Adaptive Cache Placement for Scientific Computation

Subramanian Ramaswamy and Sudhakar Yalamanchili
Computer Architecture and Systems Laboratory
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332
ramaswamy@gatech.edu,sudha@ece.gatech.edu

Technical Report: GIT-CERCS-07-03

Abstract

The central data structures for many applications in scientific computing are large multidimensional arrays. These arrays dominate memory accesses and are often accessed with strides that vary across orthogonal dimensions posing a central and critical challenge to develop effective caching strategies. We propose a novel technique to optimize cache placement for multidimensional arrays with the focus on minimizing conflict misses in the cache hierarchy. We propose architectural extensions for adaptive cache placement that are exercised under software control to reduce conflict misses for various access patterns to array data structures. Adaptive cache placement complements existing compiler optimizations, offering a new degree of freedom in optimizing the memory system performance and can be used by dynamic optimizers. Our implementation is compared to traditional caches for a range of common scientific loop based kernels and applications, and is observed to reduce, and in some cases, eliminate conflict misses in the L2 cache to array data structures. We explore the effect of cache parameters such as cache size and associativity on global miss rates, average memory access time, area and power for these benchmarks.

1 Introduction

A dominant research theme in the high performance computing community is memory system optimization focused on pushing back the memory wall [17]. Memory and processor speeds continue to diverge and cache hierarchies are becoming deeper and larger. We propose an approach to significantly lower conflict misses in L2/L3 caches by relaxing one of the core cache design constraints—fixed cache placement. Conventionally, caches are designed with a fixed *placement* function, i.e., the location of a memory line in the cache is fixed at design time. Increasing the associativity improves the sharing of cache lines by main memory lines and consequently reduces miss rate. However, increasing associativity comes at the expense of increased hit time, increased energy in tag matching logic, and reduction in the number of sets (for a fixed size cache) rendering fully associative caches impractical for L2 and L3 levels of the hierarchy.

We propose a software-controlled alternative to increasing associativity for improving sharing of cache resources across memory lines—the generation and implementation of adaptive placement functions. Our approach is to improve sharing of cache lines by relaxing the restriction of fixed placement and enabling software programmable placement functions that can be customized to the memory referencing behavior of specific domains, in this case, scientific computation. The principle difficulty with adaptive placement lies in the complexity of address decoding at the cache interface if arbitrary placement functions are permitted. However, we have discovered that for strided memory accesses dominant in scientific computing applications, we can use structured placement functions for accesses to multidimensional arrays which dominate memory accesses in these applications.

Strided array accesses are a common feature of many scientific computing kernels. These strides are either statically computable at compile-time or are dynamically computed at run-time. Sequences of accesses across orthogonal dimensions of multidimensonal arrays typically generate many conflict misses. As we report, such structured access patterns can particularly benefit from adaptive placement with dramatic reductions in conflict misses with low investment in hardware and software overhead. The result is a large reduction in the average memory access time (AMAT). For example, a 256KB 8-way adaptive placement cache can provides L2 local miss rates that are 40–90% lower compared to a traditional 8-way cache.

The compilation model is one wherein program hot spots, e.g., nested loops, are preceded by operations that modify the cache placement function. The address decoder at the cache interface is modified only during execution of these program regions. The proposed technique, architectural extensions, and the associated abstractions provide primitives to guide extensions to existing compiler-based memory system optimizations as well as conceive of new ones. Specifically, the proposed approach is not intended as a replacement for existing techniques. Furthermore, they provide an opportunity for dynamic optimizers that employ run-time stride prediction/measurement. The following section places our proposal in the context of existing and prior work followed by descriptions of the programming model, architectural extensions, and development of the key concepts as applied to multi-dimensional arrays. The paper concludes with results of simulation experiments and the anticipation of some fruitful directions for future work.

2 Related Work

There have been significant contributions reported in literature towards compile-time optimization of cache performance for strided array accesses. A dominant category includes data pre-fetching techniques [15, 20] for reducing compulsory misses as well as approaches that optimize the data layout (physical or virtual) or the data alignment with reference to application memory access patterns and cache parameters [25, 7, 22, 5]. A common theme is to reorder memory reference streams via program transformations including loop unrolling,

loop interchange and loop fusion [18, 21]. All of these optimizations are targeted to fixed cache designs, i.e., those with fixed placement functions. In general, compilers can control data layout and scheduling of accesses. Solutions for either are formulated and indirectly coupled by how regions of memory compete for resources (cache lines). By making this coupling flexible via variable placement, schedulers and data layout techniques have greater opportunity, e.g., a specific layout may no longer preclude certain instruction schedules for good performance. Although this increases the complexity (and perhaps feasibility) of newly enabled optimizations, domains such as scientific computation for which there is a history and body of knowledge with respect to understanding memory behaviors can benefit as we show in this paper.

In contrast, existing hardware techniques for improving memory behavior typically focus on program agnostic techniques to reduce conflict misses through techniques such as increasing effective associativity, including innovations in indexing and hashing [24, 11, 23, 8, 29, 4, 1, 13, 8]. These approaches seek to find better *fixed* design time placement functions than traditional placement and some of them can be recognized as specific instances of variable placement policies. Further, our proposed work couples a class of placement functions to a domain of applications in a manner that can be reprogrammed as guided by analysis (either statically via compilers or at run-time via dynamic optimizers). For example, stride predictors can be used to determine the programmable placement function [28, 10, 27]. The existing hardware techniques do not attempt to similarly benefit from analysis.

The techniques we propose are also accompanied by abstractions that enable reasoning about cache behavior from the perspective of *both* hardware techniques and compiler/dynamic optimizers. Thus we build on existing and past work by enhancing the scope of future refinement/extensions as well as open the door to new opportunities. This paper demonstrates the benefits of these abstractions and techniques.

3 Customizing Cache Placement

3.1 Execution Model and Architectural Extensions

Program execution consists of a sequence of *phases*, where a phase corresponds to the execution of a program region such as a loop nest or function. Adaptive placement is applied to phases that contain strided memory accesses to multidimensional arrays. The stride values must be known prior to execution of the region and cannot be computed during execution of the regions, e.g., through indirect array references. The compilation model is one wherein each phase is analyzed to compute stride information, and using cache parameters and array dimensions, compute and effect a custom placement function for that phase. Program regions not so optimized retain the use of the standard modulo placement function. At the current time we use a simple form of implementation wherein new placement functions are invoked at phase boundaries as shown in the three examples in Figures 2 - 4 which illustrates how the insertion of calls for adaptive placement is influenced by the availability of stride information. On exit from the optimized program phase, the placement function reverts to the traditional modulo placement policy or another adaptive placement function is invoked for a following phase.

This paper focuses on uniprocessor architectures with level 1 and level 2 caches. Adaptive placement is applied on a per program phase basis and is implemented for the L2 cache as shown in Figure 1. With multiple threads, the placement function will have to preserved as a part of the thread state. The hardware overhead for multiple threads is relatively small and is discussed in greater detail in Section 3.7.

