**NC State University**

**Department of Electrical and Computer Engineering**

**ECE 463/521: Spring 2012 (Rotenberg)**

**Project #1: Cache Design, Memory Hierarchy Design (Version 1.0)**

**Due: Friday, February 24, 11:59 PM**

# 1. Groundrules

This is the first project for the course, so let me begin by discussing some groundrules:

1. All students must work alone. The project scope is reduced (but still substantial) for ECE 463 students, as detailed in this specification.
2. Sharing of code between students is considered cheating and will receive appropriate action in accordance with University policy. The TAs will scan source code (from current and past semesters) through various tools available to us for detecting cheating. Source code that is flagged by these tools will be dealt with severely: 0 on the project and referral to the Office of Student Conduct for sanctions.
3. A Wolfware message board will be provided for posting questions, discussing and debating issues, and making clarifications. It is an essential aspect of the project and communal learning. Students must not abuse the message board, whether inadvertently or intentionally. Above all, never post actual code on the message board (unless permitted by the TAs/instructor). When in doubt about posting something of a potentially sensitive nature, email the TAs and instructor to clear the doubt.
4. It is recommended that you do all your work in the C, C++ or Java languages. Exceptions must be approved by the instructor.
5. Use of the Eos Linux environment is *required*. This is the platform where the TAs will compile and test your simulator. (WARNING: If you develop your simulator on another platform, get it working on that platform, and then try to port it over to Eos Linux at the last minute, you may encounter major problems. Porting is not as quick and easy as you think unless you are an excellent programmer. What's worse, malicious bugs can be hidden until you port the code to a different platform, which is an unpleasant surprise close to the deadline.)

# 2. Project Description

In this project, you will implement a flexible cache and memory hierarchy simulator and use it to compare the performance, area, and energy of different memory hierarchy configurations, using a subset of the SPEC-2000 benchmark suite.

# 3. Specification of Memory Hierarchy

Design a generic cache module that can be used at any level in a memory hierarchy. For example, this cache module can be "instantiated" as an L1 cache, an L2 cache, an L3 cache, and so on. Since it can be used at any level of the memory hierarchy, it will be referred to generically as CACHE throughout this specification.

## *3.1. Configurable parameters*

CACHE should be configurable in terms of supporting any cache size, associativity, and block size, specified at the beginning of simulation:

- SIZE: Total bytes of data storage.
- ASSOC: The associativity of the cache (ASSOC = 1 is a direct-mapped cache).
- BLOCKSIZE: The number of bytes in a block.

There are a few constraints on the above parameters: 1) BLOCKSIZE is a power of two and 2) the number of *sets* is a power of two. *Note that ASSOC (and, therefore, SIZE) need not be a power of two*. As you know, the number of sets is determined by the following equation:

$$\#sets = \frac{SIZE}{ASSOC \times BLOCKSIZE}$$

## *3.2. Replacement policy*

CACHE should use the LRU (least-recently-used) replacement policy.

*Note regarding simulator output: When printing out the final contents of CACHE, you must print the blocks within a set based on their recency of access, i.e., print out the MRU block first, the next most recently used block second, and so forth, and the LRU block last. This is required so that your output is consistent with the output of the TAs' simulator.*

## *3.3. Write policy*

CACHE should use the WBWA (write-back + write-allocate) write policy.

- Write-allocate: A write that misses in CACHE will cause a block to be allocated in CACHE. Therefore, both write misses and read misses cause blocks to be allocated in CACHE.
- Write-back: A write updates the corresponding block in CACHE, making the block dirty. It does not update the next level in the memory hierarchy (next level of cache or memory). If a dirty block is evicted from CACHE, a writeback (*i.e.*, a write of the entire block) will be sent to the next level in the memory hierarchy.

## *3.4. Allocating a block: Sending requests to next level in the memory hierarchy*

Your simulator must be capable of modeling one or more instances of CACHE to form an overall memory hierarchy, as shown in Fig. 1.

CACHE receives a read or write request from whatever is above it in the memory hierarchy (either the CPU or another cache). The only situation where CACHE must interact with the next level below it (either another CACHE or main memory) is when the read or write request misses in CACHE. When the read or write request misses in CACHE, CACHE must "allocate" the requested block so that the read or write can be performed.

Thus, let us think in terms of allocating a requested block X in CACHE. The allocation of requested block X is actually a two-step process. *The two steps must be performed in the following order*.

