2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks

Smaller split L-1 data caches for multi-core processing systems

Oluwayomi Adamo, Afrin Naz, Tommy Janjusic and Krishna Kavi
Department of Computer Science and Engineering
University of North Texas, Texas, USA
oluwayomi.adamo@unt.edu, afrin.naz@unt.edu, tommy.janjusic@gmail.com, kavi@cse.unt.edu

Chung-Ping Chung
Department of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu, Taiwan.
cpchung@csie.nctu.edu.tw

Abstract— As more cores (processing elements) are included in a single chip, it is likely that the sizes of per core L-1 caches will become smaller while more cores will share L-2 cache resources. It becomes more critical to improve the use of L-1 caches and minimize sharing conflicts for L-2 caches. In our prior work we have shown that using smaller but separate L-1 array data and L-1 scalar data cache, instead of a larger single L-1 data cache, can lead to significant performance improvements. In this paper we will extend our experiments by varying cache design parameters including block size, associativity and number of sets for L-1 array and L-1 scalar caches. We will also present the affect of separate array and scalar caches on the non-uniform accesses to different (L-1) cache sets exhibited while using a single (L-1) data cache. For this purpose we use third and fourth central moments (skewness and kurtosis), which characterize the access patterns. Our experiments show that for several embedded benchmarks (from MiBench) split data caches significantly mitigate the problem of non-uniform accesses to cache sets (leading to more uniform utilization of cache resources, reduction of conflicts to cache sets, and minimizing hot spots in cache). They also show that neither higher set-associativities nor large block sizes are necessary with split cache organizations.

Keywords - Cache memories, Split data cache, uniform cache access patterns.

I. INTRODUCTION

Existing cache organization suffers from the inability to distinguish different types of localities rather than making any attempt to take special advantage of the locality type. This causes unnecessary movement of data among the levels of the memory hierarchy, significant interference between unrelated data inside the cache, removal of potentially useful data causing cache pollution, unnecessary increases in miss ratio and memory access times. At the same time, because of non-uniformity in memory access pattern, some cache sets are accessed heavily, while others remain underutilized. In order to solve this problem, in our previous work [1, 2], we have proposed Split Data cache architecture, in which the memory accesses are grouped as scalar or array references according to their inherent locality and each group subsequently mapped to a dedicated cache partition, equipped with architectural constructs built to exploit that particular locality type. In this system, since the scalar references and array references are no longer negatively affecting each other, cache interference, thrashing and pollution problems are diminished, delivering better performance. In our design, not only both caches designed more optimally according to their specific needs, it will simplify some other general issues and concerns in cache design, such as the associativity, cache block size or cache capacity. The selection of proper block size or associativity to maximize performance while staying within the cost are the hardest choices in designing cache memories. In case of embedded systems, total cache size is also a big concern. By partitioning the cache, our cache system can implement different configurations exploiting different cache parameters more selectively and effectively. The "array cache" is a direct mapped cache with small stream buffer to exploit spatial localities more aggressively by (pre)fetching multiple neighboring small blocks on a cache miss. Whereas the "scalar cache" is a 2-way (or 4-way) set associative cache with smaller block sizes to exploit temporal locality. The combination of different block sizes and associativities together with partitioned cache architectures provides an effective solution for alleviating the existing problems in cache designs and maximizes the effective cache memory space for any given cache size and cost. Since significant amounts of compulsory and conflict misses are avoided, the size of each cache (i.e., array and scalar), as well as the combined cache capacity can be reduced. In this work we performed comprehensive analysis of cache miss rates by including different combinations of cache size, block size and associativity. We also report on the frequency of accesses to different cache sets by using third and fourth central moments (skewness and kurtosis). In this work we have shown that use of separate L-1 array data and L-1 scalar data cache can lead to significant decrease in cache size and number of misses. In this paper we also show that using smaller array and scalar caches significantly mitigate the problem for embedded benchmarks in terms of improving uniformity of accesses to cache sets.

The rest of the paper is organized as follows. To motivate the reader, in Section 2 we discuss related issues and performance metrics in more detail. Section 3 describes benchmarks and experimental set up used in our evaluation, while section 4 presents the results. We present our conclusions in section 6.