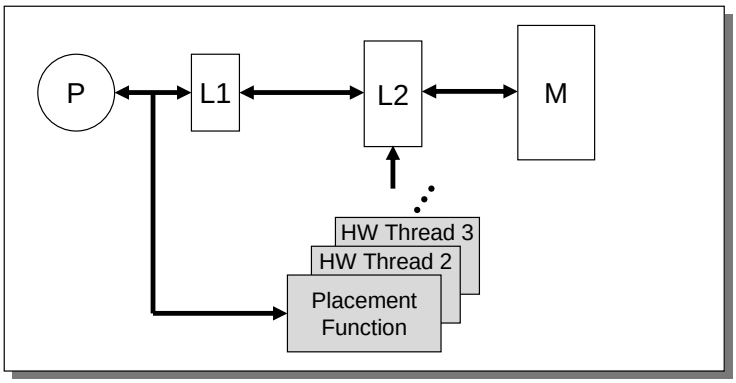

Figure 1: Architecture Model

```
1: a = const;
2: b = const;
3: c = const;
4: placement(a, b, c)
5: for i = 1 to N do
6:    for j = 1 to N do
7:       for k = 1 to N do
8:          X[ai + bj + ck] + +;
9:       end for
10:   end for
11: end for
```

Figure 2: Example 1

```
1: b = const;
2: c = const;
3: for i = 1 to N do
4:    a = expr;
5:    placement(a, b, c)
6:    for j = 1 to N do
7:       for k = 1 to N do
8:          X[ai + bj + ck] + +;
9:       end for
10:   end for
11: end for
```

Figure 3: Example 2

```
1: c = const;
2: for i = 1 to N do
3:    a = expr;
4:    for j = 1 to N do
5:       b = expr;
6:       placement(a, b, c)
7:       for k = 1 to N do
8:          X[ai + bj + ck] + +;
9:       end for
10:   end for
11: end for
```

Figure 4: Example 3

3.2 Placement Model

The placement policy used in traditional cache architectures is illustrated in Figure 5. A memory line at address L is placed in, or mapped to, the cache set $L \ mod \ S$ where there are S sets in the cache. The memory line may be placed within any cache line in the set, i.e., placement is fully associative within the set. We will refer to the memory lines mapped to a cache set as a *conflict set* and the preceding placement policy will be referred to as *modulo placement*. Thus, S contiguous lines in memory are mapped to S distinct sets in the cache and all conflict sets have equal cardinality. The approach proposed in this paper is one wherein the mapping from memory lines to cache lines is software programmable. An example of adaptive placement is shown in Figure 6. While arbitrary placement functions require more complex cache address decoding, the following sections will define placement functions for strided memory accesses that admit to simple and fast implementations. Adaptive placement is applied only during the execution of specific program regions (as described below) and invoked under compiler control. The hardware implementation is described in Section 3.7.

The performance gains achieved by adaptive placement lie in the constitution of conflict sets—the set of memory lines that share a cache line or set. Using a profile/program agnostic placement policy such as

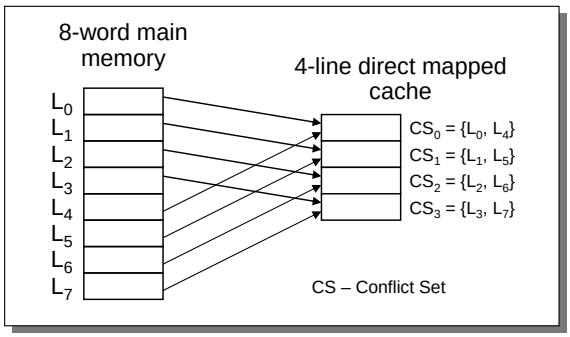

Figure 5: Cache with traditional placement

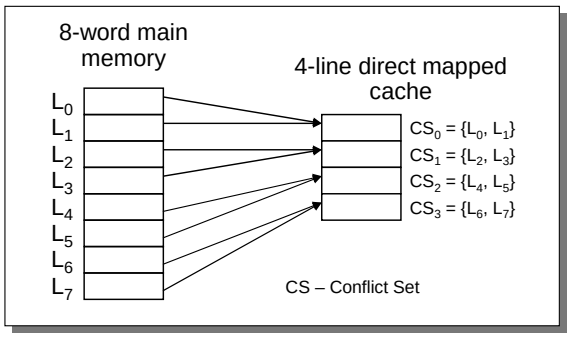

Figure 6: Cache with adaptive placement

modulo placement, the number of conflict misses can be significant in multidimensional arrays when accesses span multiple dimensions and layout is optimized for one dimension. For example, if an array is placed in row major order in memory and is accessed in column major order, lines containing column elements may be in the same conflict set resulting in all accesses generating misses. Adaptive placement can dramatically reduce, or in some instances eliminate such conflict misses.

3.3 Conflict-Free Cache Accesses in 1-D Arrays

By conflict-free accesses, we refer to the behavior where a set of memory accesses do not produce conflict misses in the cache. This section outlines some basic properties of cache placement and its impact on conflict misses for array accesses. Memory accesses to multidimensional arrays are realized as accesses to addresses in a linear memory address space. Therefore this section develops the concepts and basic relationships for conflict free accesses to one dimensional arrays. The extensions to multidimensional arrays is straightforward and is developed in the following sections.

Consider a one dimensional array, $A[N]$, with $N = 2^n$ elements where a memory block or *line* consists of $x = 2^b$ array elements. Thus, the array A is stored in $L = N/x$ contiguous memory lines, addressed $l_0, l_1 \cdots l_i \cdots l_{L-1}$, with line l_i containing the elements $A[ix], A[ix+1] \cdots A[ix+x-1]$. Traditional caches use a fixed placement policy where memory line l_i is stored in cache set $l_i \ mod \ S$, where there are S sets in the cache. As described earlier, this placement policy is referred to as *modulo* placement.

An access pattern is a sequence of memory accesses to the array. Now consider an access pattern that begins with an access to element $A[0]$ and has a stride $k = 2^p$, i.e., consecutive accesses are to elements $A[k], A[2k], A[3k]$, and so on. This corresponds to a memory *line access stride* of $K = k/x$, i.e., memory lines are accessed in the order $(l_0, l_K, l_{2K} \cdots l_{(L-1)-(L-1) \ mod \ K)})$. When $k < 2x$ all lines in the array are touched in a single array traversal. When $k \geq 2x$ all lines are not touched in a traversal. For example for $k = 2x$, every even line is touched when traversal is started at any of the elements $A[0] \cdots A[x-1]$, and every odd numbered line is touched when traversal starts at element any of the elements $A[x] \cdots A[2x-1]$. The *stride set* is defined as the set of lines accessed during one traversal of the array with a stride of k. Thus, there will be K stride sets, each with cardinality L/K. The cardinality of a stride set corresponding to an access of stride k is referred to as the *stride number*, sn_k. The preceding concepts are graphically illustrated in Figure 7.

Now consider a traditional direct mapped cache with modulo placement and S lines (equivalently S sets with each set consisting of only one line). For an access pattern with stride k, if $S = K$, all of the lines

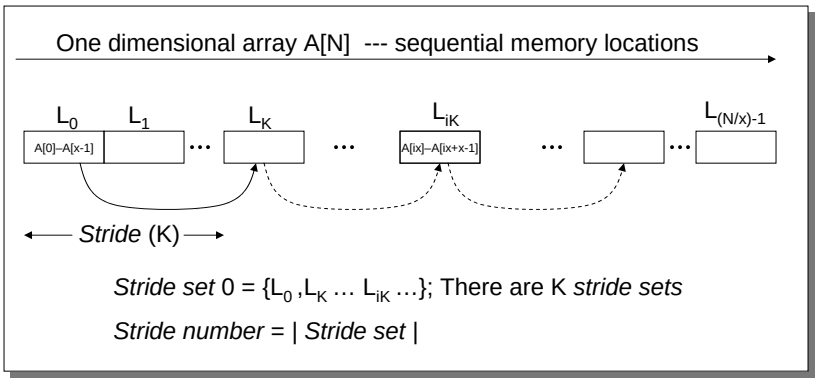

Figure 7: Basic concepts

in a stride set belong to the same conflict set. Thus strided accesses to lines in this set, with stride k, will produce a conflict miss on *every access*! To eliminate conflict misses, lines $l_0, l_K, l_{2K} \cdots$ can be mapped to separate cache lines via the following adaptive placement referred to as *strided* placement.

Strided Placement: Memory line l_i is mapped to cache set $\lfloor l_i/K \rfloor \ mod \ S$

Now the first S strided accesses to lines in this set, with stride k, will not produce any conflict misses! When $S \geq sn_k$, all lines in a stride set can be resident in the cache and thus *all conflict misses can be eliminated* leaving only compulsory misses. In this example, the number of conflict misses that would have occurred with modulo placement is sn_k for each traversal of the array. As we will show later, this is particularly significant for column major access to matrices stored in row major order where the number of conflict misses that are eliminated can be as high as $sn_k * (x - 1)$. Figure 8 illustrates this property.

