Improving Uniformity of Cache Access Pattern using Split Data Caches

Afrin Naz
University of North Texas Dallas

Oluwayomi Adamo, Krishna Kavi[1] and Tomislav Janjusic
University of North Texas

Abstract

In this paper we show that partitioning data cache into array and scalar caches can improve cache access pattern without having to remap data, while maintaining the constant access time of a direct-mapped cache and improving the performance of L-1 cache memories. By using 4 central moments (mean, standard-deviation, skewness and kurtosis) we report on the frequency of accesses to cache sets and show that split data caches significantly mitigate the problem of non-uniform accesses to cache sets for several embedded benchmarks (from MiBench) and some SPEC benchmarks.

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

1. Introduction

In this paper, we investigate methods for improving hit rates in the first level of memory hierarchy and show that the inclusion of partitioned data cache architectures provides an effective solution for alleviating existing problems in cache designs by creating better footprint through more uniform cache access behavior.

The design of a first level cache always involves fundamental tradeoffs between miss rates and access times. If the cache sets are accessed in a balanced manner, meaning all sets are accessed with equal frequency, then the cache misses can be reduced significantly without increasing cache access time. However as memory accesses in a program are not uniformly distributed, some cache sets will be accessed heavily, while others remain underutilized. Direct-mapped caches exhibit faster access time, but poor hit rates, compared with same sized set-associative caches because of non-uniform accesses to the cache sets. Although caches with higher associativity exhibit improvement in cache access behavior, most of the sets still remain underutilized. In order to access all sets more evenly, in the recent past researchers have proposed variations that includes multiple mapping functions; whereby data mapped to heavily utilized sets are remapped to underutilized sets. However these approaches result in hardware overhead, and extra cycles to locate these relocated cache sets. In this paper we show that our split array and scalar caches can improve cache access pattern without having to remap data, while maintaining the constant access time of a direct-mapped cache.

In our previous work we have shown that split data cache architectures [1,2] provide an effective solution for enhancing the use of cache memory space for a given cache size and cost. A split cache provides architectural support for distinguishing between memory references that exhibit spatial (viz., array data) and temporal locality (viz., scalar data) and mapping them to separate caches. In the previous research we used such performance metrics as cache miss rates, power consumption, silicon area needed and execution cycles. However we did not analyze the frequency of access to different cache sets. In this work we report on the frequency of accesses to cache sets when split data caches are used. Our split data cache architecture does not include any additional hardware. However in future we will explore the benefits of involving compiler time relocation of data to minimize conflicts and spread accesses more uniformly across cache sets.

Compared with others methods, in addition to showing performance gains, we also utilize well known statistical analyses to provide insights into uniformity of accesses. The most significant aspect of our work is its simplicity. We obtain more balanced cache accesses and reduce the accesses to heavily used sets without dynamically detecting the cache set usage information and using hardware to remap data to underutilized sets. We increase the access to the underutilized cache sets by incorporating more uniform locality pattern into the cache access behavior by using separate data caches.

However, we do not claim that split data caches completely solve the non-uniformity of cache accesses, In addition, the split caches are useful for L-1 data caches and not for L-1 instruction caches. 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. And in some cases dynamic remapping using (hardware) programmable decoders [7] are needed. In this paper we will show that split data caches significantly mitigate the problem for several embedded benchmarks (from MiBench) and some SPEC benchmarks.

The rest of the paper is organized as follows. Section 2 provides a survey and analysis of related research while, section 3 discusses related issues and performance metrics

[1] All correspondence should be addressed to Krishna Kavi, Dept. of CSE, University of North Texas, 1155 Union Circle, # 311366, Denton, 7620-5017 or by email: kavi@cse.unt.edu

in more detail. Section 4 presents the results. Section 5 provides a brief synopsis of our work, drawing conclusions from our experimental results.