II. CONCEPTS

In this section, we first briefly introduce issues in general cache design. Then we will demonstrate how to examine cache sets' usage during a program's execution. After that we will describe related statistical concepts. Finally we

978-0-7695-3908-9/09 $26.00 © 2009 IEEE
DOI 10.1109/I-SPAN.2009.118

will briefly describe our split data cache architecture.

A. *ABC's of Cache*

For a cache, its performance is dictated by a number of parameters, including *Associativity, Block size and Cache size*. Our work is motivated by the observation that it is not possible to design a single cache that works well for different types of localities and data types. We propose multiple data caches designed with different parameters to meet the needs of the different data types.

1) Cache Size: Increasing cache size will reduce capacity misses; however as cache size increases, a capacity miss will become a conflict miss [6]. On the other hand, Jouppi [7] reported that for stream data type, increasing cache capacity actually increases cold-start or compulsory misses.

2) Block Size: The selection of block size depends on the needs of the different data types. Increasing block size also implies prefetching of data for applications exhibiting greater spatial localities, such as the array references. For scalar references, it is better to have smaller cache block sizes and more cache lines to eliminate conflict misses and even capacity misses when smaller caches are used [6].

3) Associativity: Direct mapped caches are simpler, easier to design. The main disadvantage of a direct mapped cache is the high conflict miss ratetypically 40% of direct-mapped cache misses [7]. Conversely for caches with higher associativity the main advantage is lower miss rate, but they are more expensive and incur longer access times on hit.

More information about different cache parameters can be found in [6].

B. *Non-Uniform Accesses to Cache Sets*

Zhang [3, 4] reported that with direct mapped L-1 caches not all cache sets are equally accessed and the heavily accessed sets lead to most of the conflict misses and thus to poor performance. Zhang [3, 4] classified cache sets as frequent hit sets (FHS) and frequently missed sets (FMS) if the number of hits and misses are more than twice the average and least accessed sets (LAS) if the accesses are one half of the average accesses. In "unpublished"[1] we repeated Zang's experiments with a subset of SPEC benchmarks, some bio-informatics and embedded benchmarks (from MiBench suite). In order to more formally describe the behavior of cache access patterns, in this work we will convert the accesses and misses into probability distributions. We will then measure various statistical values knows as central-moments. Most commonly used moments are: mean (first moment) and standard-deviation (second moment). Higher moments describe the shape of the distribution. The shape of a uniform access distribution will have a flat shape compared to a normal distribution with a few values clustered around the mean and long tails. We will report skewness and kurtosis values associated with (data) cache access patterns. In order to be self contained, we will describe these statistical parameters and their value to our analyses.

1) Skewness: Skewness (third central moment) is a measure of symmetry, or more precisely, the lack of symmetry.

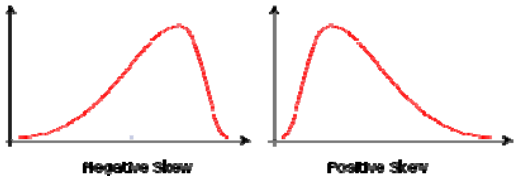

Figure 1: Positive and negative skewness

A distribution, or data set, is symmetric if it looks the same to the left and right of the center point (mean). If the left tail is more pronounced than the right tail, the function is said to have negative skewness. If the reverse is true, it has positive skewness. If the two are equal, it has zero skewness.

*2) Kurtosis:*Kurtosis (fourth central moment) is a measure of whether the data are peaked or flat relative to a normal distribution. That is, data sets with high Kurtosis tend to have distinct peaks near the mean, decline rather rapidly, and have long tails. This also indicates very few values near the peak. Data sets with low Kurtosis tend to have a flat top near the mean rather than a sharp peak. A uniform distribution would be the extreme case (with zero Kurtosis). For our pupose, a highly non-uniform behavior results in a high Kurtosis, while a more uniform access behavior leads to lower Kurtosis.

