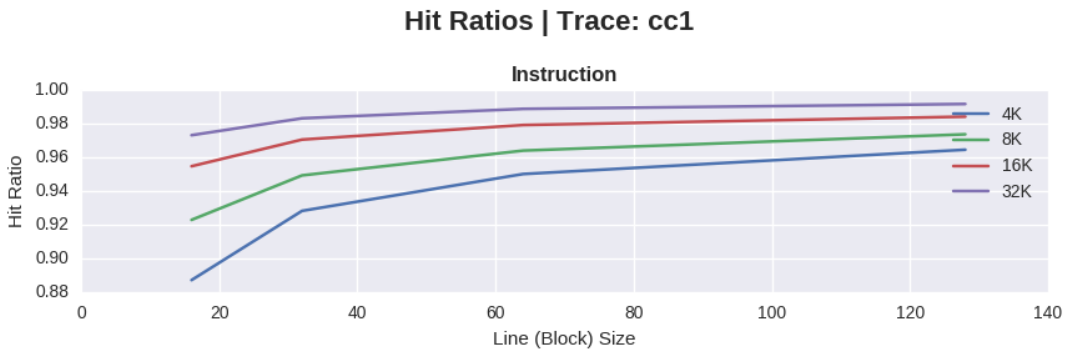
Assignment 2

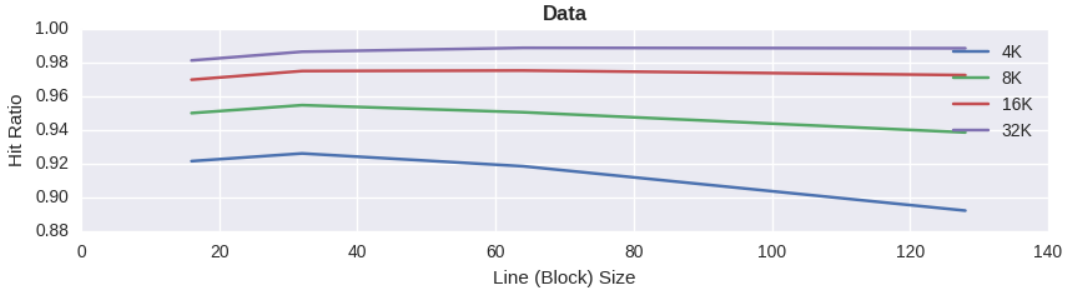
Cache Memory Simulations

Justin Clark

Part A: Hit Ratios w/ Varying Block & Cache Sizes



	16	32	64	128
4K	0.8874642	0.9284325	0.9502259	0.9646104
8K	0.9230928	0.949423	0.9640862	0.9737318
16K	0.9548288	0.9706037	0.9792181	0.9841789
32K	0.9731983	0.9831899	0.9887712	0.9916656

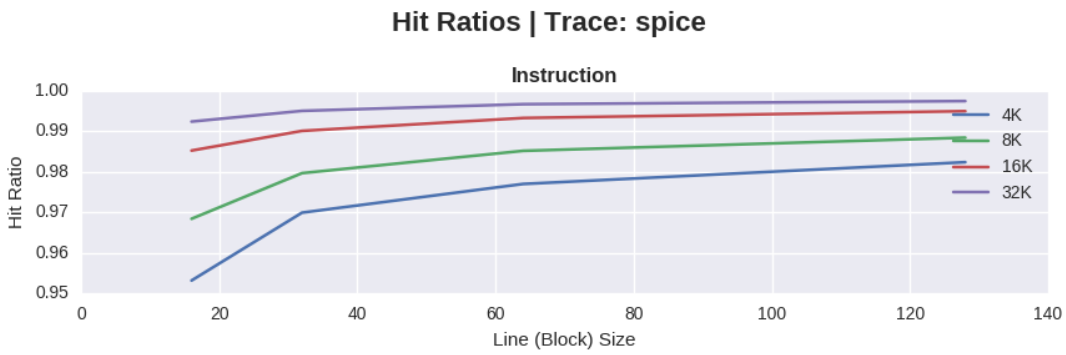


	16	32	64	128
4K	0.9214748	0.9260203	0.9184459	0.8921994
8K	0.9498972	0.9545786	0.9503876	0.9384574
16K	0.9696614	0.9747838	0.975064	0.9723647
32K	0.9810476	0.9861989	0.988486	0.9882429

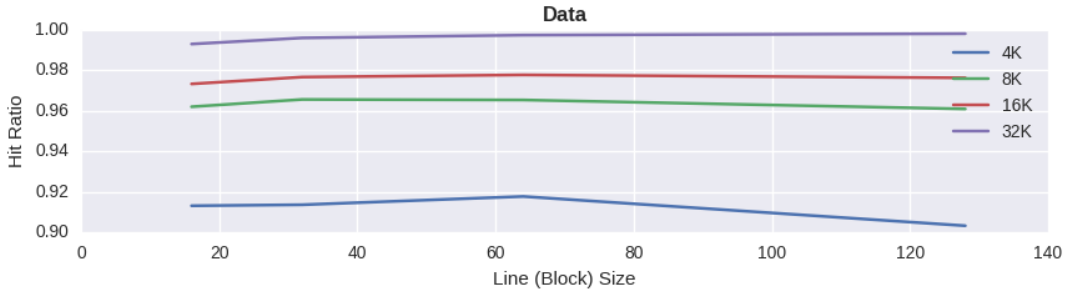
For the *cc1* trace file, the instruction cache hit ratios trended upward as the line size increased. One thing to note here is that the larger instruction cache sizes led to higher hit ratios. Specifically, the 32K cache size had the highest hit ratios for a given line size, the 16K had the next highest hit ratio for each line size, and so on.

Another interesting thing to note about the instruction cache here is that the larger cache sizes had smaller variation in hit ratios between the smaller and larger line sizes. Along similar lines, the spread between hit ratios of different cache sizes (on each line size), decreased as the line size increased.

The results of the data cache hit ratios in the simulation are the converse of the instruction cache hit ratios in two respects. First, the hit ratios tend downward (or barely move) as line size increased (excluding the small spike at the line size of 32 bytes). Second, the spread in hit ratios for a given line size *increases* as the line size increases. However, larger data cache sizes still outperformed smaller data cache sizes. Additionally, there is lower variation among the larger cache sizes. Further discussion on the nature of the results will follow in the “Access Time” section for the *cc1* trace.



