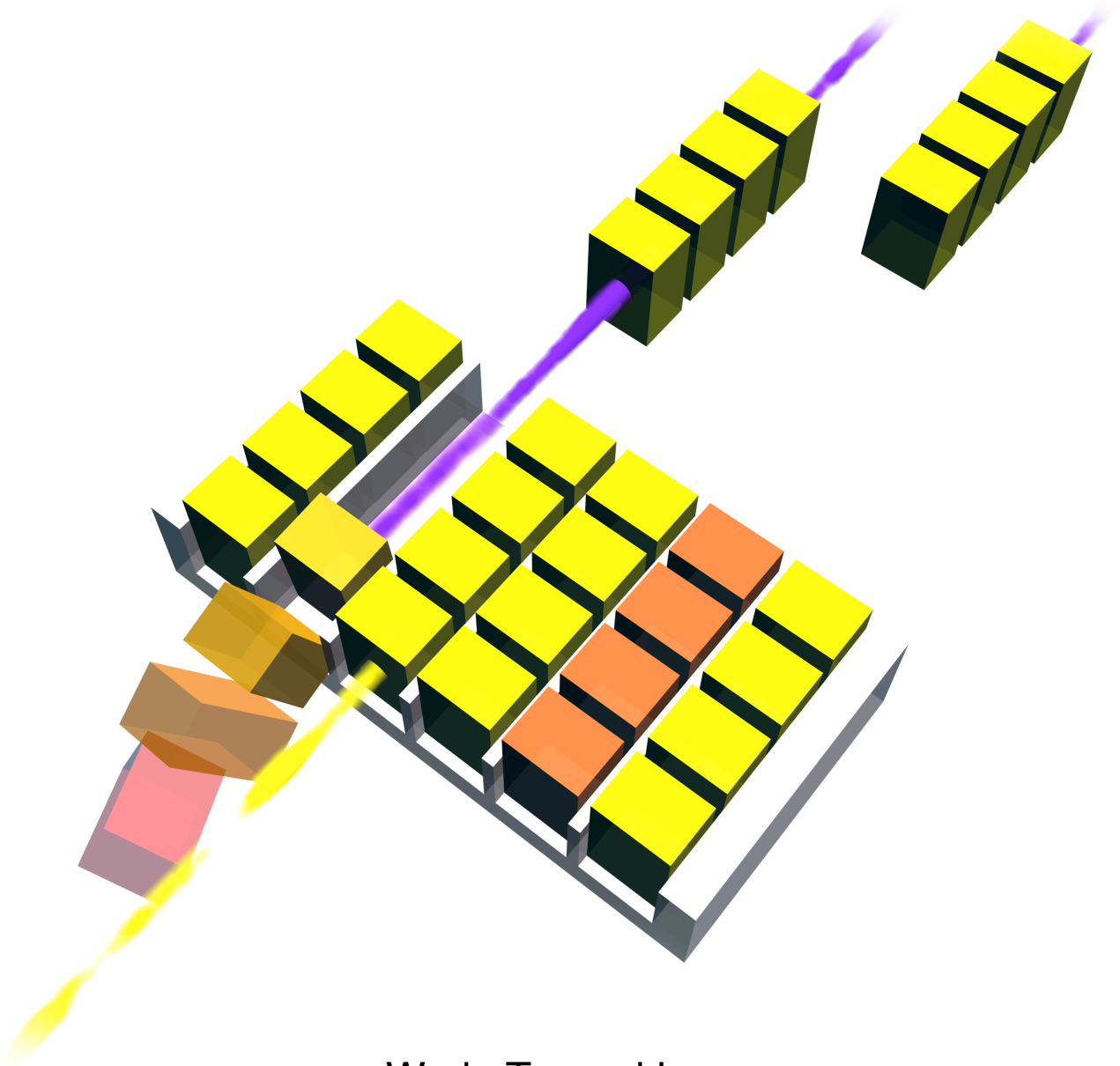
CSE32ARC

SimpleCacheSim



Wade Tregaskis

02557793

Monday, 10 October 2005

Contents

Introduction

Overview

SimpleCacheSim is a small program for simulating the operation of memory caches. It collects statistics about the general operation of the cache under simulation - such as how many requests to the cache hit, how many missed, etc. It breaks the statistics down by operation type and exact miss type, producing a thorough 6x8 matrix of results.

It is capable of simulation several replacement policies, and caches of any size or reasonable configuration. It allows specification of block (a.k.a. line) sizes and the system word size, which then allows it to detect misaligned requests - and optionally handle them, invoking multiple cache accesses from a single request.

It also features a batch-orientated interface and mode of operation, allowing a range of parameters to be specified. The program then simulates every valid permutation of these parameters, outputting the results en-mass. It has been used for batches involving over 14,000 simulations at a time. It's output can be configured to be in numerous forms, including tabular and SQL-like, allowing results to be imported into a database (e.g. PostgreSQL) with ease.

What it does not do is simulate multiple levels of cache, nor non-unified caches. For such simulation you may wish to try the free dineroIV cache simulator.

Additionally, it does not perform any timing simulation. As such, all accesses are sequential. In reality many modern hardware memory caches allow multiple concurrent requests, and many also support pipelining. There are no immediate plans, at this point in time, to add such functionality.

Usage

The program has a typical *nix command line interface. It is invoked from a terminal with various parameters defining the simulation(s) to perform. It requires a list of cache requests to simulate, which can be provided via standard input or from an input file specified as a program parameter. The program is invoked as such:

```
SimpleCacheSim [OPTIONS] [INPUT]
```

All parameters are optional - without any other direction, the program assumes a generic cache configuration and awaits input. If a file is named in the parameters, that file is opened and read from instead of standard input. This has the same effect as piping the contents of the file to the program.

The options that can be specified to the program are provided on the following pages.

Program Parameters

--addressbase/-A	The numerical base of addresses provided as input. Useful if the addresses are hexadecimal but without a preceding "0x", which would otherwise confuse the input parser. Valid values are 2 through 36 inclusive, or the special value 0, which allows any base by trying to detect the base in use (e.g. any input starting with "0x" is hexadecimal, any starting with "0" is assumed to be octal, and if all else fails the input is assumed to be decimal. Defaults to 0.
--misalignment/-M	The types of misalignment to permit, if any. Valid values are "none", "word", "block" (which implies allowing word misalignment as well) or "forced" (where all addresses are forced to align). Defaults to None.
--SQL/-S	SQL functionality. Takes a string as an argument, which may be either "create" or "insert". The "create" directive instructs the program to output the SQL CREATE statements that generate a table and summary views suitable for holding the tabular output from this program. These statements can be executed in a database client - e.g. psql for PostgreSQL. The "insert" directive instructs the program to output all tabular results as SQL INSERT statements. The output can then be executed directly in a database client to import data into the table(s) generated by the "create" directive. Note: if your database supports a COPY command, it is usually much faster to use this on the normal tabular output than to use INSERT statements. The "insert" directive is only meaningful if a negative verbosity is specified.
--associativity/-a	The associativity of the cache, which may be either "direct", "full" or a numeric value indicating the number of ways mapped. Multiple associativities may be separated by commas, and/or provided as ranges distinguished by hyphens. Defaults to Full.
--blocksize/-b	The size (in bytes) of each cache block. This is the smallest unit that the cache will be capable of holding (not to be confused with the smallest unit which can be transferred, which is the word size). Must be an integer multiple of the word size. Multiple block sizes may be separated by commas, and/or provided as ranges distinguished by hyphens. Defaults to 0.
--cachesize/-c	The size (in bytes) of the cache. Must be an integer multiple of block size and associativity. Multiple cache sizes may be separated by commas, and/or provided as ranges distinguished by hyphens. Defaults to 16384.
--help/-h	Prints this usage information and exits.
--memorysize/-m	The size (in bytes) of the memory behind the cache. Optional, used only to validate input values. A value of 0 indicates no defined limit. Note that only one value is permitted, unlike other parameters, as this is purely for input validation. Defaults to 0.

--replacement/-r	The replacement policy in N-way associative caches (N > 0). May be any of "random", "LRU" (Least Recently Used), "FIFO" (First In First Out), "LRR" (Least Recently Read) or "LRW" (Least Recently Written). Multiple policies may be separated by commas. Defaults to LRU.
--verbosity/-v	Sets the verbosity of the output, where 0 corresponds to silent operation, 1 prints out the simulation parameters and other key information, 2 also prints out information about every cache operation as it is simulated, and 3 also prints out even more details about what's happening with each and every cache access. A negative verbosity indicates tabular output; -1 outputs only the raw data, while -2 or lower also outputs a header on the first row, indicating what value each column represents. Use high verbosities with caution - the volume of output can be tremendous. Defaults to 0.
--wordsize/-w	The size (in bytes) of one word, which is the smallest unit of data that can be transferred to or from cache. Multiple word sizes may be separated by commas, and/or provided as ranges distinguished by hyphens. Defaults to 4.

Output Formats

The final output of the program has three forms. The first is “standard” output, which is a form designed to be succinct and human-readable. This is the form used for a verbosity setting of 0 or above. An example is shown below. If a non-essential value is zero, it is omitted from the output for brevity.

```
Hits: 91922
        Read: 12998
        Write: 11213
        Instruction: 67711
Capacity Misses: 0
Compulsory Misses: 12
        Read: 3
        Write: 2
        Instruction: 7
Conflict Misses: 8066
        Read: 2690
        Write: 1343
        Instruction: 4033
Misalignments straddling blocks: 0
Misalignments within block: 0

Total misses: 8078
Total cache accesses: 100000

Miss rate: 0.080780
        Compulsory: 0.000120
        Capacity: 0.000000
        Conflict: 0.080660
```

Another form is for a verbosity setting of -1 or lower, without specifying the SQL INSERT format (see the --SQL/-S parameter in the parameter list, previous). This is standard tabular output, which is well suited for import into a database or spreadsheet for further processing. It is also explicit in all values - that is, it outputs all values regardless of whether they are zero or even directly relevant to the exact simulation performed. This consistency is essential for ease of use in automated systems. An example of tabular output is shown below.

```
100000  Direct  LRU     64      512     1       12998   11213   67711   0       0
  0       0       0       0       0       0     00        0       0       0       3
    2       7       0       0       0       0       0       2690    1343    4033    0
      0     00        0       0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0     0
```

If the verbosity is set to -2 or lower, a heading is also printed on the first line, identifying the meaning of each column.

The last format is using the SQL INSERT functionality, by providing a negative verbosity setting as well as invoking the --SQL/-S parameter with an argument of “insert”. The output in this case looks like the example shown below.

```
INSERT INTO SimpleCacheSim VALUES (100000, 'Direct', 'LRU', 64, 512, 1, 12998, 11213,
67711, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 2, 7, 0, 0, 0, 0, 0, 2690, 1343,
4033, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
```

In addition to these final outputs, a high positive verbosity setting also causes information to be output as the simulation is run - such as what each access is, notification of block movement in and out of the cache, etc. The higher verbosity settings (2 and above) are not recommended for normal use - their primary purpose is as debugging aids or teaching tools for very small simulations.

Additional Notes

Program performance is generally very good - for reference, on a 1.5 GHz Powerbook a simulation with 100,000 cache requests takes [on average] approximately 100ms. Note however that some simulations will take substantially longer than others - the worst offenders in this regard are simulations of fully associative cache, particularly large such caches.

