Reduction of Cache Interference Misses through Selective Bit-permutation Mapping

Santosh G. Abraham and Henky Agusleo
Dept. of EECS, Univ. of Michigan
1301 Beal Avenue, Ann Arbor, MI 48109-2122
{sga,patreg}@eecs.umich.edu

ABSTRACT

Cache miss rates have a large and increasing impact on overall performance. In this report, we address the problem of cache interference in regular numerical programs dominated by strided memory access patterns. In our scheme, the interfering strides in each region of memory may be annotated by the programmer, detected at compile time, or even at run time. The algorithm developed in this report uses this stride information to compute a permutation of address bits for each memory region. During page placement, the operating system ensures that some low-order bits of the virtual page number are included in the permutation that forms the physical page number. Simulation results show that this scheme can reduce overall miss rates by nearly a factor of three for the `fftpde` benchmark from the NAS suite. In an extension of this scheme, the hardware provides support for selecting the address bits used for the set and line fields by permuting the address bits in the desired manner before each cache access. Simulation results show that the selective bit-permutation scheme yields a factor of four improvement in overall miss rates compared to the standard bit-selection scheme for the blocked matrix-multiplication and LU-decomposition programs on typical first-level cache configurations. The `tomcatv` and `swm256` benchmarks have regular strides and the highest miss rates on typical first-level data cache configurations in the SPEC92 suite. Selective bit-permutation achieves a factor of two improvement in overall miss rates on these two benchmarks.

Key Words: cache interference, cache conflicts, bit-selection mapping, cache modeling, exploiting reuse

1 Introduction

Processor performance has been increasing at 50% per year but memory access times have been improving at 5-10% per year only. As a result, the latency of cache misses in processor cycles is increasing rapidly. Even a 6% first-level cache miss rate can halve the performance of the DEC Alpha processor compared to a system with an ideal memory system [18]. Another trend in computer architecture is toward processors that issue more than one instruction/operation per cycle, such as VLIW or superscalar processors. The latency of cache misses will be higher in such future processors with higher clock rates and wider issue widths. Therefore, it is important to develop techniques to reduce cache miss rates.

Caches are characterized by the following three major parameters: cache size, line size and associativity [5]. In the *standard bit-selection* mapping, the binary representation of an address is divided into three contiguous fields: tag, set, and line fields. The *set field* is used to index into one of the sets of the cache. The *tag field* is compared to the tags of the lines within a set. If the tag field of the incoming address matches with one of the tags in the set, there is a *cache hit*. On a cache hit, the line field of the incoming address is used to select one of the words in the matching line. If the tag field does not match any of the tags in the set, there is a *cache miss* and the next lower level in the memory hierarchy is accessed. In a *k-way set-associative cache*, each set in the cache contains k distinct lines. In a *direct-mapped cache*, each set contains only one line. In contrast to the standard bit-selection mapping, we propose *selective bit-permutation* mapping where the set and line fields are obtained by selecting certain bits from the address under program control and the bits in a field are generally not contiguous.

Intuitively, there are two major sources of avoidable cache misses in practical cache organizations. *Capacity misses* occur when the working set of a program does not fit in the cache. For regular numerical programs, there are restructuring techniques such as blocking that can reduce capacity misses. *Conflict misses* occur when the addresses in the working set are mapped unevenly into the sets of the cache and the number of addresses (or rather cache lines) mapping into some sets exceeds the associativity of the cache. Though conflict misses can be reduced by increasing cache associativity, a larger associativity can influence other parameters of interest unfavorably, such as silicon area (or number of chips) and cache access time. For instance, Jouppi describes why direct-mapped caches are desirable in first-level CPU caches [9].

Conflict misses can account for a large fraction of overall cache misses, especially in direct-mapped caches.[1] Recently, several cache organizations have been proposed for reducing cache conflict misses. In the victim cache organization, a small fully-associative cache maintains a set of lines that have been evicted due to conflicts in the main set-associative cache [8]. In the victim cache approach, the size of the victim cache limits the number of conflicts that can be handled. In column-associative caches, a conflicting line uses a different hashing function to map into another set in the cache [2]. With this organization, the cache usually has the fast access time of a direct-mapped cache but a miss ratio close to a two-way set-associative cache. In the column-associative cache, the number of conflicts that can be handled is proportional to the cache size. The skewed-asociative cache also uses a secondary hash if the primary hash fails [13].

In regular numerical programs typical of many engineering and scientific applications, the access streams have regular strides. Traditionally, numerical programs have been optimized for vector machines. Recently, there has been considerable interest in migrating this workload to workstation type systems because of their greater perceived cost-effectiveness. Though workstation systems have comparable processing capabilitites, they do not have the memory bandwidth of vector systems. Also,

[1] Though there are some differences in the precise concept, conflict misses are also referred to as cache interference misses and mapping misses.

```
      parameter (n = 1024)
      real  ..., B(n, n)

      do 10 k = 1, n
      ...
         do 10 j = 1, n
            ...
            ... = ... B(i,j)
            ...
 10   continue
```

Figure 1: Sample program segment illustrating stride conflicts

even though numerical programs have strong spatial locality properties, these locality properties are not exploited directly by the memory hierarchy of workstations. As a result, such applications tend to have large cache conflict miss ratios, especially for certain program and cache parameters.

Access streams with certain strides use only a small fraction of the sets in the cache. For instance, consider the Fortran program segment in Fig. 1. The $B(i, j)$ reference has a stride of 4K because column-major storage is used and each array element is 4 bytes. A direct-mapped cache of size 8KB with a 32-byte line size has sufficient capacity to store the entire row of B occupying 4KB. However, the entire access stream is mapped on to two sets by the standard bit-selection mapping and the cache misses on every access. In this case, the standard technique for reducing bank conflicts in vector machines is to choose an array dimension of 1025. However, even when this technique is applied, four distinct accesses are mapped to each set and the miss rate is one. Hashing the entire address can solve the cache conflict-problem. However, such hashing can potentially decrease the performance for the more common case of unit strides.

In this report, we address the cache interference miss problem for regular numerical programs. We present an algorithm for analyzing strides and determining the appropriate bits to be used for the set field and line field. For instance, in the example described above, our method chooses the bits 0,1,12(i.e. $log 4K$),13,14 for the line field and the bits 15 to 22 for the set field. Each of the accesses in the stride maps to a different cache line There are no cache conflicts with this bit-permutation mapping and the miss rate is zero. We also present three schemes to implement the algorithm. In large physically-indexed caches, the set field overlaps with the page frame number and these bits are affected by the virtual-to-physical translation. These bits can be controlled during the placement of pages by the operating system [10]. Thus, in the first scheme called the Limited Selective Bit-Permutation ($LSelBiP$), the application provides the operating system with the desired permutation of the address bits (excluding the bits that are part of the page offset). The operating system uses this information in the mapping process. The speed advantage of the standard bit-selection mapping is preserved, but the specific bits selected for the set field are tailored to the access pattern into regions in the memory space. This scheme requires an interface through which the application can provide desired permutation information to the operating system and a modification in the page mapper routine in the operating system. This scheme does not require any other changes to the hardware or the operating system and hence is a low-cost way of improving performance. In the second scheme, Selective Bit-Permutation ($SelBiP$), hardware support is provided for permuting all the address bits before each cache access. SelBiP is more powerful because the address bits used for the entire set field and the line field are controllable. SelBiP provides greater improvement in cache miss rates than LSelBiP. However, SelBiP requires hardware modifications and increases cache access

times. The third scheme is an entirely-hardware approach called the Dynamic Set Selection, where the decision to permute an address and the actual permutation, if it is decided to do so, are made in hardware. Finally, we present experimental results comparing LSelBiP and SelBiP to the standard bit-selection scheme used in most caches. For `tomcatv` and `swm256`, two floating-point SPEC92 benchmarks, and `fftpde`, a floating-point NAS benchmark, we obtain a factor of two improvement in cache miss ratios using our scheme and we almost completely eliminate the conflict misses in these programs.