	16	32	64	128
4K	0.9532528	0.9699897	0.9770595	0.9824277
8K	0.9684694	0.9797282	0.9852254	0.9884793
16K	0.9852906	0.9901452	0.9933211	0.99501
32K	0.9924345	0.9950803	0.9967232	0.9974909

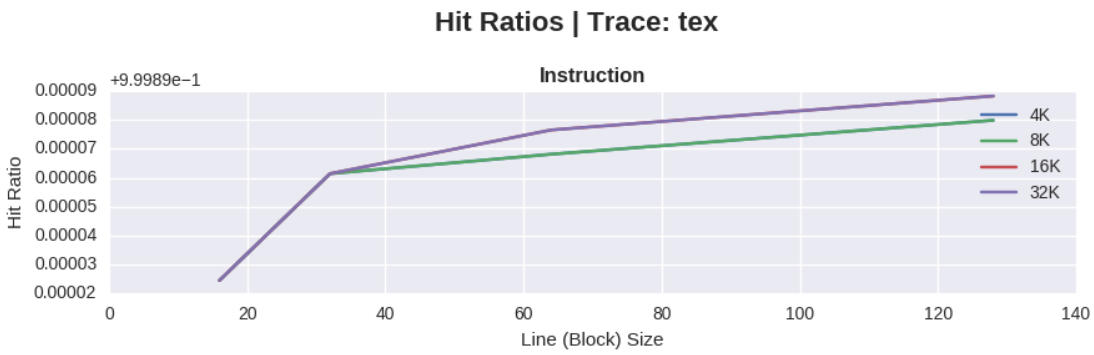


	16	32	64	128
4K	0.9130489	0.913546	0.9176337	0.9032025
8K	0.9618711	0.9654571	0.9652131	0.9607848
16K	0.9731215	0.9765003	0.9775775	0.9761136
32K	0.9927867	0.9957926	0.9971736	0.9978871

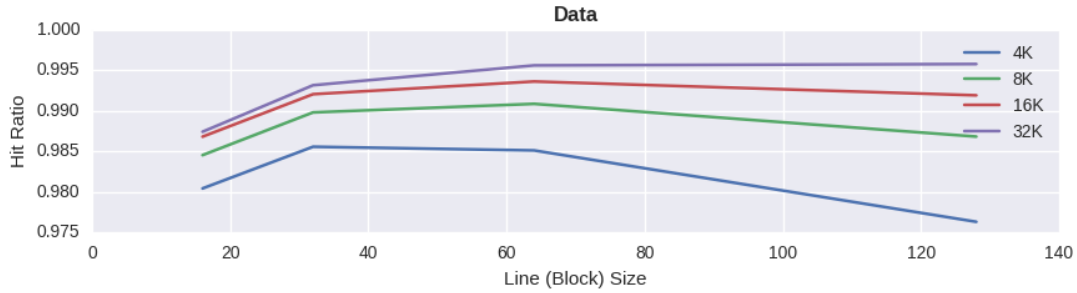
There are certainly similarities between the *spice* trace and the previous *cc1* trace. The same general trends could be said about these results, but there are some subtle differences. The *spice* hit ratios are higher than their *cc1* trace counterparts. Concretely, the 4K instruction cache starts at a .95325 hit ratio and increases to a .98243 hit ratio for the *spice* trace and in the *cc1* trace, the hit ratio starts at .88746 and increases to .96461. Further, the tables of the instruction cache for both the *spice* and *cc1* traces illustrate the variance of all hit ratios for a given cache size is lower for the *spice* trace.

The data cache for the *spice* trace is also similar to the data cache for the *cc1* trace. However, for the *spice* trace, we notice that there is extremely little variance for a give cache size across all line sizes. This implies that the line size has a small effect on data cache.

Conversely, the data cache size certainly matters in terms of hit ratio - especially for the 4K cache size. Interestingly, the 8K, 16K, and 32K cache sizes in the *spice* trace outperform their *cc1* counterparts. However, the 4K cache size in the *spice* trace actually underperforms its *cc1* counterpart. More discussion on the results will follow in the “Access Times” section.



	16	32	64	128
4K	0.9999146	0.9999514	0.9999581	0.9999699
8K	0.9999146	0.9999514	0.9999581	0.9999699
16K	0.9999146	0.9999514	0.9999665	0.9999782
32K	0.9999146	0.9999514	0.9999665	0.9999782



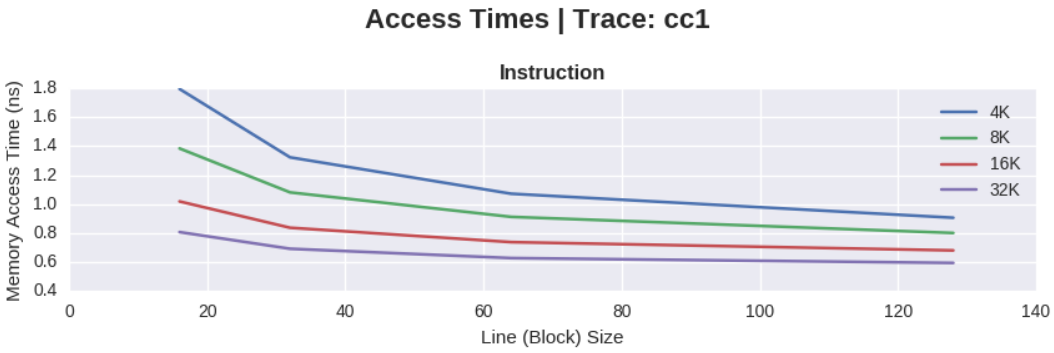
	16	32	64	128
4K	0.980397	0.985538	0.985083	0.976285
8K	0.9845004	0.989769	0.9908236	0.9867882
16K	0.9867754	0.9920142	0.9935791	0.9918739
32K	0.9873835	0.9931198	0.9955649	0.9957307

The instruction cache for the *tex* trace is unlike any of the other hit ratio results. For one, the 4K and 8K cache sizes line up exactly, as do the 16K and 32K cache sizes. Another thing to note is how high the instruction cache hit ratios are. Even the lowest hit ratios are a few orders of magnitude greater than the highest hit ratio of either the *cc1* or *spice* traces.