1. *Make space for the requested block X*. If there is at least one invalid block in the set, then there is already space for the requested block X and no further action is required (go to step 2). On the other hand, if all blocks in the set are valid, then a victim block V must be singled out for eviction, according to the replacement policy (Section 3.2). If this victim block V is dirty, then a write of the victim block V must be issued to the next level of the memory hierarchy.
2. *Bring in the requested block X*. Issue a read of the requested block X to the next level of the memory hierarchy and put the requested block X in the appropriate place in the set (as per step 1).

To summarize, when allocating a block, CACHE issues a write request (*only* if there is a victim block and it is dirty) followed by a read request, both to the next level of the memory hierarchy. Note that each of these two requests could themselves miss in the next level of the memory hierarchy (if the next level is another CACHE), causing a cascade of requests in subsequent levels. *Fortunately, you only need to correctly implement the two steps for an allocation locally within CACHE. If an allocation is correctly implemented locally (steps 1 and 2, above), the memory hierarchy as a whole will automatically handle cascaded requests globally.*

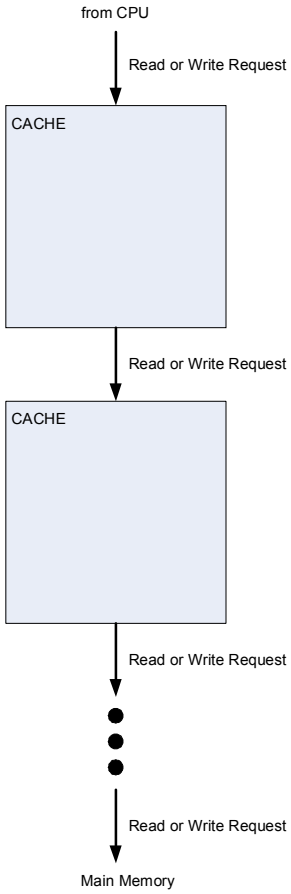


**Fig. 1: Your simulator must be capable of modeling one or more instances of CACHE to form an overall memory hierarchy.**

### *3.5. Updating state*

After servicing a read or write request, whether the corresponding block was in the cache already (hit) or had just been allocated (miss), remember to update other state. This state includes LRU counters affiliated with the set as well as the valid and dirty bits affiliated with the requested block.

## 4. ECE 521 Students: Augment CACHE with Stream-Buffer Prefetching

Students enrolled in ECE 521 must additionally augment CACHE with a prefetch unit. The prefetch unit implements Stream Buffers.

In this project, consider the prefetch unit to be an extension implemented within CACHE. This preserves the clean abstraction of one or more instances of CACHE interacting in an overall memory hierarchy (see Fig. 1), where each CACHE may have a prefetch unit within it.

### *4.1. Prefetch unit can be enabled or disabled*

Your simulator should be able to specify, for each CACHE, whether or not its prefetch unit is enabled.

## 4.2. Prefetch unit parameters

The prefetch unit has N Stream Buffers. Each Stream Buffer contains M memory blocks. Both N and M should be configurable. Setting N=0 disables the prefetch unit.

## 4.3. Operation of a single Stream Buffer

A Stream Buffer is a simple queue that is capable of holding M *consecutive* memory blocks.

When CACHE receives a read or write request, both CACHE and the *first entry* in its Stream Buffer are checked for a hit. There are three possible scenarios:

1. **Scenario #1: Request hits in CACHE:** In this case, nothing happens with respect to the Stream Buffer.
2. **Scenario #2: Request misses in CACHE and misses in first entry of Stream Buffer:** Handle the miss in CACHE as usual (see Section 3.4). In addition to fetching the requested block X into CACHE, prefetch the next M consecutive memory blocks into the Stream Buffer. That is, prefetch memory blocks X+1, X+2, …, X+M into the Stream Buffer, thereby replacing the entire contents of the Stream Buffer. Note that prefetches are implemented by issuing read requests to the next level in the memory hierarchy.[1] Also note that replacing the contents of the Stream Buffer does not involve any writebacks from the Stream Buffer: this will be explained in Section 4.5.
3. **Scenario #3: Request misses in CACHE and hits in first entry of Stream Buffer:** Perform an allocation in CACHE as follows. First, make space in CACHE for the requested block X (as described in Section 3.4). Second, instead of fetching the requested block X from the next level in the memory hierarchy, move the requested block X from the first entry in the Stream Buffer into CACHE (since the first entry of the Stream Buffer contains the requested block X, in this third scenario). Note that the requested block X is removed from the Stream Buffer. This frees up the first entry in the Stream Buffer. The remaining M-1 blocks in the Stream Buffer are "shifted up" such that the next consecutive block, X+1, is now in the first entry and the last entry is now free. The last entry should be refilled by prefetching a single memory block that is consecutive with respect to the last memory block currently in the Stream Buffer, thereby continuing the prefetch stream.