2. Previous Work

In the recent past researchers have proposed variations that result in miss rates like a 2-way set associative cache, but hit-times like a direct mapped cache [3, 4]. These "sequential search" of set associative caches or probe caches are based on the key observation that associativity is needed only for conflicting blocks and should not be provided at the expense of higher hit latencies for all accesses. At the same time, implementation of different mapping options also reduce the accesses to heavily used sets without dynamically detecting the cache set usage information. In most of these schemes, a traditional direct-mapped cache is conceptually partitioned into sets with 2 (or more) blocks per set. Cache access is sequential; first one block is probed, and if the tag doesn't match the second block in the set is probed. In order to achieve more balanced cache access behavior different probe caches use different data structures, ranging from very simple hash bit to complicated way prediction mechanisms. Hash-Rehash cache (HR cache) uses fixed probe order with different hash functions to search the cache [3]. The column associative cache [4] improves on HR cache by associating rehash information with each block for dynamically selecting alternate hashing functions. Column-associative cache can be extended to include multiple alternative locations, which are described in [4]. In adaptive group-associative cache (AGAC) Peir *et al.* [12] attempts to use cache space intelligently by taking advantage of the cache holes during the execution of a program. AGAC needs three cycles to access these relocated cache lines. The skewed-associative cache [5] is a 2-way cache that exploits two or more indexing functions derived by XORing two *m*-bit fields from an address to generate an *m*-bit cache index to achieve more uniform cache access pattern to lower the miss rate. In B-cache design, by using programmable decoders, Zhang [7] proposes to increase the decoder length and reduce the accesses to heavily used sets.

In our previous work [1, 2] we have shown the performance improvements that can be achieved by using separate L-1 data caches for array (or streams) and scalar data objects of a program. In many cases, the combined size of array and scalar caches are much smaller than a unified L-1 data cache to achieve the same level of performance (miss rates, execution times, energy consumption). For example, figure 1 shows the percentage of reductions in execution times, power consumption and silicon area needed for split data caches when compared with a 8Kbyte unified cache. The sizes of array and scalar cache for each benchmark (from Mibench suite) are optimized in this experimentation[2].

In this paper we expand on our previous research, but our work differs from other research efforts in two ways. First, our cache design does not include any additional hardware or complicated mapping techniques, hence does not result in increasing access times or other performance overheads. Second, unlike the other reported studies (except B-cache), we perform analyses not only with miss rates but also cache access patterns to different cache sets. Even for B-cache, the analysis of cache access patterns is very simplistic while we use sound statistical analyses to determine the shape (kurtosis) and skewness of access patterns.

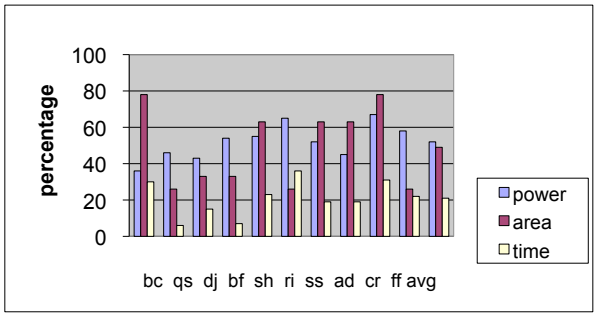

Figure 1: Reductions in execution times, power consumption and silicon area using split data caches

3. Background

In this section, we first demonstrate how to examine the cache sets usage during a program's execution. Since we will implement statistical analysis, we will briefly describe related statistical concepts. Finally we will briefly describe our split data cache architecture.

3.1 Non-Uniform Accesses to Cache Sets

Zhang [7] 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 [7] 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. We repeated Zang's experiments with a subset of SPEC benchmarks, some bio-informatics and embedded benchmarks (from MiBench suite). Consider for example figure 2, which shows the accesses and misses to the L-1 data cache caused by SPEC 2000 benchmark parser. The cache is a direct-mapped, 8kb cache with a line size of 32 bytes (or 256 sets). Here 11 sets (or 7.4%) are in FHS category while 7 sets (or 2.7%) are in FMS and 88% of the sets are in LAS categories. Although not shown in this document,

L-1 instruction cache exhibits similar (non-uniform) access behavior.

In figure 3 we show the cache hit and miss numbers on each set of data cache of the same benchmark with increased associativity. From figure 3, we can see that set-associativity does not fully address the non-uniformity of accesses, although associativity will reduce conflict misses, and increases access times. If we can find a way to balance the mappings of a direct-mapped cache, such that the accesses to direct-mapped cache sets are more evenly distributed across the cache sets, we can reduce miss rate of direct-mapped caches without increasing the cache's access time.