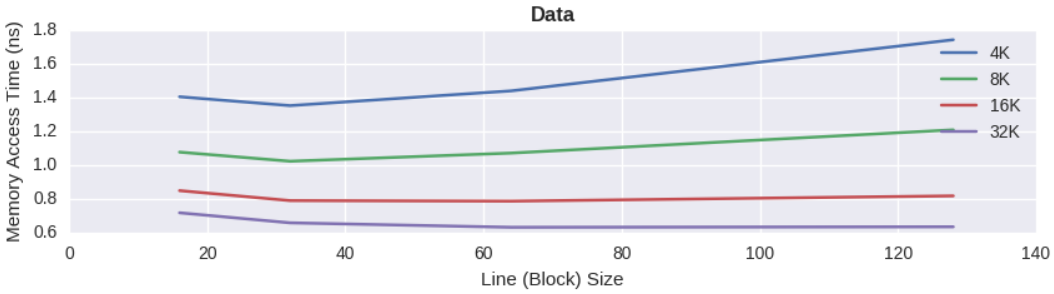
The data cache is more in line with what we have seen in the previous two trace programs. The difference to note is that there is a clear upward trend for the line size of 32 bytes and then a plateau or decrease in hit ratio as the line size increases further. The variation between the data cache hit ratios of the *tex* trace are much smaller than the variation of the data cache hit ratios in either of the other two trace programs.

Additionally, it is worth noting that the performance for the 32K data cache of the *tex* trace is very similar to the 32K cache of the *spice* trace. However, the 4K, 8K, and 16K cache sizes of the *tex* trace outperform their *spice* trace counterparts. Further discussion on the nature of the performances will follow in the “Access Times” section.

Part A: Access Times w/ Varying Block & Cache Sizes



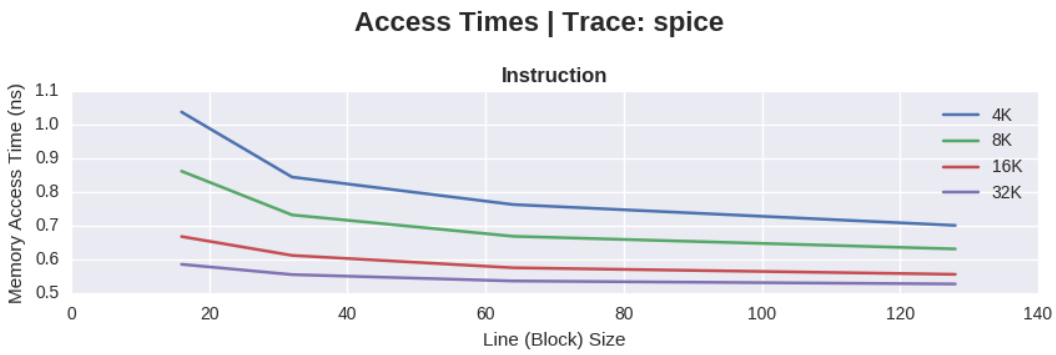
	16	32	64	128
4K	1.7941621	1.3230262	1.0724027	0.9069805
8K	1.3844332	1.081635	0.9130088	0.8020845
16K	1.0194688	0.8380571	0.7389921	0.6819431
32K	0.8082192	0.6933165	0.6291307	0.5958459



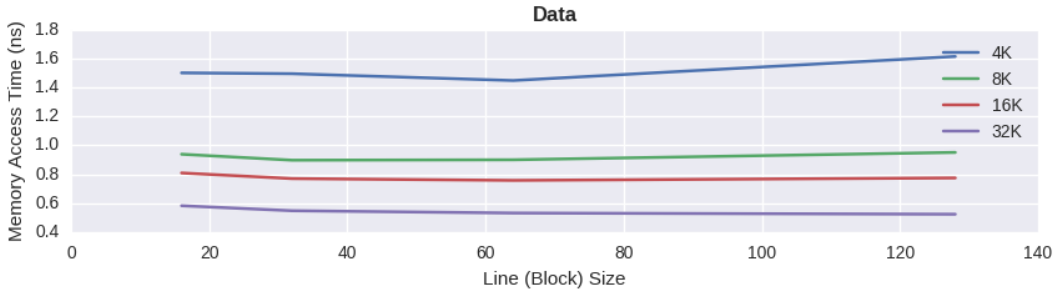
	16	32	64	128
4K	1.4030396	1.3507671	1.4378722	1.7397068
8K	1.0761824	1.022346	1.0705429	1.2077404
16K	0.8488941	0.7899869	0.7867643	0.8178055
32K	0.7179522	0.6587132	0.6324111	0.6352071

As discussed in the hit ratio section of the *cc1* trace, the variation of the instruction cache performance decreases as the line size increases. Why? The answer could lie in the type of instructions specified by the *cc1* trace program. If the performance increases with the line size, it is likely that the instructions are relatively complicated (i.e. *long*). This could also explain why there is more variance in performance for the smaller line sizes; with smaller line sizes, more memory locations are required, so the smaller cache sizes run out of cache memory locations and have to access main memory more often. The effect on the larger cache sizes is smaller since there are more locations to store the complicated instructions.

The data cache performance could be explained as follows: the *tex* trace program simulates relatively small operands (data), so larger line sizes are relatively useless. Additionally, the larger line sizes can actually take longer to parse. All else being equal, a line size of 32 bytes seems to be the optimal point when weighing hardware overhead and performance of the data cache. Of course, the 32K would be the optimal choice of cache size in both instruction and data caches.



	16	32	64	128
4K	1.0375924	0.8451187	0.7638158	0.702082
8K	0.8626016	0.7331252	0.6699075	0.6324882
16K	0.6691583	0.6133305	0.5768073	0.5573851
32K	0.5870032	0.5565771	0.5376838	0.5288542

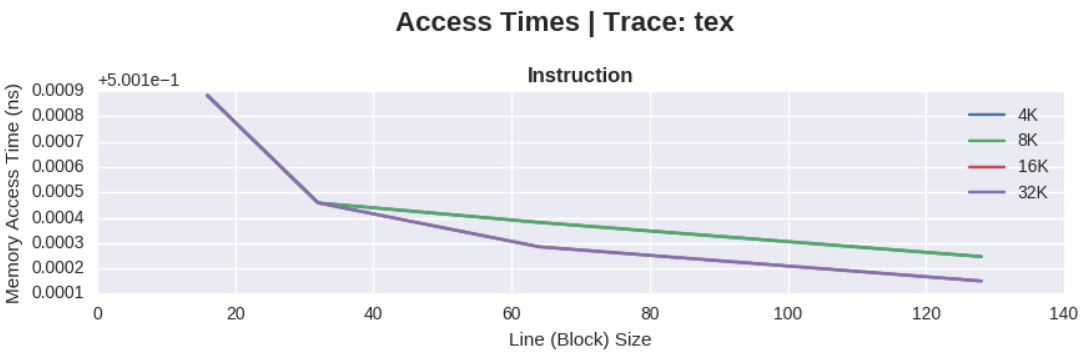


	16	32	64	128
4K	1.4999379	1.4942206	1.447212	1.6131713
8K	0.9384819	0.8972436	0.9000493	0.9509752
16K	0.8091025	0.7702463	0.7578589	0.7746931
32K	0.5829532	0.5483849	0.5325037	0.5242983