## 4.4. Multiple Stream Buffers

The operation of a single Stream Buffer, described in the previous section, extends to multiple Stream Buffers. The main difference is that the first entries of all Stream Buffers are checked for a hit. Scenario #1 (request hits in CACHE) and Scenario #3 (request misses in CACHE and hits in the first entry of one of the Stream Buffers) are unchanged. For Scenario #2 (request misses in CACHE and misses in the first entries of all Stream Buffers), the only difference is that one of the Stream Buffers must be chosen for the prefetch stream: select the least-recently-used Stream Buffer, i.e., apply the LRU policy to the Stream Buffers as a whole. When a new stream is

[1] For accurate performance accounting using the Average Access Time (AAT) expression, you will need to convey to the next level in the memory hierarchy that these read requests are prefetches. This will enable the next level in the memory hierarchy to distinguish between 1) its read misses that originated from normal read requests versus 2) its read misses that originated from prefetch read requests. Note that this is only needed for accurate performance accounting.

prefetched into a particular Stream Buffer (Scenario #2) or a particular Stream Buffer supplies a requested block to CACHE (Scenario #3), that Stream Buffer becomes the most-recently-used buffer.

## *4.5. Invalidating blocks in Stream Buffers*

A Stream Buffer never contains dirty blocks, that is, it never contains a block whose content differs from the same block in the next level of the memory hierarchy. The benefit of this design is that replacing the contents of the Stream Buffer will never require writebacks from the Stream Buffer.

This design does introduce a complication, however. Consider that a block Y may exist in CACHE as well as in a Stream Buffer. If there is a write request to block Y, block Y is updated in CACHE (making it dirty in CACHE) but is not updated in the Stream Buffer (since dirty blocks are not permitted in the Stream Buffer). This means the Stream Buffer contains a stale copy of block Y. This is not a problem as long as dirty block Y remains in CACHE, since any accesses to block Y will hit in CACHE. On the other hand, if dirty block Y is evicted from CACHE, then the stale copy of block Y in the Stream Buffer could cause incorrect operation in the future. The solution to this problem is as follows. When a dirty block Y is evicted from CACHE (i.e., when there is a writeback), any copies of block Y within the Stream Buffers are invalidated. This implies that a Stream Buffer has M valid bits (one for each of its consecutive memory blocks).

Invalidations form “gaps” in a Stream Buffer. Recall that a hit to the first entry of the Stream Buffer causes entries to shift up (Scenario #3). Therefore, it is possible for an invalid entry, i.e., a “gap”, to eventually percolate up to the first entry. This is not treated in any special way: just let it happen.

# 5. Memory Hierarchies to be Explored in this Project

While Fig. 1 illustrates an arbitrary memory hierarchy, you will only study the memory hierarchy configurations shown in Fig. 2a (ECE 463) and Fig. 2b (ECE 521). Also, these are the only configurations the TAs will test.

For this project, all CACHEs in the memory hierarchy will have the same BLOCKSIZE.