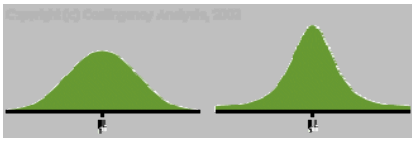

Figure 2: Kurtosis values

Fig. 3 shows distributions associated with cache hits and misses to different sets. We show the distribution with a single 64 sets of 32Byte unified data cache, and for 32 sets of array and 32 sets of scalar data caches (using our split

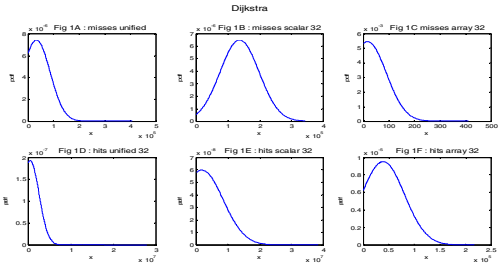

Figure 3: Distribution of cache accesses

data caches), for benchmark dijkstra (from Mibench). The main goal of this figure is to illustrate the importance of the

shape of the accesses and misses, when the accesses are converted to probability distributions.

C. Split Cache Design

Fig 4 shows our Split Data cache architecture, with array and scalar data caches. Our split data cache

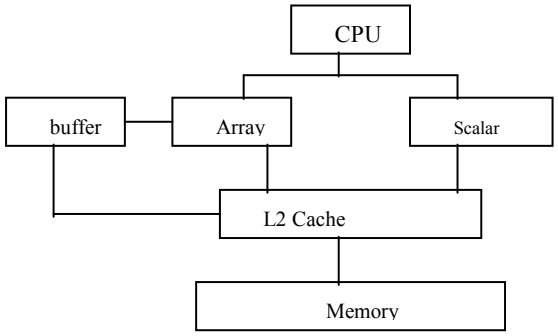

Figure 4: split data cache

architecture consists of an "array cache" and a "scalar cache". Memory accesses are grouped as scalar or array references according to their inherent locality and each data group is mapped to a dedicated cache partition. Our array cache is also equipped with a small 10-line stream prefetching buffer. The stream buffer is a fully associative, FIFO buffer specially designed to support direct-mapped cache through hardware based prefetching [7]. A miss induces the fetching of the missed block along with next block stored in the buffer. Our intent is to use the stream buffer for prefetched blocks and avoid cache pollution (premature data displacement). In this system, since scalar references and stream references no longer negatively affected each other, cache interference, thrashing and pollution problems will be diminished, delivering better performance. Specially for array cache, since there is no more contamination of scalar data the stream buffer will provide significant decrease in cold misses.

III. SIMULATION ENVIRNMENT

The descriptions of the benchmarks used in our studies are listed in Table 1.We use selected benchmark programs from the MiBench suite [9]. MiBench includes benchmarks from

TABLE 1: Descriptions of benchmarks

Benchmarks	Description	Name
bit counts	Test bit manipulation	bc
qsort	Computational Chemistry	qs
dijkstra	Shortest path problem	dj
blowfish	Encription/decription	bf
AES	Advanced Encryption Standard	AE
CRC	Cyclic Redundancy Check	CR
String search	Search mechanism	ss

several representative embedded application domains. In this paper we included selected programs from these application domains: (1) Automotive and Industrial Control, (2) Office Automation, (3) Networking, (4) Security, (5) Consumer and (6) Telecommunications.

Our experimental environment builds on the SimpleScalar (version 3.0d) simulation tool set [5] modeling an out-of-order speculative processor with a two-level cache hierarchy. We rely on default parameters defined by SimpleScalar [5].

IV. RESULTS

The next subsection presents total references and misses. Then we show the values of third and fourth central moments (skewness and kurtosis) to study non-uniform access to cache sets.

A. Results with different parameters

By changing the block size, cache capacity and associativity, attempt is made to obtain the best configurations for array and scalar caches.