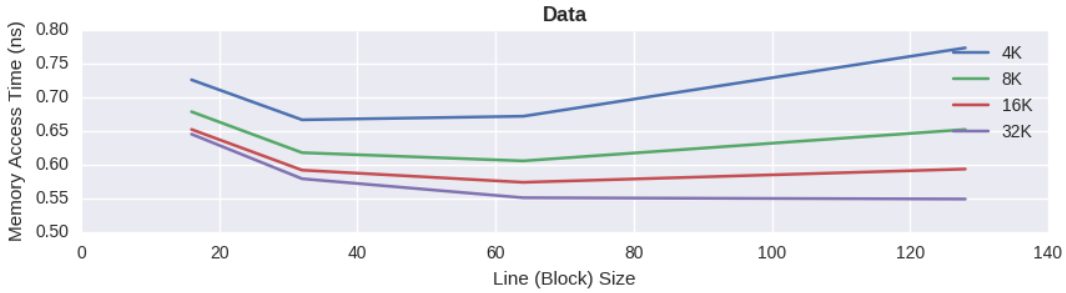
I would echo much of what is explained in the previous section in regard to the causes of the performance of the caches when simulated with the *spice* trace. However, in the instruction cache there is not much of a performance benefit to having a 32K cache over a 16K cache. This reason for this minor difference could be that there is a smaller variation in the instructions provided by the *spice* trace compared to the *cc1* trace.

As with the *cc1* trace, the data cache for the *spice* trace shows extremely small variation in memory access times between line sizes. This could mean that, like the *cc1* trace, the fetched operands are very small and the increase in line size offers no improvement and actually decreases performance for the 128 byte line size in all but the 32K cache.

For the *spice* trace program, the ideal cache size is either 16K or 32K. The exact specification depends on the how much overhead can be tolerated. In many systems, a 32K cache size should be well in the limit, but there are certainly some systems (embedded systems, for example), where 32K might be pushing the limit and 16K would be a good choice.



	16	32	64	128
4K	0.5009819	0.5005583	0.5004813	0.5003466
8K	0.5009819	0.5005583	0.5004813	0.5003466
16K	0.5009819	0.5005583	0.5003851	0.5002503
32K	0.5009819	0.5005583	0.5003851	0.5002503



	16	32	64	128
4K	0.7254346	0.666313	0.6715454	0.7727221
8K	0.6782449	0.6176563	0.6055288	0.6519361
16K	0.6520828	0.5918365	0.5738408	0.5934502
32K	0.6450899	0.5791222	0.551004	0.5490968

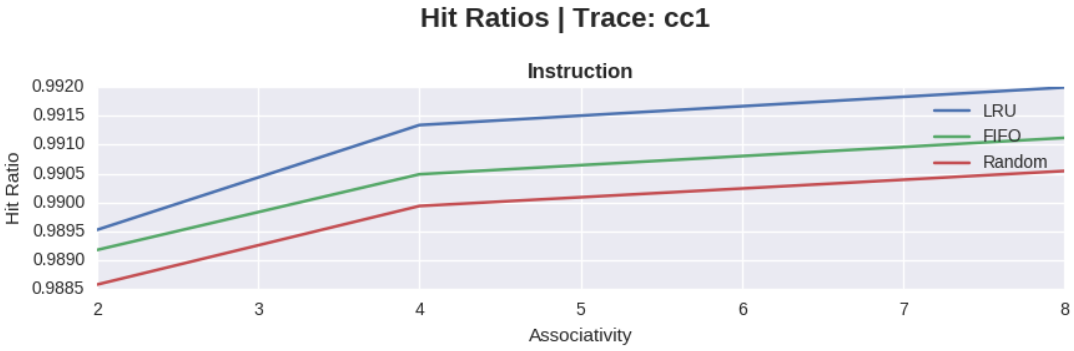
The performance of the simulated instruction cache on the *tex* trace implies that there is likely a relatively small number of instructions. The reason for this explanation is that there are only minor differences in performance between different cache sizes (if any at all). However, since performance increases with larger line sizes, the instructions could be relatively large.

The data cache has interesting access time (and hit ratio) curves. The small variation at smaller line sizes could be because there are fewer operands fetched by memory. As line size increases, we notice performance either stagnates or decreases. This could mean that operands are of “moderate” length. So, in other words, the operands are big enough to see a performance increase when the line size increases from 16K to 32K, but they are small enough that further increases in line size are either detrimental or negligible.

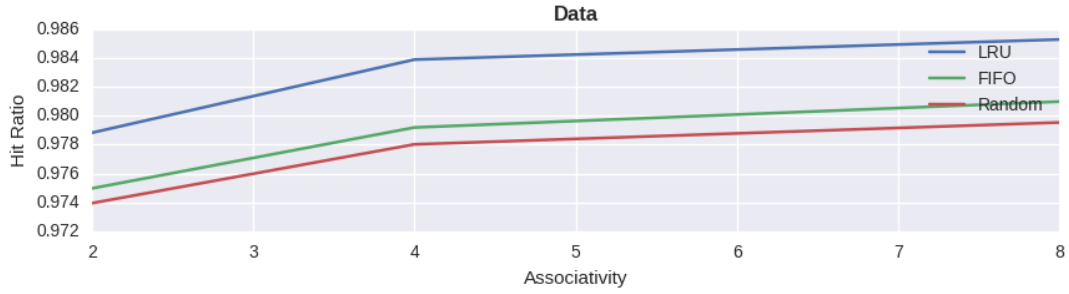
Part A: Conclusion

Given the performances of instruction and data cache discussed in the last three pages, the design would be as follows: the instruction cache size should be 32K and the line size should be 128 bytes. The data cache is more nuanced and requires weighing trade-offs. The cache size should almost certainly be 32K, but 16K would not be too detrimental in most cases (like the *tex* trace above). The line size would be almost optimal at 64 bytes, but, as with the cache size, there would not be too much of a performance loss if the line size was 32 bytes.