Increasing the associativity of the traditional cache will not improve matters considerably as all the lines being accessed still belong to a few conflict sets (all the lines will be grouped into k_a sets, where k_a is the associativity) and we have a sequential strided access pattern. To have an impact in the traditional cache, an associativity of sn_k is required, which eliminates all conflict misses in a single array traversal with stride k. This is generally undesirable and clearly infeasible for $S < sn_k$. In contrast, adaptive placement can eliminate conflict misses for this case as described in the preceding paragraph. For the case, $sn_k = S$ the number of conflict misses corresponding to the adaptive placement in the preceding paragraph equals that of a fully associative cache with S lines. The reduction in miss rate is achieved without the area and power penalties of a fully associative cache at the expense and a modest increase in address translation complexity that can be removed from the critical path (as described in Section 3.7). This is illustrated in Figure 9.

The preceding discussion motivates line sizes for which $S \geq sn_k$. Smaller block sizes have the additional benefit of lowering the miss penalty. If the miss rate can be significantly reduced with adaptive placement, then the benefits of smaller blocks sizes can be realized. Further, scientific applications tend to be amenable to pre-fetching strategies. Additionally, in the case of large arrays or matrices, sub-blocking techniques are often employed, and the sub-block sizes can be chosen such that $S \geq sn_k$ to minimize conflict misses with strided array accesses.

The extension to multidimensional arrays is based on the key insight of the relationship between conflict sets and access patterns. The goal is to have a strided access pattern, sequentially reference elements in distinct conflict sets. Thus the lines corresponding to the accessed elements can be concurrently resident in the cache. Conflict set formation is defined by the definition of a placement function relative to the storage

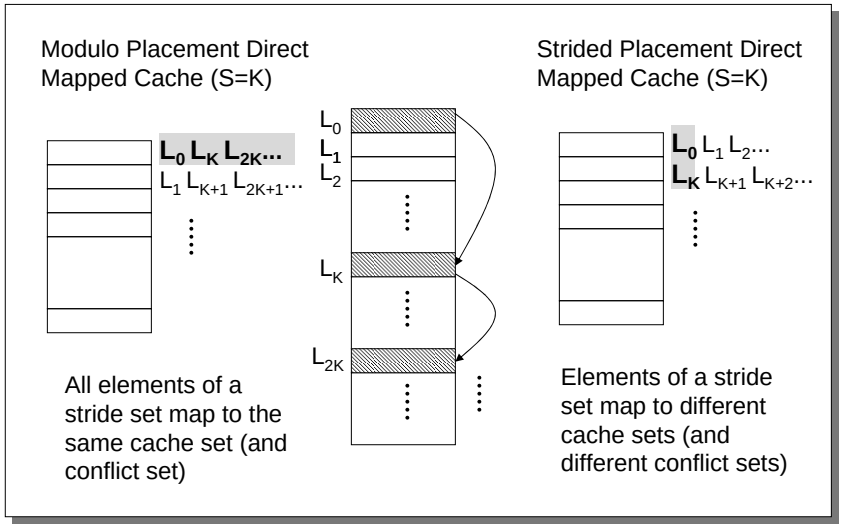

Figure 8: Strided placement

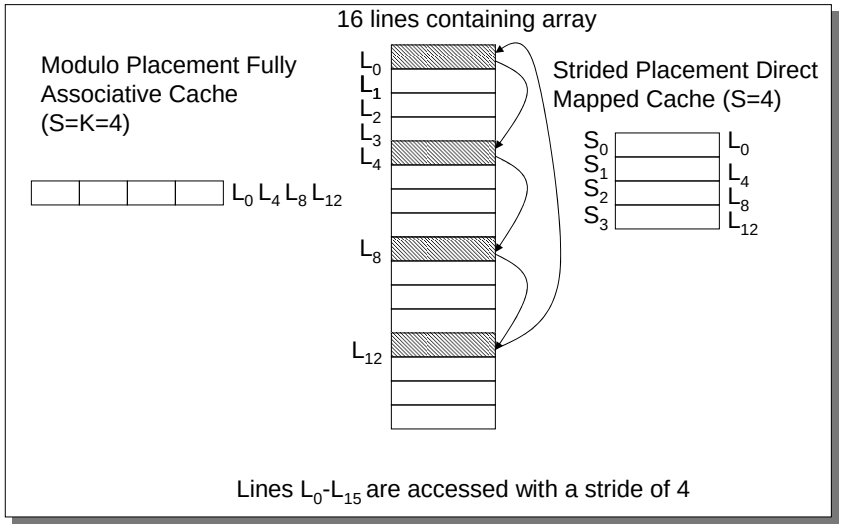

Figure 9: A comparison between strided placement and fully associative placement

scheme for the multidimensional array. For example, if we wished to have conflict-free access to all elements of a column in a matrix stored in row-major order, we define a placement function that places each row in a distinct conflict set. This is elaborated on and applied to multidimensional matrices in the following sections.

3.4 Adaptive Placement for 2-D and 3-D Arrays

The current work was motivated by analogy with data skewing techniques for concurrent access to interleaved memory banks [6, 26]. We view cache sets as the memory banks and benefit from that analysis. Accordingly from that analysis the criterion for conflict free accesses using modulo placement has the following conditions: $S \geq sn_k$, and, $S \geq sn_k.gcd(S, K)$ where $gcd(S, K)$ represents the greatest common divisor between the number of cache sets and the stride K. If S and K are co-prime, the requirement reduces to $S \geq sn_k$ for conflict-free access. If S is even, and the stride is even, modulo placement cannot provide conflict free accesses. With adaptive placement, the criterion for conflict free access is just $S \geq sn_k$ for conflict free accesses. This is because the stride can be made equal to 1 with adaptive placement.

For a two dimensional array with dimensions M, N (stored in row-major order with M rows and N columns), row-major accesses translates to an elemental stride of unity, and column-major access translates to an elemental stride of N. Therefore, the placement function that minimizes conflict misses for column order access will map line l_i to set $\lfloor l_i/\lfloor N/x \rfloor \rfloor \ mod\ S$ in the cache. This is illustrated in Figure 10. When it is desired to traverse a matrix in both row and column-major orders, storage can be row-major and the placement be optimized for column-major order to minimize conflict misses. The same placement policy can be used for forward and back diagonal accesses, because the conflict set construction remains the same.

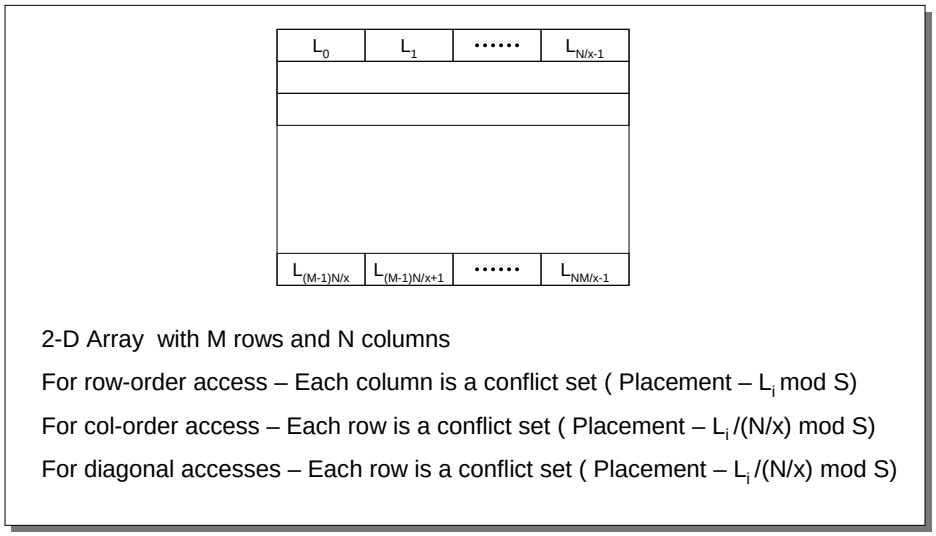

Figure 10: Adaptive placement for matrices

For a three dimensional array with dimensions M, N, O, and row-major storage, row-major accesses correspond to a stride of unity, column-major accesses to an elemental stride of N, and the O-dimension-major accesses to an elemental stride oF NM. Thus the adaptive placement policies suited to these strides can be applied, as shown in Figure 11. This approach can be extended easily to n-dimensions. The fundamental

principle in computing a placement function for a given access pattern is that each successive reference in an access pattern references an element stored in a distinct conflict set. This approach reduces conflict misses, and in some instances (depending on specific relationships between cache parameters and array dimensions) can eliminate conflict misses altogether.