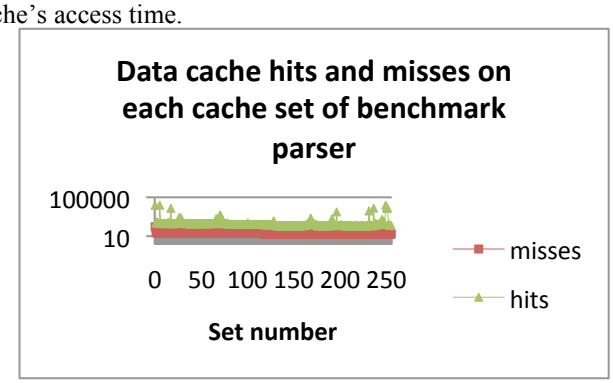

Figure 2: Data cache hits and misses on each cache set for benchmark Parser

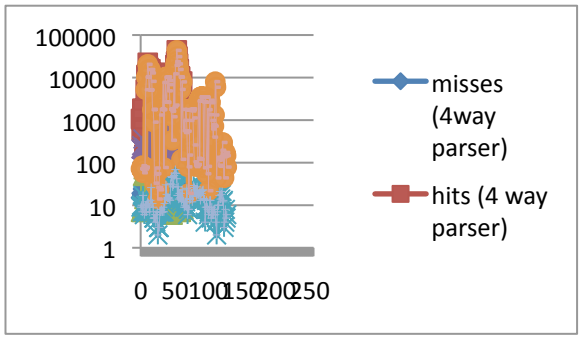

Figure 3: Cache hit and miss numbers on each set of data cache with increased associativity

Benchmark	FMS	FHS	LAS	FMS (%)	FHS (%)	LAS (%)
176.GCC	4	18	219	1.5	7	48.8
164.GZIP	7	11	3	2.7	4.3	1.2
181.MCF	0	22	0	0	8.6	0
CLUSTALW	11	16	170	4.2	6.2	66
175.VPR	45	15	129	18	5.8	65
197.PARSER	7	20	166	2.7	7.8	65
String search	154	17	254	60.2	6.6	99.2
Qsort	3	22	4	1.2	8.5	1.6
dijkstra	9	6	185	3.5	2.3	72.3
AES	5	14	113	1.9	5.5	44.1

Table 2: frequent hit sets (FHS), frequently missed sets (FMS) and least accessed sets (LAS) values.

Zhang [7] proposed the use of a programmable decoder to remap data to different cache sets (on conflict misses) as a technique to achieve more uniform accesses to L-1 cache sets. This hardware solution may not be cost-effective for some applications since the access path is lengthened by the programmable decoders. We repeated the analysis for benchmark programs from the SPEC 2000 [14], Bioinformatics [13] and MiBench suite [15]. The FHS, FMS and LAS were calculated based on (1), (2), and (3). These measures are the same as those described in Zhang [7].

$$FHS(\%) = \frac{fhs(value)}{256} \times 100 \quad (1)$$

$$FMS(\%) = \frac{fms(value)}{256} \times 100 \quad (2)$$

$$LAS(\%) = \frac{las(value)}{256} \times 100 \quad (3)$$

Our experimental environment builds on the SimpleScalar (version 3.0d) simulation tool set [10] modeling an out-of-order speculative processor with a two-level cache hierarchy. We rely on default parameters defined by SimpleScalar. In order to obtain the statistical values we used Mathlab [11].

Table 2 shows our results for the L-1 data cache with a subset of SPEC benchmarks, some bio-informatics and embedded benchmarks. The last three columns indicate the fraction of the sets in the FMS, FHS and LAS categories, while the first 3 columns indicate the number of sets in these categories. Consider for example, VPR, Parser (from SPEC) and CLUSTALW (a bio-informatics benchmark), string search and dijkstra (from MiBench). These benchmarks show that a very high number of cache sets fall in the LAS category. Moreover very few sets (FMS value) cause most misses. On the other hand, GZIP shows a more uniform access pattern since only 1.2% of the sets fall in the LAS category (a significant number of the sets receive between 0.5 to 2 times the average number of accesses) but very few sets cause very high miss rates. MCF (from Spec) shows a different access pattern: not only do sets exhibit more uniform access patterns, they also exhibit more uniform miss behaviors.

Thus different solutions are needed for different applications. For example instead of a programmable hardware address decoder, it may be possible for a software solution where program variables are remapped to different addresses to minimize cache conflicts. In some cases, increasing set associativity alleviates the problem. We feel that our split L-1 data caches may also mitigate the non-uniform access patterns and conflict misses. In this paper we will explore the effect of split caches.

However, to understand the nature of the non-uniform accesses, we need sound statistical analyses. We propose to use several central-moments for this purpose.