Part B: Hit Ratios w/ Varying Cache Replacement Policies & Set-Associative Mappings



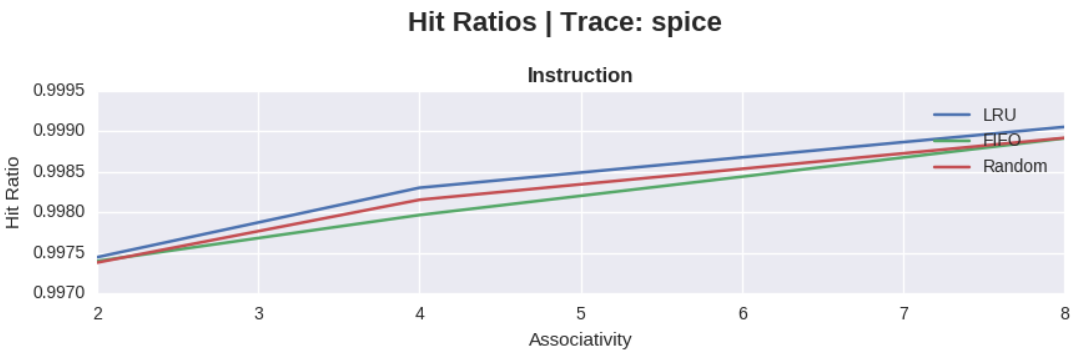
	2	4	8
LRU	0.9895292	0.9913394	0.9919891
FIFO	0.9891819	0.9904891	0.9911176
Rand	0.9885837	0.9899411	0.9905459



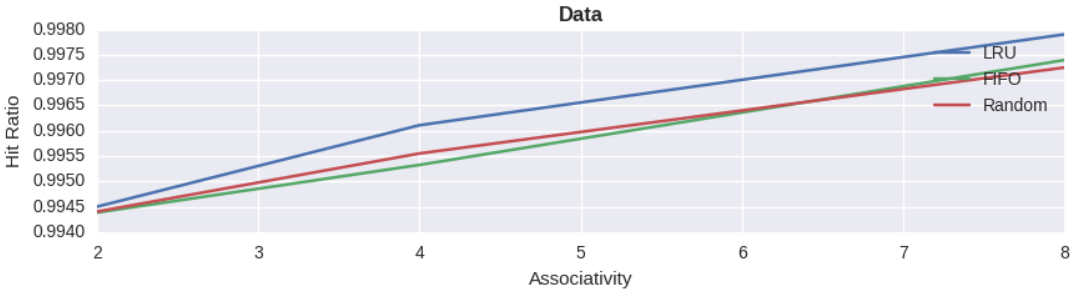
	2	4	8
LRU	0.9788223	0.9838746	0.9852634
FIFO	0.9749774	0.979185	0.9809735
Rand	0.9739554	0.9780187	0.9795311

For both the instruction cache and data cache, the performances are relatively similar. For one, the LRU replacement policy outperforms the other two at all levels of set-associative mapping. Second, there is a spike in performance at the 4-way associativity and then a smaller performance increase.

The difference in the instruction and data caches lies primarily in the actual values of the hit ratios. The instruction hit ratios are noticeably higher than the hit ratios of the corresponding replacement policies and associativities in data cache. Nature of the performances will be discussed in the *cc1* trace “Access Times” section.



	2	4	8
LRU	0.9974501	0.998306	0.9990559
FIFO	0.9974015	0.99797	0.9989154
Rand	0.9973836	0.9981578	0.9989205

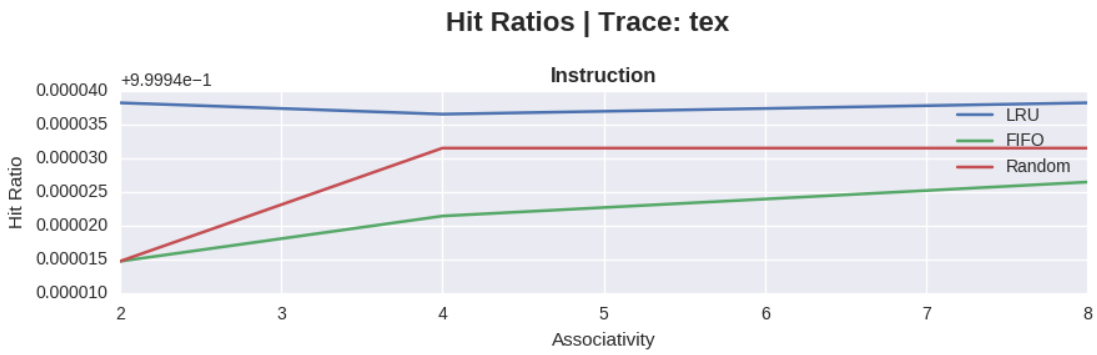


	2	4	8
LRU	0.9945037	0.9961102	0.9979009
FIFO	0.9943886	0.9953277	0.9973946
Rand	0.994407	0.9955532	0.9972472

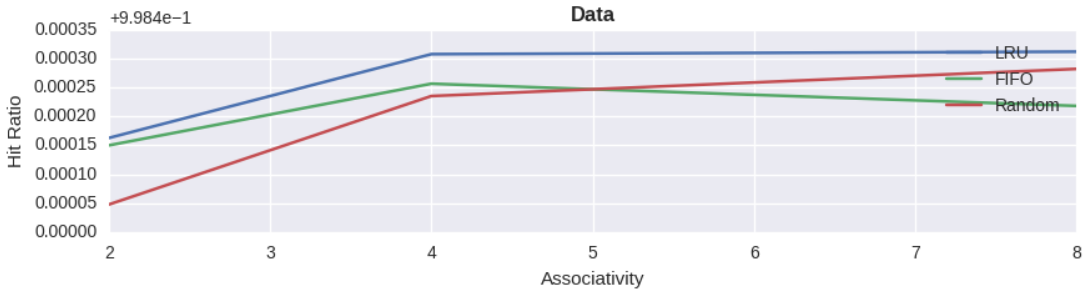
For the *spice* trace program, LRU remains the top performing replacement policy for all levels of associativity. However, the benefit of using LRU over FIFO or a random replacement policy is almost negligible for the instruction cache. In the data cache, LRU is noticeably higher, but the absolute differences are very small.

For both the instruction and data caches, there are minor differences between the FIFO and random replacement policies. In fact, FIFO turns the tide and outperforms the random replacement policy with an 8-way associativity.