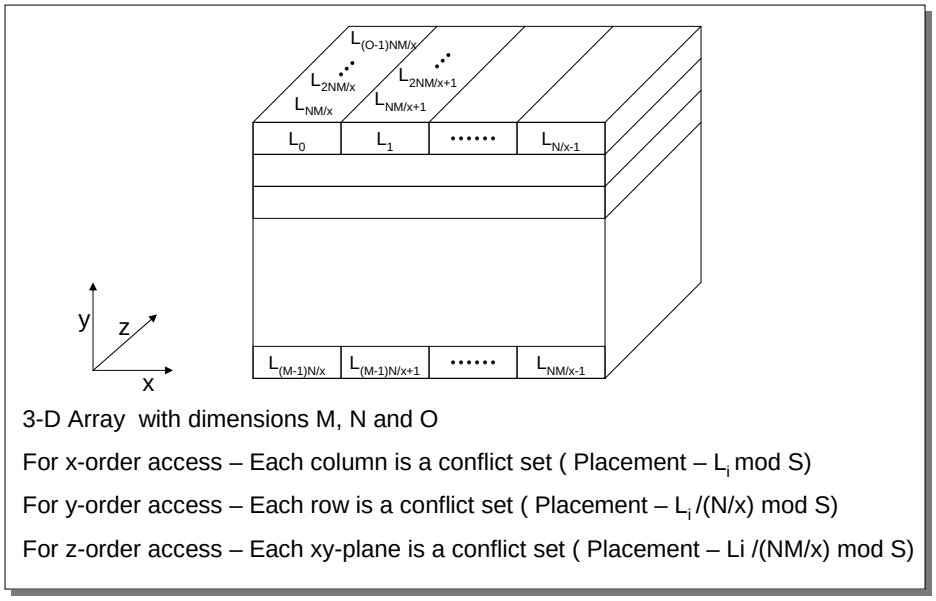

Figure 11: Adaptive placement for 3-D arrays

3.5 Array Accesses with Multiple Strides

If a one dimensional array, $A[]$ is accessed with strides P and Q in distinct program regions $seg1$ and $seg2$, distinct placement functions can be employed in each program region. Now consider the case where A is accessed with line strides P and Q in the same program region. The stride that has to be optimized for with adaptive placement will now be $K' = gcd(P, Q)$. This placement policy will yield conflict free accesses as long as the condition $S \geq sn_{K'}$ is satisfied. This can be extended to more than two strides by choosing a placement function optimized to the stride $K' = gcd(P, Q, \cdots)$.

If there are two arrays with the same dimension accessed within the same loop nest, with the same stride, K, the placement policy is optimized for that stride K. However, to avoid conflict misses between accesses to two arrays, one would need $S \geq sn_k$, with an associativity of two. This can be extended to accesses to multiple arrays. If two arrays have different strides P, and Q, the placement policy must be optimized for the stride $K' = gcd(P, Q)$. Accesses will be conflict free as long as the condition, $S \geq 2 * sn_{K'}$ is satisfied for direct mapped caches, or if the cache is two-way associative, $S \geq Sn_{K'}$ is satisfied. All these relations can be extended to multiple arrays with multiple strides.

Another approach applies different placements at different levels of the cache hierarchy. For example, adaptive placement optimized to stride P is used in one cache, say the L1 cache, and that optimized to stride Q is used in the L2 cache to reduce conflict misses.

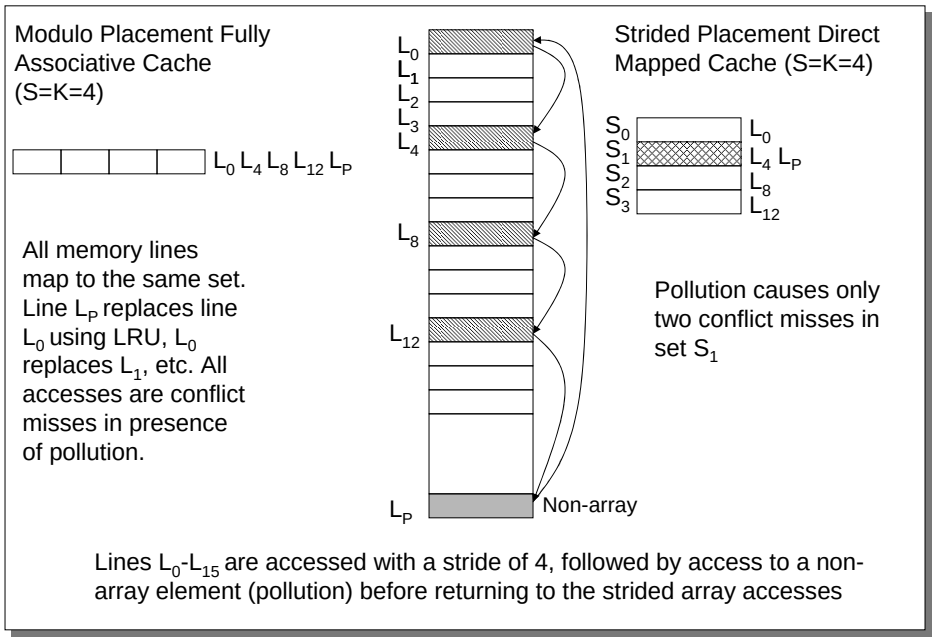

Figure 12: Effects of pollution

3.6 Practical Effects—Pollution

The analytical models and relationship derived earlier were based on the cache being free from non-array accesses, i.e., there was no pollution of array data in the cache. However, in real systems, variables and data other than arrays will be accessed inside loop nests. Although the effects of pollution are implicit in our performance evaluation, there are a few points worth considering. In a direct-mapped cache with adaptive placement, for every memory line accessed other than array data (the "polluting" line in this analysis), there will be two conflict misses—one caused by the polluting line and one caused by the next array access that maps to that line. If the cache is set-associative, with $S = sn$, for every polluting access, there will be $sa + 1$ misses, where sa is the associativity of the cache. This is due to the fact that the polluting access modifies the stack distance of the LRU replacement policy [2]. Accordingly, traditional fully associative caches with $S = sn$ are very susceptible to pollution effects with strided accesses, for example, the well known case where $K + 1$ elements are repeatedly accessed in a fully associative cache with associativity K. This effect is captured in Figure 12.

3.7 Hardware Implementation

While adaptive placement can be implemented at all levels of the cache hierarchy in this paper we limit ourselves to the L2 cache, primarily for two reasons. First any increase in address translation time can be overlapped with L1 access and thus hidden. Second, we are focused in reducing AMAT for which L2 cache performance is crucial.