Additionally, larger block and word sizes typically improve performance (as larger block sizes reduce the total number of blocks in the simulation).

Design

See Appendix III for the source code.

The program is written in C++, but favours a procedural approach. There is one core function - performSimulation() - which does the body of the work. main() parses input parameters and drives the simulation(s), and various other utility functions aid it in that endeavour. Execution flow is fairly trivial, and what isn’t intuitive is covered in the code by appropriate commenting and inline documentation.

In terms of data structures and flow, again, the design is fairly simple. There are three primary structures - CacheAccess, CacheEntry and CacheSet. The first simply stores a single cache request, which involves recording the address, the access type and any other miscellaneous information necessary (such as which line the access came from in the original input). A single vector of such CacheAccess's are created in main() by reading from the appropriate input stream, and this vector is passed to performSimulation(). It is reused across all calls to performSimulation() for performance reasons.

The CacheEntry structure represents a single block that is or has been in cache. It records the base address of the block, temporal information regarding when it was last loaded/used/written/read/etc, flags indicating whether it is dirty, whether it was prefetched, etc, and - importantly - a field containing the eviction reason for the last time it was thrown out of cache. The reason for this field's importance will be detailed shortly.

The CacheSet structure is trivial - it contains two vectors, one of which stores all the CacheEntry's currently "in" the cache, while the other stores every CacheEntry that is or ever has been in the cache. The latter vector is used, obviously, primarily for determining why a block is not already in cache (i.e. if it does not exist in the vector, it is a compulsory miss, otherwise the reason it is not in cache can be read from the eviction reason field).

The CacheSet structure also contains an 'isFull' flag, indicating whether the cache has reached capacity. For performance reasons the vector of entries "in" cache is fixed at the number of possible entries, with empty spots being represented by NULL pointers. Thus size() or similar methods of the vector are not useful for indicating it's state. The 'isFull' flag is used as a common-case optimisation, as the capacity status of the cache is checked with every new block loaded into the set.

Finally, performSimulation() has a local variable - an array of CacheSet's, representing the cache as a whole. A vanilla C array was used for simplicity and performance - the array is static for the life of the performSimulation() function, and need only be indexed; never resized or otherwise modified.

The following two subsections contrast the good and bad parts of this design, both in an absolute sense and also with an eye towards dineroIV, a "competing" cache simulator.

Design Wins

The design is simple and elegant. The regular approach and use of data - regardless of replacement algorithms or any other parameters - makes the implementation single-minded and straightforward. The use of vectors instead of linked lists, especially given adding elements is rare (and removals never) is a big performance win (in contrast to dineroIV, primarily).

In terms of features, there are some minor ones worth mentioning, as they are relatively rare in cache simulators. The first is the flexibility of input parameters. For example, it is perfectly valid to use a word size of 3, provided all other values (such as block size, cache size, etc) are multiples of 3. No other simulator known to the author supports this, and while it may seem gimmicky, it is testimony to the elegant and un-assuming design.

Additionally, several replacement policies are available that are usually overlooked, because they are not used in the real world. These are LRR (Least-Recently-Read) and LRW (Least-Recently-Written). While not representative of any real world caches, they are useful for academic purposes.

Arguably the biggest win, however, is the method for distinguishing between capacity and conflict misses. dineroIV and many other simulators have a serious design flaw in this regard - they distinguish between the two based on the state of the cache at the time of a miss. Unfortunately, the cache will likely have changed by this time, perhaps significantly. For example, at the time the block was actually evicted from cache, it may have been because of a conflict, even though the cache was not full. If the cache does become full before a miss occurs on that block, dineroIV and others will register the miss as a capacity miss, not a conflict miss. This is seen clearly in some of the comparative results in Appendix I, where dineroIV and SimpleCacheSim both recognise exactly the same total misses, and compulsory misses, but dineroIV overstates the portion of those which are capacity misses.

SimpleCacheSim stores the eviction reason in the CacheEntry as it is removed from cache, as previously noted. Thus, the eviction reason is based on the state of the cache at the actual time of the eviction, and consequently is always accurate.

Design Losses

One big area of concern is performance. While SimpleCacheSim performs admirably in most cases, for some borderline scenarios it's performance rapidly deteriorates. For example, high associativies are expensive, as are access patterns which hit a very large number of unique addresses only a few times each, causing a large number of CacheEntry's to be produced and stored, increasing the search time for each subsequent cache miss.

Another problem is scalability of the design going forward, when looking to introduce new functionality. A more object-orientated implementation would be more flexible in this regard. For example, the current implementation has no support for multiple caches, and cannot be easily configured as such.

Other small flaws include the inability for the program to issue more than one additional cache operation per access. This relates to how the program handles misalignment, or more generally multi-block references. Ideally the block size could be completely independent of word size, and the cache could generate as many block accesses as necessary for any given request. The current implementation does not support this, although to add such functionality would be relatively trivial.

In terms of implementation details, there are several instances where expensive iteration is used unnecessarily. For example, to check if the cache is full requires iteration through the entire array of CacheSets, checking their 'isFull' flag. A much better way would be to count how many are full, which requires only a single integer comparison for the same effect.

Another notable set of "problems" (really just possible avenues for future work) is the missing features that would enrich the program. Multiple levels of cache is one clear such suggestion, as well as support for non-unified caches. Furthermore, support for multiport access, pipelining and prefetching would be desirable. Also of use might be statistics regarding the bandwidth used to and from the cache to service all the requests (as dineroIV provides, albeit in a limited capacity).

And beyond that, on the completely wishful-thinking list, is pervasive support for timing. This includes calculating statistical data such as cycles spent waiting on misses, average request latency, etc. Such functionality is not trivial to add, as the timing of a real world cache is far from a simple "1 cycle hits, 50 cycle misses". Indeed, a much more modular implementation would be required, as the timing simulation itself could easily swamp the core cache simulation.

Sample Simulation

To demonstrate the use of the simulator, the following section documents the output from a simple test set of data. The sample access pattern contains 100,000 requests, which are a mixture of instruction reads, data reads and data writes. There are no advanced requests (such as cache line invalidates/flushes, or prefetches). Due to it's size, it is not included in the printed form of this document - it should however be bundled with the electronic copy.

To be more specific, the sample access pattern used contains 7,261 unique addresses, which with 64-byte blocks (as used extensively in this document) map to 129 unique blocks. There are 15,691 data read requests, 12,558 data write requests, and 71,751 instruction read requests.

Method

To gather data for this analysis, the program was run with the command:

```
./SimpleCacheSim -A 16 -M block -a 0-16 -b 1,2,4,8,16,32,64,128,256,512,1024 -c
1,2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384,32768,65536,131072 -r ran-
dom,LRU,FIFO,LRR,LRW -v -1 -w 1-16 ../../smalltex.din > results
```

That is - all associativities from direct to 16-way and then full, as well as all the first 10 powers of two for block sizes, cache sizes in powers of two up to 128kiB, every replacement policy available and word sizes from 1 to 16 bytes.

This set of parameters generates 14,100 simulations, which are output to the "results" file. This was then copied into a PostgreSQL database (as defined using the --SQL/-S create parameter) so that it could be easily manipulated.

Tiny subsets of the results were selectively copied into Excel, in order to generate the simplified tables and diagrams shown throughout this document. Due to the functional limitations of Excel, the charts are not optimal, but do suffice for the most part.

Data

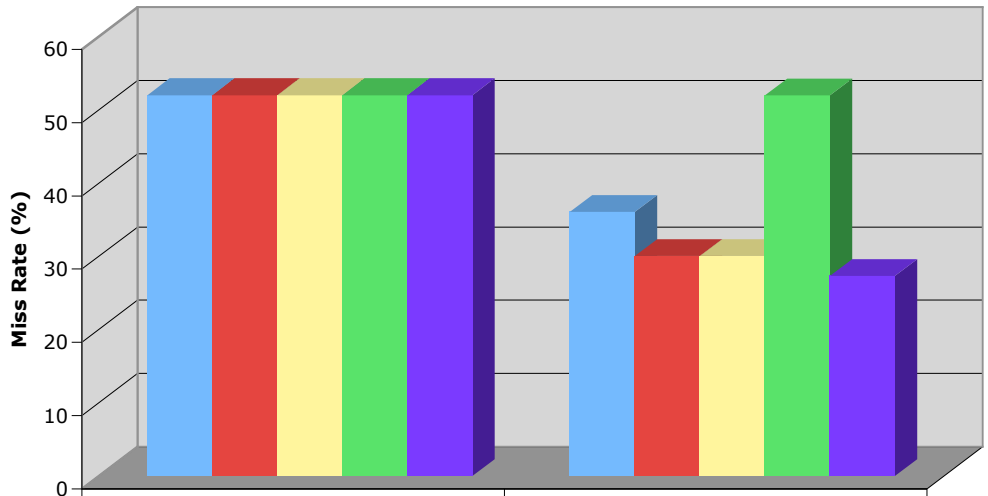
Appendix I contains the raw data used in this section, in tabular form. Italic entries indicate results from dineroIV. Entries coloured light yellow indicate discrepancies as a result of the randomness inherent in random replacement. Entries coloured blue indicate discrepancies as a result of errors in dineroIV.

Performance as related to cache size

This section looks at the performance of caches of various sizes, using a fixed block size of 64 bytes (and a word size of one byte). The miss rates are shown in the following six charts. Keep in mind that the scales are not consistent between the charts, as the range of values varies significantly between them.

Total Miss Rates

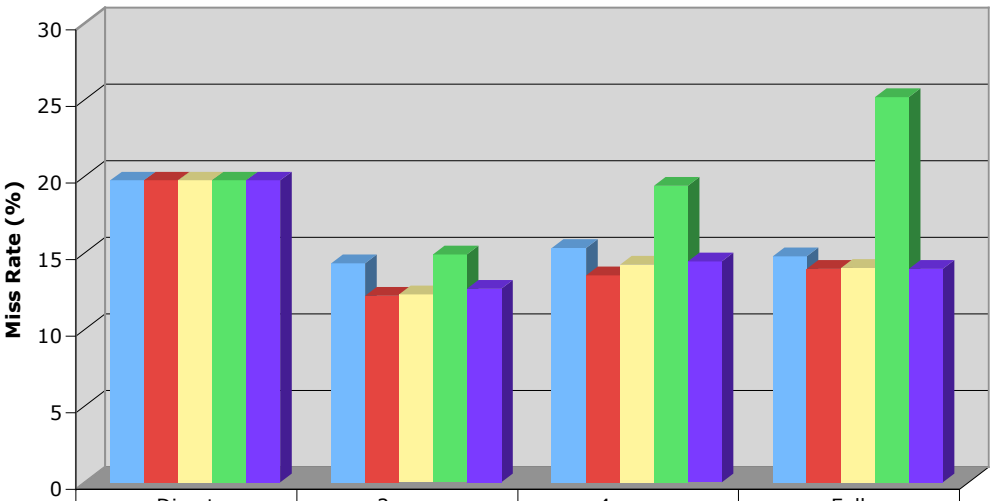
Miss Rate over Associativity and Replacement Algorithm (64-byte blocks, 128-byte cache, 1-byte words)



	Direct	Full
Random	52.024	36.09
LRU	52.024	30.048
LRR	52.024	30.048
LRW	52.024	52.012
FIFO	52.024	27.358