In general, the hit ratios of the *spice* trace are better than those of the *cc1* trace. The potential causes will be discussed in the “Access Time” sections.



	2	4	8
LRU	0.9999782	0.9999766	0.9999782
FIFO	0.9999548	0.9999615	0.9999665
Rand	0.9999548	0.9999715	0.9999715

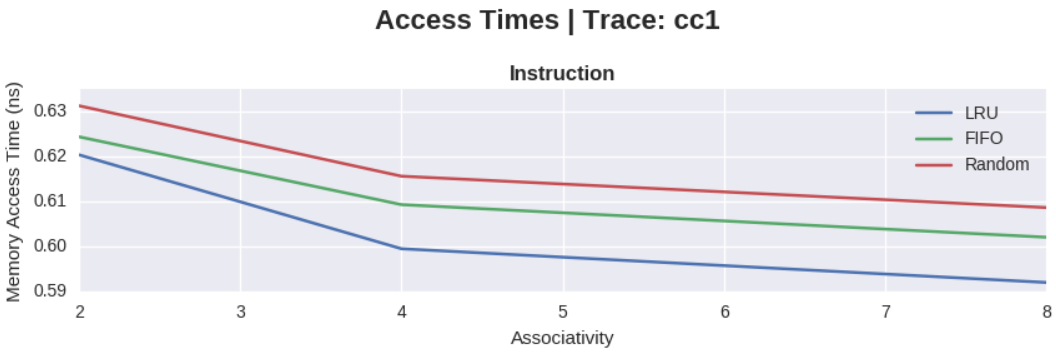


	2	4	8
LRU	0.9985627	0.9987073	0.9987116
FIFO	0.99855	0.9986563	0.998618
Rand	0.9984479	0.998635	0.9986818

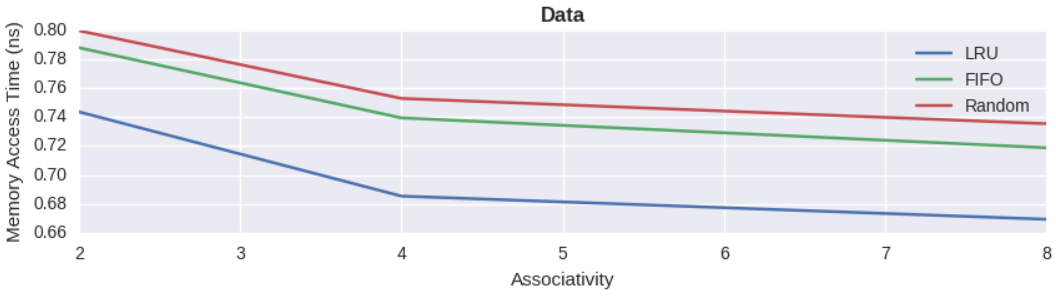
The hit ratios of the cache simulation on the *tex* trace are equally high (when compared to the other trace programs) as they were in part A. One of the things that really stands out here is the LRU curve of the instruction cache. There is very little (or no) change between different associativities. Conversely, the random and FIFO replacement show improvement as the associativity increases. Another thing to note is that, like the instruction cache of the *spice* trace and unlike the instruction cache of the *cc1* trace, the order of performance is 1) LRU, 2) random, and 3) FIFO.

There are more pronounced differences between the instruction and data caches in the *tex* trace when compared to the other two traces. The two most noticeable differences in the *tex* trace are the LRU curves and the fact that FIFO actually outperforms the random replacement policy in the 2- and 4-way associativities for the data cache. However, the random does end back up on top for the 8-way associativity. More discussion will follow in the “Access Times” section.

Part B: Access Times w/ Varying Cache Write Policies & Set-Associative Mappings



	2	4	8
LRU	0.6204147	0.5995965	0.5921256
FIFO	0.6244083	0.6093754	0.6021475
Rand	0.631287	0.6156771	0.6087225

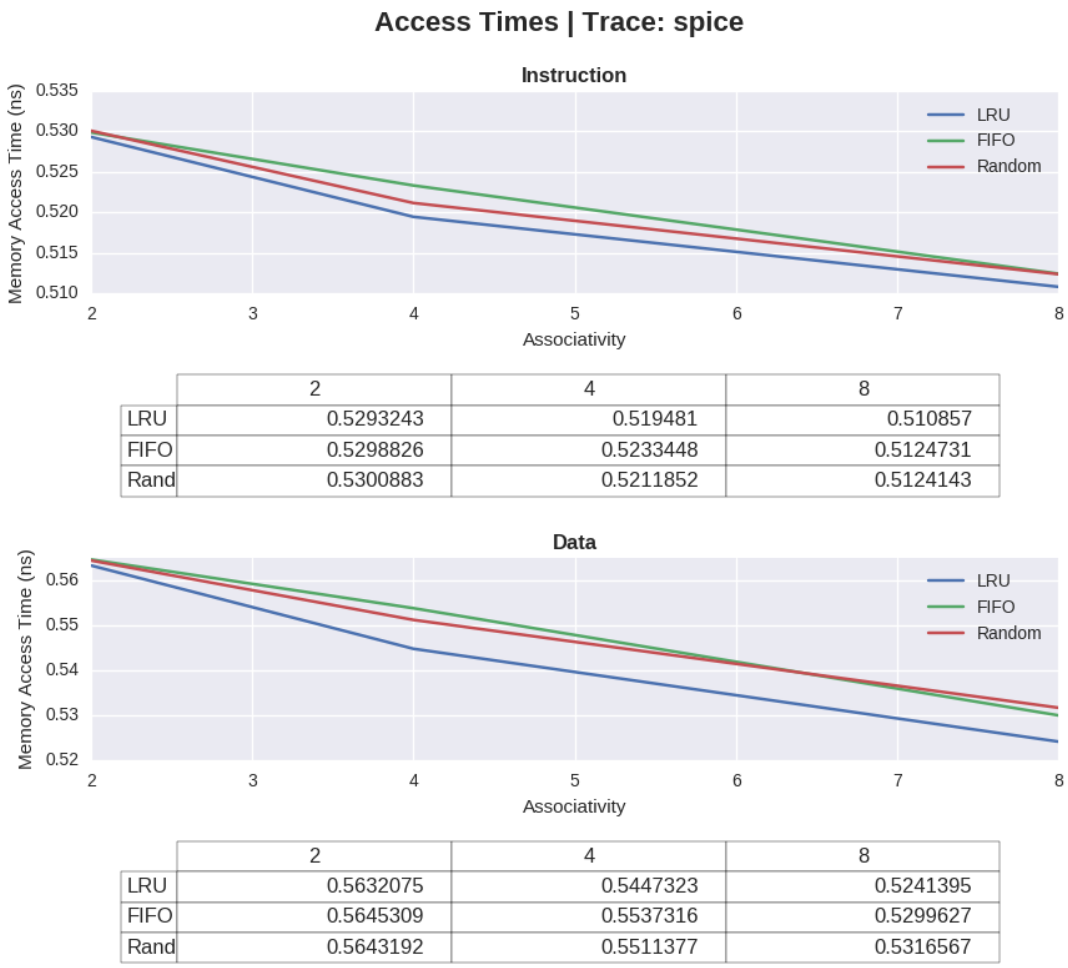


	2	4	8
LRU	0.7435435	0.6854418	0.669471
FIFO	0.7877595	0.739373	0.7188052
Rand	0.7995125	0.7527847	0.7353922