There are some limitations to our approach. Firstly, our approach is targeted to a specific class of programs where the memory accesses are dominated by streams with regular strides. These strides are known at compile time or can be calculated at run time before accessing the memory region with stride conflicts. Therefore this scheme is targeted toward caches being designed for numerical applications. Secondly, in the SelBiP scheme, special instructions that can install and remove address masking are needed for accessing data structures with potential cache conflict problems. Furthermore, the SelBiP scheme requires a modification of the TLB-to-cache access path and an augmentation of the page table data structures. In the LSelBiP scheme, no special instructions are needed to carry out the required processes. Thirdly, the Dynamic Set Selection scheme needs a moderate amount of expansion of the tag directory of the cache. Finally, our evaluations have been limited to small cache sizes because we need much longer simulation runs for evaluating larger cache sizes typical of second-level caches.

The rest of the report is structured as follows. In Section 2, we contrast the proposed scheme with related work. In Section 3, we describe the selective bit-permutation algorithm. In Section 4, we consider the implementation options. In Section 5, we present experimental results comparing the performance of LSelBiP and SelBiP to that of the standard bit-selection mapping. In Section 6, we present conclusions of our work.

2 Related Work

In this section, we review previous work on characterizing cache misses into various categories as well as strategies for reducing cache misses in each category. We describe in greater detail hardware and software strategies that have been developed to reduce conflict misses.

Thiebaut and Stone [14] [19], Agarwal [1], Hill [6] [5] and others have proposed models which can explain or classify misses. These models are useful both for gaining the insight required to develop caching strategies and for evaluating these strategies. In Thiebaut and Stone's model and in Agarwal's model, expressions for miss rates are derived in terms of a few trace dependent parameters. In Hill's three C's model, misses are classified into three componenets: compulsory misses – misses that occur on first time reference to lines; capacity misses – additional misses in a fully-associative LRU cache; and conflict misses – additional misses due to the constraints of limited associativity. In the OPT model [15], capacity misses are redefined as the non-compulsory misses from a fully-associative cache with OPT replacement rather than LRU replacement. Instead of conflict misses, two other miss types are defined for a k-way cache: mapping misses – additional misses in a k-way OPT cache due to the mapping of addresses to sets; replacement misses – additional misses due to the sub-optimal replacement strategy. For the SPEC89 benchmarks, the mapping component accounts for 10-20% of the data cache miss ratio. This report is directed at reducing the mapping component for programs with regular strides because such programs tend to have a large mapping miss component.

Capacity misses can be reduced by reordering the access stream so that the reuse distance (i.e. the number of distinct accesses between two references to the same address) is reduced to less than

the cache size. Eisenbeis *et al* [3] and Wolf and Lam [21] have developed data locality optimizing algorithms that automatically block a class of regular numerical programs. Subsequently, Lam *et al* [11] and Temam *et al* [17] show that cache interference effects in blocked programs can greatly reduce the benefit of blocking using these data locality optimization techniques. In many cases, Lam *et al* found that block sizes that are often only a tenth of the computed optimal block sizes yield the best performance on practical set-associative caches because these smaller block sizes have much less cache interference even though they have some additional capacity misses. In this report, we show that bit-permutation mapping can be used to almost eliminate cache interference misses in blocked programs such as matrix-multiply and LU-decomposition so that block sizes close to the computed optimal can be used effectively.

Several hardware and software methods have been proposed for reducing mapping (or conflict) misses in instruction and data caches. McFarling [12] and Hwu and Chang [7] statically remap basic blocks in memory to eliminate instruction cache mapping misses on frequently executed basic blocks. Temam [18] develops techniques to analyze data cache interference misses in a class of regular numerical programs. As described briefly in the introductory section, victim caches [8], column-associative caches [2] and skewed-associative caches [13] are cache organization schemes that reduce cache interference misses. As illustrated in the example program described in the introductory section, the number of conflicting lines in numerical programs can be much larger than can be handled by any of the above schemes. Our approach uses the stride information to change the mapping of addresses to sets and remove the conflicts. Even when there are a large number of conflicting lines under the standard bit-selection mapping, there are few mapping conflicts in the reorganized mapping. Kessler and Hill [10] have investigated several *careful mapping* algorithms to be used by an operating system to place memory pages with the objective of reducing cache conflicts in large physically-indexed caches. These algorithms are designed to spread the distribution of accesses of a process over the entire cache without using any information about the memory access pattern of the process. Our LSelBiP scheme also place pages with the objective of reducing cache conflicts but requires information on the access pattern within the memory regions of a particular process.

3 Selective bit-permutation algorithm

In this section, we describe the algorithms used to determine the bit-permutation mapping. We first describe some concepts and terminology associated with the cache interference problem. We then describe the software algorithm used to compute the bit-permutation mapping when the strides are known at compile time or at run-time. We will use the blocked matrix multiplication program as an example as we go along. In this example, the strides of accesses are determined by the programmer.

Some of the concepts and terminology described here are based on [21] and [18]. Particularly, we will employ the terms *self-* and *cross-interference* for our subsequent discussion. In addition, we define some new terms relevant to the development of the algorithm. *Reuse* occurs when the same location in memory is accessed more than once. *Self-reuse* occurs when a static array reference in a program accesses the same memory location, possibly on two distinct points in the iteration space. *Cross-reuse* occurs when two static array references in a program access the same memory location. The *reuse distance* is the number of distinct addresses accessed between two successive accesses to the same location. A reuse is *exploitable* if the reuse distance is less than the cache size. Since the cache line size is the smallest unit of allocation/deallocation, these definitions should really be restated in terms of cache lines. Thus, reuse occurs even if two distinct memory locations are accessed but those locations lie on the same cache line.

```
      parameter (n = 128, m = 43)
      integer A(n,n), B(n, n), C(n,n)

      do 10 jj = 1, n, m
        do 10 kk = 1, n, m
          do 10 i = 1, n
            do 10 j = jj, min(jj+m-1,n)
              do 10 k = kk, min(kk+m-1,n)
                C(i,j) = C(i,j) + A(i,k)
                         * B(k,j)
10    continue
```

Figure 2: Blocked matrix multiplication program (MM)

In blocked matrix multiplication (MM), the blocking algorithm attempts to reorder the memory access streams so that the reuse distance is less than the cache size for a large majority of the references. In other words, such algorithm attempts to increase the possibility of exploitable reuse. Our objective is to ensure that as much as possible of the exploitable self-reuse is utilized by reducing cache interference between array references.

In our algorithm, we first identify all pairs of array references that can result in exploitable reuse. We identify the loop level(s) through which the reuse occurs. For instance, an array reference *A(i,j)* is reused at the innermost loop level that does not iterate on either i or j. For each pair of array references with exploitable reuse, we determine the *reuse path* which consists of the static references that are executed between a use and its exploitable reuse. Thus, an exploitable reuse can only be utilized by the cache if the static array references do not access addresses that map into the same set as the reused location.

The only kind of reuses that occur in MM is self-reuse, and they occur at the j-loop for matrix A, at the i-loop for matrix B, and at the k-loop for matrix C. Fig. 3 shows some of the exploitable reuse paths in MM. [2] Let us begin by examining the shortest reuse paths in the program; the reuse paths of C (RPC's). Reuse occurs between the same references to C. In this case, no self-interference takes place because there are no interfering references to C. Cross-interference can occur between a reference from A and that from C, and between one from B and one from C. Upon inspection, we find that the stride between an A reference and a C reference is non-constant going from one RPC to another. The same is true for the stride between a B reference and a C reference. For instance, the A-C stride in RPC(3,6) is @(C)-@(A)-n, where 3,6 denote the starting and ending line references, and @(C),@(A) denote the starting address of C and A. The same stride in RPC(9,12), however, is @(C)-@(A). Likewise, the B-C strides in RPC(3,6) and in RPC(15,18) are @(C)-@(B)-1 and @(C)-@(B) respectively. Therefore, when an A or B reference interferes with the reuse in one RPC, it may or may not interfere with the reuse in another RPC. In other words, the interferences between A or B references and the reuse do not happen in a regular basis throughout the program. Based on this observation, we only account for consistent interferences (in this example, they are self-interferences) that occur in each reuse path. In an RPB, we have two strides, one of value 1 (e.g. line 5-line 2) and the other of value n (e.g. line 8-line 2). In an RPA, we only have strides of value n (e.g. line 4-line 1, line 16-line 13). In general, we identify all reuse paths in the program. For each reference in a reuse path, we compute a symbolic expression for the difference between the location accessed by the reference and the reused location. This expression is a function of array dimensions and loop indexes. As illustrated in the example, such an expression is used to characterize the strides of an interference.