Adaptive placement is invoked for specific program phases. Accordingly, the base L2 cache design uses traditional modulo placement. When a program phase utilizing adaptive placement is encountered, an

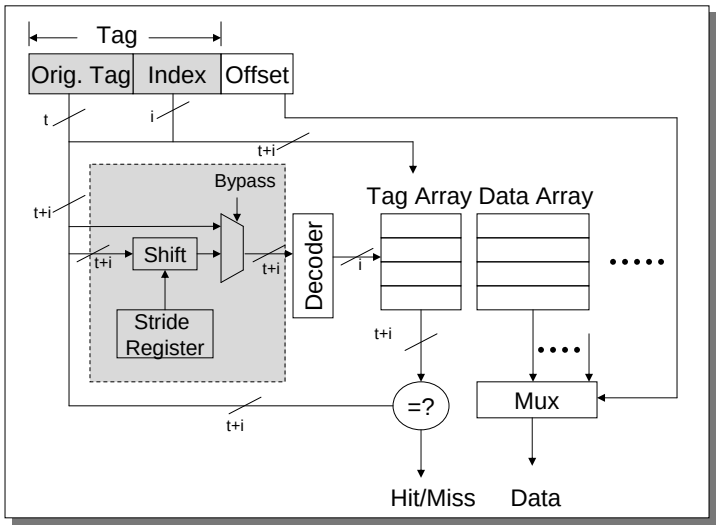

Figure 13: Run-time address translation

additional address transformation step is included in the cache address decode path as shown in Figure 13. In strided placement, memory line l_i is placed in set $\lfloor l_i/K \rfloor \ mod \ S$, where K equals $\lfloor k/x \rfloor$. Because x, the number of array elements per cache line, is always a power of two, the value $K = \lfloor k/x \rfloor$ is easily obtained with a shift operation. Furthermore, if the resulting value of K is also a power of 2, the computation of the placement reduces to a shift operation on the line address. Alternatively, compilers have been known to pad arrays to simplify addressing and maximize pre-fetch mechanisms. Those techniques are certainly feasible here. In the worst case, the address decoder will require a divide operation. Although this can still be overlapped with the L1 access, the overhead cannot be ignored. However, the placement can be optimized for a value close to the stride, and is a power of two. This will not result in conflict free accesses, but will still reduce conflict misses with relatively modest associativities. If the stride is not a power of two, one can adapt the placement for a stride K' which is a power of two, where $K' = gcd(K, K')$.

The hardware implementation of the run-time address decoding for adaptive placement depicted in Figure 13 consists of a stride register that contains the number of bits by which the line address (tag plus index) must be shifted to compute the set address, i.e., $\lfloor l_i/K \rfloor \ mod \ S$ prior to traditional tag look-up The additional logic is shown shaded in the figure. Additional bits are required in the tag because the cache tag array may contain tags from adaptive as well as modulo placement. Hence the tag used for matching in the case of adaptive placement functions is the original tag augmented with the original cache set index. Traditional modulo placement can be invoked either by writing a zero into the stride register, or by invoking the bypass mechanism which bypasses the optional address translation. Writing to the stride register can be implemented as a custom instruction, or by making the stride register an addressable region in memory.

With multiple software threads, the stride register state will have to be saved along with the thread state on context switches. In the case of hardware threads (simultaneous multithreading), there must be separate stride registers for each hardware thread which maintains the cache placement information on a per-thread basis.

Benchmark	Group	Description
doolittle	Linear algebra [16]	LU decomposition using Doolittle's algorithm (dimension 512)
svd		UDV' decomposition using Golub and Reinsch's technique(512)
lowinv		Calculates the inverse of a lower triangular matrix (512)
crout		LU decomposition using Crout's algorithm (512)
hydroexp	Livermore Loops [19]	2-D explicit hydrodynamics routine (kernel 23) (6x4096)
linrec		General linear recurrence equations (kernel 6) (1024)
ADI		Alternating direction implicit integration (kernel 8) (4096,2x4096x5)
mmult		Matrix Multiplication (kernel 21) (1024)
179.art	SPEC CPU2000 [30]	Adaptive Resonance Theory 2 image recognition (100x10)
183.equake		Simulation of seismic wave propagation in large basins (3x30169x3,3x220546)
LINPACK	LINPACK [14]	Collection of subroutines that analyze and solve linear equations and linear least-squares problems (1024x1024)

Table 1: Description of Benchmarks

4 Performance Evaluation

This section provides empirical support for the improved performance indicated by the analytical models derived in Section 3. We chose a set of benchmarks that are representative of program loops in large array based application domains including scientific computing, image and video processing, sonar and radar processing, and weather forecasting, where multidimensional arrays are accessed with varying strides. A description of the benchmarks and their characteristics are provided in Table 1.

4.1 Simulation Methodology

Simulation experiments were performed using the *cachegrind* cache simulator, a part of the *valgrind* profiling suite [31]. The *cachegrind* cache simulator was modified to accommodate adaptive placement policies which are invoked via calls to the *placement*() function. The simulations serve to indicate how much effect pollution has on the adaptive cache performance, as described in Section 3. The simulations were conducted for a variety of cache configurations similar to those in existing Intel Pentium 4 architectures. Adaptive placement is employed only in the L2 cache—a dominant consumer of area and power. All the benchmarks were compiled using *gcc*3.4.6 with the highest optimization level $O3$ chosen. The area and power estimates for the various cache configurations were obtained using *cacti* [9] for the 70nm technology node.

The kernels were manually modified to include calls to functions that modified the placement (as shown in Figure 2) prior to entering loops with strided array accesses. Upon exiting the loops, the placement function reverts back to the traditional modulo placement. The *cachegrind* simulator adopts the modified the placement function. The L1 cache was fixed as a 16KB 4-way cache with 64 byte lines.

4.2 Results and Discussion

Adaptive placement cache (APC) provides better sharing of cache resources and consequently improved miss rates over traditional caches across size and degrees of associativity. Figure 14 compares the miss rates of a 256KB 8-way APC to larger traditional set associative caches and Figure 15 compares it to traditional caches with higher associativity. The effect of increased associativity on performance for traditional caches was found to be negligible for some benchmarks. This is because the stride characteristics of certain benchmarks lead

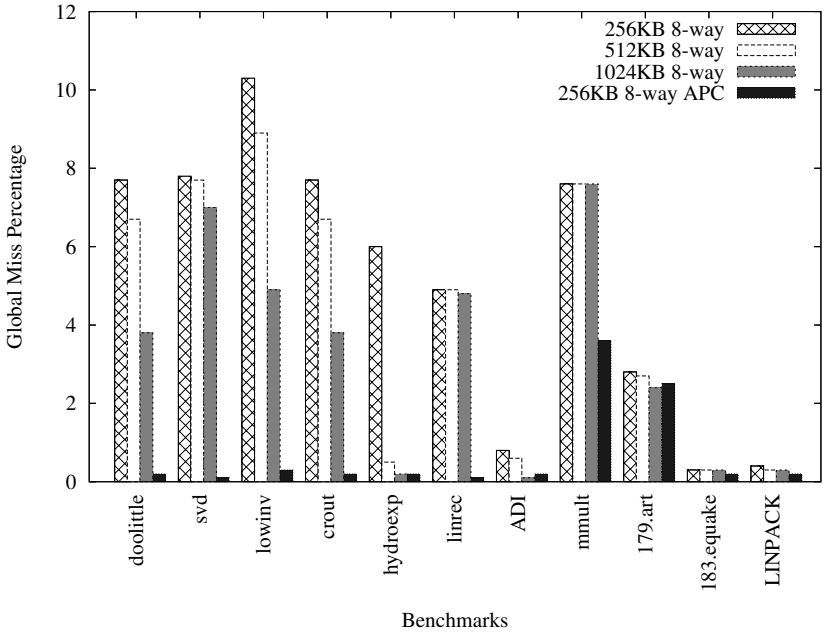

Figure 14: Adaptive placement versus larger traditional caches

to poor cache usage with traditional placement, and therefore very large associativities (512 or greater) are required to have any appreciable effect on miss rates. Traditional caches with larger sizes bring down miss rates, but are outperformed by adaptive placement caches of lower sizes. The primary reason why adaptive placement caches offer higher performance than traditional caches of higher associativity and larger sizes is better sharing of cache resources among memory lines, i.e., the cache resources are shared in a manner to suit the strided access pattern.

The AMAT for a 256KB 8-way traditional L2 cache is plotted with its adaptive placement counterpart in Figure 16. The L1 access time was assumed to be two cycles, and the L2 access time 10 cycles, and L2 miss penalty was assumed to be 300 cycles comparable with numbers for the Pentium 4. The AMAT for the adaptive placement cache is considerably lower than the traditional placement cache. The access to the L2 cache is overlapped with the L1 cache access hiding any overhead of address translation due to the placement function. However, the assessment of a single cycle penalty for adaptive placement in L2 does not alter the performance difference appreciably.