As we saw in the “Hit Ratio” section, the *cc1* trace had a stable increase in performance as the associativity increase. This could be explained by the fact that the trace provides a reasonably diverse set of instructions and operands, which leads to a higher number of replacements needed. Finding cache in the higher associativities takes a little bit longer, but there is much more flexibility in which locations can be overwritten. However, it looks like the cache does not need *too* much flexibility given that there are small differences in performance between the 4-way and 8-way associativities for both the instruction and data caches.

I am unsurprised that the random replacement policy performs worse than the LRU and FIFO policies given that both of those policies at least attempt to be intelligent in their replacement decisions. This implies that the *cc1* is somewhat repetitive in its instructions and operands.

LRU is clearly the top performer in both caches, but one might choose either 4- or 8-way associativity, depending on how important hardware simplicity is.



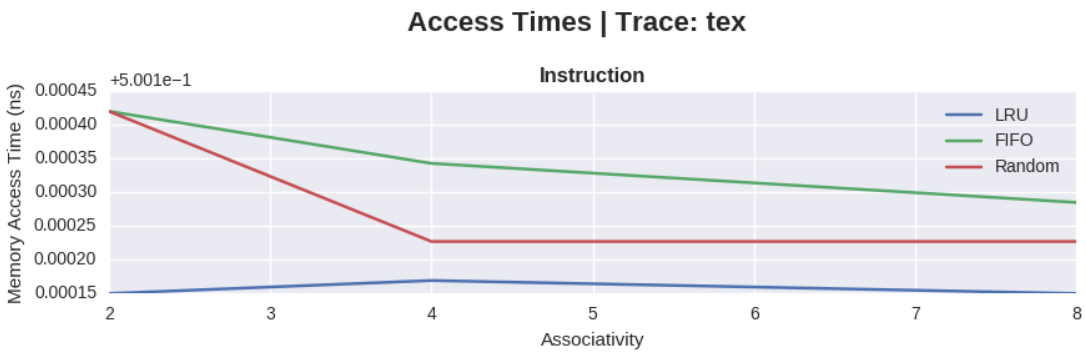
	2	4	8
LRU	0.5293243	0.519481	0.510857
FIFO	0.5298826	0.5233448	0.5124731
Rand	0.5300883	0.5211852	0.5124143

	2	4	8
LRU	0.5632075	0.5447323	0.5241395
FIFO	0.5645309	0.5537316	0.5299627
Rand	0.5643192	0.5511377	0.5316567

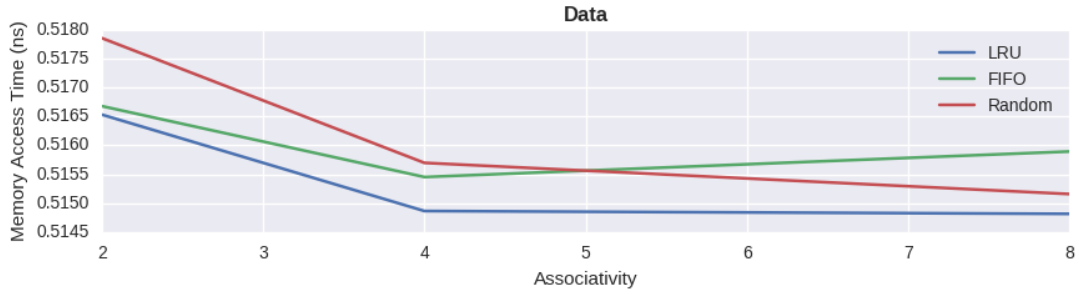
The most surprising thing about this graph is the relative equality in performance (with LRU being at least marginally better). How could it be that a random replacement policy can outperform FIFO? At least FIFO tries *something*. However, the nature of the *spice* trace program could be relatively random (even if it *varies* less than the *cc1* trace).

As with the *cc1* trace, performance improves from 2- to 4-way set-associativity. However, there is an almost linear improvement that continues from 4- to 8-way associativity - unlike the *cc1* trace program. Could it be the case that a more random instruction and operand set lends itself better to higher associativities? Another explanation is that the nature of the *spice* instruction set is such that higher associativities lead to less conflict on replacement.

In designing a system that runs only the *spice* program, the replacement policy would not make much of a difference, but the higher the associativity, the better.



	2	4	8
LRU	0.5002503	0.5002695	0.5002503
FIFO	0.5005198	0.5004428	0.5003851
Rand	0.5005198	0.5003273	0.5003273



	2	4	8
LRU	0.5165286	0.514866	0.5148171
FIFO	0.5166753	0.5154528	0.5158929
Rand	0.5178489	0.5156973	0.5151594

The thing that stands out about the instruction cache results is the 2-way + LRU variation. The 2-way associative mapping actually performs better than 4- and 8-way associativity, unlike any other cache variation in part B. This could be explained by the *tex* program having relatively few instructions (as conjectured in part A), which leads to less conflict upon replacement of a cache memory location.

For the data cache, the interesting swap in performance between FIFO and random policies from 4- to 8-way associativity (discussed in the corresponding “Hit Ratio” section) could be explained if the low flexibility of 2-way associativity lends itself better to a FIFO (somewhat intelligent) strategy, but with relatively few operands of the *tex* trace, the FIFO strategy for the more flexible 8-way associativity is essentially “over-thinking” and thus a random strategy is better. In general, the LRU is probably the best way to go in both cache types.