1) Selection of Cache size

Several experiments are performed to determine the optimum cache size for each data type and the results for each benchmark are shown in TABLE 2. This table shows the number of references and misses with increasing cache sizes for unified cache and for split cache (each direct

TABLE 2: Number of misses with different cache sizes (Direct Mapped)

name	size	Unified		Array		Scalar	
		Total Ref.	Total Miss	Total Ref.	Total Miss	Total Ref.	Total Miss
AE	8K	133927460	8392267	77551107	16	55977972	3448827
	4K	133927597	15893061	77377863	29	56187955	10224831
	2K	133913470	37844548	78162635	7093	55512769	15972202
	1K	133903267	50335956	79229364	20536	54480746	19787067
	512	133890217	57455389	79518950	2657048	54209594	23439879
dik	8K	81701808	4630365	1491373	25	80237588	968410
	4K	81696527	7383228	1489128	48	80226623	2503228
	2K	81667486	13695622	1231050	207	80463571	7002213
	1K	81647846	18529657	1192468	1184	80505698	12323751
	512	81659148	23944713	1129962	8561	80559269	14993876
bf	8K	109796377	2947143	40587790	13	69121381	1192624
	4K	109790778	5923918	40599233	31	69105144	9945644
	2K	109785876	19574058	41030991	2424	68677410	20406296
	1K	109776066	27166084	41172494	5659	68533133	26199914
	512	109756413	31925415	41206622	28101	68490653	29701443
bc	8K	5043729	2076	607732	10	4436091	330
	4K	5043900	2125	607721	16	4436083	437
	2K	5043225	2376	607578	25	4435896	662
	1K	5043471	11667	607557	46	4435973	1236
	512	5044080	321924	607661	117	4435961	1783
CR	8K	186797013	1989776	133058204	13	53745373	166
	4K	186803525	2288668	133058201	22	53745373	177
	2K	186803526	5873674	133006233	26018	53797339	230
	1K	186803833	9509752	132954272	52024	53849327	297
	512	186803982	23073115	132850321	104053	53953267	432
ss	8K	1116726	26131	635129	21	483129	14044
	4K	1121016	66634	634226	45	484718	39032
	2K	1120220	113937	634405	93	485622	44493
	1K	1120722	198475	635142	216	486710	78177
	512	1121665	330717	634353	3815	489159	102722
qs	8K	137307939	5969335	47823891	2655	89477832	4975916
	4K	137315853	6946743	47813528	7849	89495840	5982930
	2K	137326611	8698193	47424282	25464	89918603	8246678
	1K	137507348	11065903	46859407	66274	90557423	10992870
	512	137534354	15252322	46961945	165054	90786913	14323730

mapped and with 8 bytes block size). Here it should be mentioned that the total size of Array and Scalar cache is

equal to the size of unified cache. Hence for the row with cache size 8K means that we have 4k-array and 4k Scalar caches. From TABLE 2 we can see that our split data cache has significantly decreased the number of misses for all cache sizes for all benchmarks except for two cache sizes 4k and 2k for the benchmark "blowfish". If we compare results 8k uniformed cache, 4k-array and 4k Scalar caches), we see 58.90%, 79.08%, 59.53%, 83.62%, 99.99%, 46.10% and 16.59% reductions in misses for the benchmarks "AES", "dijkstra", "blowfish", "bitcount", "CRC", "stringsearch" and "qsort" respectively. For both unified and split caches we see the gradual decrease misses as we increase cache sizes. As mentioned in section 2, an important criterion for selecting cache size is the frequency of capacity misses. We expect that when separate scalar and array caches are used, the scalar cache can be small since the number of capacity misses is small with scalar data items. We also expect that using a small stream buffer with array cache will allow us to significantly reduce the cache size of array cache. For two benchmarks, "bc" and "CRC", the number of capacity misses are so low that a tiny 256 byte scalar cache is providing better performance (95% better for "CRC") than 8k unified cache. For other benchmarks (except "qsort" and "blowfish") 1k scalar cache and 512 byte array cache (with a 10-line stream buffer) provide better results than 8k unified cache. For "blowfish" a large 4k scalar cache is needed with a smaller array cache. And t"qsort", 8k unified cache.

