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
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 rate typically 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 known 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.

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 purpose, a highly non-uniform behavior results in a high Kurtosis, while a more uniform access behavior leads to lower Kurtosis.

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 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 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 affect 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 ENVIRONMENT

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 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 mapped).

<table>
<thead>
<tr>
<th>Name</th>
<th>Benchmark</th>
<th>Description</th>
<th>Unified</th>
<th>Array</th>
<th>Scalar</th>
<th>Total Ref.</th>
<th>Total Miss</th>
<th>Total Ref.</th>
<th>Total Miss</th>
<th>Total Ref.</th>
<th>Total Miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit counts</td>
<td>Test bit manipulation</td>
<td>bc</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sport</td>
<td>Computational/Chemistry</td>
<td>qs</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dijkstra</td>
<td>Shortest path problem</td>
<td>dj</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>blowfish</td>
<td>Encryption/decryption</td>
<td>bf</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AES</td>
<td>Advanced Encryption Standard</td>
<td>AE</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CRC</td>
<td>Cyclic Redundancy Check</td>
<td>CR</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>String search</td>
<td>Search mechanism</td>
<td>ss</td>
<td>7</td>
<td>7</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TABLE 2: Number of misses with different cache sizes (Direct Mapped)
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 block sizes for all benchmarks except for two cache sizes 4k and 2k for the benchmark “blowfish”. If we compare results 8k unified 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 is so low that a tiny 256 byte scalar cache is needed with a smaller array cache. And t“qsort”, 8k unified cache, 4k-array and 4k Scalar caches (with a 10-line stream buffer) provide better results than 8k unified cache. For “blowfish” a large 4k 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 8k unified cache). For other benchmarks (except “qsort” and “blowfish”) 1k scalar cache and 512 byte array cache (with 8k unified cache) we see 47.65%, 43.76%, 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 is so low that a tiny 256 byte scalar cache is needed with a smaller array cache. And t“qsort”, 8k unified cache.

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 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 misses 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 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 “qsort”) we have considerable decrease in kurtosis values.

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

TABLE 8: 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 we compare last column of TABLE 5 and last 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