

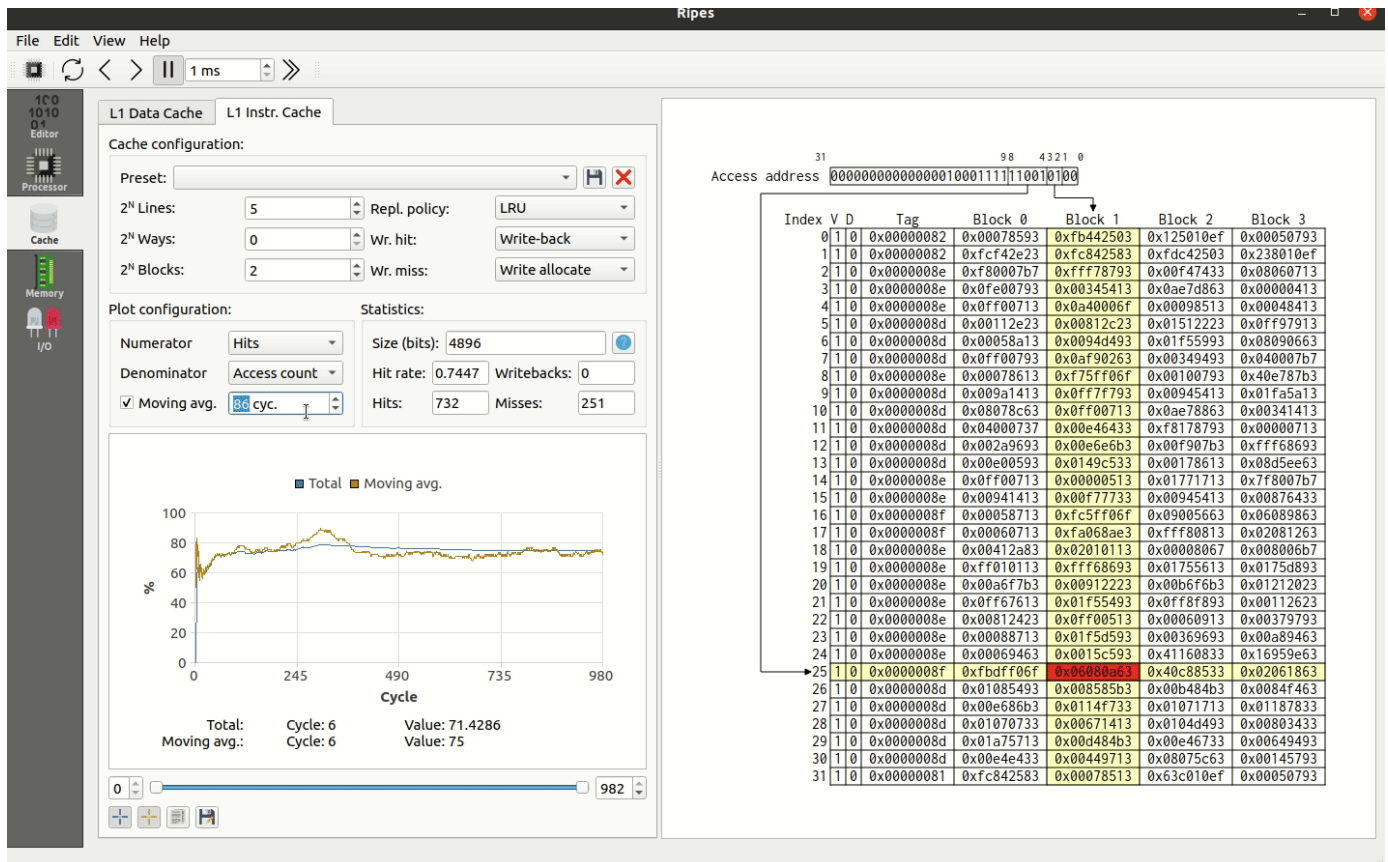
Cache Simulation

[Jump to bottom](#)

Morten Borup Petersen edited this page on 21 May · 8 revisions

Cache Simulation

As of version 2.1.0, Ripes includes cache simulation. The cache simulator simulates L1D (data) and L1I (instruction) caches, wherein it is possible to configure the layout and behavior of each cache type. Given this, we are able to analyse the cache performance of our programs to see how different cache designs interact with the memory access patterns which our programs exhibit.