2) Selection of Block size

TABLE 3 shows the number of references and misses with increasing block sizes usingn 8k unified cache and 8k split cache (4k-array and 4k Scalar caches). From TABLE 3 we can see that our split data cache has significant decrease in number of misses for all block sizes for all benchmarks.

TABLE 3: Number of misses with different block sizes

Name	SIZE	Unified		Array		Scalar	
		Total Ref.	Total Miss	Total Ref.	Total Miss	Total Ref.	Total Miss
AE	32	133910390	8290178	77534341	21	56031082	3081563
	16	133921183	8239429	77432837	18	56108021	2933398
	8	133927460	8392267	77551107	16	55977972	3448827
dik	32	81699450	1617205	1508783	20	80207908	805859
	16	81700436	2768324	1508683	27	80210176	856485
	8	81701808	4630365	1491373	25	80237588	968410
bf	32	109797005	3301215	41507287	16	68206865	1393702
	16	109796375	3560287	39883295	16	69832598	1415416
	8	109796377	2947143	40587790	13	69121381	1192624
bc	32	5043447	660	607679	9	4435988	196
	16	5043405	1123	607581	9	4435925	236
	8	5043729	2076	607732	10	4436091	330
CR	32	186797011	1312730	133058201	16	53745378	88
	16	186797012	1338548	133058203	16	53745371	116
	8	186797013	1989776	133058204	13	53745373	166
ss	32	1116059	13071	635188	25	482724	7912
	16	1116328	16927	635080	25	483039	10243
	8	1116726	26131	635129	21	483129	14044
qs	32	137296899	2105032	48459615	4094	88852271	1822313
	16	137301183	3211978	48125758	3160	89221813	2919856
	8	137307939	5969335	47823891	2655	89477832	4975916

Array and scalar data items exhibit different localities, requiring different block sizes. For array data, the simplest way to reduce miss rate is to use large block sizes. However in a fixed sized cache increasing block size will decrease the number of lines, leading to an increase in conflict and capacity misses, which is harmful for scalar data. In our architecture we can take advantage of both techniques - by using larger cache blocks for array caches and smaller block sizes for scalar cache. work [2] For this work, as we are using stream buffer with our array cache we increase the block size.

unified cache, for all benchmarks (except AES) changing block size has significant effect on number of misses. Whereas for scalar cache, only two benchmarks, "qsort" and "stringsearch", showed similar behavior.

3) Selection of associativity

TABLE 4 shows the number of references and misses with increasing cache sizes in unified cache and split cache with 2-way set associativity and 8 bytes block size. In our test suite, after removing array references, for our scalar cache, conflict misses are the main concern. As a result increasing the associativity to 2-way leads to significant

TABLE 4: Number of misses with different cache sizes (2-way)

Nname	size	Unified		Array		Scalar	
		Total Ref.	Total Miss	Total Ref.	Total Miss	Total Ref.	Total Miss
AE	8K	133713457	987382	77848529	16	55831952	1805293
	4K	133717655	2696452	77389542	29	56261928	4729790
	2K	133728044	14677571	77560782	3217	56089551	10957994
dik	8K	81722379	2884265	1524280	16	80209668	301108
	4K	81721479	3940987	1498052	27	80235733	849784
	2K	81708055	6501822	1466076	87	80262740	2166106
bf	8K	109717681	35830	42516551	13	67197858	25267
	4K	109717724	1370778	41047216	24	68663526	1260416
	2K	109707348	3945244	40516095	3212	69189954	12661708
bc	8K	5043878	1957	607494	10	4435784	538
	4K	5043539	1976	607592	16	4435896	417
	2K	5043737	2030	607796	25	4436160	429
CR	8K	186803516	2400	133058219	11	53745335	437
	4K	186803520	2428	133058210	20	53745358	302
	2K	186803530	2022743	133006234	26016	53797345	221
SS	8K	1117744	9980	633596	18	483288	7012
	4K	1118454	15056	634794	34	483338	13545
	2K	1119473	30948	634895	69	483530	15663
qs	8K	137288564	4782955	47864653	2468	89433832	4275591
	4K	137291461	5490387	47819079	6372	89480927	4854223
	2K	137295563	6467058	47764804	23609	89548311	5441755