[2] From hereafter, when we mention 'reuse', we mean 'self-reuse'.

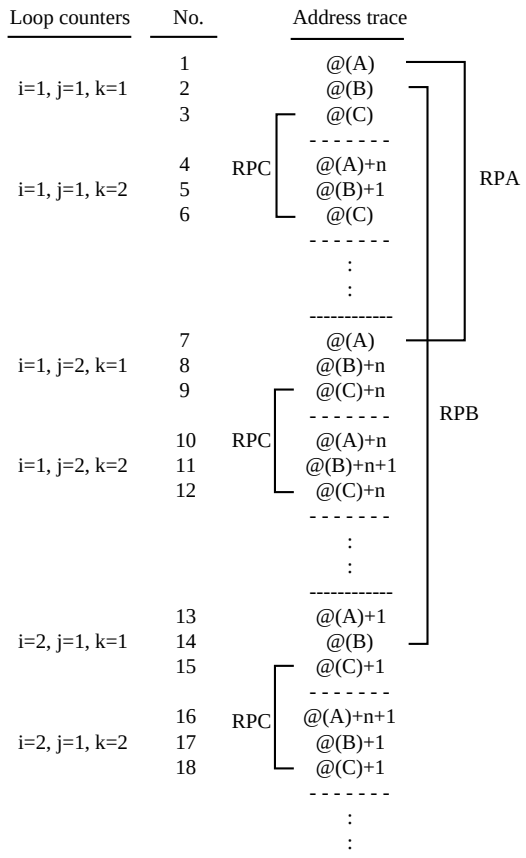


Figure 3: Reuse paths of the matrices in MM

To represent in an intuitive, yet accurate, manner the interferences in a program, we construct an interference graph. Each static array reference is a node in the graph. For each interfering reference along each reuse path, we determine the stride between the reference and the exploitable reuse associated with the reuse path. If such a stride exists, then we draw an arc from the array reference to the source of the interference. Thus, an arc between two distinct nodes represents cross-interferences between the two, and a self-arc represents self-interferences. The *length* of this interference is the maximum value of the loop index associated with the stride. The *weight* of this interference is the total number of times the interference appears in the reuse path which contains it. We annotate each arc in the interference graph with its stride, weight, and length.

Fig. 4 is the interference graph for MM. Recall that there are no consistent cross-interferences in MM. In an RPA, the interference has stride n (e.g. address 4-address 1 in Fig. 3). Since an RPA spans a complete set of iterations on k, the length of this interference corresponds to k_{max} which is m. The weight of the interference is the number of RPA's in MM, which is the total number of iterations divided by the length of an RPA. There are n^3 iterations in the program; therefore, the weight of the interference is n^3/m. The same information can be obtained from RPB's in Fig. 3 to get the stride, length, and weight values for the B-node. In an RPC, there is no interference, and therefore the node is left as it is.

The binary representation of the stride between an array reference in the reuse path and the array reference generating exploitable reuse can be written as $s_n, s_{n-1}, ..., s_0$. Consider the case where the

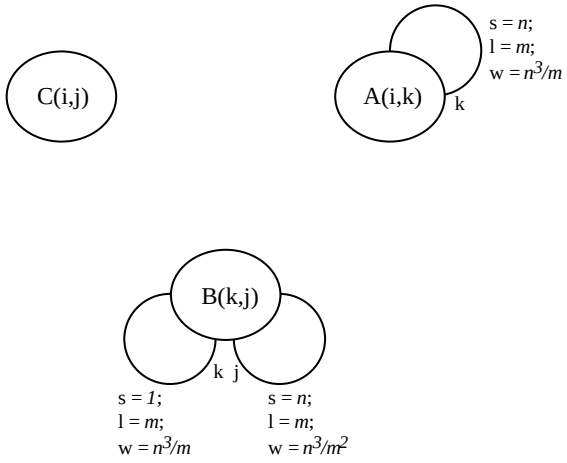


Figure 4: Interference graph of MM

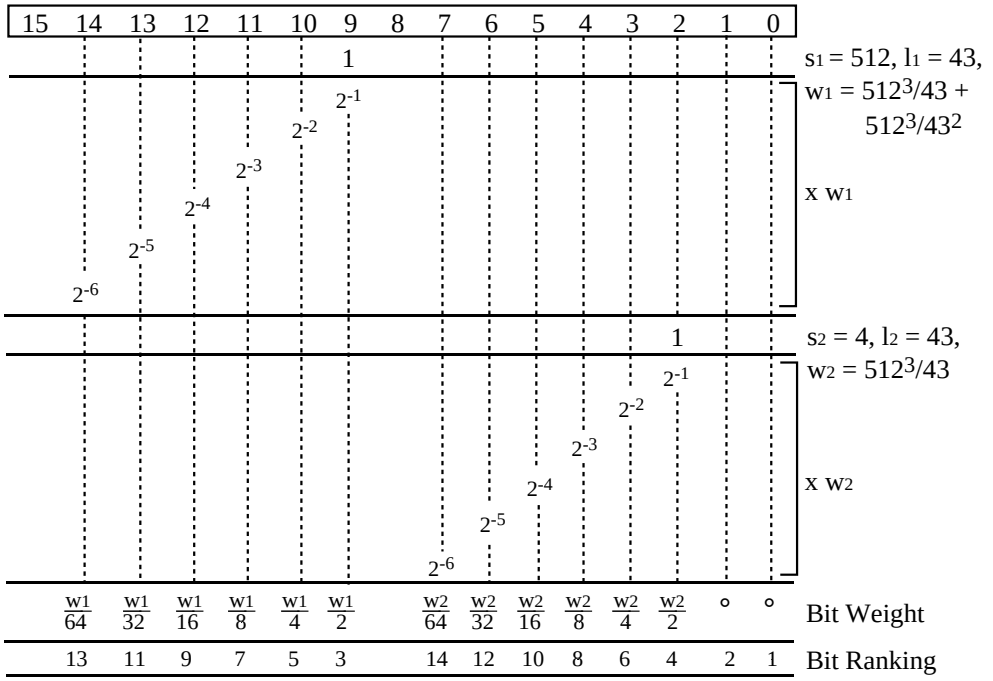
stride is a power of two. In this case, the binary representation of the stride has a single bit, s_k, that is one. Thus, it is clear that if some subset of the bits $s_{k-1}, ..., s_0$ are used for the set and line fields, then the array reference in the reuse path will consistently interfere with the exploitable reuse. However, if the bits s_k and above are chosen for the set and line fields the array reference may be located at a different set location depending on the precise multiple. Thus, consistent interference can be reduced by selecting bits that are to the left of the ones in the binary representation of the stride.

The next step is to construct the interference table consisting of n columns for a machine with an n-bit address space. [3] The purpose of constructing the table is to assign to each address bit a weight denoting the desirability of choosing that bit for the set field. Each interference arc is represented by a group of $log(length)$ rows in the interference table. The binary representation of the stride is obtained and signed-bit recoding is used to reduce the number of ones in the binary representation. [4] The negative and positive ones are both represented by 1, because both have the same effect in our application. For each interference arc, the first row is formed by entering a 1/2 at each position in the binary representation of the stride with a positive or negative one. The remaining $log(length) - 1$ rows are formed by placing a copy of the previous row and dividing each entry by 2 and shifting each entry to the left by one. Now, each row in the table is multiplied by the weight of its corresponding interfering arc. We then compute the sum of the numbers in each column. Each of the sums is the weight of its corresponding bit and denotes the desirability of choosing the bit for inclusion in the set field. The *log(size of datum)* lowest-order address bits are assigned the weight of infinity. Thus, for a 4-byte integer, bits 0 and 1 will have infinite weights. This is done to ensure that these bits will never be permuted. In the LSelBip implementation scheme, it is necessary to keep the *log(page size)* lowest-order bits unpermuted.

