page table pointing to this L2 page table. In other words, let's try and capture the idea that we would not really need to allocate L2 page tables for which all entries are invalid.

	Page-Table Entry	
Paddr	**Valid**	**Ptr or PPN**
0xffffc		
0xffff8		
0xffff4		
0xffff0	1	0xfffb0
0xfffec		
0xfffe8		
0xfffe4		
0xfffe0		
0xfffdc		
0xfffd8		
0xfffd4		
0xfffd0		
0xfffcc		
0xfffc8		
0xfffc4		
0xfffc0		
0xfffbc		
0xfffb8		
0xfffb4		
0xfffb0		

Figure 6: Contents of Physical Memory with Page Tables

Part 4.B Translation-Lookaside Buffer

A two-level page table requires two additional memory accesses for every instruction or data memory request. A translation-lookaside buffer (TLB) can be used to cache translations and provide single-cycle mappings between virtual to physical addresses. Each TLB entry includes a valid bit, virtual page number (VPN), and physical page number (PPN). TLBs are usually flushed on a context swap. This is one step in implementing memory protection. Flushing the TLB prevents one program from accidentally using an old translation in the TLB to access physical memory allocated to a different program. Unfortunately, this results in TLB misses when a program restarts execution.

We will assume that program A was in the middle of copying a large amount of data from the stack to the heap when it was context swapped. Now that program A is restarting, it will continue copying the data from the stack to the heap. This results in the following address stream:

```
0xeff4, 0x2ff0, 0xeff8, 0x2ff4, 0xeffc, 0x2ff8, 0xf000,
0x2ffc, 0xf004, 0x3000, 0xf008, 0x3004, 0xf00c, 0x3008
```

We will be focusing on a two-entry, fully associative TLB exclusively for data memory accesses (i.e., instruction memory accesses use a different TLB). Assume the TLB uses a least-recently used replacement policy. **Create a table similar to the one shown in Figure 7 which shows the state of the TLB during the given sequence of data memory request transactions.** To get you started, we have filled in the table for the first transaction. Use a dash (–) to indicate an invalid TLB entry (recall that all TLB entries are initially invalid). Fill in the VPN and page offset for each transaction before updating the VPN and PPN of each TLB entry after each transaction. Indicate which accesses result in a TLB miss or hit. Indicate the *total* number of memory accesses for each transaction (i.e., include any accesses to the page tables and the actual access corresponding to the memory transaction). Include the total number of TLB misses and the TLB miss rate in your table. *You only need to fill in elements in the table when the value changes! Remember that the TLB entries should always reflect the state of the TLB before the corresponding transaction on that row executes!*

Transaction		Page		Total Num Mem	TLB Way 0		TLB Way 1	
Address	VPN	Offset	m/h	Accesses	VPN	PPN	VPN	PPN
0xeff4	0xe	0xff4	m	3	–	–	–	–
0x2ff0	...				0xe	0x07		
0xeff8	...							
Number of Misses =								
Miss Rate =								

Figure 7: TLB Contents Over Time