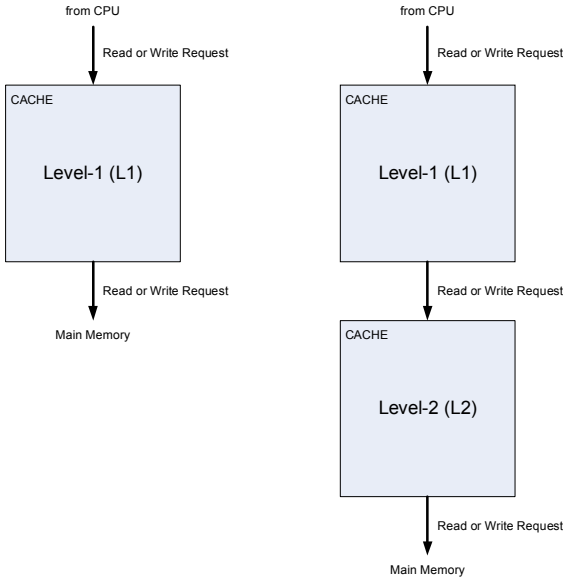


**Fig. 2a: ECE463: Configurations to be studied.**

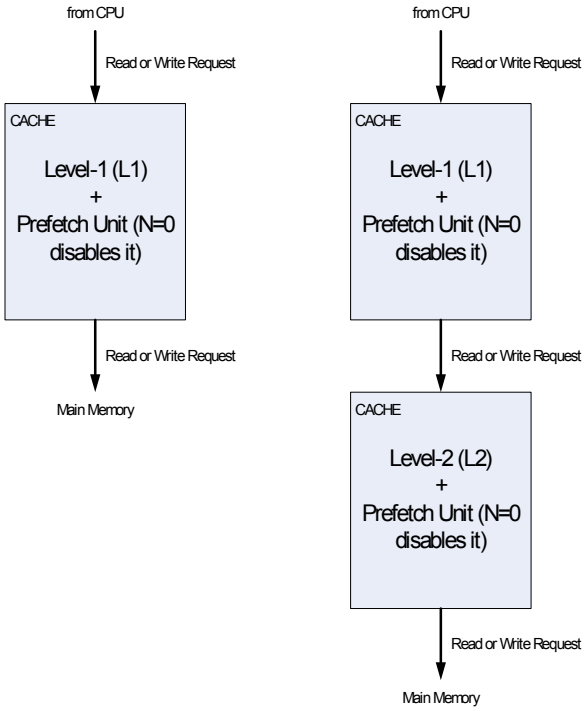


**Fig. 2b: ECE521: Configurations to be studied. Note that for each CACHE, its prefetch unit can be independently enabled or disabled.**

## 6. Inputs to Simulator

The simulator reads a trace file in the following format:

```
r|w <hex address>
r|w <hex address>
...
```

"r" (read) indicates a load and "w" (write) indicates a store from the processor.

Example:

```
r ffe04540
r ffe04544
w 0eff2340
r ffe04548
...
```

Traces are in the directory /afs/eos.ncsu.edu/courses/ece/ece521/lec/001/www/projects/proj1/traces. (It will be possible to download the traces directly from the web.)

**NOTE:**
All addresses are 32 bits. When expressed in hexadecimal format (hex), an address is 8 hex digits as shown in the example trace above. In the actual trace files, you may notice some addresses are comprised of fewer than 8 hex digits: this is because there are leading 0's which are not explicitly shown. For example, an address "`ffff`" is really "`0000ffff`", because all addresses are 32 bits, *i.e.*, 8 nibbles.

# 7. Outputs from Simulator

## *7.1. Raw measurements*

The simulator outputs the following "raw" measurements after completion of the run: (*note that "miss" means neither the cache nor its stream buffers hit*)

a. number of L1 reads
b. number of L1 read misses, *excluding L1 read misses that hit in the stream buffers if L1 prefetch unit is enabled*
c. number of L1 writes
d. number of L1 write misses, *excluding L1 write misses that hit in the stream buffers if L1 prefetch unit is enabled*
e. L1 miss rate = $\boxed{MR_{L1}}$ = (L1 read misses + L1 write misses)/(L1 reads + L1 writes)
f. number of writebacks from L1 to next level
g. number of L1 prefetches (*prefetch requests from L1 to next level, if prefetch unit is enabled*)
h. number of L2 reads that did not originate from L1 prefetches (*should match b+d: L1 read misses + L1 write misses*)
i. number of L2 read misses that did not originate from L1 prefetches, *excluding such L2 read misses that hit in the stream buffers if L2 prefetch unit is enabled*
j. number of L2 reads that originated from L1 prefetches (*should match g: L1 prefetches*)
k. number of L2 read misses that originated from L1 prefetches, *excluding such L2 read misses that hit in the stream buffers if L2 prefetch unit is enabled*
l. number of L2 writes (*should match f: number of writebacks from L1*)
m. number of L2 write misses, *excluding L2 write misses that hit in the stream buffers if L2 prefetch unit is enabled*
n. L2 miss rate (*from standpoint of stalling the CPU*) = $\boxed{MR_{L2}}$ = (item i)/(item h)
o. number of writebacks from L2 to memory
p. number of L2 prefetches (*prefetch requests from L2 to next level, if prefetch unit is enabled*)
q. total memory traffic = number of blocks transferred to/from memory
   (*with L2, should match i+k+m+o+p:*
   *all L2 read misses + L2 write misses + writebacks from L2 + L2 prefetches*)
   (*without L2, should match b+d+f+g:*
   *L1 read misses + L1 write misses + writebacks from L1 + L1 prefetches*)

## 7.2. Performance, Power, and Area

Report the following performance, power, and area metrics:
1. **Average access time (AAT)**: See Section 7.2.1 for how to calculate this metric.
2. **Energy-delay product (EDP)**: See Section 7.2.2 for how to calculate this metric.
3. **Total area of caches (Area)**: See Section 7.2.3 for how to calculate this metric.