The final step in the process is forming the permuted address. First, we rank the bits according to their respective weights. We then pick the *log(line size)* heaviest bits for the line field of the address. Note that this selection will always include the infinitely-weighted bits. The next *log(number of sets)* heaviest bits are selected for the set field. The rest of the bits are placed in the tag field. For

[3] If we want to restrict the number of address bits that are permuted, we can restrict n to be less than the total number of address bits.

[4] In signed-bit recoding, a string of ones is replaced by a positive one (1) and a negative one ($\overline{1}$). For instance, 01111001 is replaced by $1000\overline{1}001$.



Comparison (8KB direct-mapped cache, 32-byte line size):

StaBitS:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SelBiP:	15	8	7	14	13	12	11	6	5	4	3	10	9	2	1	0

Figure 5: Interference table of MM

example, to form a 32-bit selective bit-permutation mapping which will be used to address an 8KB direct-mapped cache with 32-byte line size, we take the five heaviest bits for the line field, the next eight heaviest bits for the set field, and the rest for the tag field.

We are now ready to build an interference table of MM for such cache configuration. Substituting variables with values, and acknowledging that the size of an integer in Fortran is 4 bytes, we obtain the actual value of each stride and its properties. Fig. 5 is the final form of the table. Note that we keep bits 1 and 0 in their places as the two least-significant address bits. Again, this is because the unit size is 4 bytes. We then compare the address bits under selective bit-permutation and the standard bit-selection.

The entire selective bit-permutation algorithm is summarized in Fig. 6.

4 Implementation issues

To implement the algorithm, we present three implementation possibilities. The first is LSelBiP, in which both the stride determination and the bit permutation are done in software. The second is SelBiP, in which the strides are determined in software but the actual bit permutation is done in hardware. The third is Dynamic Set Selection, in which both the stride determination and the bit

```
SelBiP(input: program segment) {

  /* Declare reuse paths */
  Determine all exploitable reuse paths in the segment;

  /* Build interference graph */
  For (each reference in each reuse path) do
    If (conflict with reuse) then {
      Determine the symbolic expression of the difference between the
        accessed and the reused locations;
      /* This expression is the stride of the conflict */
      Calculate the length and weight of the conflict;
    }

  /* Build interference table */
  Make a table of n columns and some number of rows for an n-bit address space;
  For (each stride of conflict found) do {
    Obtain its binary representation;
    For (each 1 in such representation) do {
      power = 1;
      Current bit position is the position of 1;
      Start from the top-most row;
      For (power < length of conflict) do {
        Assign the weight (2^-power * weight of conflict) to current bit
          position;
        power = power+1;
        Shift left the current bit position by one;
        Go to the next row;
      }
    }
  }

  /* Calculate the total weight of each column */
  For (each column in the table) do
    If (column is non-empty) then
      Sum all the weights in that column;
    else
      Assign the total weight of 0 to that column;

  /* Each total weight of a column denotes the desirability of choosing the
     corresponding bit position for the set and line field */
  Rank the bit positions according to the total weight of each;
  For (each memory reference in the segment) do {
    Assign the address bits corresponding to the log(line size) heaviest
      bit position to the line field;
    Assign the address bits corresponding to the next log(number of sets)
      heaviest bit position to the set field;
    Assign the rest of the address bits to the tag field;
  }
}
```

Figure 6: The Selective Bit-permutation algorithm

permutation are done in hardware.

4.1 The LSelBiP scheme

The motivation for an entirely-software scheme for selective bit-permutation is its ease of implementation. The delivered improvement can be enjoyed via moderate modifications to the system softwares, i.e. compilers and operating system, without any additional hardware. The approach that we take assumes that the computer system supports both cache and virtual memory. From the point of view of a cache, an address consists of a tag field, a set field, and a line field. From the stand point of a virtual memory, the same address consists of a page number and a page offset.

A cache selects a set for an incoming reference by extracting bits from the set field of the reference's address. When the address is virtual (not translated), the cache is said to be virtually-indexed; when it is physical, the cache is physically-indexed. Taylor *et al* [16] and Wang *et al* [20] elaborate on why physically-indexed caches are desirable, especially for cases where the needed address translation does not seriously impair access time. Often times, the size of a physically-indexed cache is so much bigger than the size of a memory page that the set field of an address overlaps with the page number. Fig. 7 illustrates this case for 64KB direct-mapped cache with 32-byte line size and 4KB page frames. Thus, address translation affects large physically-indexed cache performance because page translation is partially responsible for selecting a cache set. In the same spirit as Kessler and Hill's careful-mapping algorithms, we introduce a *selective-mapping* algorithm, which employs a variant of the selective bit-permutation algorithm, as another solution to the problem when additional information regarding the memory access pattern is available.

In this scheme, the compiler determines the stride of a conflict and its associated properties and calculates the bit permutation. It then generates a mask that carries information regarding the permutation and inserts calls in the resulting object code to communicate the masking information to the operating system. Preferably, this routine is executed at the beginning of the program; however, strictly speaking, it can be executed anytime before the relevant memory references. In the case where the stride of a conflict can only be determined during run-time, the compiler will also need to add the mask-generating routines to the object code. The page-fault service routine of the operating system uses as its page placement algorithm a variant of the selective bit-permutation algorithm, which permutes only those address bits in the page-number field that overlap with the set field. For large physically-indexed caches, the permuted bits will include several high-order set bits. In our example memory parameters, four highest-order set bits are permuted. On a page fault, the page fault service routine inspects the masking information. If a mask is found, the routine uses it to permute the bits in the fault-causing virtual page number. The permuted bits form the low-order part of the physical page number. The service routine has the flexibility to select the high-order part of the physical page number from the pool of available page numbers with matching low-order part. Fig. 8 depicts the interaction between the compiler, the object code, and the operating system in this scheme.

4.2 The SelBiP scheme

The LSelBiP scheme has an inherent limitation in that it limits the degree to which address bit permutation is performed. Here, we present an approach where the selective bit-permutation algorithm is employed to its fullest extent.

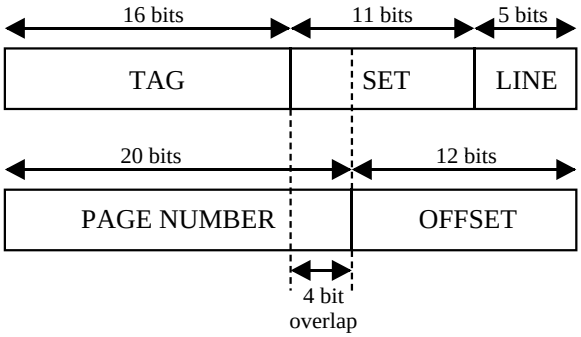


Figure 7: Address Field Overlap

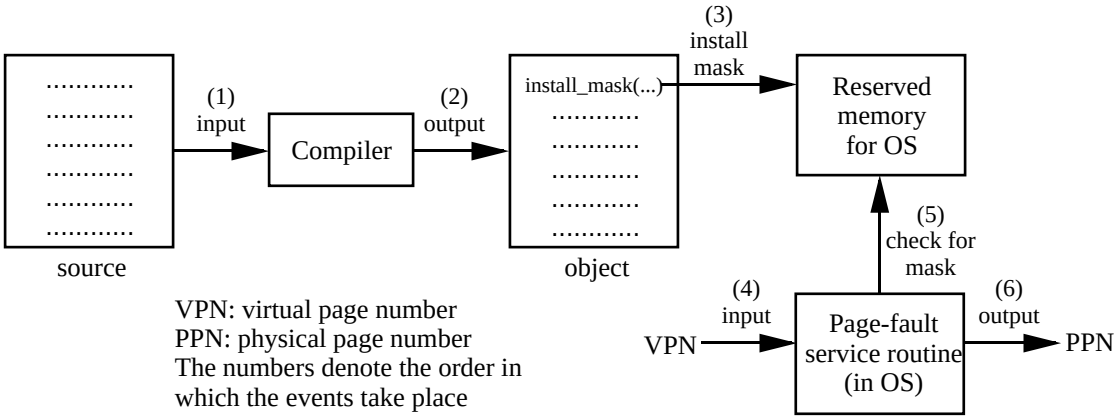


Figure 8: The LSelBiP Scheme

4.2.1 Virtually-addressed caches

In this scheme, the compiler calculates the strides of accesses in a program segment, and uses information on the cache parameters to calculate the SelBiP mappings to avoid the detected interferences. It then encloses the program segment with a special pair of instructions which tell the hardware that an alternative mapping should be used for that region of the program. The instruction pair, which acts as would a pair of *begin* and *end* statements, install and remove the SelBiP mapping from a set of special registers in hardware. In addition, they also invalidate all cache entries at both the entry and the exit points of such region in the program to avoid aliasing problems.