Figure 17 shows that the performance of 256KB 8-way APC compares favorably to a fully associative cache of the same size and outperforms the fully associative cache for many benchmarks. A fully associative 256KB cache with 256 byte lines has an associativity of 1024 which while not practical serves to illustrate the efficiency that can be achieved via adaptive placement when stride values are known. Adaptive placement caches outperform fully associative caches for some benchmarks due to the susceptibility of fully associative caches to pollution effects. Considering that caches with high associativities and larger sizes are expensive and power hungry, adaptive placement caches offer an effective and efficient alternative. The line size chosen in the plots was 256 bytes.

A comparison of power and area costs of traditional placement caches versus adaptive placement caches

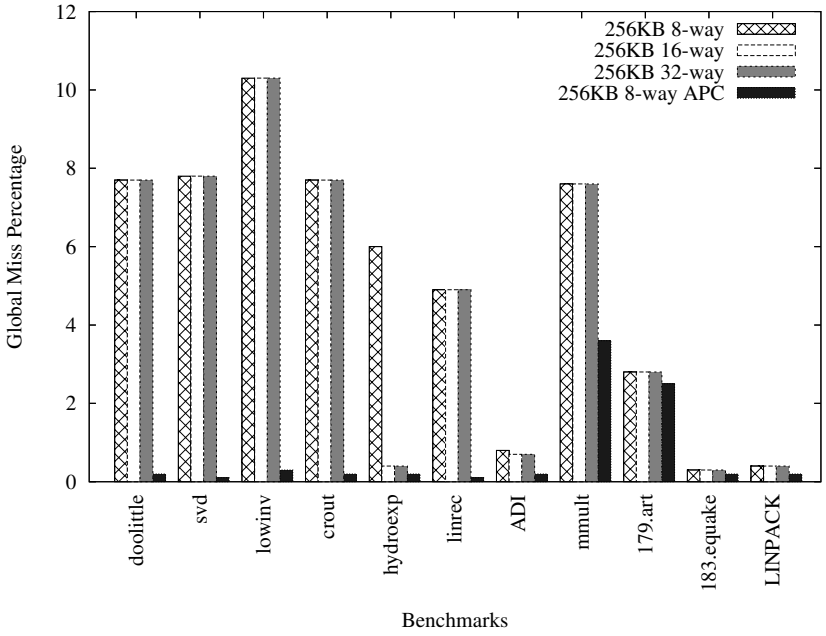

Figure 15: Adaptive placement versus higher associativity traditional caches

are shown in Figures 18 and 19. The data points reflect global miss rates averaged across all benchmarks versus increasing cache sizes and associativity. A line parallel to the y-axis reflects the averaged miss rate for the same configuration of traditional and adaptive placement. For traditional placement caches, miss rates lower considerably as cache sizes and associativity increase relative to the adaptive placement caches. Extrapolating the curves indicate that very large cache sizes and high associativity are required for traditional caches to provide miss rates comparable to adaptive placement caches. The additional area costs for the adaptive placement cache consists of the extra tag bits required for identifying memory addresses uniquely, i.e., the tag matching logic requires the original tag and the original index to be stored. The stride register does not add any appreciable area costs. Thus, for a 256KB 8-way L2 cache with 256 byte lines, an additional seven bits per set has to be stored, a total of 896 bits, which is an addition of less than 0.1% area to the cache.

Some of the benchmarks had a non power of two dimension and as a result, there were accesses with strides that were non power of two. The placement function applied was optimized for a new stride $K' = gcd(K, K')$, where K was the original stride, and K' is a power of two, as discussed in Section 3.7 and the performance of these benchmarks also improved with adaptive placement. We studied two SPEC2000 benchmarks which had the major data structures accessed in unit strides. Thus, the adaptive placement chosen is expected to be the same as traditional caches. The benchmark 179.art had a major portion of the misses contributed to by non array structures, and therefore we mapped the major array data structures to just a few sets (eight) in the cache. This offered a slight performance improvement because the array data structure was relegated to eight sets in the cache and therefore the other sets were not polluted with the array data, and because the array was accessed with unit stride touching all array elements, forcing the array to map to a few cache sets did not cause any negative effects on performance. For the benchmark 183.equake a similar

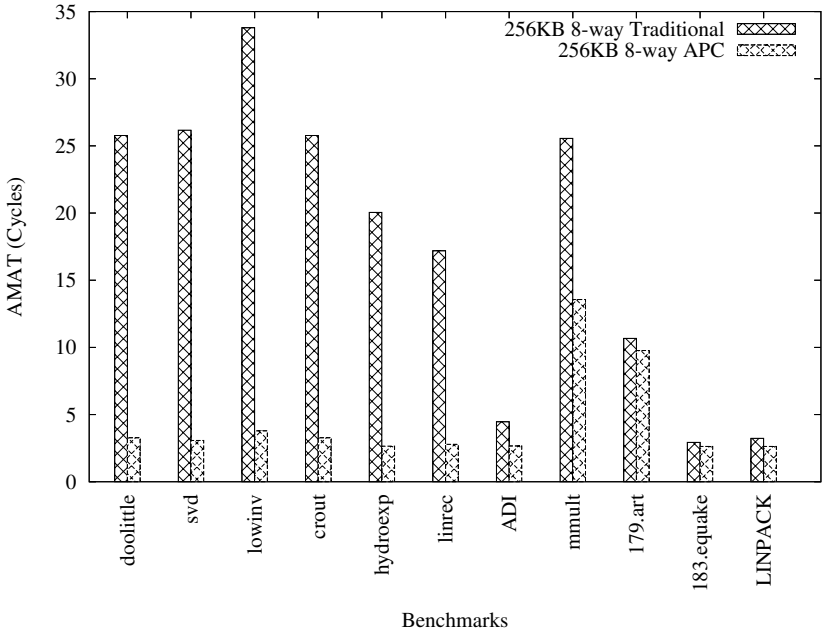

Figure 16: AMAT comparison of traditional and adaptive placement

policy was followed—whenever the major arrays were accessed the entire memory region was mapped to just sixteen sets in the cache. Again, there was no appreciable drop in performance when the entire memory was mapped to a few sets in the cache. This suggests that substantial power savings with very low, if any, performance degradation can be obtained by turning off a large number of cache sets, for certain access patterns (example, when an array is accessed with unit strides) under compiler control using precise domain specific information when available. Analyzing, exploiting, and quantifying the performance improvements of this form of optimization is the focus of some of our current efforts.

We evaluated the utility of adaptive placement at higher associativity and larger cache sizes. Figure 20 compares miss rates across various adaptive cache configurations. Higher associativity and larger cache sizes were found to have very limited impact (0.1-0.2% difference in miss rates). We surmise that this is due to the fact that the 256KB cache was sufficient to capture the memory footprint or working set of the benchmarks we analyzed. The data is further evidence of the increased efficiency of caches with adaptive placement [3]. In an environment where applications other than scientific computations are being executed, we can envision the proposed techniques being used to turn off ways and sets (as others have proposed [12, 32]) guided by compilers or dynamic optimizers.