3.2 Statistical Analysis

As stated above, in order to more formally describe the behavior of cache access patterns, we will convert the accesses and misses into probability distributions. We can 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.

For this purpose we will convert accesses (and misses) to cache sets into a probability distribution and analyze 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 mean, standard deviation, 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.

3.2.1 *Standard Deviation*

Standard Deviation is a measure of the dispersion of a set of data from its mean. A low standard deviation indicates that the data points tend to be very close to the same value (the mean), while high standard deviation indicates that the data are "spread out" over a large range of values. A zero standard deviation implies a uniform distribution. Figure 4 includes a plot of a standard normal distribution (or bell curve). Each band has a width of one standard deviation.

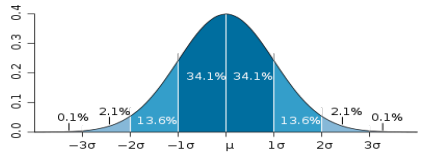

Figure 4: A plot of a standard normal distribution

3.2.2 Skewness

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

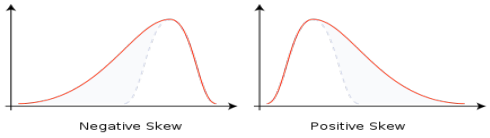

Figure 5: Positive and negative skewness

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 two are equal, it has zero skewness.

3.2.3 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.

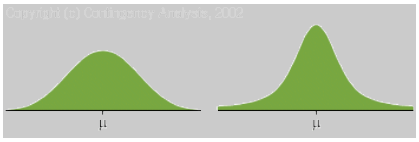

Figure 6: Kurtosis values

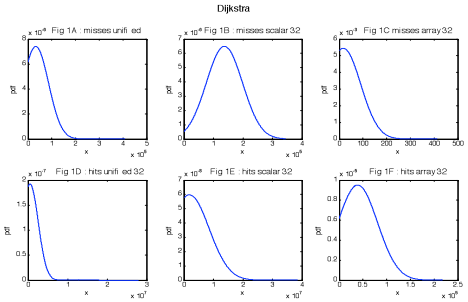

Figure 7: Distribution of cache accesses

Figures 7 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 data caches), for benchmark dijkstra (from Mibench). The main goal of this figure is to illustrate the importance of the shape of the accesses, when the accesses are converted to a probability distribution.

3.3 Split Cache Design

Our split data cache architecture consists of an "array cache" and a "scalar cache". Memory accesses are distinguished as scalar or array references and mapped to a either the scalar or array cache portions. 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 [1, 2].

	Unified			Scalar			Array		
Benchmark	Misses	Hits	Accesses	Misses	Hits	Accesses	Misses	Hits	Accesses
Qsort	2105032	135191867	137296899	1822313	83668050	85490363	4094	48455521	48459615
Dijkstra	1617205	80082245	81699450	805859	61729147	62535006	20	1508763	1508783
AES	8508635	120939232	129447867	295558	5086418	5381976	21	7448490	7448511
String Search	9704	2234608	2244312	4293	1398267	1402560	24	790896	790920
GCC	11846853	337815632	349662485	12467659	220657330	233124989	506	106923052	106923558

Table 3: Number accesses, hits and misses when using a unified data cache and when using split data caches

	Unified				Scalar				Array			
Measure / Benchmark	Mean	Standard Deviation	Skewness	Kurtosis	Mean	Standard Deviation	Skewness	Kurtosis	Mean	Standard Deviation	Skewness	Kurtosis
Qsort	528093.23	1647803.73	5.370	36.07	653656.64	1477732.06	4.13	22.73	378558.76	1066926.36	4.24	22.75
Dijkstra	312821.27	2388115.68	15.80	251.83	482258.96	3373641.56	11.13	125.25	11787.21	30503.16	5.450	40.31
AES	472418.87	1545251.35	7.69	68.67	39737.64	133728.19	8.61	85.48	58191.33	156514.26	6.19	43.555
String Search	8728.94	41807.25	9.36	99.36	10923.96	48048.54	6.88	54.29	6178.87	13202.82	4.86	31.73
GCC	1319592.3	7855808.33	11.81	159.73	1723885.39	5808867.2	7.25	65.06	835336.35	5177289.20	9.38	96.27