improvement. If we compare 4k direct mapped scalar cache (last column of TABLE 2) with 4k 2-way set associative scalar cache (last column of TABLE 4) we see 47.65%, 68.90%, 97.88%, 50.07% and 14.07% decrease in misses for the benchmarks "AES", "dijkstra", "blowfish", "stringsearch" and "qsort" respectively. However for two benchmarks, "CRC" and "bitcount" very unusual result is obtained. With 8k unified cache, 4k-array and 4k Scalar caches there is actually an increase in the number of misses with 2-way cache when compared to a direct mapped cache. Because of the lack of temporal locality, the stream references will cause more compulsory misses than conflict misses and direct mapping with prefetching will be the better option for an array cache.

B. Statistical Analysis

In this section, we are going to perform statistical analysis to more carefully analyze the benchmarks.

1) Selection of Cache size

TABLE 5 shows the skewness and kurtosis values with increasing cache sizes for unified cache and for split cache (each direct mapped and with 8 bytes block size). From the values presented in TABLE 5 for smaller caches, scalar

TABLE 5: Skewness and Kurtosis values with different cache sizes (Direct Mapped)

Name	Cache size	Unified		Array		Scalar	
		Skewness	Kurtosis	Skewness	Kurtosis	Skewness	Kurtosis
AES	8K	4.14	34.04	16.34	316.07	18.95	396.03
	4K	3.41	19.82	13.08	191.42	7.27	83.99
	2K	1.66	5.24	9.40	96.09	2.14	10.35
	1K	0.74	2.65	5.56	35.23	1.24	4.84
	512	0.97	4.04	2.61	9.67	0.69	2.55
dijk	8K	16.96	324.10	12.28	175.58	6.17	62.0
	4K	4.24	29.99	12.81	186.86	3.61	23.49
	2K	0.94	4.18	9.34	94.15	2.75	11.54
	1K	0.34	2.19	5.60	33.21	1.53	5.55
	512	0.08	1.91	1.65	4.52	3.12	11.18
bf	8K	0.33	1.78	17.83	360.09	6.35	47.62
	4K	18.67	390.47	14.51	223.21	2.79	9.08
	2K	12.08	172.50	11.18	126.00	2.20	6.79
	1K	7.41	70.94	7.702	60.83	1.42	3.50
	512	4.38	28.58	4.69	24.74	0.56	1.64
bc	8K	0.33	1.78	17.83	357.29	6.57	67.97
	4K	4.36	27.79	13.54	201.89	3.87	19.11
	2K	5.44	38.18	10.37	113.80	2.26	8.21
	1K	11.17	125.96	7.49	58.73	2.12	8.62
	512	5.38	30.04	4.60	24.08	1.21	4.27
CR	8K	6.91	106.39	19.71	416.44	5.37	44.26
	4K	6.30	62.65	14.84	230.42	5.02	34.80
	2K	7.60	66.33	11.18	126.00	3.98	22.21
	1K	6.29	40.62	7.81	62.01	3.46	16.25
	512	4.28	19.37	2.78	8.77	5.38	30.03
SS	8K	25.18	736.84	20.19	436.30	3.00	16.14
	4K	6.28	42.83	15.44	243.96	10.02	112.64
	2K	5.25	36.06	10.96	122.78	6.84	53.94
	1K	2.32	8.76	7.54	59.12	2.75	9.99
	512	1.53	4.61	1.17	2.95	3.56	17.57
qs	8K	2.02	11.96	16.46	291.93	2.02	8.22
	4K	1.66	6.98	12.47	168.57	10.30	114.38
	2K	9.43	97.18	9.54	98.11	4.31	20.45
	1K	3.79	16.38	6.54	46.97	1.63	4.06
	512	1.45	3.68	0.89	3.02	1.29	4.36

cache portion of split data caches show better uniformity (smaller kurtosis and skewness), except rAES. For most of the benchmarks, array cache portion of the split cache also show better uniformity when using smaller caches. For several applications, array caches do not appear to be very useful. The advantage of split caches, in terms of uniformity, disappears with larger caches.