For your reference, Table 1 gives names and descriptions of parameters that appear throughout this section.

**Table 1. Parameters, descriptions, and how you obtain these parameters.**

| *Parameter* | *Description* | *How to get parameter* |
|---|---|---|
| $MR_{L1}$ | L1 miss rate. | From your simulator. See Section 7.1. |
| $MR_{L2}$ | L2 miss rate (from standpoint of stalling the CPU). | |
| $HT_{L1}$ | Hit time of L1. | "Access time (ns)" from CACTI tool. The CACTI executable is available in the course locker. The Project-1 website explains how to call CACTI directly from your simulator. |
| $HT_{L2}$ | Hit time of L2. | |
| Miss_Penalty | Time to fetch one block from main memory. | From Project-1 website. |
| $E_{L1}$ | Energy consumed by a single access to L1. | "Total dynamic read energy per access (nJ)" from CACTI tool. The CACTI executable is available in the course locker. The Project-1 website explains how to call CACTI directly from your simulator. |
| $E_{L2}$ | Energy consumed by a single access to L2. | |
| $E_{mem}$ | Energy consumed by a single access to main memory. | From Project-1 website. |
| $A_{L1}$ | Die area of L1. | "Cache height x width (mm)" from CACTI tool. You must compute the die area: it is the product of cache height and width ($mm^2$). The CACTI executable is available in the course locker. The Project-1 website explains how to call CACTI directly from your simulator. |
| $A_{L2}$ | Die area of L2. | |

## 7.2.1. Average Access Time (AAT)

For memory hierarchy *without* L2 cache:

$$\boxed{\text{Total access time} = (\text{L1 reads} + \text{L1 writes}) \cdot HT_{L1} + (\text{L1 read misses} + \text{L1 write misses}) \cdot \text{Miss\_Penalty}}$$

$$\boxed{\text{Average access time (AAT)} = \frac{\text{Total access time}}{(\text{L1 reads} + \text{L1 writes})}}$$

$$\begin{aligned} AAT &= HT_{L1} + \left( \frac{\text{L1 read misses} + \text{L1 write misses}}{\text{L1 reads} + \text{L1 writes}} \right) \cdot \text{Miss\_Penalty} \\ &= HT_{L1} + MR_{L1} \cdot \text{Miss\_Penalty} \end{aligned}$$

For memory hierarchy *with* L2 cache:

$$\boxed{\text{Total access time} = (\text{L1 reads} + \text{L1 writes}) \cdot HT_{L1} + (\text{L1 read misses} + \text{L1 write misses}) \cdot HT_{L2} + (\text{L2 read misses not originating from L1 prefetches}) \cdot \text{Miss\_Penalty}}$$

$$\boxed{\text{Average access time (AAT)} = \frac{\text{Total access time}}{(\text{L1 reads} + \text{L1 writes})}}$$

$$\begin{aligned} AAT &= HT_{L1} + \left( \frac{\text{L1 read misses} + \text{L1 write misses}}{\text{L1 reads} + \text{L1 writes}} \right) \cdot HT_{L2} + \left( \frac{\text{L2 read misses not originating from L1 prefetches}}{\text{L1 reads} + \text{L1 writes}} \right) \cdot \text{Miss\_Penalty} \\ &= HT_{L1} + MR_{L1} \cdot HT_{L2} + \left( \frac{\text{L2 read misses not originating from L1 prefetches}}{\text{L1 reads} + \text{L1 writes}} \right) \cdot \text{Miss\_Penalty} \\ &= HT_{L1} + MR_{L1} \cdot \left( HT_{L2} + \left( \frac{1}{MR_{L1}} \right) \cdot \left( \frac{\text{L2 read misses not originating from L1 prefetches}}{\text{L1 reads} + \text{L1 writes}} \right) \cdot \text{Miss\_Penalty} \right) \\ &= HT_{L1} + MR_{L1} \cdot \left( HT_{L2} + \left( \frac{\text{L1 reads} + \text{L1 writes}}{\text{L1 read misses} + \text{L1 write misses}} \right) \cdot \left( \frac{\text{L2 read misses not originating from L1 prefetches}}{\text{L1 reads} + \text{L1 writes}} \right) \cdot \text{Miss\_Penalty} \right) \\ &= HT_{L1} + MR_{L1} \cdot \left( HT_{L2} + \left( \frac{\text{L2 read misses not originating from L1 prefetches}}{\text{L1 read misses} + \text{L1 write misses}} \right) \cdot \text{Miss\_Penalty} \right) \\ &= HT_{L1} + MR_{L1} \cdot \left( HT_{L2} + \left( \frac{\text{L2 read misses not originating from L1 prefetches}}{\text{L2 reads not originating from L1 prefetches}} \right) \cdot \text{Miss\_Penalty} \right) \\ &= HT_{L1} + MR_{L1} \cdot (HT_{L2} + MR_{L2} \cdot \text{Miss\_Penalty}) \end{aligned}$$