Table 4: Mean, Standard deviation, skewness and kurtosis values for hits

	Unified				Scalar				Array			
Measure / Benchmark	Mean	Standard Deviation	Skewness	Kurtosis	Mean	Standard Deviation	Skewness	Kurtosis	Mean	Standard Deviation	Skewness	Kurtosis
Qsort	8222.78	20045.83	13.56	197.37	14236.82	12533.69	6.705	49.88	31.98	242.21	11.15	125.49
Dijkstra	6317.21	9123.08	9.04	94.03	6295.77	6605.39	3.21	18.20	0.156	1.17	10.56	116.57
AES	33236.85	54126.65	4.70	30.25	2309.05	4010.85	7.68	69.29	0.164	1.018	9.65	102.16
String Search	37.91	187.97	12.62	178.22	33.54	51.13	2.45	9.34	37.91	187.97	10.44	114.76
GCC	46276.77	446648.20	14.03	210.79	97403.59	391157.16	6.13	47.29	46276.77	446648.20	11.17	125.91

Table 5: Mean, Standard deviation, skewness and kurtosis values for misses

	Scalar				Array			
Measure / Benchmark	Mean	Standard Deviation	Skewness	Kurtosis	Mean	Standard Deviation	Skewness	Kurtosis
Dijkstra (32-32)	1866309.56	6678402.38	5.378	29.96	38894.91	41944.15	2.71	11.44
Dijkstra (64-64)	955699.13	4749340.75	7.79	61.80	23017.78	40869.39	3.47	15.85
Dijkstra (a32-s64)	955698.72	4749340.83	7.79	61.797	46020.34	53023.72	2.29	7.86
Dijkstra (a32-s128)	482258.99	3373641.56	11.13	125.25	47132.19	58919.24	2.56	9.3934
AES (a32-s32)	1137002.13	2519099.83	4.43	23.06	2438624.5	3053485.24	2.77	9.97
AES (a64-s64)	647832.72	1873827.11	6.23	44.76	1211850.16	2260792.6	4.22	21.007
AES (a32-s64)	647530.5	1873635.69	6.231	44.776	2423287.16	3065156.78	2.781	9.99
AES (a32-s128)	370643.17	1370940.37	8.82	88.54	2422399.41	3066587.1	2.793	10.04

Table 6: Results with variable size Array and Scalar caches (hits)

	Scalar				Array			
Measure / Benchmark	Mean	Standard Deviation	Skewness	Kurtosis	Mean	Standard Deviation	Skewness	Kurtosis
Dijkstra (32-32)	137281.4062	61404.8151	1.9006	5.915	17.25	73.2525	5.1275	28.0888
Dijkstra (64-64)	32165.5469	16284.4695	1.8282	8.5207	1.0781	6.6172	7.7257	61.1232
Dijkstra (a32-s64)	32165.5156	16284.4528	1.8282	8.5208	17.4688	76.7211	5.2023	28.6579
Dijkstra (a32-s128)	6295.7578	6605.3763	3.2048	18.197	32383.5078	53792.3955	5.2023	28.6579
AES (a32-s32)	485188.625	150513.208	0.5424	2.4639	941.2812	4365.5708	5.1901	28.5629
AES (a64-s64)	159564.9531	89381.895	1.6113	5.6647	0.6719	3.6255	7.6274	60.1015
AES (a32-s64)	159556.8906	89390.1215	1.6112	5.6629	885.4375	4088.4034	5.1557	28.2783
AES (a32-s128)	24076.2969	41659.1486	7.6905	69.4319	8442.0156	59650.6805	5.0916	27.7194

Table 7: Results with variable size Array and Scalar caches (misses)

4. Results

In this section, we are going to use 4 central moments (mean, standard-deviation, skewness and kurtosis) to more carefully analyze the benchmarks. Before we describe the affect of our split caches on the non-uniformity of accesses to L-1 data cache sets, we want to show that split data caches do reduce the total number of misses (as we have described in our previous or split caches research). Table 3 shows number access and misses when using a unified data cache and when using separate array and scalar caches. Here we are using 256 sets (32Byte per line) unified cache and 128 sets each for array and scalar caches. In principle the total number data access should be the same whether we are using a unified or split caches. However, the out-of-order Simplescalar simulator generates slightly different number of access under different runs, where the differences are very small (less than 1%).

We now show the 4 central moments (mean, standard-deviation, skewness and kurtosis) for cache accesses and misses. The data is shown in Tables 4 (hits) and 5 (misses). The tables include data for both unified data cache and split data caches. It should be noted that for our purpose (exploring uniformity of accesses), Kurtosis is more useful than other moments. Since our distribution only contains non-negative probabilities (in terms of hits and misses), skewness is not as useful. Looking at the data in Table 4, it appears that split data caches reduce Kurtosis, implying that the accesses to cache sets exhibit more uniform behavior when compared with unified cache. For some benchmarks, the reduction in Kurtosis is very significant.