Before getting started, here are some general notes on cache simulation in Ripes:

- It is **recommended** to use the single-cycle processor for cache simulation, given that:
 - If simulating with the pipelined processor models, it may happen that we are stalling a stage which is currently reading from memory. If a stage is stalled, each stalled cycle



will count as an additional memory access. This is implementation-specific behavior and as such *may or may not* be similar to other computing systems. Such behavior is avoided with the single-cycle model.

- The single-cycle processor model has a significantly faster execution rate as compared to the pipelined processor models.
- The processor models do **not** access the cache simulator when accessing memory. Instead, the cache simulator hooks into a processor model and analyses memory accesses within each clock cycle. Then, these memory accesses are used as a trace for performing our cache simulation. The implications of this are:
 - Ripes does not simulate cache access latency. As such, Ripes does not provide any estimations of actual CPU execution time, but can solely provide insight into factors such as miss, hit and writeback rates.
 - Dirty cache lines (when the cache is configured in write-back mode) will still be visible in the memory view. In other words, words are always written *through* to main memory, even if the cache is configured in write-back mode¹.

The Cache

Cache Configuration

Cache configuration:

Preset:	32-entry 4-word fully associative				
2 ^N Lines:	0	Repl. policy:	LRU		
2 ^N Ways:	5	Wr. hit:	Write-back		
2 ^N Blocks:	2	Wr. miss:	Write allocate		

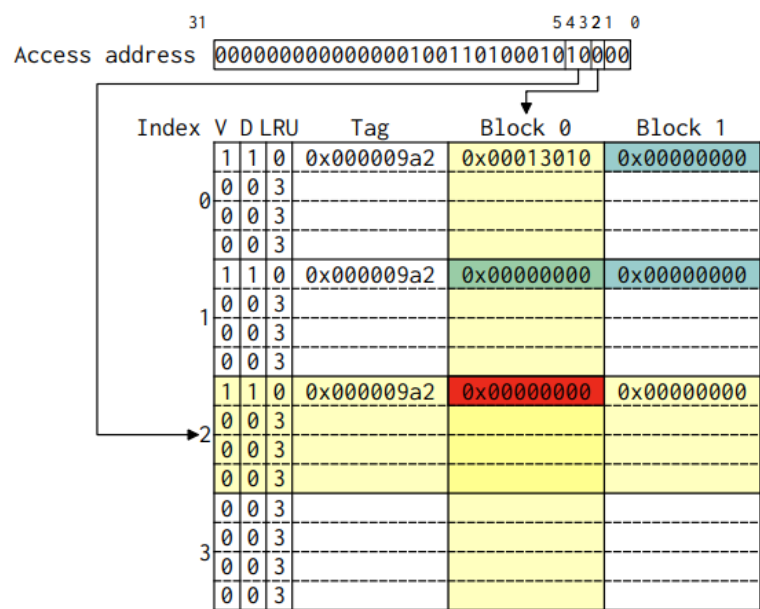
The cache is configurable through the following options:

- **Ways:** Associativity specification. Specified in a power of two (ie. a "two-way set associative cache" will have `ways=1` ($2^1 = 2$ ways) whereas a "direct mapped cache" will have `ways=0` ($2^0 = 1$ way)).
- **Lines:** Number of cache lines. The number of cache lines will define the size of the `index` used to index within the cache. Specified in a power of two.
- **Blocks:** Number of blocks within each cache line. The number of blocks will define the size of the `block index` used to select a block within a cache line. Specified in a power of two.
- **Wr. hit/Wr. miss:** Cache write policies. Please refer to [this Wikipedia article](#) for further info.
- **Repl. policy:** Cache replacement policies. Please refer to [this Wikipedia article](#) for further info.

Furthermore, a variety of presets are made available, and you are able to store your own presets for future reference..