2) Selection of Block size

TABLE 6 shows the skewness and kurtosis values with increasing block sizes using 8k unified cache and 8k split cache (4k-array and 4k Scalar caches). For most benchmarks and block sizes (except all block sizes for "AES" and block 8 size for benchmarks "blowfish" and "bitcount") the scalar data shows more uniformity resulted in lower kurtosis values. In some benchmarks ("dijkstra", "CRC", "stringsearch" and "qsort") we have huge reductions in kurtosis values for scalar caches. For array cache, again for most benchmarks (except all block sizes for "AES" and "bitcount", block 32 for "dijkstra", block 8 for benchmarks "blowfish" and "qsort") we have considerable decrease in kurtosis values.

TABLE 6: Skewness and Kurtosis values with different block sizes

Name	Block Size	Unified		Array		Scalar	
		Skewness	Kurtosis	Skewness	Kurtosis	Skewness	Kurtosis
AES	32	4.77	30.87	10.48	115.3	7.69	69.44
	16	7.55	74.92	12.68	182.1	13.17	191.82
	8	4.14	34.04	16.34	316.0	18.95	396.03
dik	32	9.04	94.03	10.56	116.5	3.2	18.2
	16	11.83	156.03	11.26	144.6	5.44	47.36
	8	16.96	324.10	12.28	175.5	6.17	62.06
bf	32	15.75	250.70	9.97	107.4	7.03	59.71
	16	22.26	500.84	14.18	215.9	6.52	54.26
	8	0.33	1.78	17.83	360.1	6.35	47.62
bc	32	5.25	36.63	9.21	94.42	3.44	16.33
	16	4.98	38.43	13.11	190.1	5.34	39.93
	8	0.33	1.78	17.83	357.2	6.57	67.97
CR	32	13.63	205.98	10.90	121.7	6.57	49.07
	16	14.44	275.19	15.51	245.2	5.56	40.40
	8	20.04	414.42	19.71	416.4	5.37	44.26
ss	32	12.48	179.70	10.53	116.1	1.94	6.05
	16	18.17	376.43	14.96	233.4	2.54	9.86
	8	25.18	736.84	20.19	436.3	3.00	16.14
qs	32	13.56	197.36	11.14		6.70	49.87
	16	19.71	423.02	15.20	237.5	13.97	212.71
	8	2.02	11.96	16.46	291.9	2.02	8.22

3) Selection of associativity

TABLE 7 shows the the skewness and kurtosis values with increasing cache sizes in unified cache and split cache, each with 2-way set associativity and 8 bytes block size. If

TABLE 7: Skewness and Kurtosis values with different cache sizes (2-way)

Name	size	Unified		Array		Scalar	
		Skewness	Kurtosis	Skewness	Kurtosis	Skewness	Kurtosis
AES	8K	6.45	64.27	16.34	316.0	14.36	270.21
	4K	17.16	350.28	14.18	216.5	3.80	30.96
	2K	2.43	13.38	11.18	126.0	3.53	22.07
dik	8K	1.52	18.02	15.00	261.2	12.88	182.62
	4K	10.60	144.00	14.05	212.4	7.61	70.21
	2K	3.10	15.87	10.59	116.6	3.65	18.92
bf	8K	0.33	1.78	17.83	360.1	7.27	64.77
	4K	13.77	198.88	13.82	208.7	2.67	8.33
	2K	4.16	25.33	11.18	126.0	1.53	3.49
bc	8K	0.33	1.78	17.83	357.3	0.88	4.62
	4K	0.99	3.77	13.54	201.9	1.18	6.11
	2K	0.64	2.94	10.37	113.8	3.23	15.88
CR	8K	0.33	1.78	18.14	362.7	1.28	7.28
	4K	2.95	20.73	14.50	222.7	2.52	14.91
	2K	-0.11	1.35	11.18	126.0	4.28	25.65
SS	8K	4.48	33.17	17.20	343.2	2.21	8.41
	4K	2.93	14.58	15.26	240.3	4.86	37.28
	2K	13.60	204.20	10.93	122.4	4.62	26.17
qs	8K	1.56	4.02	16.76	301.3	1.57	6.387
	4K	1.36	3.78	12.12	159.6	1.42	7.64
	2K	0.53	1.72	9.60	99.33	3.00	16.34

we compare last column of TABLE 5 and last column of TABLE 7, we can say that for scalar caches higher associativity improves uniformitity. On the other hand, for Array caches, if we compare the kurtosis values for direct mapped and 2-way (sixth column of TABLE 5 and sixth column of TABLE 7) we can see that direct mapping is definitely the better option with more uniformity.