When it comes to misses (Table 5), the Kurtosis value alone does not show the affect of our split caches. For example, for GCC, the Kurtosis values for both scalar and array cache are higher than for the unified cache. However, it is necessary to consider the actual number of cache misses for the two cache designs. Our split caches results in a significant reduction in actual cache misses (see Table 3). But the higher Kurtosis means that our split caches are causing more conflict misses on some specific sets. This is in part because array and scalar caches are smaller than the unified cache. As we have shown in our previous work [2], different applications need different sizes for array and scalar caches. With this in mind, we repeated our simulations using different sized array and scalar caches for two benchmarks: dijkstra and AES, two benchmarks that have shown increased values for kurtosis with cache misses. From the following tables (Tables 6 and 7) one can see the for both benchmarks, a smaller array and a larger scalar cache result in better performance (smaller Kurtosis values).

It should be noted that for these benchmarks (AES and Dijkstra) large scalar and array caches cause higher Kurtosis values (both for hits and misses). This implies that, for these applications, larger caches are not very useful since the number of sets utilized (hits and misses) remain small. To fully use large L-1 caches, it will be necessary to use more complex techniques to spread data accesses across available sets. In our current research we are investigating profiling cache accesses to identifying program variables that are mapping to highly utilized sets and reassigning them to new memory addresses.

5. Conclusions

In this paper we show that split data caches significantly mitigate the problem for several embedded benchmarks (from MiBench) and some SPEC benchmarks, in terms of improving uniformity of accesses to cache sets. However, we do not claim that split data caches completely solve the non-uniformity of cache accesses for all applications. Thus different applications need different approaches to solve the non-uniform accesses. In some cases our split-caches are adequate. However in some cases profiling and compile time analyses or additional hardware may be needed to relocate data that maps to highly utilized sets. Currently we are also exploring how profiling and compile time analyses can be used to uniformly distribute data among all cache sets.

6. Acknowledgment

This research is supported in part by NSF grants #0649748, #0855939 and by the NSF Net-Centric IUCRC industrial memberships.

7. References

[1] A. Naz, K. Kavi, P. Sweany and W. Li. "A study of reconfigurable split data caches and instruction caches", *in the Proceedings of the 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" *in the Proceedings of the 18th International Symposium on High Performance Computing Systems and Applications (HPCS 2004)*, Winnipeg, Manitoba, Canada, May 2004.

[3] A. Agarwal, J. Hennessy, and M. Horowitz. Cache performance of operating systems and multiprogramming. *ACM Transactions on Computer Systems,* 6(4):39343 1, Nov.1988.

[4] A. Agarwal and S. Pudar. “Column associative caches: A technique for reducing miss rate of direct-mapped caches”, in *Proceedings* of *the 20th Annual International Symposium on Computer Architecture,* May 1993.

[5] A. Seznec. “A case for two-way skewed-associative caches”, In *Proceedings of the InternationalSymposium on Computer Architecture,* June 1993.

[6] B. Chung and J. PMF. “LRU-based column associative caches”, *Camp. Arch. News 26*, 2, 9–17, 1998.

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

[8] C. Zhang. “Balanced cache: Reducing conflict misses of direct-mapped caches through programmable decoders”, In *Proceedings of the International Symposium on Computer Architecture*, June , 2006.

[9] C. Zhang, X. Zhang, and Y. Yan. “Two fast and high-associativity cache schemes”, *IEEEMicro 17*, 1997.

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

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

[12] J. Peir, Y. Lee, and W. Hsu. “Capturing dynamic memory reference behavior with adaptive cache topology”, In *Proceedings of the 8th International Conference on Architectural Support forProgramming Language and Operating Systems,* 1998.

[13] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C. Tseng, and D. Yeung, “BioBench: A benchmark suite of bioinformatics applications”, *in Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software*, Austin TX, March 2005.

[14] L. Henning. "SPEC CPU2000: Measuring CPU Performance in the New Millennium", *IEEE Computer*, 33(7), pp. 28-35, July 2000.

[15] M. Guthaus, J. Ringenberg, T. Austin, T. Mudge, R. Brown, "MiBench: A free, commercially representative embedded benchmark suite, *in Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*," Austin, TX, December 2001.