The Cache View

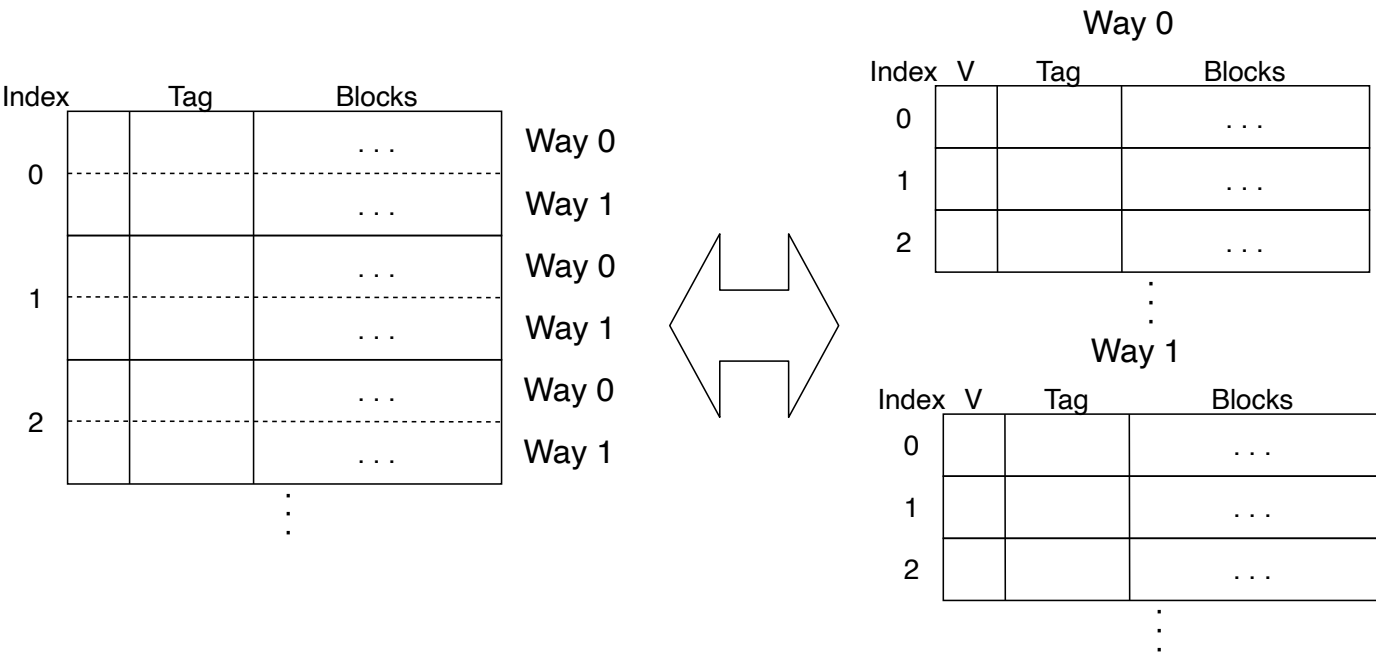
Based on the current cache configuration, a visualization of the current state of the cache is provided.



The cache is drawn as a table wherein rows are defined as:

- **Cache lines** are delimited with **solid** lines. The indices (index column) represents the index of each cache line.
- **Cache ways** are contained within a cache line. Ways which map within the same cache line are delimited with **dashed** lines.

Commonly, a set-associative cache will be drawn as separate tables for each way. This representation is equivalent with the representation used in Ripes, as follows:



Columns within the cache view are defined as:

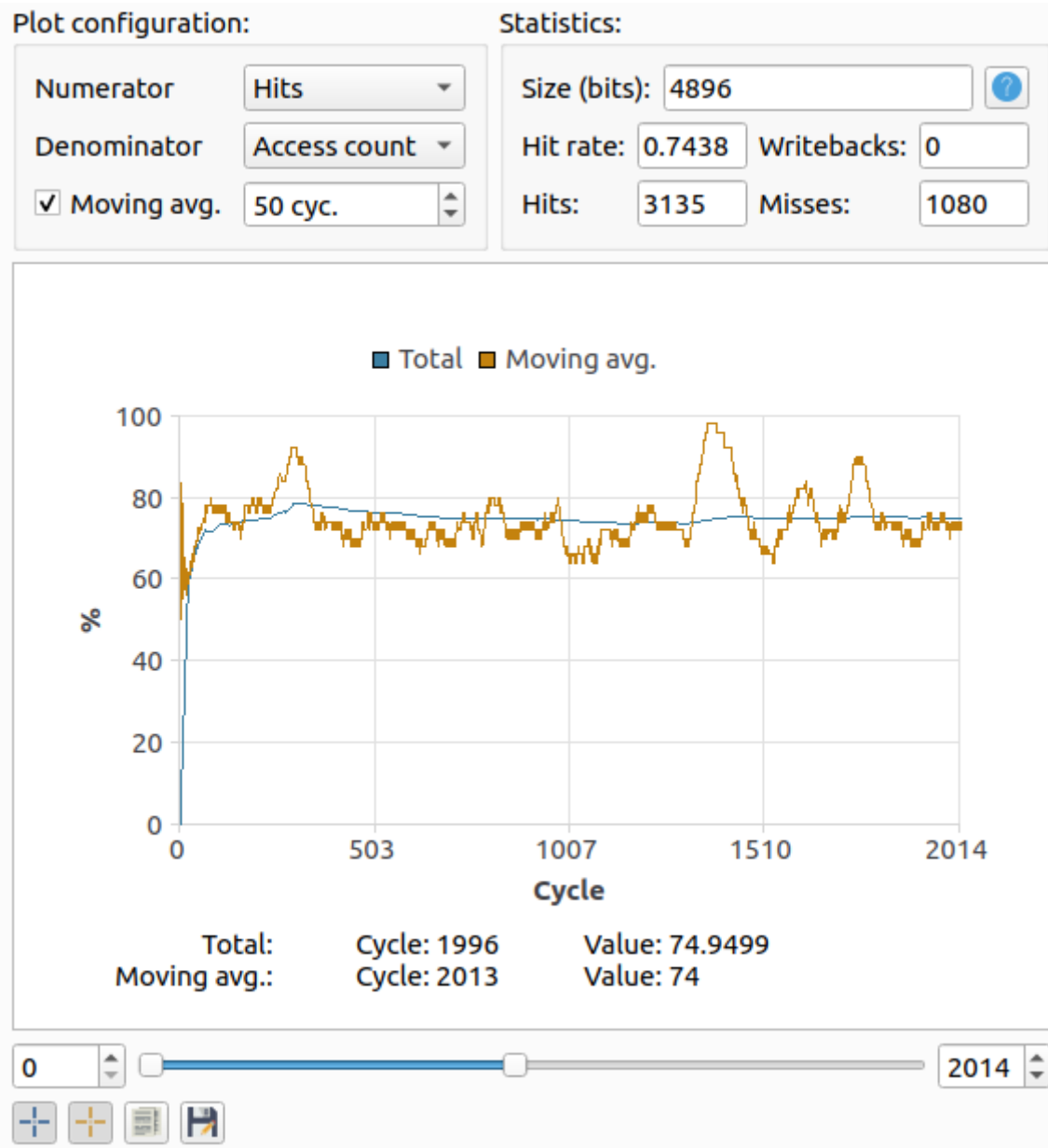
- **V:** Valid bit. Whether the cache way contains valid data.
- **D:** Dirty bit. Whether the cache way contains dirty data (the cache way was written to, in write-back mode).
- **LRU:** Visible when $\text{ways} > 0$ and $\text{Repl. policy} = \text{LRU}$. A value which was just accessed will have $\text{LRU} = 0$, and a value which is about to be evicted will have an LRU value of $\text{LRU} = 2^{(\text{ways})} - 1$.
- **Tag:** Current tag of the cached way.
- **Block #:** Cached data.

The cache view may be interacted with as follows:

- Hovering over a block will display the physical address of the cached value
- Clicking a block will move the memory view to the corresponding physical address of the cached value.
- The cache view may be zoomed by performing a `ctrl+scroll` operation (`cmd+scroll` on OSX).

When the cache is indexed, the corresponding *line* row and *block* column will be highlighted in yellow. The intersection of these corresponds to all the cells which may contain the cached value. Hence, for a direct mapped cache, only 1 cell will be in the intersection whereas for an N -way cache, N cells will be highlighted. In the 4-way set associative cache picture above, we see that 4 cells are highlighted. A cell being highlighted as green indicates a cache hit, whilst red indicates a cache miss. A cell being highlighted in blue indicates that the value is dirty (with write-hit policy "write-back").

Cache Access Statistics & Plotting






To provide insight to the cache simulation, various cache parameters may be plotted in real-time. Each cycle, the following cache access information is recorded:

- **Writes:** # of times the cache was accessed through a write
- **Reads:** # of times the cache was accessed through a read
- **Hits:** # of times a cache access (read or write) was a hit
- **Misses:** # of times a cache access (read or write) was a miss
- **Writebacks:** # of times a cache line was written back to memory
- **Access count:** # of times the cache was accessed. Effectively reads+writes or hits+misses

From this, it is possible to plot the ratio between any of these, by selecting the Numerator and Denominator. For instance, to see the cache hit rate over time, select Hits and Access count. A moving average plot of the selected variables can be enabled as well. This is useful when identifying at what points in the program that hit rate significantly changes.