Associativity

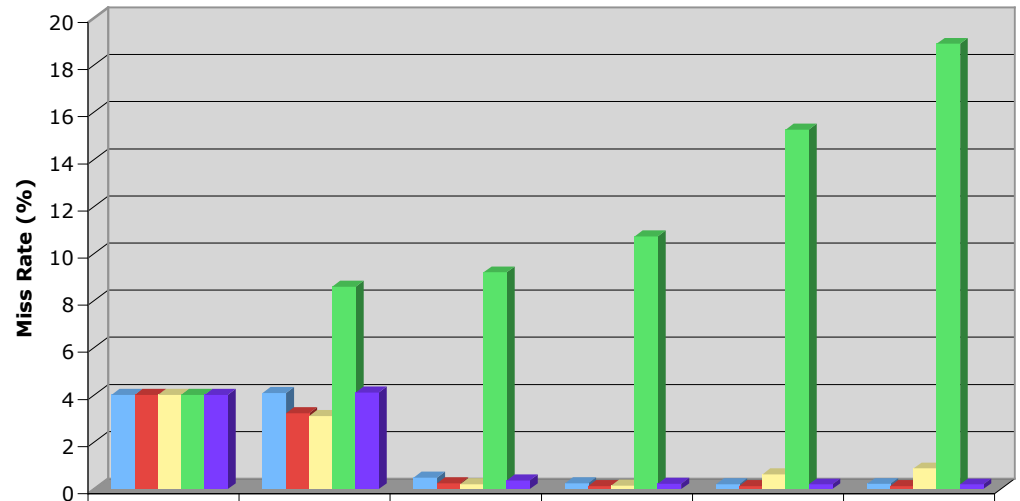
Miss Rate over Associativity and Replacement Algorithm (64-byte blocks, 512-byte cache, 1-byte words)



	Direct	2-way	4-way	Full
Random	19.73	14.316	15.316	14.777
LRU	19.73	12.21	13.54	13.96
LRR	19.73	12.301	14.226	14.016
LRW	19.73	14.899	19.37	25.159
FIFO	19.73	12.637	14.436	13.96

Associativity

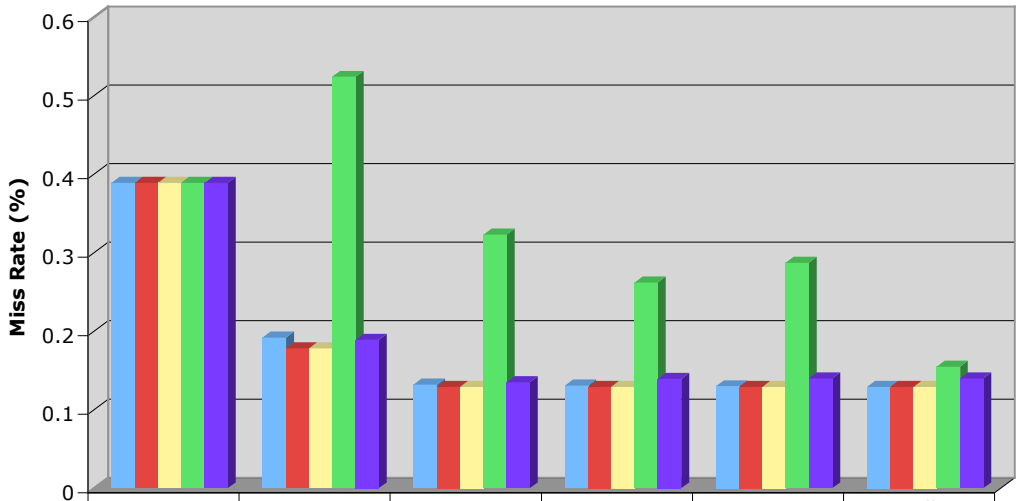
Miss Rate over Associativity and Replacement Algorithm (64-byte blocks, 2048-byte cache, 1-byte words)



	Direct	2-way	4-way	8-way	16-way	Full
Random	4.009	4.083	0.482	0.255	0.216	0.218
LRU	4.009	3.219	0.245	0.129	0.129	0.129
LRR	4.009	3.103	0.215	0.154	0.627	0.879
LRW	4.009	8.588	9.202	10.73	15.26	18.902
FIFO	4.009	4.109	0.371	0.227	0.199	0.199

Associativity

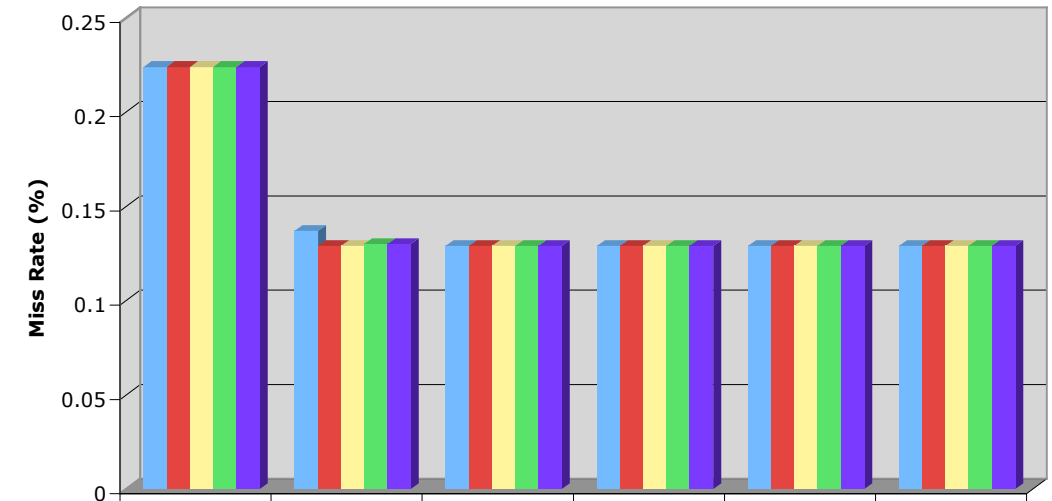
Miss Rate over Associativity and Replacement Algorithm (64-byte blocks, 8192-byte cache, 1-byte words)



	Direct	2-way	4-way	8-way	16-way	Full
Random	0.389	0.192	0.132	0.131	0.13	0.129
LRU	0.389	0.178	0.129	0.129	0.129	0.129
LRR	0.389	0.178	0.129	0.129	0.129	0.129
LRW	0.389	0.524	0.323	0.262	0.287	0.155
FIFO	0.389	0.189	0.135	0.139	0.14	0.14

Associativity

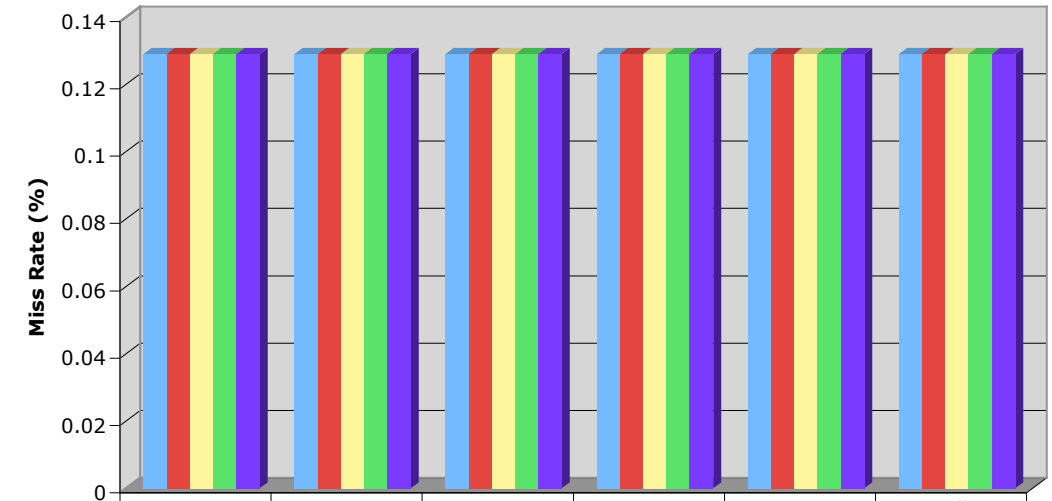
Miss Rate over Associativity and Replacement Algorithm (64-byte blocks, 16384-byte cache, 1-byte words)



	Direct	2-way	4-way	8-way	16-way	Full
Random	0.224	0.137	0.129	0.129	0.129	0.129
LRU	0.224	0.129	0.129	0.129	0.129	0.129
LRR	0.224	0.129	0.129	0.129	0.129	0.129
LRW	0.224	0.13	0.129	0.129	0.129	0.129
FIFO	0.224	0.13	0.129	0.129	0.129	0.129

Associativity

Miss Rate over Associativity and Replacement Algorithm (64-byte blocks, 32768-byte cache, 1-byte words)



	Direct	2-way	4-way	8-way	16-way	Full
Random	0.129	0.129	0.129	0.129	0.129	0.129
LRU	0.129	0.129	0.129	0.129	0.129	0.129
LRR	0.129	0.129	0.129	0.129	0.129	0.129
LRW	0.129	0.129	0.129	0.129	0.129	0.129
FIFO	0.129	0.129	0.129	0.129	0.129	0.129

Associativity

Note that at and above 32kiB (shown in the last chart, above) the results do not change at all - the sample access pattern is too small to have any further effect. All misses at this point are compulsory, with no evictions from cache ever occurring. Obviously a “serious” real program would perform several orders of magnitude more requests than we have simulated, which may require a far larger cache to reach optimal efficiency.

Discussion

There are several obvious conclusions to draw form these charts. The first is that, as noted previously, our sample set of requests requires only 32kiB of cache to achieve maximum efficiency. This obvious proof is intuitive, as with sufficient size the cache no longer acts as cache, but rather just a flat memory.

Another note is that even with a tiny cache - fitting just two blocks - there is still a marked improvement over no cache, with only slightly more than 50% of requests missing. More optimistically, this means nearly 50% of requests hit. Given the minimal latency a real cache would introduce, relative to the cost of going to the next level of memory, a 50% hit rate delivers fantastic performance gains.

Of interest is the varying performance of more esoteric replacement policies - particularly LRW (Least-Recently-Written). This policy performed terribly with small caches, particularly the 2kiB cache, where it’s miss rate was an order of magnitude greater than every other policy. However, for larger caches the performance improved again - although in the tests above never equalling the other policies. Clearly LRW is not a good policy for the sample access pattern used - intuitively this is true, as memory is more often read than written, and programs do not generally overwrite the same memory frequently, or in quick succession.

Overall the best policy was clearly LRU (Least-Recently-Used), which in nearly every case at least equalled the other policies. In only a handful of cases was it slightly outperformed by LRR (Least-Recently-Read), but these are so infrequent and the differences so small as to be insubstantial. But LRR was still a reasonable policy, and for some data sets could hypothetically be significantly better than LRU - e.g. when iteratively reading a small set of addresses, and writing a large set of varying addresses. In this case LRU might tend to throw out from the small set of read addresses, whereas LRR would favour keeping them, flushing instead the writes - which in this scenario would be unlikely to hit again frequently, if at all.