### 7.2.2. Energy-Delay Product (EDP)

In computer design, power consumption has become as important as performance. The *energy-delay product* is a combined metric that is sometimes used to account for both factors together: performance and power consumption. A combined metric is convenient because performance and power consumption are often at odds. Optimizing a combined metric leads to a balanced compromise between performance and power consumption.

Energy-delay product is energy multiplied by time (delay). Since, ideally, we want the lowest possible energy consumed as well as the lowest possible execution time, the goal is to find a design that minimizes the energy-delay product.

Below is a *very* rough energy model for your memory hierarchy.

For memory hierarchy *without* L2 cache:

$$\begin{aligned}&\text{Total energy} = \\ &(\text{L1 reads} + \text{L1 writes}) \cdot E_{L1} + (\text{L1 read misses} + \text{L1 write misses}) \cdot E_{L1} + \\ &(\text{L1 read misses} + \text{L1 write misses}) \cdot E_{mem} + (\text{L1 prefetches}) \cdot E_{mem} + (\text{writebacks from L1}) \cdot E_{mem}\end{aligned}$$

Explanation of above expression:

- The first term accounts for accessing the L1 cache for all reads and writes.
- The second term accounts for putting a new block in the L1 for all L1 read and write misses. (Unfortunately we are not accounting for transferring blocks from the stream buffer to the cache, since L1 miss + Stream Buffer Hit is not counted as a miss. That would require an additional raw measurement.)
- The third term accounts for fetching blocks from main memory.
- The fourth term accounts for prefetching blocks from main memory.
- The fifth term accounts for writing blocks to main memory.

For memory hierarchy *with* L2 cache:

$$\begin{aligned}&\text{Total energy} = \\ &(\text{L1 reads} + \text{L1 writes}) \cdot E_{L1} + (\text{L1 read misses} + \text{L1 write misses}) \cdot E_{L1} + \\ &(\text{all L2 reads} + \text{L2 writes}) \cdot E_{L2} + (\text{all L2 read misses} + \text{L2 write misses}) \cdot E_{L2} + \\ &(\text{all L2 read misses} + \text{L2 write misses}) \cdot E_{mem} + (\text{L2 prefetches}) \cdot E_{mem} + (\text{writebacks from L2}) \cdot E_{mem}\end{aligned}$$

Explanation of above expression:

- The first term accounts for accessing the L1 cache for all reads and writes.
- The second term accounts for putting a new block in the L1 for all L1 read and write misses.
- The third term accounts for accessing the L2 cache for all L2 reads and writes.
- The fourth term accounts for putting a new block in the L2 for all L2 read and write misses.
- The fifth term accounts for fetching blocks from main memory.
- The sixth term accounts for prefetching blocks from main memory.
- The seventh term accounts for writing blocks to main memory.

For the delay component, we will use the "Total access time" from Section 7.2.1, which is reproduced below.

For memory hierarchy *without* L2 cache:

$$\text{Total access time} = (\text{L1 reads} + \text{L1 writes}) \cdot HT_{L1} + (\text{L1 read misses} + \text{L1 write misses}) \cdot \text{Miss\_Penalty}$$

For memory hierarchy *with* L2 cache:

$$\text{Total access time} = (\text{L1 reads} + \text{L1 writes}) \cdot HT_{L1} + (\text{L1 read misses} + \text{L1 write misses}) \cdot HT_{L2} + (\text{L2 read misses not originating from L1 prefetches}) \cdot \text{Miss\_Penalty}$$

Finally, the expression for the Energy-Delay Product (EDP) is as follows. The unit of EDP is joules·seconds.

$$EDP = (\text{Total energy}) \cdot (\text{Total access time})$$

### 7.2.3. Total area of caches (Area)

For a given memory hierarchy configuration, provide the total area of the caches.

$$\text{Area} = A_{L1} + A_{L2}$$

If a particular cache does not exist in the memory hierarchy configuration, then its area is 0.