For an n-bit address space, the special registers, which we will denote collectively as the *selectors*, is an array of n registers whose width is $log(n)$ bits. Thus, for a 32-bit address space, it is a collection of 32 5-bit registers. The output port of each of these registers is connected to the select line of a 1-bit $n \times 1$ multiplexer. The input to the n multiplexers is an address, which could be either physical or virtual depending on the architecture of the memory hierarchy. The output of the multiplexers represent the new address obtained under the SelBiP mapping, and go to an n-bit register. The content of this register is then used to address the cache. Fig. 9 depicts this hybrid scheme for a 32-bit address space.

Assuming that the selectors and the input address are stable before the rising edge of the clock, the critical path of this implementation is the propagation delay of the 32x1 multiplexers. Based

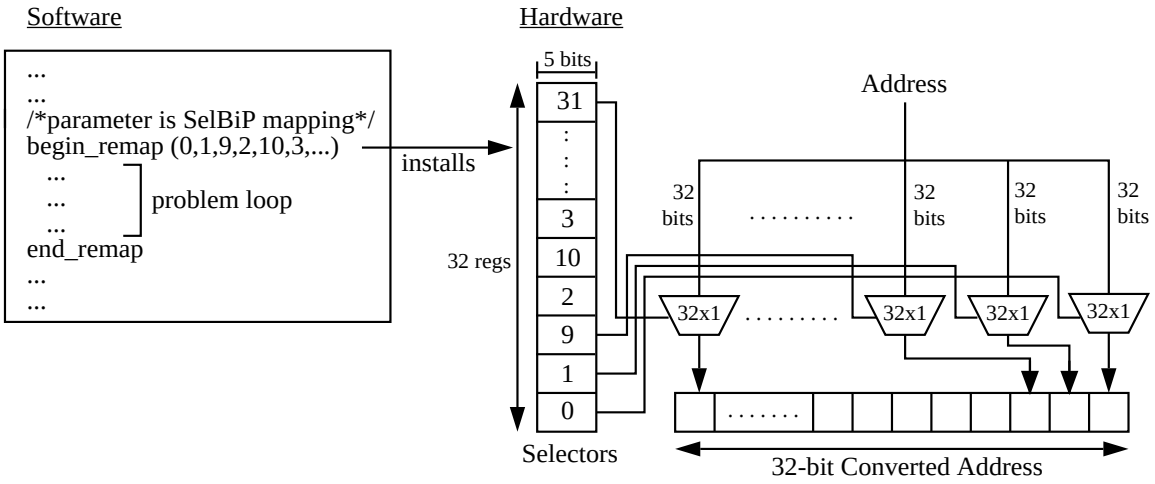


Figure 9: The SelBiP Scheme

on consultations with VLSI designers, this delay should be less than one-fourth of the CPU clock cycle. In addition, since we are using this scheme only on program segments with strided accesses and bypassing it for the rest of the program, the average cache latency for the entire program should increase only slightly.

In many numerical programs, the dimensions of various arrays are dependent on external input. The strides are often a function of the dimensions of some of the arrays and are not known at compile time. In this case, the compiler will add run-time code to calculate and install at run time the SelBiP mapping once the needed information is gathered. The new mapping can then be used to avoid conflicting set accesses. A cache miss is incurred the first time the data is accessed under the new mapping.

4.2.2 Physically-addressed caches

In physically-addressed caches, virtual addresses are translated into physical addresses using a TLB (Translation Lookaside Buffer) lookup. The physical address is then partitioned into three fields and used to access the cache.

The key feature of SelBiP implementation for physically-addressed caches is that each page table and TLB entry is augmented with an additional field to specify the address permutation to be employed for all accesses into that page. Thus, the new field functions as would the selector registers described previously. On the required TLB lookup for each address, we also obtain additional information on the set and line mapping at the same time. The physical page number, the page offset, and the permutation information are fed to a permutation hardware that will give the permuted address to be used for cache access.

4.3 The Dynamic Set Selection scheme

Loop-dense numerical applications (LDNA's) have been the mainstay of the workload of engineering workstations ever since their conception. These applications have a characteristic that is

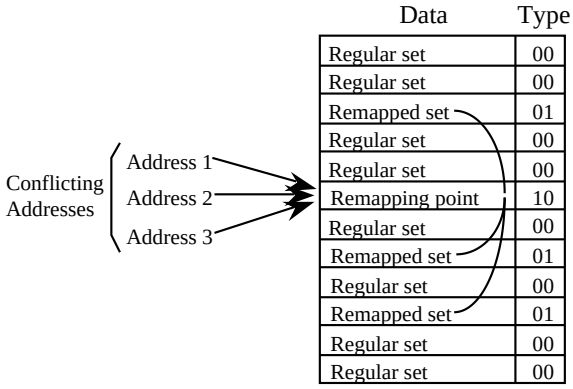


Figure 10: Dynamic Set Selection Scheme

particularly attractive for the application of SelBiP [5]; namely, they have regularly-strided accesses. That is, the loops inside an LDNA access matrices in regular (constant) strides. The desire to exploit this phenomenon is what motivates us to develop an entirely hardware implementation. As before, we assume a write-through policy.

The idea behind dynamic set selection is to use the cache itself as a secondary cache after a first time miss. The first access is always conducted using the standard bit-selection (StaBitS). When this is a miss, a second access is conducted using SelBiP. Thus, like the column-associative cache and victim cache, a single cache access is required for non-conflicting addresses in the cache and two cache accesses are required for conflicting ones. Fig. 10 provides a conceptual overview of the dynamic set selection.

Under the scheme, each cache set can be one of three types. A cache set can contain data installed under the regular StaBitS mapping (*regular set*). Alternatively, a cache set can contain data installed under the SelBiP remapping (*remapped set*). Finally, a cache set can trigger a remapping to another set because of conflicting addresses accessing it. Such set then acts as a *remapping point*. The type of sets in the cache is maintained by a two-bit tag. Finally, a remapping point and all its remappings that were caused by the same constantly-strided conflicts belong to a *remapping tree*.

Imagine that a regular, non-permuted address a is used in a write to the cache. We call the set accessed by that StaBitS address the *base* set. Fig. 11 illustrates the next steps taken under the scheme. A StaBitS access is a hit if, in addition to the usual tag match, the the base set is a regular set. When this happens, data is written to both the set and lower-level memory. Otherwise, the cache control logic checks the type of the base set. If it is either a regular set or a set that has been declared as a remapping point, the cache controller continues to see if a is a part of a strided access stream. If the base set is a remapped set, data is written to it and to lower-level memory.

When the cache controller detects that the miss-causing address a is part of a strided access stream, it tries to remap the address to another set using SelBiP. When the remapping is successful, data is written to the new set which we call the *target* set. However, the controller may find out that the target set is a remapping point and thus a replacement policy for it is in order. As indicated by Fig. 11, most of the time we keep the remapping point around and write the data directly to lower-level memory. It is imperative, nevertheless, that we not keep remapping points unreplaced all the time because each of them will outlive its usefulnes at some point. We suggest a construction of an *aging mechanism* for the remapping points which guarantees that a remapping point will be

[5] In this section, the mnemonic SelBiP refers to the algorithm, not to the implementation scheme.