While multiple threads in a scientific application naturally compete for cache resources, the placement function is thread specific and thus remains a part of the hardware thread context. While we have not completed an implementation to support multithreading, the basic idea is as shown in Figure 1 - the placement function must be saved across hardware thread context switches. An important question is the interactions of references in the L2 across threads. Note that the placement function is determined based on structured accesses across the entire multidimensional array. Thus conflicts are optimized across the *entire* set of defined, structured, accesses. A multithreaded implementation of the benchmarks simply determines *when*

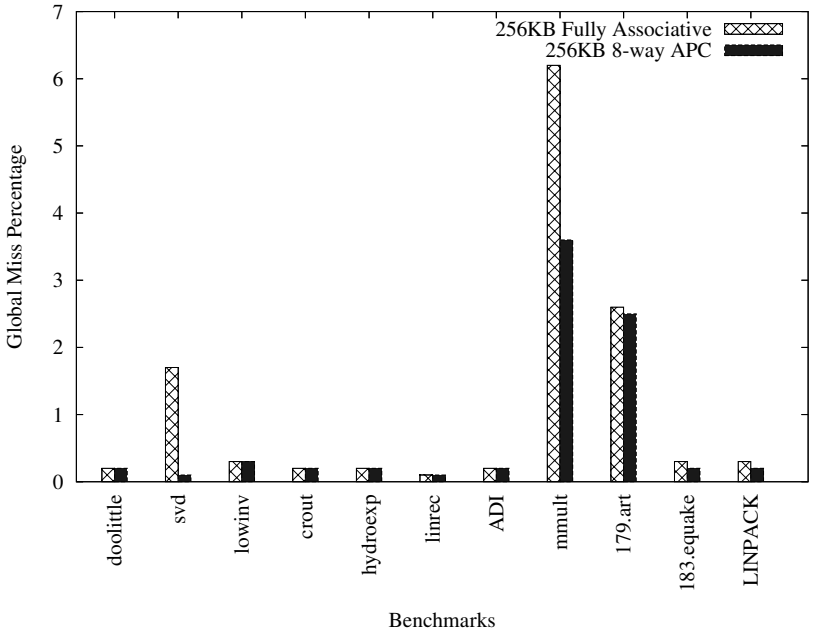

Figure 17: Comparison of an 8-way APC to a fully associative traditional cache

these accesses occur, and therefore aside from pollution effects does not change the basic analysis. Experiments to quantify the impact of multithreading is a current activity. Even if the threads were independent, adaptive placement at worst can lead to an increase in the number of cold start misses when switching contexts. Adaptive placement caches not only provide better sharing of cache resources, but, also offer better control of cache resources. In this context, analysis of multiple threads can lead to the selection of placement functions for each thread that reduces thrashing across threads. This analysis is also part of our current studies.

5 Relation to Compiler Optimizations

The performance evaluation section presented results for adaptive placement in isolation and in comparison to implementations that do not employ many sophisticated compiler optimizations because it is not offered as an alternative to compiler optimizations, but rather as an additional tool for compilers to exploit. *Adaptive placement does not alter the memory layout or memory access pattern in any manner as most compiler optimizations do, it merely changes the placement of a memory location in the cache, and is therefore complementary to compiler optimizations.* A thorough evaluation of the interaction between adaptive placement and existing compiler optimizations is beyond the scope of this paper. However, we provide some examples and discussion of how compiler optimizations can be be further enhanced with adaptive placement.

Data re-layout exploits group temporal locality, i.e., the temporal locality across references, and converts that into spatial locality. For example, if access to memory address A is followed by access to memory address B, spatial locality can be enhanced by laying out data at address A adjacent to address B. If access strides

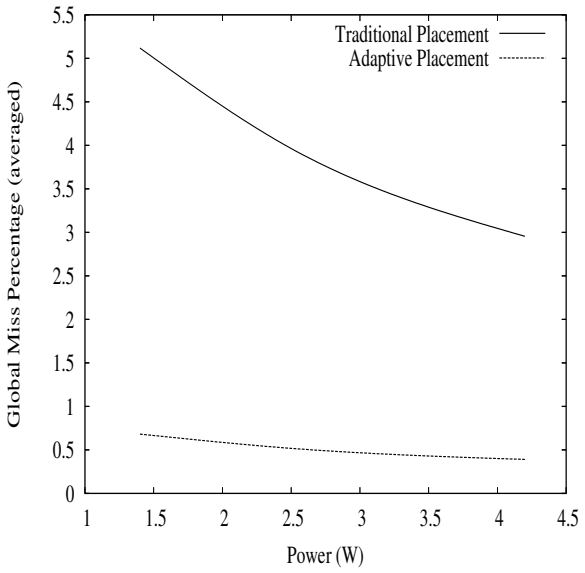

Figure 18: Comparison of power costs for traditional placement caches and APC

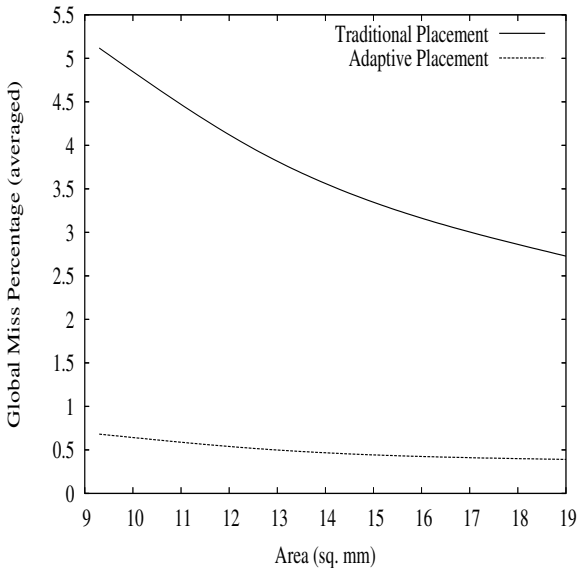

Figure 19: Comparison of area costs for traditional placement caches and APC

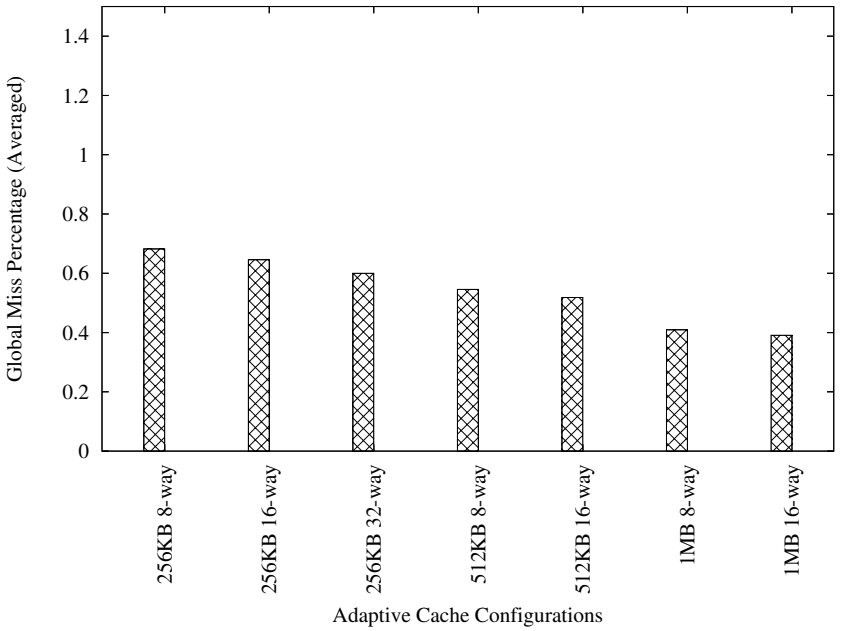

Figure 20: Miss rates of adaptive placement cache configurations

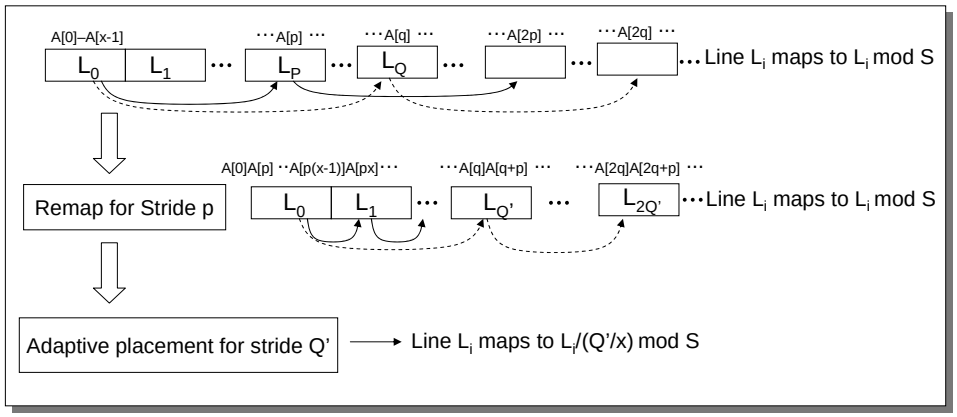

Figure 21: Combining data re-layout optimizations with adaptive placement

vary at run-time and across program segments, the memory accesses required to re-layout memory can offset any potential performance gains. Adaptive placement, on the other hand, enhances temporal locality in the cache, i.e., it tries to ensure that cache lines contain data that are likely to be reused, i.e., tries to increase the cache *efficiency*. If access strides vary at run-time, the hardware mechanism for adaptive placement is simple enough to be adapted with very little cost and does not require expensive re-layout of memory.