Note that it is difficult to estimate the area of the prefetch unit using CACTI due to the specialized structure of the Stream Buffers.

# 8. Validation and Other Requirements

## *8.1. Validation requirements*

Sample simulation outputs will be provided on the website. These are called "validation runs". You must run your simulator and debug it until it matches the validation runs.

Each validation run includes:
1. The memory hierarchy configuration.
2. The final contents of all caches in the memory hierarchy.
3. CACTI outputs for each cache.
4. All measurements described in Section 7 (raw measurements, AAT, EDP, Area).

Your simulator must print outputs to the console (i.e., to the screen). (Also see Section 8.2 about this requirement.)

Your output must match both numerically and in terms of formatting, because the TAs will literally "diff" your output with the correct output. You must confirm correctness of your simulator by following these two steps for each validation run:
1) Redirect the console output of your simulator to a temporary file. This can be achieved by placing `> your_output_file` after the simulator command.
2) Test whether or not your outputs match properly, by running this unix command:
`diff –iw <your_output_file> <posted_output_file>`
The –iw flags tell "diff" to treat upper-case and lower-case as equivalent and to ignore the amount of whitespace between words. Therefore, you do not need to worry about the exact number of spaces or tabs as long as there is some whitespace where the validation runs have whitespace.

## *8.2. Requirements for compiling and running simulator*

You will hand in source code and the TAs will compile and run your simulator. As such, you must meet the following strict requirements. Failure to meet these requirements will result in point deductions (see section "Grading").

1. You must be able to compile and run your simulator on Eos Linux machines. This is required so that the TAs can compile and run your simulator. If you are logging into an Eos machine remotely and do not know whether or not it is Linux (as opposed to SunOS), use the "uname" command to determine the operating system.
2. Along with your source code, you must provide a **Makefile** that automatically compiles the simulator. This Makefile must create a simulator named "sim_cache". The TAs should be able to type only "make" and the simulator will successfully compile. The TAs should be able to type only "make clean" to automatically remove object (.o) files and the simulator executable. An example Makefile will be posted on the web page, which you can copy and modify for your needs.
3. Your simulator must accept exactly 10 command-line arguments in the following order:

```
sim_cache     <BLOCKSIZE>
              <L1_SIZE>  <L1_ASSOC>  <L1_PREF_N>  <L1_PREF_M>
              <L2_SIZE>  <L2_ASSOC>  <L2_PREF_N>  <L2_PREF_M>
              <trace_file>
```

- *BLOCKSIZE*: Positive integer. Block size in bytes. (Same block size for all caches in the memory hierarchy.)
- *L1_SIZE*: Positive integer. L1 cache size in bytes.
- *L1_ASSOC*: Positive integer. L1 set-associativity (1 is direct-mapped).
- *L1_PREF_N*: Positive integer. Number of Stream Buffers in the L1 prefetch unit. *L1_PREF_N* = 0 disables the L1 prefetch unit.
- *L1_PREF_M*: Positive integer. Number of memory blocks in each Stream Buffer in the L1 prefetch unit.
- *L2_SIZE*: Positive integer. L2 cache size in bytes. *L2_SIZE* = 0 signifies that there is no L2 cache.
- *L2_ASSOC*: Positive integer. L2 set-associativity (1 is direct-mapped).
- *L2_PREF_N*: Positive integer. Number of Stream Buffers in the L2 prefetch unit. *L2_PREF_N* = 0 disables the L2 prefetch unit.
- *L2_PREF_M*: Positive integer. Number of memory blocks in each Stream Buffer in the L2 prefetch unit.
- *trace_file*: Character string. Full name of trace file including any extensions.

Example: 8KB 4-way set-associative L1 cache with 32B block size, L1 prefetch unit has 2 Stream Buffers with 4 blocks each, 256KB 8-way set-associative L2 cache with 32B block size, L2 prefetch unit has 3 stream buffers with 10 blocks each, gcc trace:

```
sim_cache  32  8192  4  2  4  262144  8  3  10  gcc_trace.txt
```

4. Your simulator must print outputs to the console (i.e., to the screen). This way, when a TA runs your simulator, he/she can simply redirect the output of your simulator to a filename of his/her choosing for validating the results.