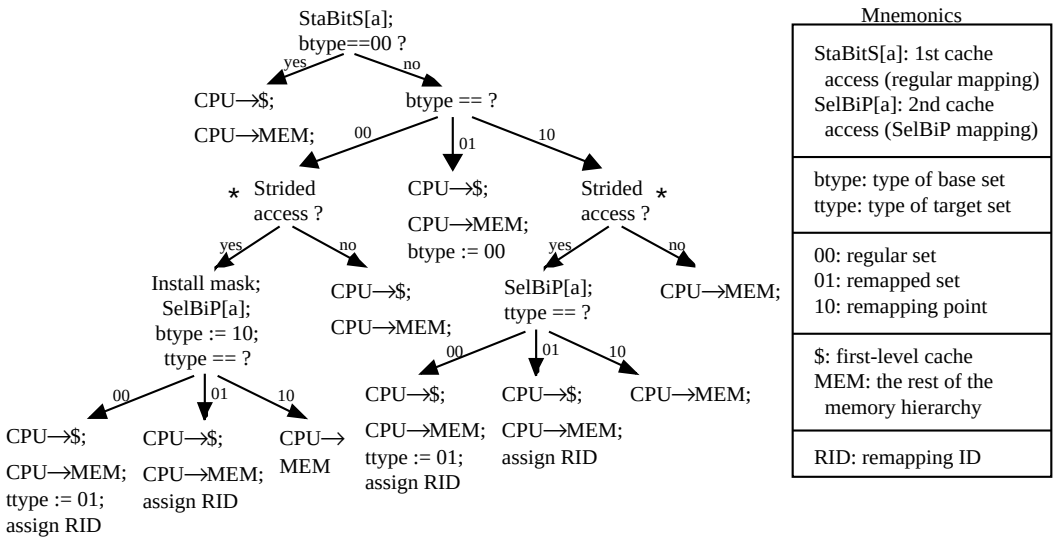


Figure 11: Write Decision Tree

eligible for replacement after some number of accesses, and that their number does not exceed a certain predetermined threshold. It will use information such as the length and the weight of an interference to determine the life span of a remapping point. Such information can be determined from an empirical study of the memory- referencing behavior of the machine's typical workload. The existence of such mechanism also resolves some issues regarding the possibility of having more than one remapping trees per remapping point. This event corresponds to a matrix being accessed in several manners, thereby resulting in several constantly-strided access streams for the same matrix. The aging mechanism needs to employ, in addition to the interference statistics, a graph-coloring scheme to make sure that a remapping point is only remapping one constantly-strided access stream at a time. Such coloring scheme is widely implemented in compilers for register allocation purposes.

The read operation has also been expressed as a decision tree in Fig. 12. In resolving misses, it follows closely the steps taken during a write operation. The difference is that on a SelBiP access after missing on a remapping point, the scheme checks if this second access is a hit. This is not required in the write operation because write-through caches do not distinguish between a write miss and a write hit. A SelBiP access is a hit if, in addition to the usual tag match, the target set is a remapped set belonging to the correct remapping tree. This new criterion brings about the need of a *remapping ID*, which will distinguish one remapping tree from another. We recommend the construction of an ID generator that will assign a number to each remapping tree. Since in general the number of remapping points corresponds linearly to the number of matrices in an application, which is relatively small compared to the number of cache sets, we expect the modification to the cache structure to be fairly moderate both in complexity and in scale.

Let us examine closer the dynamic set selection scheme in the context of an LDNA. Upon a miss, the cache controller will determine whether the missing address is part of a strided access stream. In the case of an LDNA, the most likely answer is yes because of the strided nature of the memory references of the program. The cache controller will then make an effort to assign a new set for the data. This course of action corresponds to following the left path after the starred step. In the case of other programs, for instance database applications, memory spaces are dynamically allocated and therefore not likely to be contiguous. So, the likelihood of having a regularly-strided access stream is dramatically reduced. This results in the cache controller following the usual miss-handling scheme,

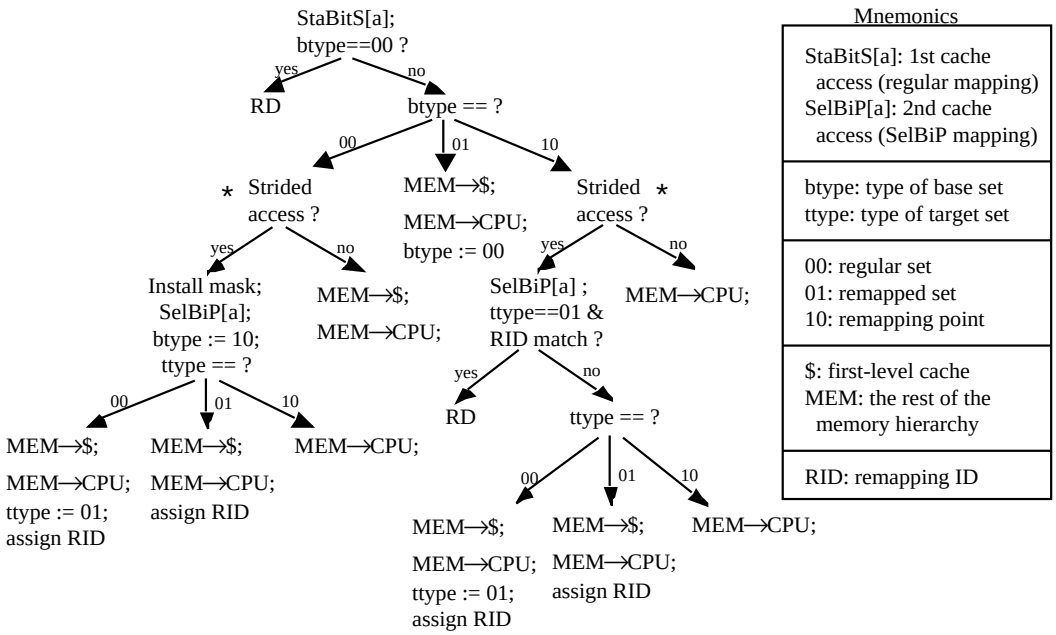


Figure 12: Read Decision Tree

which corresponds to following the right path after the starred step in either tree. In conclusion, when an LDNA is running, most of the cache sets it is using are remapped sets. In contrast, when an application of different type is running, most of the cache sets being used are regular sets.

One concern that may come to mind after closer examination of the decision trees is the possibility that remapped data and regular data are replacing one another before either one is reused. Such event could easily lead to thrashing. The question now is how likely either type of data remains intact before it is reused. To answer that, let us take a closer look at the structure of a typical LDNA. An LDNA is comprised of clusters of loops with some common matrices being operated on going from one loop cluster to another. Memory references made inside a loop cluster are regularly strided. The loop clusters are separated by initializing instructions. Most of these instructions set up the loop parameters such as loop indexes, and do not access the matrices used in the loop cluster following them. Consequently, they are more likely to manipulate single variables (registers) instead of matrix elements (memory space). Even when some of them do access matrix elements, their number is usually small enough that only a few of the remappings are replaced. Therefore, a remapping is likely to remain intact going from one loop cluster to another. As a consequence, trashing is unlikely.

As mentioned previously, in other types of applications, most of the cache sets used are regular sets instead of remapping ones. Thus, chances are slim that a regular set will be replaced by a remapping. The possibility of thrashing is, therefore, minimal.

At the appendix of the report, we present an example implementation of the dynamic set selection for the direct-mapped, write-through cache configuration.

5 Simulation Experiments

In this section, we compare and contrast the data gathered from simulations of two caches (64KB and 8KB direct-mapped; 32-byte line size) under our bit-permutation and the usual bit-selection scheme. In the first configuration, LSelBiP implementation is assumed, whereas in the second the SelBiP scheme is used. We begin by describing the metrics we will use to evaluate the performance of our algorithm, and continue with the description of the traces and the analysis of the result.

5.1 Performance Metrics

We use two cache performance metrics to evaluate our algorithm: total miss rate, and percentage of mapping misses removed.

Total miss rate is the ratio of the number of misses to the total number of references. Using the OPT model, the total miss rate can be broken down into 4 components: compulsory, capacity, mapping, and replacement misses [15]. *Mapping misses* are of particular interest to us, since they arise from the set-mapping strategy and are affected by the choice of set-mapping strategy (e.g. our bit-permutation mapping instead of the usual bit-selection scheme).