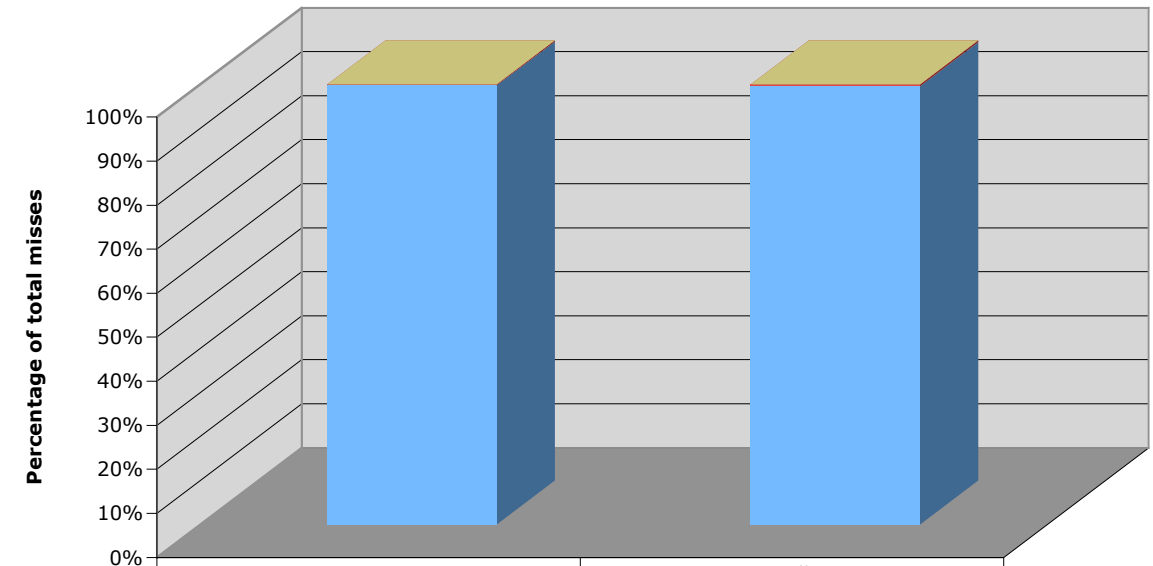
In terms of associativity, it is clear that higher associativities are general superior. In nearly every case direct mapping resulted in twice as many (or more) misses as associative configurations. It is interesting to note, however, that in some cases 2-way associativity was superior to higher associativies - such as with the 512-byte cache. Yet in others (e.g. 2048 and 8192) it was clearly inferior.

By and large associativities of four and greater were equivalent. It is generally accepted that higher associativities represent strongly diminishing returns in real world caches - 16-way associativity is generally considered equivalent to full associativity, and 8-way associative caches are by far the most common in the real world.

To be complete, it should finally be noted that random replacement, while potentially equivalent to any other policy - as well as potentially better or worse - did not perform particularly well in any of the tests. It was never substantially worse than LRU, but it was very rarely equivalent [or better]. Obviously rerunning the exact same simulations could produce different results, so the results reported above [for random replacement] are not conclusive.

Miss Breakdown

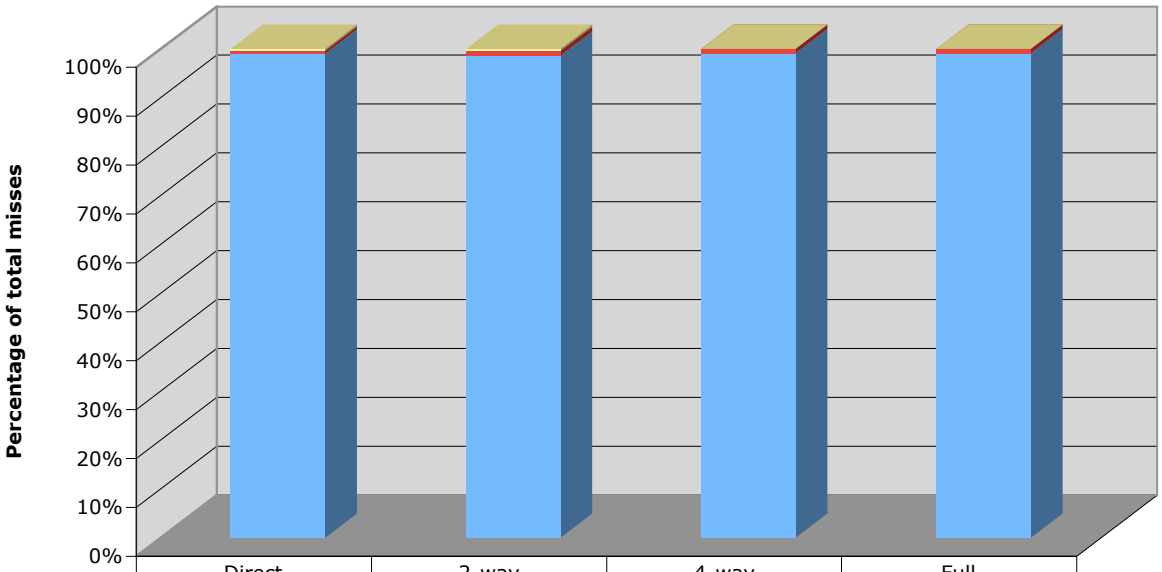
Breakdown of Miss Types for LRU
(64-byte blocks, 128-byte cache, 1-byte words)



	Direct	Full
Conflict	0.00	0.00
Compulsory	0.25	0.43
Capacity	99.75	99.57

Associativity

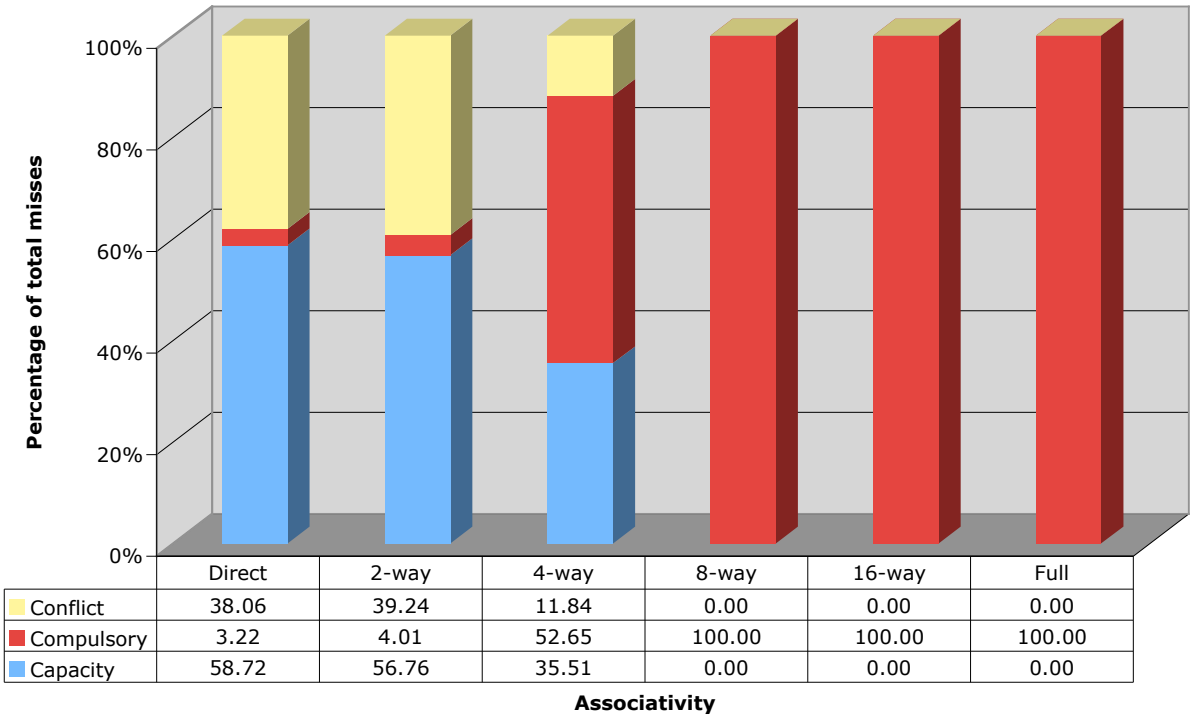
Breakdown of Miss Types for LRU
(64-byte blocks, 512-byte cache, 1-byte words)



	Direct	2-way	4-way	Full
Conflict	0.38	0.41	0.00	0.00
Compulsory	0.65	1.06	0.95	0.92
Capacity	98.97	98.53	99.05	99.08

Associativity

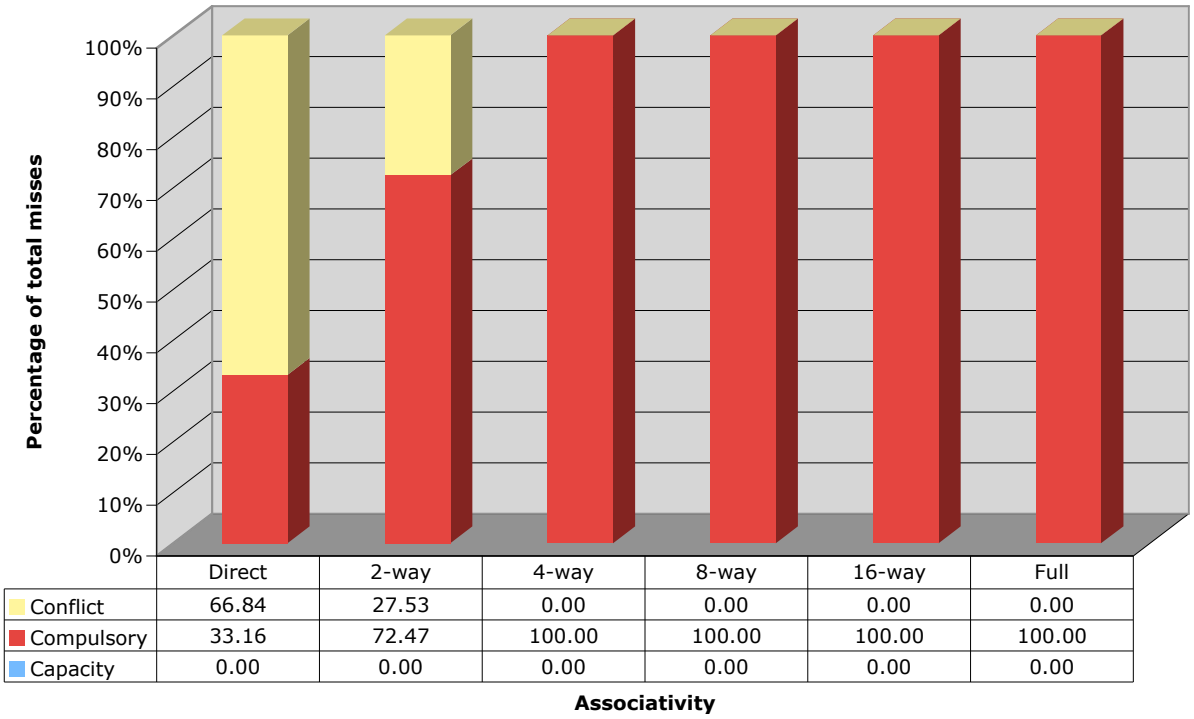
Breakdown of Miss Types for LRU
(64-byte blocks, 2048-byte cache, 1-byte words)