## 8.3. Keeping backups

It is good practice to frequently make backups of all your project files, including source code, your report, etc. You can backup files to another hard drive (Eos account vs. home PC vs. laptop … keep consistent copies in multiple places) or removable media (Flash drive, etc.).

## 8.4. Run time of simulator

*Correctness* of your simulator is of paramount importance. That said, making your simulator *efficient* is also important for a couple of reasons.

First, the TAs need to test every student's simulator. Therefore, we are placing the constraint that your simulator must finish a single run in 2 minutes or less. This includes the time it takes to do a system call to CACTI from your simulator, to obtain CACTI-generated parameters. CACTI 6.0 may take up to a minute depending on the size and complexity of the cache configuration, and

your cache simulator should take on the order of seconds for the small traces we use. If your simulator takes longer than 2 minutes to finish a single run, please see the TAs as they may be able to help you speed up your simulator.

Second, you will be running many experiments: many memory hierarchy configurations and multiple traces. Therefore, you will benefit from implementing a simulator that is reasonably fast.

One simple thing you can do to make your simulator run faster is to compile it with a high optimization level. The example Makefile posted on the web page includes the –O3 optimization flag.

Note that, when you are debugging your simulator in a debugger (such as gdb), it is recommended that you compile without –O3 and with –g. Optimization includes register allocation. Often, register-allocated variables are not displayed properly in debuggers, which is why you want to disable optimization when using a debugger. The –g flag tells the compiler to include symbols (variable names, etc.) in the compiled binary. The debugger needs this information to recognize variable names, function names, line numbers in the source code, etc. When you are done debugging, recompile with –O3 and without –g, to get the most efficient simulator again.

## *8.5. Running simulator+CACTI in VCL environment*

We are finding that CACTI 6.0 requires a lot of virtual memory, so much so, that it sometimes runs out of virtual memory on *remote-linux.eos.ncsu.edu* and *grendel.ece.ncsu.edu* machines. To deal with this, you can reserve more powerful machines in the Virtual Computing Lab (VCL). The following website describes how to use VCL:

http://vcl.ncsu.edu/

From the website:
1) select “Reservation System > Make a Reservation”,
2) select “NCSU WRAP” and “Proceed to Login”,
3) login,
4) for the environment, select “Linux Lab Machine (Realm RHEnterprise 4)” or “Linux Lab Machine (Realm RHEnterprise 5)”,
5) you may choose to reserve a shell for up to 4 hours beginning now or later (with the possibility to extend reservation before it expires),
6) click “Create Reservation” button,
7) wait until the screen updates with a reservation and click “Connect!”,
8) you will be given an internet address that you can ssh to, using putty or other ssh clients, the same way that you remotely and securely login to other linux machines.

## 9. Experiments and Report

TO BE FILLED IN BY INSTRUCTOR IN NEXT VERSION OF SPEC.

# 10. What to Submit via Wolfware

You must hand in a single zip file called **project1.zip** that is no more than 1MB in size. Please respect the size limit on behalf of fellow students, as the Wolfware submission space is limited and exceeding your quota will cause problems come submission time. (Notify the TAs beforehand if you have special space requirements. However, a zip file of 1MB should be sufficient.)

Below is an example showing how to create **project1.zip** from an Eos Linux machine. Suppose you have a bunch of source code files (*.cc, *.h), the Makefile, and your project report (report.doc).

```
zip  project1  *.cc  *.h  Makefile  report.doc
```

**project1.zip** must contain the following (any deviation from the following requirements may delay grading your project and may result in point deductions, late penalties, etc.):

1. **Project report**. This must be a single MS Word document named **report.doc** *OR* a single PDF document named **report.pdf**. The report must include the following:
    - A cover page with the project title, the Honor Pledge, and your full name as electronic signature of the Honor Pledge. A sample cover page will be posted on the project website.
    - See Section 9 for the required content of the report.
2. **Source code**. You must include the commented source code for the simulator program itself. You may use any number of .cc/.h files, .c/.h files, etc.
3. **Makefile**. See Section 8.2, item #2, for strict requirements. If you fail to meet these requirements, it may delay grading your project and may result in point deductions.

## 11. Grading

TO BE FILLED IN BY INSTRUCTOR IN NEXT VERSION OF SPEC.