Percentage reduction of mapping miss rate is the percentage by which the number of mapping misses obtained under the usual bit-selection scheme is reduced by our bit-permutation scheme. Thus, it follows that:

$$\%mm_{reduced} = \frac{mm_{bit-selection} - mm_{bit-permutation}}{mm_{bit-selection}} \times 100\%$$

where mm stands for mapping misses, and $mm_{bit-selection}$ and $mm_{bit-permutation}$ denote such misses obtained under the usual scheme and under our bit-permutation, respectively. This metric allows us to focus more on the effectiveness of our scheme (i.e. reducing conflict misses), since it excludes such miss components as compulsory and capacity misses that any cache configuration must suffer.

5.2 Environment and Trace Description

We compiled six Fortran programs on a DECstation 5000/120, and then used the Mips Pixie tool to generate address traces to feed directly to a modified Cheetah cache simulator [15] that simulated the configurations.

The programs are given in Table 1, and the trace of each is run to completion. [6] We selected the programs because each has regular strided accesses. In addition, MM and LU contain loops that are typical in many engineering and scientific applications, whereas `tomcatv` and `swm256` are two of the SPEC92 benchmark programs that exhibit the largest data cache miss rates [4]. `fftpde` manipulates matrices that are larger than those present in the SPEC92 suite and thus is suitable for illustrating misses in bigger caches. The blocking factors for blocked programs are determined from the formulas $n \geq m^2 + 3m$ and $n \geq m^2$ respectively where n is the cache size and m the desired blocking factor [21].

[6] Except `fftpde`, whose run is simulated up to 50 million data references.

Name	Description
MM	Blocked multiplication of 2 128x128 matrices; blocking factor = 43
MMbig	Blocked multiplication of 2 256x256 matrices; blocking factor = 254
LU	Blocked LU-decomposition of a 128x128 matrix; blocking factor = 45
tomcatv	Mesh-generation program (SPEC92)
swm256	Shallow water equation solver (SPEC92)
fftpde	3-D Fast Fourier transform (NAS)

Table 1: Description of benchmark traces

Pro-	Total Miss Rate		Mapping Miss Rate		
gram	StaBitS	LSelBiP	StaBitS	LSelBiP	% reduction
MMbig	0.3565	0.1996	0.3252	0.1644	49.5%
tomcatv	0.0948	0.0640	0.0533	0.0226	57.6%
fftpde	0.0699	0.0253	0.0528	0.0083	84.3%

Table 2: Miss rates for 64KB direct-mapped cache

5.3 Simulation Results and Analysis

We present our simulation results in Tables 2 and 3. Note that if the mapping miss rate obtained under the bit-permutation scheme is higher than that obtained under the usual scheme, the percentage of mapping misses removed becomes a negative quantity.

5.3.1 Performance of LSelBiP on 64KB cache

Our simulation assumed that a physical page frame with the desired characteristics is always available. The results therefore represent the improvement of our scheme over the page-coloring scheme [10] when under each scheme the page mapper can successfully find an available physical page frame with the appropriate characteristics. Even when permutation is restricted to the bits in a page number, the LSelBiP scheme manages to achieve substantial reduction of misses. With only four set bits to replace, it halves the total miss rate of each trace on average. The scheme also successfuly removed approximately 50% of mapping misses of MMbig and `tomcatv`. In `fftpde`, it reduces the mapping misses to less than 1% the original count. This suggests that limited permutation is a very attractive implementation option.

5.3.2 Performance of SelBiP on 8KB cache

The SelBiP scheme performs very well for MM and LU traces, reducing the total miss rate of each trace by as much as a factor of 4. Mapping misses for each trace is also greatly decreased in both cache configurations. The scheme successfuly removed up to 75% of the existing mapping misses. The success of our scheme lies mainly in the ease at which the strides of accesses in both

Pro-	Total Miss Rate		Mapping Miss Rate		
gram	StaBitS	SelBiP	StaBitS	SelBiP	% reduction
MM	0.3818	0.0932	0.2475	0.0760	69.3%
LU	0.4878	0.1447	0.3779	0.0901	76.2%
tomcatv	0.1701	0.0649	0.1258	0.0217	82.7%
swm256	0.1820	0.0990	0.0750	0.0302	59.8%

Table 3: Miss rates for 8KB direct-mapped cache

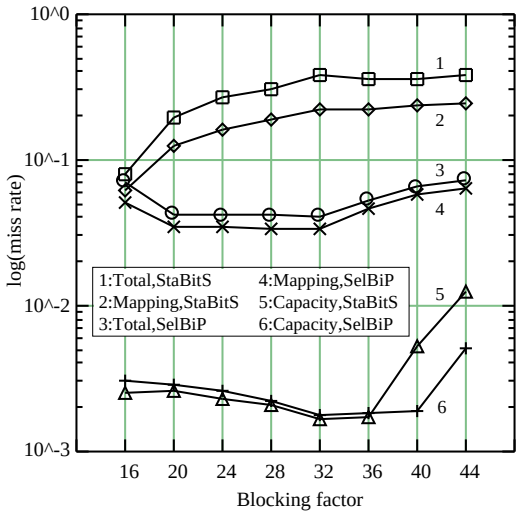


Figure 13: Miss rates of MM trace simulated on 8KB cache

programs can be detected. In the traces of `tomcatv` and `swm256`, the strides of accesses are not as obvious as those in the previous traces. Nevertheless, our bit-permutation managed to achieve, on average, a factor of two improvement in total miss rate and a 75% reduction in mapping misses for `tomcatv` trace.

The plot in Fig. 13 reveals that mapping misses comprise an overwhelming majority of the total miss rate for blocked MM. By reducing such misses, we are bringing down the total miss rate as well. Note that as the blocking factor is increased from 40 to 44, capacity misses for both mappings started to rise. This is due to the fact that blocks are getting too big to fit in the cache. The difference of the manner in which each capacity miss grows, however, stems from the fact that we are using different bits for the line field in each mapping. This causes different fully-associative miss rates, which in turn causes different capacity miss rates. Nevertheless, the selective bit-permutation mapping is able to maintain a superior performance relative to the standard bit-selection mapping, as the plot evidently shows.

6 Conclusion

Compared to the rapid improvements in cycle time and issue widths in high performance processors, the improvement in memory access times has been modest. Cache misses have a large and increasing impact on overall performance.

Virtually all caches have used bit-selection mapping in which the binary representation of the address is partitioned into a contiguous tag, set and line fields. In this report, we propose selective bit-permutation mapping, where the address bits are permuted before being partitioned into the tag, set and line fields. We develop an algorithm that computes the permutation required for each data structure in the program. We use the compiling technology that has been developed for automatically blocking programs. The objective of our algorithm is to choose a bit-permutation mapping so that the exploitable reuse in the program is utilized by the cache. The process involves first determining pairs of array references that result in exploitable reuse and the reuse path consisting of other array references between the use and the exploitable reuse. For each array reference along the reuse path, we determine if there is a consistent stride that can interfere with the reuse. For each such potentially interfering reference we create an arc and record its stride, length, and weight to form the interference graph. The interference table is formed from the interference graph, where the weight of each entry denotes the desirability of choosing the corresponding bit for inclusion in the set field. The permuted address is then formed by ranking the bits according to their respective weights, and assigning the heaviest bits to the line field, the less heavy ones to the set field, and the light ones to the tag field.

We conduct trace-driven simulation of six benchmark programs (2 variants of blocked matrix multiplication, blocked LU-decomposition, `tomcatv`, `swm256`, and `fftpde`) on 8KB and 64KB direct-mapped caches. The selective bit-permutation mapping obtains as much as a four fold and a six fold reduction of total and mapping miss rates respectively in the three kernel programs. In the case of the more complex SPEC92 and NAS programs, the reductions range from two to three fold for total miss rate, and from two to six fold for mapping miss rate.

We discuss the implementation options of our algorithm. We present three schemes: an entirely-software scheme, a hybrid software-hardware one, and an entirely-hardware approach. The first two options requires compiling techniques such as stride determination. The compiler provides the stride of each access, typically in regions of the program that can have high cache conflict miss ratios. It then uses this information to generate masks that will select which address bits to use for set selection or, in cases where the strides are not known until run-time, insert code that will generate such mask. In the entirely-software approach, the operating system uses the stored mask to permute the address bits during the virtual-to-physical address translation; thus it is only applicable to physically-indexed caches. In the hybrid approach, the hardware provides support for carrying out the actual bit permutation. In the entirely hardware approach, the mask generation and the bit permutation are done by the cache logic. Any of the last two schemes can be applied to either physically- or virtually-indexed caches.