	Direct	2-way	4-way	8-way	16-way	Full
Conflict	38.06	39.24	11.84	0.00	0.00	0.00
Compulsory	3.22	4.01	52.65	100.00	100.00	100.00
Capacity	58.72	56.76	35.51	0.00	0.00	0.00

Associativity

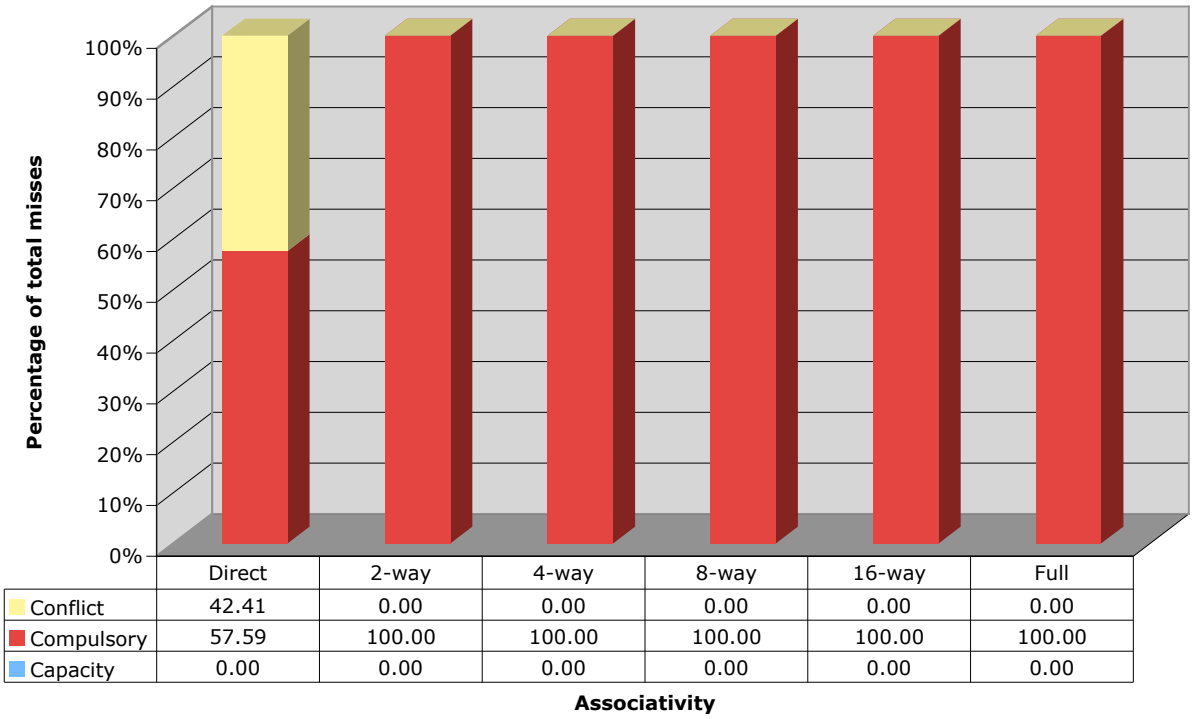
Breakdown of Miss Types for LRU
(64-byte blocks, 8192-byte cache, 1-byte words)



	Direct	2-way	4-way	8-way	16-way	Full
Conflict	66.84	27.53	0.00	0.00	0.00	0.00
Compulsory	33.16	72.47	100.00	100.00	100.00	100.00
Capacity	0.00	0.00	0.00	0.00	0.00	0.00

Associativity

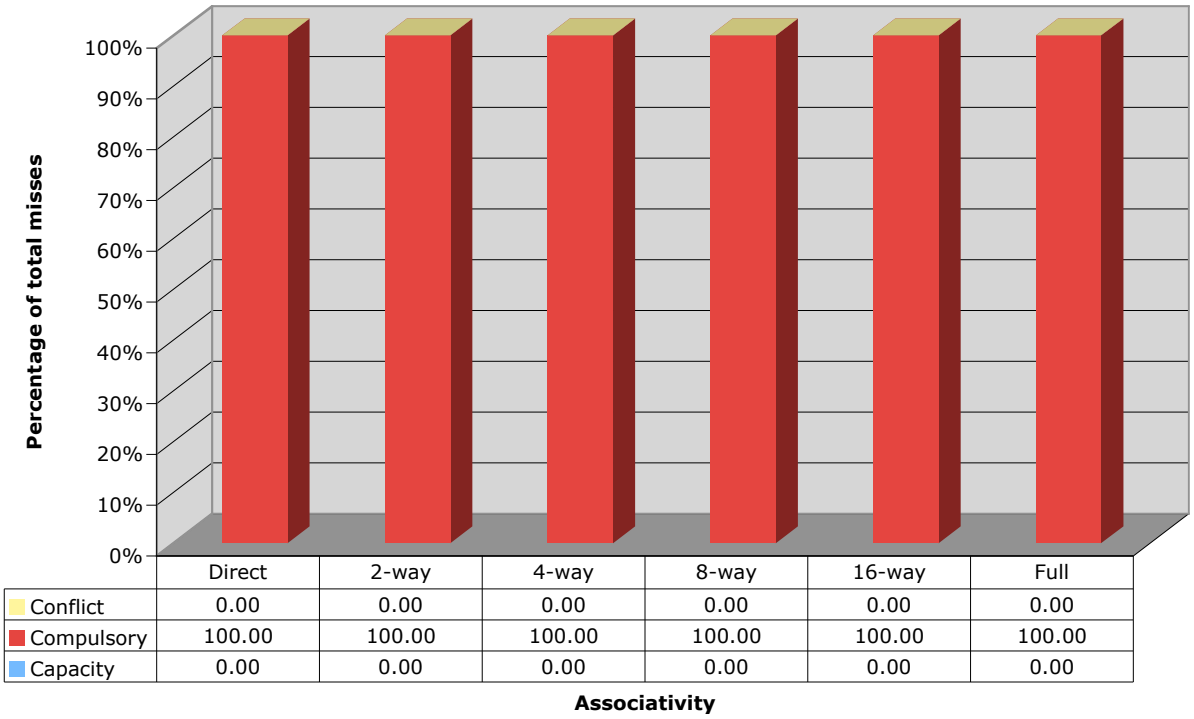
Breakdown of Miss Types for LRU (64-byte blocks, 16384-byte cache, 1-byte



	Direct	2-way	4-way	8-way	16-way	Full
Conflict	42.41	0.00	0.00	0.00	0.00	0.00
Compulsory	57.59	100.00	100.00	100.00	100.00	100.00
Capacity	0.00	0.00	0.00	0.00	0.00	0.00

Associativity

Breakdown of Miss Types for LRU (64-byte blocks, 32768-byte cache, 1-byte



	Direct	2-way	4-way	8-way	16-way	Full
Conflict	0.00	0.00	0.00	0.00	0.00	0.00
Compulsory	100.00	100.00	100.00	100.00	100.00	100.00
Capacity	0.00	0.00	0.00	0.00	0.00	0.00

Associativity

Discussion

Keep in mind when reviewing the charts that there are, in every case, 129 compulsory misses - since there are only 129 blocks (for 64-byte blocks) touched by the sample access pattern. Thus the size of the red segment can be used as a yardstick.

The data here is more simplistic, but shows strong conclusions. The most obvious is the transition from capacity misses as the dominant type, to compulsory misses, as cache size increases. Similarly, for caches “in the middle”, size-wise, conflict and capacity misses dominate only for the low associativities.

Overall it is clear that conflict misses are the dominant problem - both by nature and by portion of misses. A conflict miss indicates that there is room in the cache to store a new block, but the cache structure does not permit the exact block in question to be stored in any of the available locations. Such misses can occur for a cache of any size, given an appropriate access pattern. While capacity misses are relatively trivial to reduce - simply increase the cache size - conflict misses require an increase in associativity, which is a far less trivial modification.

From this we could hypothesise that policies favouring efficiency at lower associativities would be the most critical for real world caches. LRU and LRR both represent this in the results demonstrated so far.

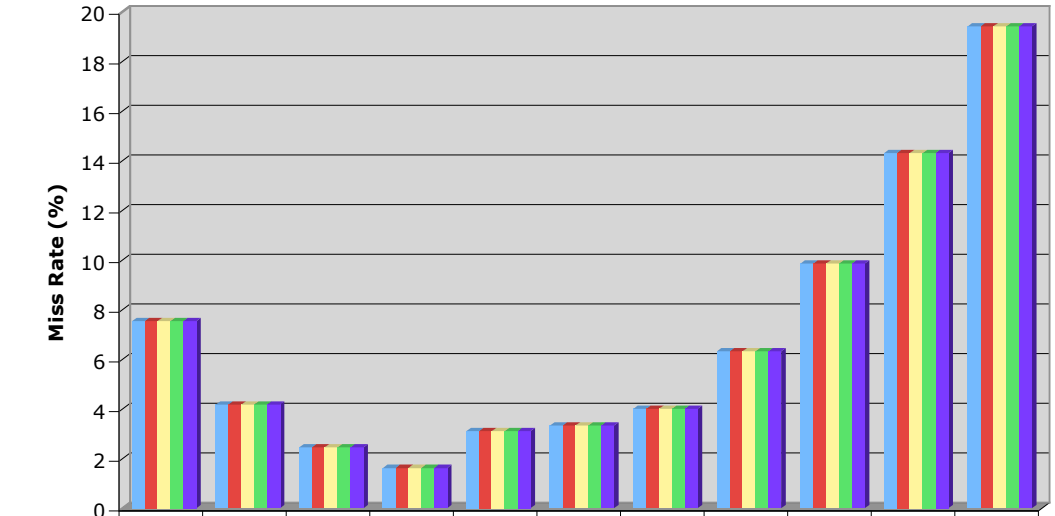
Performance as related to block size

This section looks at the performance of various replacement policies and block sizes, for a 2048-byte cache, with 1-byte words and considering associativities from direct to 16-way and then full. As in the preceding section, keep in mind that scales on the charts are not necessarily consistent.

A cache size of 2048-bytes was chosen because in the previous results it seemed to represent a key point in the results, below which the performance of the cache was poor regardless of replacement policy, and above which it quickly became close to ideal for all policies. Thus 2048-bytes was chosen to ensure some interesting contention.