V. CONCLUSION

The goal of a computer architect is to maximize performance while staying within the cost and power

constraints. It is difficult to achieve a compromised cache design that works with data exhibiting conflicting behaviors. This work shows that using separate (data) caches for array data and scalar data items, we can separate these concerns and design caches that achieve optimal performance for different data items. Another significant achievement of this work is the ability to include prefetching into embedded systems. While traditional prefetching techniques have been explored [6], (premature) prefetching can adversely affect performance if it leads to cache pollution by displacing needed data in an untimely manner. This is the primary reason for not using pre-fetching in embedded systems. However we show that a carefully designed cache system not only solves the deficiencies of general prefetching, it also solves the problems of stream buffers. Jouppi's analysis [7] included a stream buffer for a unified data cache, and the buffer was flushed every time a scalar data is accessed (since stream buffers assume contiguous data items). In our study, because we are removing scalar data from array caches, stream buffers associated with array cache are flushed less frequently and provide a decrease in the number of misses in the array cache.

As more cores (processing elements) are included in a single chip, it is likely that the sizes of per core L-1 caches will become smaller and it becomes more critical to improve the use of L-1 caches. In this work we have shown that using smaller but separate L-1 array data and L-1 scalar data cache, instead of a larger single L-1 data cache, can lead to significant performance improvements. We have not only achieved significant reduction in cache size, number of misses, we also have showed that as we reduce the cache sizes the uniformity in cache access pattern improves significantly. However, we do not claim that split data caches completely solve the non-uniformity of cache accesses. We contend that different applications need different approaches to solve the non-uniform accesses. In some cases our split-caches are adequate. In some cases profiling and compile time analyses may be adequate to relocate data that maps to highly utilized sets. Currently we are exploring how profiling and compile time analyses can be used to uniformly distribute data among all cache sets.

ACKNOWLEDGMENT

This research is supported in part by the Net-Centric IUCRC industrial memberships and in part by NSF funds for the IUCRC.

REFERENCES

[1] A. Naz, A. Adamo, K. Kavi and T. Janjusic, "Improving Uniformity of Cache Access Pattern using Split Data Caches", Proc. *19th ISCA Parallel and Distributed Computing Systems,* San Francisco Sept 2006.

[2] A. Naz, K.M. Kavi, P.H. Sweany and M. Rezaei, "A study of separate array and scalar caches", Proc. *18th International Symposium on High Performance Computing Systems and Applications (HPCS 2004)*, Winnipeg, Manitoba, Canada, May 2004.

[3] C. Zhang, "Reducing Cache Misses Through Programmable Decoders", ACM Transactions on Architecture and Code Optimization, Vol. 4, No. 4, Article 24, January 2008.

[4] C. Zhang, "Balanced cache: Reducing conflict misses of direct-mapped caches through programmable decoders", Proc. *International Symposium on Computer Architecture*, June , 2006.

[5] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0", *Tech. Rep. CS-1342*, University of Wisconsin-Madison, June 1997.

[6] J. L. Hennessy and D. A. Patterson, Computer Architecture A Quantitative Approach, Morgan Kaufmann Publishers, Third Edition 2003, pp 423-430.

[7] N. P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully Associative Cache and Prefetch Buffers," Proc. *17th ISCA*, May 1990, pp. 364-373.

[8] http://www.mathworks.com/

[9] M. Guthaus, J. Ringenberg, T. Austin, T. Mudge, and R. Brown, "MiBench: A free, commercially representative embedded benchmark suite", Proc. *IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, Dec. 2001.