Part B: Conclusion

In designing an ideal system which runs all three programs, it is clear that LRU is the optimal strategy. It is also clear that 4-way is better than 2-way associativity. However, deciding between 4- and 8-way associativity could depend on the complexity tolerated by the computer designer. The *cc1* and *tex* traces show negligible improvements with the 8-way associativity, but 8-way associativity is clearly better in the *spice* trace.

Appendix:

Source code for scripts that ran tests and created visualizations and tables:

Part A (two lines change to convert hit-ratios to memory access times):

```
import os
import matplotlib.pyplot as plt
import seaborn as sns

traces      = ['cc1', 'spice', 'tex']
cache_sizes = ['4', '8', '16', '32']
block_sizes = ['16', '32', '64', '128']
table_labels = [i + 'K' for i in cache_sizes]

if __name__=='__main__':

    for trace in traces:
        data_labels = []
        fig, _ = plt.subplots(figsize=(10, 8))
        fig.suptitle('Hit Ratios | Trace: '+ trace, fontsize=16, fontweight='bold')
        itable = []
        dtable = []
        for cache_size in cache_sizes:
            sns.set_style("darkgrid")
            idata = {}
            idata_lines = []
            ddata = {}
            ddata_lines = []
            for block_size in block_sizes:
                output_file = './results/problem_A_' + trace \
                + '_' + cache_size + '_' + block_size + '.txt'
                command = './dineroIV -l1-isize ' \
                        + cache_size + 'k -l1-ibsize ' \
                        + block_size + ' -l1-dsize ' \
                        + cache_size + 'k -l1-dbsize ' \
                        + block_size + ' -informat d < ' \
                        + trace + '.din > ' + output_file
                print command
                os.system(command)

                with open(output_file, "r") as f:
                    lines = [line.strip() for line in f if line.strip()]
                    ihit_ratio = 1-float(lines[35].split()[3])/float(lines[33].split()[3])
                    dhit_ratio = 1-float(lines[49].split()[4])/float(lines[47].split()[4])
                    idata[float(block_size)] = ihit_ratio
                    ddata[float(block_size)] = dhit_ratio

            ilists = sorted(idata.items())
            x, y = zip(*ilists)
            itable.append([round(i,7) for i in y])
            plt.subplot(211)
            idata_lines += plt.plot(x, y, label=block_size)

            dlists = sorted(ddata.items())
            w, z = zip(*dlists)
            dtable.append([round(i,7) for i in z])
            plt.subplot(212)
            ddata_lines += plt.plot(w, z, label=block_size)

            print 'instruction', idata
            print 'data', ddata
            data_labels.append(cache_size+'K')

        plt.subplot(211)
        plt.table(cellText=itable,
                  rowLabels=table_labels,
```

```
                  colLabels=block_sizes,
                  bbox=[0.1, -1, 0.85, 0.6])
        plt.legend(data_labels)
        plt.xlabel('Line (Block) Size')
        plt.ylabel('Hit Ratio')
        plt.title('Instruction', fontweight='bold')
        plt.subplot(212)
        plt.table(cellText=dtable,
                  rowLabels=table_labels,
                  colLabels=block_sizes,
                  bbox=[0.1, -1, 0.85, 0.6])
        plt.legend(data_labels)
        plt.xlabel('Line (Block) Size')
        plt.ylabel('Hit Ratio')
        plt.title('Data', fontweight='bold')
        plt.subplots_adjust(hspace=1.3, wspace=1.3, bottom=.23)
        plt.savefig('../hitratios_A_'+trace+'.png')
```

Part B (two lines change to convert hit-ratios to memory access times):

```
import os
import matplotlib.pyplot as plt
import seaborn as sns

traces          = ['cc1', 'spice', 'tex']
policies        = ['l', 'f', 'r']
associativities = ['2', '4', '8']
table_labels    = ['LRU', 'FIFO', 'Rand.']

if __name__=='__main__':

    for trace in traces:
        data_labels = []
        itable = []
        dtable = []
        fig, _ = plt.subplots(figsize=(10, 8))
        fig.suptitle('Hit Ratios | Trace: '+ trace, fontsize=16, fontweight='bold')
        for policy in policies:
            sns.set_style("darkgrid")
            idata = {}
            idata_lines = []
            ddata = {}
            ddata_lines = []
            for associativity in associativities:
                output_file = './results/problem_B_' + trace \
                + '_' + policy + '_' + associativity + '.txt'

                command = './dineroIV -l1-urepl ' \
                        + policy + ' -l1-uassoc ' \
                        + associativity + ' -l1-usize 32k -l1-ubsize 128 -informat d < ' \
                        + trace + '.din > ' \
                        + output_file

                print command
                os.system(command)

                with open(output_file, "r") as f:
                    lines = [line.strip() for line in f if line.strip()]
                    print float(lines[29].split()[3])
                    print float(lines[27].split()[3])
                    ihit_ratio = 1-float(lines[29].split()[3])/float(lines[27].split()[3])
                    dhit_ratio = 1-float(lines[29].split()[4])/float(lines[27].split()[4])
                    idata[float(associativity)] = ihit_ratio
                    ddata[float(associativity)] = dhit_ratio

            ilists = sorted(idata.items())
            x, y = zip(*ilists)
```

```
            itable.append([round(i,7) for i in y])
            plt.subplot(211)
            plt.plot(x, y, label=associativity)

            dlists = sorted(ddata.items())
            w, z = zip(*dlists)
            dtable.append([round(i,7) for i in z])
            plt.subplot(212)
            plt.plot(w, z, label=associativity)

            print 'instruction', idata
            print 'data', ddata
            if policy=='f': data_labels.append('FIFO')
            elif policy=='l': data_labels.append('LRU')
            elif policy=='r': data_labels.append('Random')

        plt.subplot(211)
        plt.legend(data_labels)
        plt.table(cellText=itable,
                  rowLabels=table_labels,
                  colLabels=associativities,
                  bbox=[0.1, -1, 0.85, 0.6])
        plt.xlabel('Associativity')
        plt.ylabel('Hit Ratio')
        plt.title('Instruction', fontweight='bold')
        plt.subplot(212)
        plt.legend(data_labels)
        plt.table(cellText=dtable,
                  rowLabels=table_labels,
                  colLabels=associativities,
                  bbox=[0.1, -1, 0.85, 0.6])
        plt.xlabel('Associativity')
        plt.ylabel('Hit Ratio')
        plt.title('Data', fontweight='bold')
        plt.subplots_adjust(hspace=1.3, wspace=1.3, bottom=.23)
        plt.savefig('../hitratios_B_'+trace+'.png')
```