Total Miss Rates

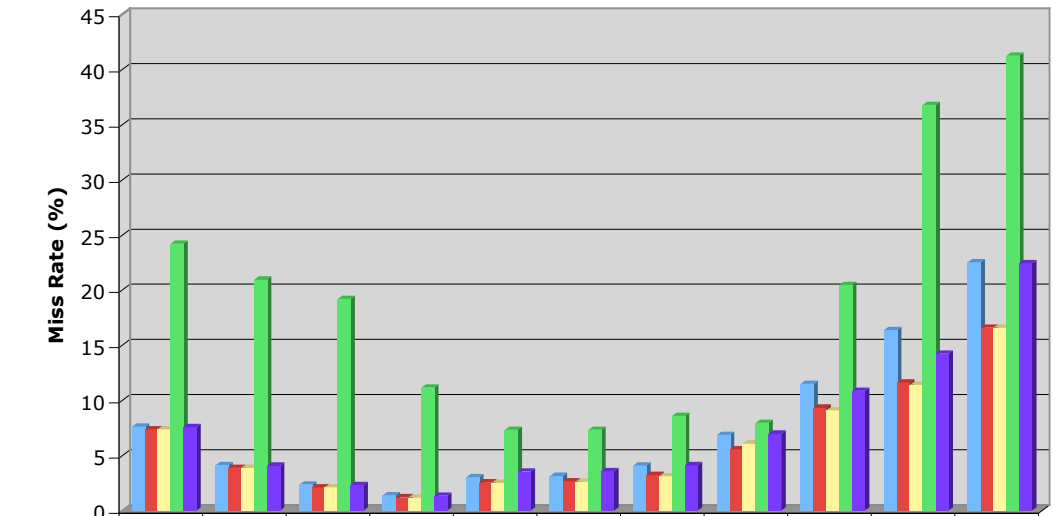
Miss rate versus block size (2048-byte cache, direct mapped, 1-byte words)



	1	2	4	8	16	32	64	128	256	512	1024
Random	7.537	4.176	2.454	1.62	3.111	3.328	4.009	6.324	9.856	14.307	19.425
LRU	7.537	4.176	2.454	1.62	3.111	3.328	4.009	6.324	9.856	14.307	19.425
LRR	7.537	4.176	2.454	1.62	3.111	3.328	4.009	6.324	9.856	14.307	19.425
LRW	7.537	4.176	2.454	1.62	3.111	3.328	4.009	6.324	9.856	14.307	19.425
FIFO	7.537	4.176	2.454	1.62	3.111	3.328	4.009	6.324	9.856	14.307	19.425

Block Size (Bytes)

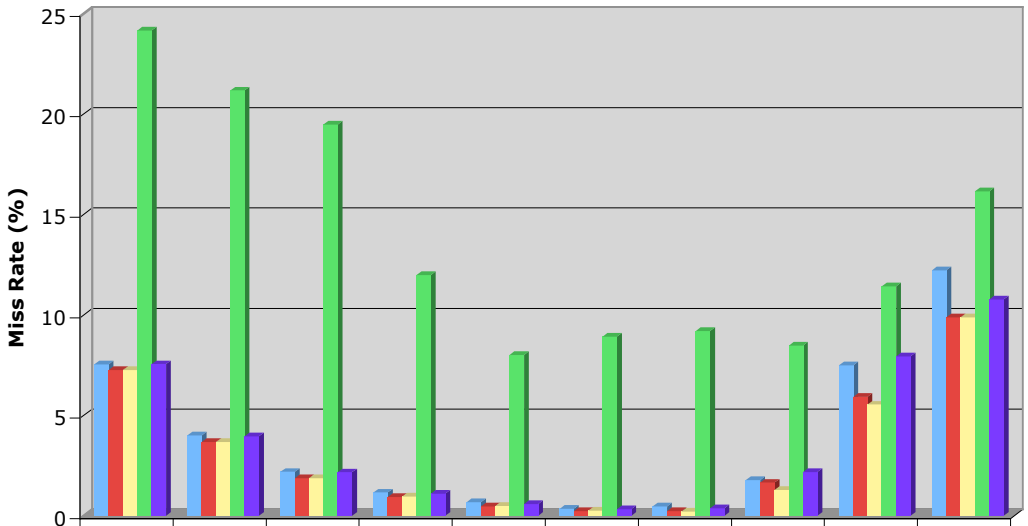
Miss rate versus block size (2048-byte cache, 2-way associative, 1-byte words)



	1	2	4	8	16	32	64	128	256	512	1024
Random	7.617	4.141	2.367	1.406	3.049	3.135	4.083	6.852	11.495	16.369	22.511
LRU	7.378	3.887	2.102	1.198	2.547	2.645	3.219	5.571	9.323	11.629	16.595
LRR	7.361	3.885	2.095	1.197	2.536	2.608	3.103	6.087	9.09	11.394	16.595
LRW	24.205	20.942	19.211	11.173	7.316	7.328	8.588	7.959	20.462	36.782	41.249
FIFO	7.596	4.063	2.305	1.334	3.533	3.577	4.109	6.969	10.856	14.227	22.425

Block Size (Bytes)

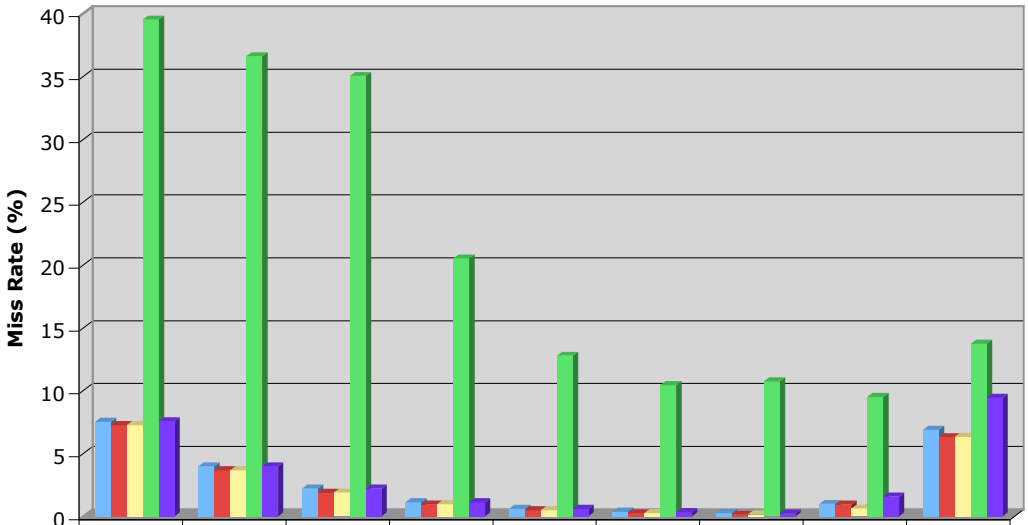
Miss rate versus block size (2048-byte cache, 4-way associative, 1-byte words)



	1	2	4	8	16	32	64	128	256	512
Random	7.545	4	2.192	1.161	0.674	0.36	0.482	1.79	7.484	12.215
LRU	7.261	3.674	1.88	0.945	0.479	0.246	0.245	1.653	5.93	9.875
LRR	7.261	3.674	1.886	0.954	0.488	0.258	0.215	1.288	5.539	9.882
LRW	24.144	21.169	19.481	11.988	8.012	8.918	9.202	8.474	11.42	16.155
FIFO	7.55	3.963	2.169	1.105	0.587	0.331	0.371	2.179	7.94	10.772

Block Size (Bytes)

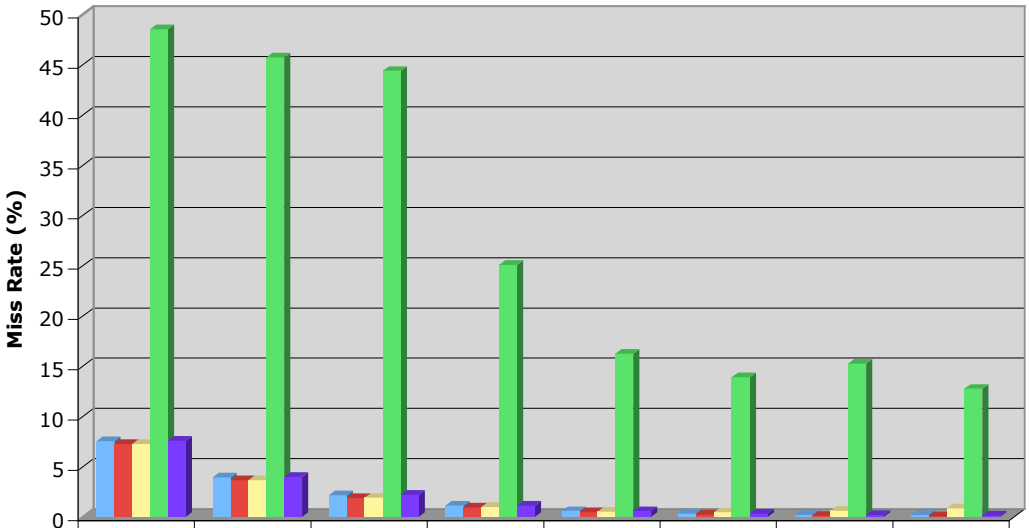
Miss rate versus block size (2048-byte cache, 8-way associative, 1-byte words)



	1	2	4	8	16	32	64	128	256
Random	7.528	3.989	2.196	1.13	0.597	0.369	0.255	0.997	6.879
LRU	7.261	3.674	1.88	0.945	0.479	0.246	0.129	0.937	6.311
LRR	7.261	3.674	1.891	0.961	0.495	0.267	0.154	0.617	6.326
LRW	39.526	36.616	35.041	20.526	12.778	10.439	10.73	9.507	13.742
FIFO	7.578	3.991	2.197	1.121	0.599	0.338	0.227	1.584	9.433

Block Size (Bytes)

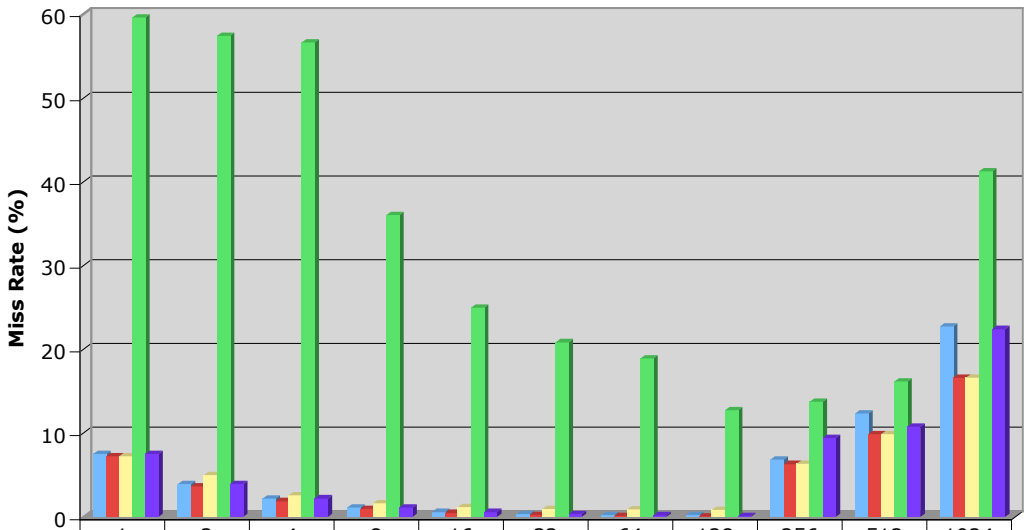
Miss rate versus block size (2048-byte cache, 16-way associative, 1-byte words)



	1	2	4	8	16	32	64	128
Random	7.541	3.95	2.154	1.116	0.595	0.347	0.216	0.205
LRU	7.261	3.674	1.88	0.945	0.479	0.246	0.129	0.069
LRR	7.261	3.674	1.901	0.976	0.511	0.456	0.627	0.851
LRW	48.56	45.751	44.4	25.098	16.258	13.923	15.26	12.763
FIFO	7.59	4.003	2.209	1.124	0.594	0.334	0.199	0.139

Block Size (Bytes)

Miss rate versus block size (2048-byte cache, fully associative, 1-byte words)



	1	2	4	8	16	32	64	128	256	512	1024
Random	7.507	3.917	2.184	1.103	0.587	0.329	0.218	0.23	6.852	12.331	22.709
LRU	7.261	3.674	1.88	0.945	0.479	0.246	0.129	0.069	6.311	9.875	16.595
LRR	7.261	4.999	2.564	1.631	1.173	0.965	0.879	0.851	6.326	9.882	16.595
LRW	59.592	57.438	56.615	36.054	24.992	20.853	18.902	12.763	13.742	16.155	41.249
FIFO	7.516	3.929	2.22	1.129	0.591	0.322	0.199	0.139	9.433	10.772	22.425

Block Size (Bytes)

Note: Obviously the replacement policy has no bearing for a direct mapped cache, as shown in the first chart above. Also keep in mind that the largest block size for each n-way associativity also corresponds to full associativity, so there is some overlap between the fully associative chart and the others.