The orthogonality of data re-layout and adaptive placement can be explained through a simple example. Figure 21 illustrates an array accessed with two strides p and q (corresponding to line strides P and Q). In addition to the techniques mentioned in Section 3.5, we can use re-mapping along with adaptive placement to accommodate multiple strides as Figure 21 illustrates. This is especially of interest if one stride is known at compile-time, and the other is computed at run-time. An example would be accesses to a single field within multiple records, where the records themselves are accessed with a stride computed at runtime. The records can be re-laid out in memory such that all the same fields are laid out contiguously in memory, following which adaptive placement can be applied customized to the run-time access stride. This strategy can be generalized to handle concurrent accesses with multiple strides.

Memory reference patterns are often altered to increase cache locality and is done by many optimizations including loop interchange and loop tiling. Loop tiling partitions the loop iteration space into smaller blocks to enable data used in a loop to be available in the cache for reuse. Loop tiling when used with modulo placement can provide enhanced cache performance, but it increases code size, and increases the number of branch misses because tile sizes will have to be kept small with traditional placement. With adaptive placement, loop tiling can benefit from increased tile sizes as a result of better sharing. Again, if the stride or the iteration space is determined at run-time, adaptive placement is beneficial. Another example is data pre-fetching which helps reduce access latency to memory. However, because it increases bandwidth demand, and with higher miss rates with traditional placement it may be infeasible in certain cases. Whereas, with adaptive placement the lower miss rates can enable various pre-fetch optimizations.

6 Conclusion and Future Work

We have described an approach to optimize the memory performance of multidimensional arrays by relaxing the fixed placement constraint currently used by cache hierarchies. By building on the work of data skewing

techniques we can analyze multidimensional array accesses and produce cache placements that can significantly reduce AMAT and power (for a given performance level). By giving software greater control over the cache structure we can recover some of the performance headroom that is obscured by fixed, design time partitioning of cache resources across memory lines.

Among the many directions for future research include optimizations for multiple strides within and across multiple arrays within a program regions and determining how to make use of the analysis at multiple levels of the cache hierarchy. In particular, we have just started this line of inquiry and expect better quantification of power and performance, as well as development of algorithmic techniques for (re-)computing placement functions.

Our current efforts include leveraging the control of cache resources offered by adaptive placement to enhance performance of shared L2 and L3 caches in multicore systems, by selecting core specific placement policies. Whereas this paper was intended to propose the approach and provide convincing evidence of the efficacy of adaptive placement, our current efforts are focused on deeper analysis of the limits and their encapsulation into engineering software infrastructures for general purpose use.

References

[1] Agarwal, A., and Pudar, S. D. Column-associative caches: A technique for reducing the miss rate of direct-mapped caches. In *ISCA* (1993).

[2] Bennet, B. T., and Kruskal, V. J. Lru stack processing. *IBM Journal of Research and Development 19*, 4 (1975), 353–357.

[3] Burger, D. C., Goodman, J. R., and Kagi, A. The Declining Effectiveness of Dynamic Caching for General-Purpose Microprocessors. Tech. Rep. UWMADISONCS CS-TR-95-1261, University of Wisconsin, Madison, January 1995.

[4] Calder, B., G, D., and Emer, J. Predictive sequential associative cache. In *HPCA* (1996).

[5] Carter, J. B., Hsieh, W. C., Stoller, L., Swanson, M. R., Zhang, L., Brunvand, E., Davis, A., Kuo, C.-C., Kuramkote, R., Parker, M., Schaelicke, L., and Tateyama, T. Impulse: Building a smarter memory controller. In *HPCA* (1999), pp. 70–79.

[6] Chang, D. Y., Kuck, D. J., and Lawrie, D. H. On the effective bandwidth of parallel memories. *IEEE Trans. Computers 26*, 5 (1977), 480–490.

[7] Chilimbi, T. M., Hill, M. D., and Larus, J. R. Cache-conscious structure layout. In *PLDI* (1999), pp. 1–12.

[8] Chiou, D., Jain, P., Rudolph, L., and Devadas, S. Application-specific memory management for embedded systems using software-controlled caches. In *DAC* (2000), pp. 416–419.

[9] David Tarjan, Shyamkumar Thoziyoor, N. P. J. CACTI 4.0.

[10] Fu, J. W. C., Patel, J. H., and Janssens, B. L. Stride directed prefetching in scalar processors. In *MICRO* (1992), pp. 102–110.

[11] Hallnor, E. G., and Reinhardt, S. K. A fully associative software-managed cache design. In *ISCA* (2000), pp. 107–116.

[12] Hu, Z., Martonosi, M., and Kaxiras, S. Improving cache power efficiency with an asymmetric set-associative cache. In *Workshop on Memory Performance Issues* (2001).

[13] Jouppi, N. P. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA* (1990), pp. 364–373.

[14] The LINPACK Benchmark. Available at http://www.netlib.org/linpack/.

[15] Luk, C.-K., and Mowry, T. C. Automatic compiler-inserted prefetching for pointer-based applications. *IEEE Transactions on Computers 48*, 2 (1999), 134–141.

[16] The Mathematics Source Library. Available at http://mymathlib.webtrellis.net/matrices.html.

[17] McKee, S. A. Reflections on the memory wall. In *CF '04: Proceedings of the 1st conference on Computing frontiers* (2004), p. 162.

[18] McKinley, K. S., Carr, S., and Tseng, C.-W. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems 18*, 4 (July 1996), 424–453.

[19] McMahon, F. The Livermore Fortran kernels: A computer test of the numerical performance range. Tech. rep., Livermore National Labaratory, 1986.

[20] Mowry, T. C., Lam, M. S., and Gupta, A. Design and evaluation of a compiler algorithm for prefetching. In *ASPLOS* (1992), pp. 62–73.

[21] Panda, P. R., Dutt, N. D., Nicolau, A., Catthoor, F., Vandecappelle, A., Brockmeyer, E., Kulkarni, C., and Greef, E. D. Data memory organization and optimizations in application-specific systems. *IEEE Design and Test of Computers 18*, 3 (2001), 56–68.

[22] Panda, P. R., Nakamura, H., Dutt, N. D., and Nicolau, A. Augmenting loop tiling with data alignment for improved cache performance. *IEEE Trans. Computers 48*, 2 (1999), 142–149.

[23] Peir, J.-K., Lee, Y., and Hsu, W. W. Capturing dynamic memory reference behavior with adaptive cache topology. In *ASPLOS* (New York, NY, USA, 1998), pp. 240–250.

[24] Qureshi, M. K., Thompson, D., and Patt, Y. N. The v-way cache: Demand based associativity via global replacement. In *ISCA* (2005), pp. 544–555.

[25] Rabbah, R. M., and Palem, K. V. Data remapping for design space optimization of embedded memory systems. *ACM Transactions in Embedded Computing Systems 2*, 2 (2003), 186–218.

[26] Raghavan, R., and Hayes, J. P. Reducing Interference Among Vector Accesses in Interleaved Memories. *IEEE Trans. Comput. 42*, 4 (1993), 471–483.

[27] Sair, S., Sherwood, T., and Calder, B. A Decoupled Predictor-Directed Stream Prefetching Architecture. *IEEE Transactions on Computers 52*, 3 (2003), 260–276.

[28] Sazeides, Y., and Smith, J. E. Modeling program predictability. In *ISCA* (1998), pp. 73–84.

[29] Seznec, A. A case for two-way skewed-associative caches. In *ISCA* (1993), pp. 169–178.

[30] The SPEC CPU2000 Benchmarks.

[31] Valgrind tool suite version - 2.1.2. Available at http://www.valgrind.org.

[32] Zhang, M., and Asanovi, K. Fine-grain CAM-tag cache resizing using miss tags. In *ISLPED* (2002), pp. 130–135.