To see a breakdown of the theoretical cache size (in bits), press the ? button.

At the bottom of the view the following actions are available::

- : Copy cache access information for all cycles to clipboard
- : Save plot to file
- : Enables plot markers for the total and moving average plots.

Example

The following example illustrates how different cache configurations may have an impact on the hit-rate of a cache. This is shown through the execution of an example program using different cache configurations.

Example Program

The example program allows us to specify some memory access pattern wherein adjusting this access pattern will have an impact on cache performance.

The example program is sketched out as a C program and compiled to RISC-V assembly using [Compiler Explorer](#) (for tips on how to convert compiler-explorer generated RISC-V assembly to assembly compatible with the Ripes assembler, refer to [this wiki page](#)).

```
unsigned stride = 4;
unsigned accessesPerTurn = 128;
unsigned turns = 2;
unsigned* baseAddress = (unsigned*)0x1000;

void cacheLoop() {
    for(unsigned i = 0; i < turns; i++) {
        volatile unsigned* address = baseAddress;
        for(unsigned j = 0; j < accessesPerTurn; j++) {
            *address;
            address += stride;
        }
    }
}
```

Which we convert into the following assembly program:

```
.data
stride:          .word    512 # in words
accessesPerTurn: .word    2
turns:           .word    128
baseAddress:     .word    4096

.text
cacheLoop():
    lw          a6, turns
```

```

        lw      a7, baseAddress
        lw      a2, stride
        lw      a0, accessesPerTurn
        mv      a3, zero
        slli    a4, a2, 2
        j       .LBB0_3
.LBB0_2:
        addi    a3, a3, 1
        beq     a3, a6, .LBB0_5
.LBB0_3:
        add     a5, zero, a7
        add     a2, zero, a0
        beqz    a0, .LBB0_2
.LBB0_4:
        lw      a1, 0(a5)
        addi    a2, a2, -1
        add     a5, a5, a4
        bnez    a2, .LBB0_4
        j       .LBB0_2
.LBB0_5:
        # exit

```

Simulating Different Cache Configurations

Initially, we specify the following variables in the source code:

```

.data
stride:          .word    512 # in words
accessesPerTurn: .word    2
turns:           .word    128
baseAddress:     .word    4096

```

Go to the Memory tab and, for the data cache, select the cache preset 32-entry 4-word direct mapped cache. Next, press the Run button to execute the program.

We see that the cache manages a hit rate of 0.01154. In the example program, we see that we have specified a stride of 512 words. This results in the following access pattern, accessing two different memory locations:

```

1: 4096 = 0b00010000 00000000
2: 4608 = 0b00010010 00000000
3: 4096 = 0b00010000 00000000
4: 4608 = 0b00010010 00000000
...

```

For the chosen cache configuration we see that the line index of the cache is the following bitmask:

```
0b00000001 11110000
```


Applying the bitmask to the access pattern listed above, we see that all access addresses mask to $0x0$ and thus will index to the same cache line. In other words, we have no diversity with respect to the indexing in the cache for the given access pattern.

In this case, a set-associative cache could be more suitable than a direct-mapped cache.

Select the cache preset 32-entry 4-word 2-way set associative . Note that this cache design provides the *same* number of possibly cached blocks as the previous direct-mapped design. Next, rerun the program.

In this case, we see that the cache achieves a hit rate of 0.9885 . We no longer experience associativity conflicts, since each of the two accesses, whilst mapping to the same cache index, will be placed in separate ways.

1: This would be an obvious issue if Ripes was to simulate a multiprocessor system. However, given that this is not the case, and that cache latency is not simulated, this will not have any effect on cache access statistics nor execution semantics.

▼ Pages 10
Find a Page...
▶ Home
▶ Adapting Compiler Explorer generated RISC V assembly code
▶ Adding new processor models
▶ Building and Executing C programs with Ripes
▼ Cache Simulation
Cache Simulation
The Cache
Cache Configuration
The Cache View
Cache Access Statistics & Plotting
Example
Example Program
Simulating Different Cache Configurations
▶ Environment calls
▶ Memory mapped IO
▶ Release notes
▶ Ripes Introduction

- ▾ [Ripes introduction](#)
- [RISC V Assembly Programmer's Manual \(Adapted for Ripes\)](#)

Clone this wiki locally

https://github.com/mortbopet/Ripes.wiki.git