Discussion

It must also be kept in mind that although larger blocks may result in fewer compulsory misses (as shall be detailed in the next subsection), larger blocks take longer to fetch. So the above charts may not directly represent performance (as measured by latency or memory bandwidth).

The first thing that is painfully obvious is that LRW again performs terribly. Although interestingly it still follows the same sort of curve as the other policies, but has a “sweet spot” consistently above the other policies - e.g. for the fully associative cache all other policies seem to favour block sizes of around 64 bytes, while LRW favours 256 bytes.

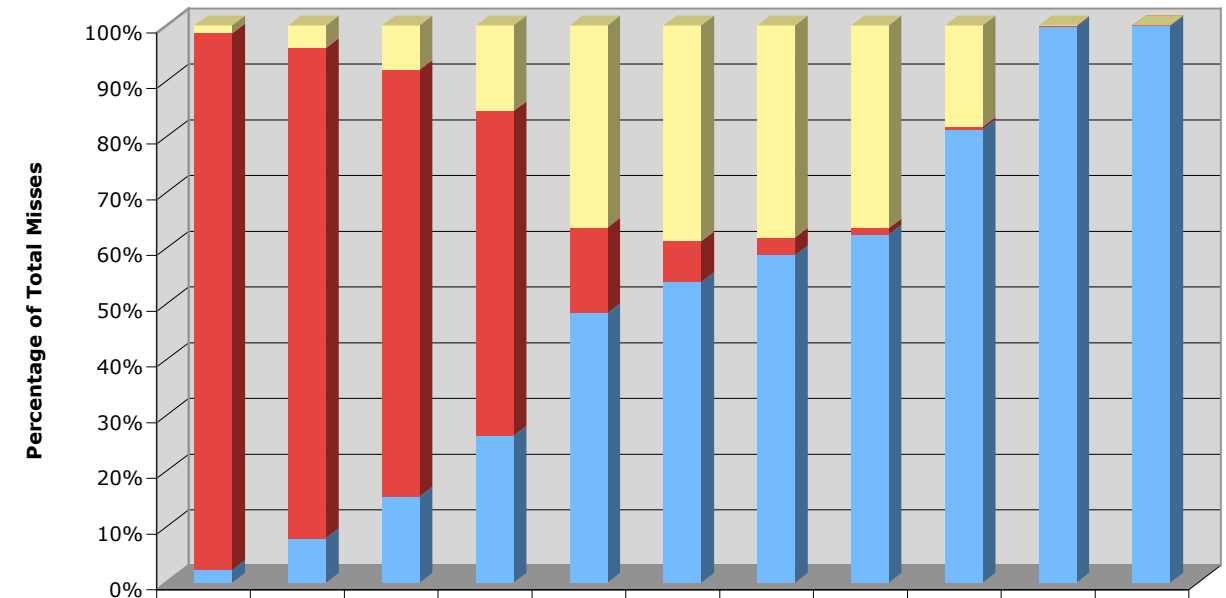
It is interesting to note that performance is much more adversely effected by too-large block sizes, as opposed to too-small block sizes. This is highly dependent on the exact request pattern issued to the cache, but is fairly intuitive - assuming every address is requested more than once, the optimal cache configuration will tend to be one which allows every required address to be in cache while maintaining the largest block size possible (to ensure as many addresses are covered by a single block as possible, thus reducing compulsory misses).

Although it is not entirely supported by the charts above, it seems likely that higher associativities do perform better, and favour larger block sizes, provided the cache is large enough to accommodate a reasonable number of sets. Looking at the fully associative results, as soon as the effective associativity drops below 8-way (128-byte blocks) performance falls off rapidly. If the cache were larger, this would not happen.

Miss Breakdown

The following charts break down the misses shown for the LRU replacement policy in the previous charts, into capacity, compulsory and conflict misses. All scales are percentages and not directly comparable, as the absolute number of misses does vary significantly between charts and over block size.

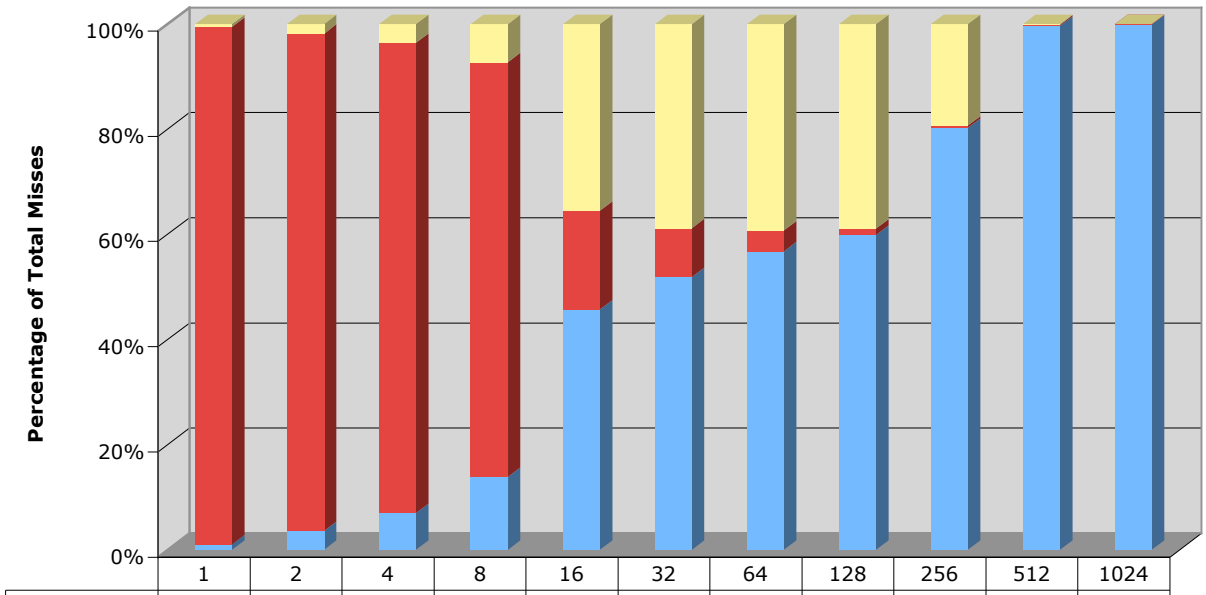
Breakdown of Miss Types for LRU (2048-byte cache, direct mapped, 1-byte words)



	1	2	4	8	16	32	64	128	256	512	1024
Conflict	1.42	4.05	7.95	15.37	36.26	38.70	38.06	36.42	18.22	0.13	0.00
Compulsory	96.34	87.98	76.61	58.33	15.40	7.39	3.22	1.09	0.41	0.17	0.08
Capacity	2.24	7.97	15.44	26.30	48.34	53.91	58.72	62.49	81.37	99.71	99.92

Block Size (Bytes)

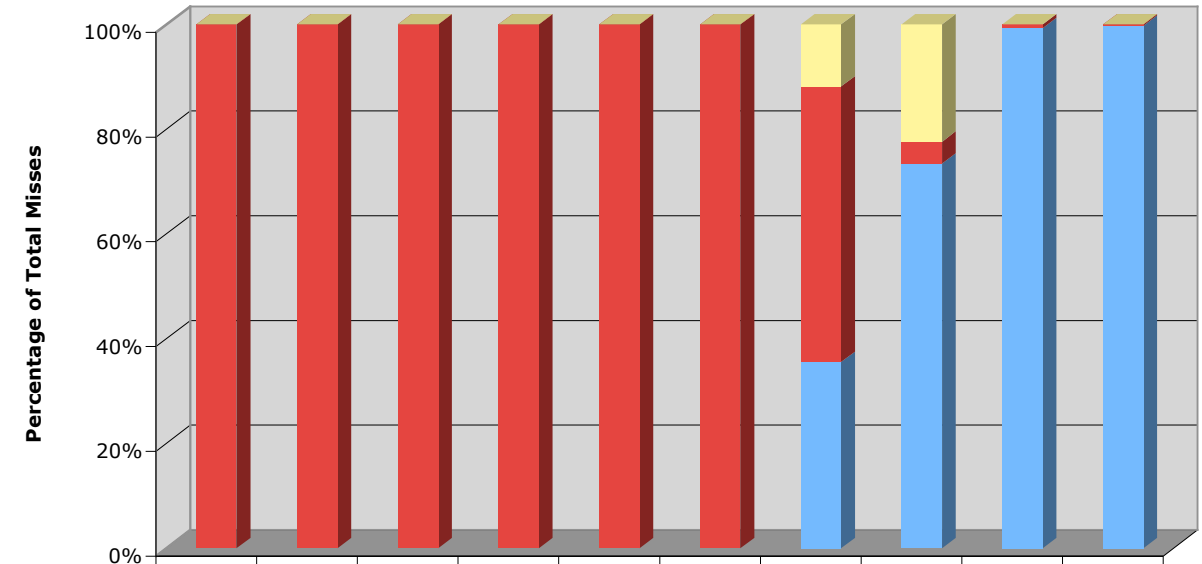
Breakdown of Miss Types for LRU
(2048-byte cache, 2-way associative, 1-byte words)



	1	2	4	8	16	32	64	128	256	512	1024
Conflict	0.54	1.85	3.57	7.26	35.49	38.79	39.24	38.88	19.33	0.15	0.00
Compulsory	98.41	94.52	89.44	78.88	18.81	9.30	4.01	1.24	0.43	0.21	0.10
Capacity	1.04	3.63	6.99	13.86	45.70	51.91	56.76	59.88	80.24	99.65	99.90

Block Size (Bytes)

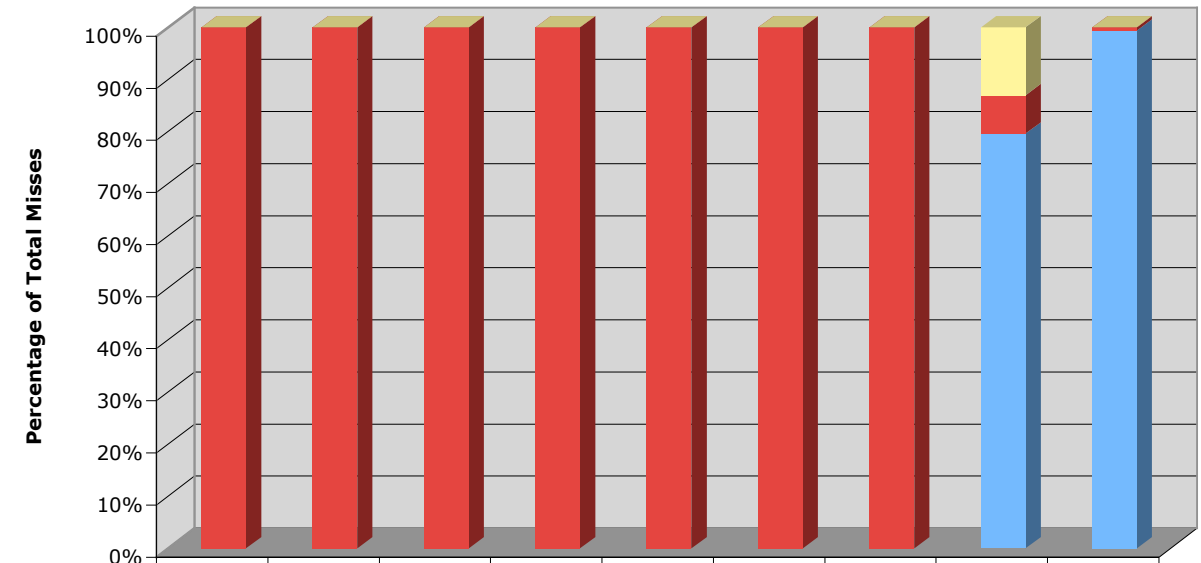
Breakdown of Miss Types for LRU
(2048-byte cache, 4-way associative, 1-byte words)



	1	2	4	8	16	32	64	128	256	512
Conflict	0.00	0.00	0.00	0.00	0.00	0.00	11.84	22.38	0.02	0.00
Compulsory	100.00	100.00	100.00	100.00	100.00	100.00	52.65	4.17	0.67	0.24
Capacity	0.00	0.00	0.00	0.00	0.00	0.00	35.51	73.44	99.31	99.76

Block Size (Bytes)

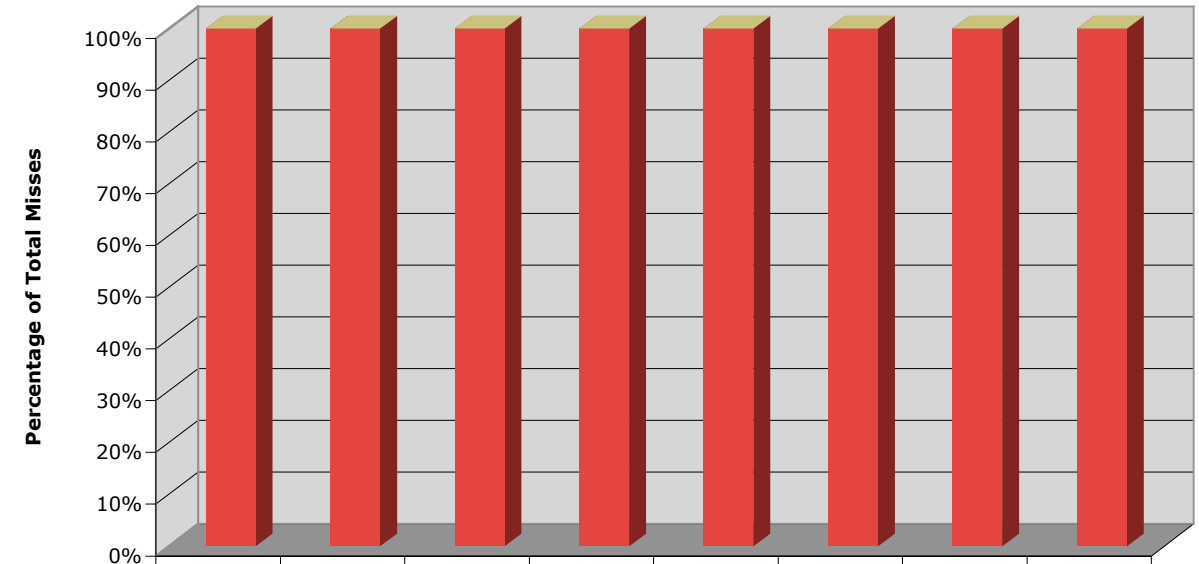
Breakdown of Miss Types for LRU
(2048-byte cache, 8-way associative, 1-byte words)



	1	2	4	8	16	32	64	128	256
Conflict	0.00	0.00	0.00	0.00	0.00	0.00	0.00	13.02	0.00
Compulsory	100.00	100.00	100.00	100.00	100.00	100.00	100.00	7.36	0.63
Capacity	0.00	0.00	0.00	0.00	0.00	0.00	0.00	79.62	99.37

Block Size (Bytes)

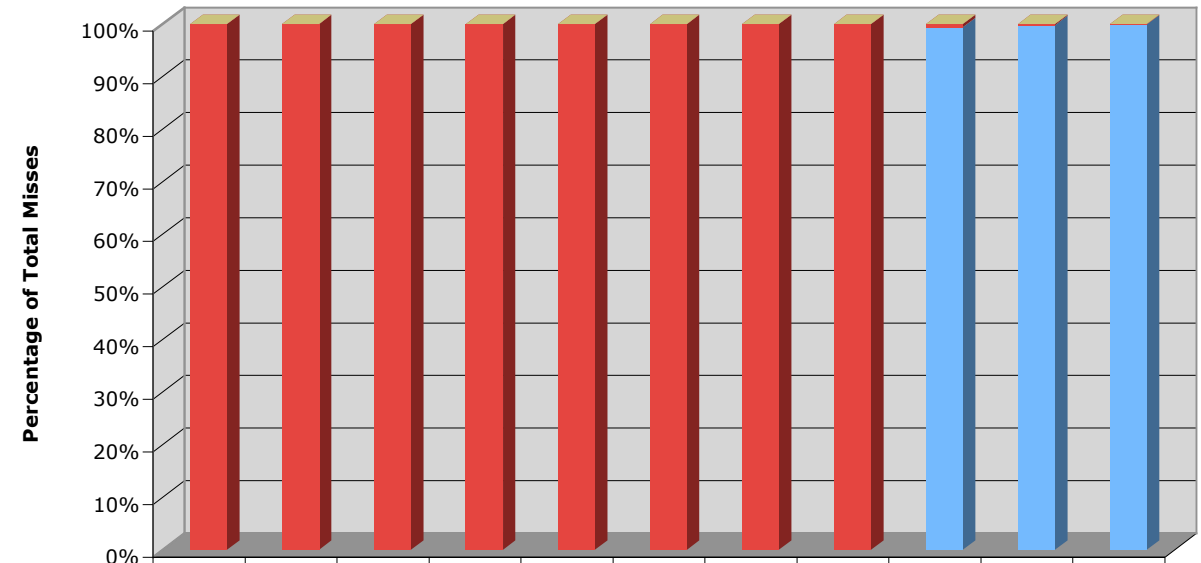
Breakdown of Miss Types for LRU
(2048-byte cache, 16-way associative, 1-byte words)



	1	2	4	8	16	32	64	128
Conflict	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Compulsory	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
Capacity	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Block Size (Bytes)

Breakdown of Miss Types for LRU
(2048-byte cache, fully associative, 1-byte words)



	1	2	4	8	16	32	64	128	256	512	1024
Conflict	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Compulsory	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	0.63	0.24	0.10
Capacity	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	99.37	99.76	99.90

Block Size (Bytes)

Discussion

Compulsory misses clearly dominate the lower block sizes, regardless of associativity - indeed, increasing the associativity only increases [slightly] the maximum block size before conflict or capacity misses become significant.

However, the results above may be misleading on their own - remember that the actual total number of misses was generally much higher for very low block sizes (4 or less), as shown in the previous subsection. This is because for very small block sizes, there is very little sharing of blocks by multiple addresses, consequently more unique blocks are required and there are vastly more compulsory misses to load these. For example, remember that a block size of 64-bytes requires 129 compulsory misses, while a block size of 1 byte requires 7,261 compulsory misses.

Also of interest is the fact that, for any real degree of associativity (i.e. not direct mapped), capacity misses dominate over conflict misses. This is relatively good, as capacity misses - as previously discussed - can be easily reduced by increasing cache size, while conflict misses are a product of cache structure (associativity), which is much harder to modify.

Going from direct to 2-way associative, it is interesting to note that the portions are fairly similar. What does change slightly is the total number of misses - as seen in the previous subsection, and indirectly visible on these charts by the larger relative portion of compulsory misses. This is indicating that the extra associativity, even just 2-way, significantly decreases the number of “unnecessary” misses - i.e. conflict and capacity misses. Unfortunately, however, once block size becomes large enough (16 in this case) performance drops of similarly. A large part of this is capacity misses, due to the reduced number of sets.

Ideal Cache

In summary, the ideal approach for designing a cache is:

- Large size
- High associativity
- Reasonably large block size

It is important to recognise at this point the limitations of SimpleCacheSim. Notably, that it does not simulate timing of any sort. All the points above tend to push latencies up and clock speed down, such that there is diminishing returns to increasing them, and at some point performance will in fact begin to decrease. And then even beyond that, there are issues such as pipelining, multiport access (concurrent requests), etc. And once you get to issues such as manufacturing yields and recovery, it’s almost anybody’s guess.

Nonetheless, given the results presented in this document, it seems a block size of around 32 or 64 bytes is appropriate. A cache size of at least 16kiB is highly desirable - especially 32kiB or above, where there are no conflict or capacity misses. And an associativity of 8 or higher produces best results (although of course with a large enough cache, direct mapping will perform much the same as higher associativities, and would be much simpler and faster in a real implementation).

Looking through the entire 14,100 sets of results produced for this document, the lowest possible number of misses is 16. This occurs only for a block size of 1024 (the largest simulated), for caches at least 16kiB in size (given high associativity - larger caches required for lower associativies), and for any replacement policy (since the 16 misses are compulsory, and there are no evictions). A full listing is included in Appendix II.

In the real world, the typical cache configurations are as shown below. These figures are derived from the specifications for the PowerPC 7400 and 970, the AMD Athlon, the Intel Pentium IV and Pentium M.

	Cache Size	**Word Size**	**Block Size**	**Associativity**
L1	32-128kiB Split	16B-32B/4*4B-4*8B	32-128B	2-8 way
L2	256-1024kiB Unified	8B-32B/4*2B-4*8B	64-128B	8 way
L3	512-2048kiB Unified	8B-16B/4*2B-4*4B	?	?

Note that some of the caches are multiported, allowing multiple requests at different addresses. Thus some L1 caches have an effective word size of 32B, composed of 8 independent requests for standard 4-byte words.

It is worth noting that in the leading edge processors today - notably the Pentium M and future processors from Intel - the block size is decreasing at all levels of cache, down to 32 bytes in the Pentium M (from 128 bytes in the Pentium IV). Smaller block sizes offer better performance when serving multiple threads concurrently, each of which will have a very disparate memory pattern - and parallelism is a high priority for future processors. Combined with modern prefetching (the 970 has eight independent prefetch streams), many compulsory misses can be avoided, reducing or even eliminating that key failing of small block sizes.

One day we may see huge, fully-associative caches with single-word block sizes and all the extra bells and whistles. For the moment, however, there are still some very big trade-offs to be evaluated. And in any case, the performance of the cache is largely dependent on the access pattern thrown at it, making it very difficult to design a cache that pleases everyone. It is strongly evident in modern processors that the caches were designed with very specific aims, aiming to be very good with certain types of programs (e.g. games) while performing miserably for everything else.

Of course, the next breakthrough may be memory which performs as fast as today's L1 or L2 caches, which will change the performance landscape massively.