7 Acknowledgements

We wish to thank Ashok Singhal of Sun Microsystems, whose suggestions led to the LSelBiP scheme, and Michael Schlansker of Hewlett-Packard Laboratories in Palo Alto, for refining the original SelBiP scheme.

References

[1] A. Agarwal. *Analysis of Cache Performance for Operating Systems and Multiprogramming.* PhD thesis, Stanford University, 1988. Available as Technical Report CSL-TR-87-332.

[2] A. Agarwal and S. D. Pudar. Column-associative caches: A technique for reducing the miss rate of direct-mapped caches. In *Proc. of 20th Intl. Symp. on Computer Architecture*, pages 179–190, 1993.

[3] C. Eisenbeis, W. Jalby, D. Windheiser, and F. Bodin. A strategy for array management in local memory. In *Proc. 3rd Wkshp. Prog. Lang. & Compilers for Par. Comp.*, 1990.

[4] J.D. Gee, M.D. Hill, D.N. Pnevmatikos, and A.J. Smith. Cache performance of the SPEC92 benchmark suite. *IEEE Micro*, pages 17–27, August 1993.

[5] J. L. Hennessy and D. A. Patterson. *Computer Architecture — A Quantitive Approach.* Morgan Kaufmann Publishers Inc., 1990.

[6] M. D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance.* PhD thesis, University of California, Berkeley, 1987. Available as Technical Report UCB/CSD 87/381.

[7] W. W. Hwu and P. P. Chang. Achieving high instruction cache performance with an optimizing compiler. In *Proc. of 16th Intl. Symp. on Computer Architecture*, pages 242–251, 1989.

[8] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proc. of 17th Intl. Symp. on Computer Architecture*, pages 364–373, 1990.

[9] N. P. Jouppi. Cache write policies and performance. In *Proc. of 20th Intl. Symp. on Computer Architecture*, pages 191–202, 1993.

[10] R. E. Kessler and M. D. Hill. Page placement algorithms for large real-indexed caches. *ACM Trans. on Computer Systems*, 10(4):338–359, 1992.

[11] M. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance of blocked algorithms. In *Proc. of ASPLOS IV*, 1991.

[12] S. McFarling. Program optimization for instruction caches. In *Proc. of ASPLOS III*, 1989.

[13] A. Seznec. A case for two-way skewed-associative caches. In *Proc. of 20th Intl. Symp. on Computer Architecture*, pages 169–178, 1993.

[14] H. S. Stone. *High-Performance Computer Architecture.* Addison-Wesley, 2^{nd} edition, 1987.

[15] R. A. Sugumar and S. G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *Proc. ACM SIGMETRICS Conf.*, pages 24–35, 1993.

[16] G. Taylor, P. Davies, and M. Farmwald. The TLB slice–a low cost high-speed address translation mechanism. In *Proc. of 17th Intl. Symp. on Computer Architecture*, pages 355–363, 1990.

[17] O. Temam, C. Fricker, and W. Jalby. Impact of cache interference on usual numerical dense loop nests. *Proc. IEEE*, 81(8):1103–1115, Aug 1993.

[18] Olivier Temam. *Study and optimization of numerical codes cache behavior.* PhD thesis, University of Rennes, 1993.

[19] D. Thiebaut. On the fractal dimension of computer programs and its application to the prediction of the cache miss ratio. *IEEE Trans. on Computers*, 38(7):1012–1026, July 1989.

[20] W. Wang, J. Baer, and H. M. Levy. Organization and performance of a two-level virtual-real cache hierarchy. In *Proc. of 16th Intl. Symp. on Computer Architecture*, pages 140–148, 1989.

[21] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proc. of the SIGPLAN '91 Conf. on Prog. Lang. Design and Implementation*, pages 30–44, 1991.

A Example implementation of the dynamic set selection

We present an example of how the dynamic set selection can be implemented. Fig. 14 displays the additional datapath and memory required in the tag store. The data store of the cache remains the same.

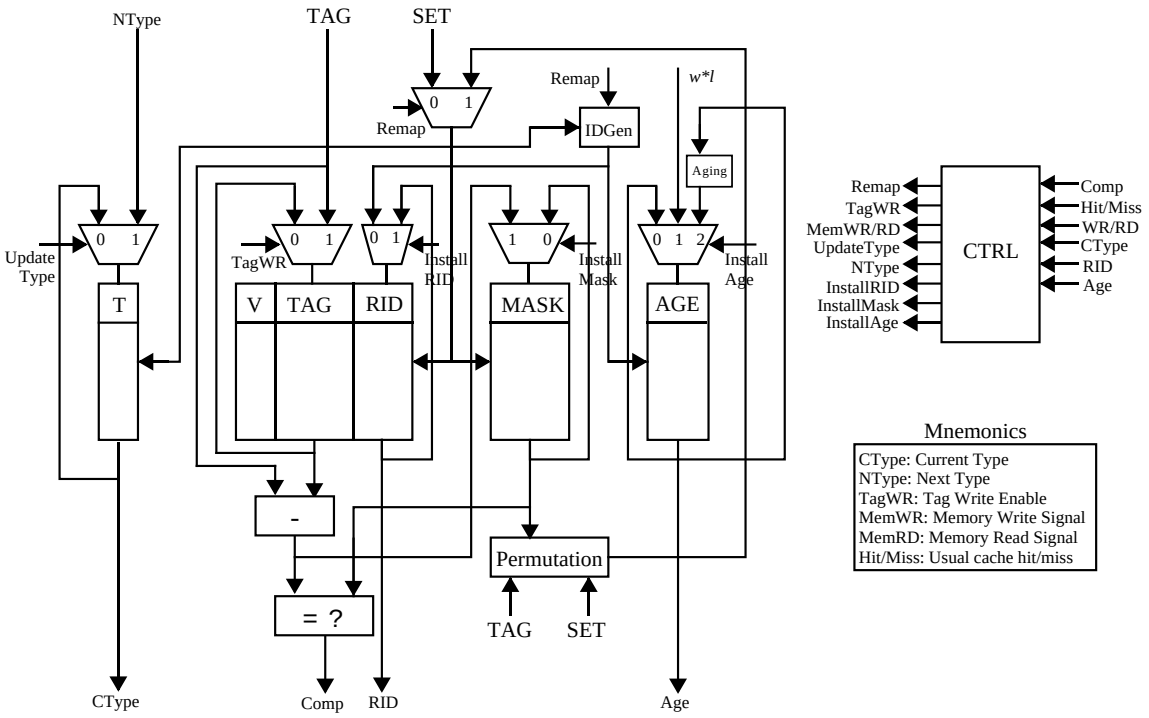


Figure 14: Example cache implementation

Suppose that a TAG,SET pair is used to address the cache. $MASK$ is used to store the value of the stride at the beginning of a constantly-strided access stream to a set, which is detected by sensing a repeated stride of addresses. The stored value is unchanged for the duration of the stream. The value stored in MASK acts as a mask in the remapping process.

The remapping employs a simplified SelBiP algorithm, in which the length of any strided access stream is assumed to be l. Theoretically, l is the length of the interference in the stream; however, we assume that the length of an interference is the same as the length of the stream containing it. Therefore, only $log(l)$ bits in the set field are changed. The actual value of l can be determined from an empirical study on the memory-referencing behavior of the typical workload of the machine. In ideal situation, the machine designer has the control of increasing the hit rate of a direct-mapped cache to that of a l-way set associative cache. It is preferred that the highest-order set bits are replaced. This is done to ensure the farthest-possible separation between remappings. The intended result would be to reduce the likelihood that the remapping is replaced before it is reused. The principle of data locality dictates that consecutive accesses are often localized in a certain region of the cache. Thus, when an intefering access replaces a remapping, other accesses that follow are likely to be contained to its neighborhood. By spreading out the remappings, we reduce the chance that other remappings are clobbered once a remapping is replaced.

To give an example, let us assume that 4 highest-order bits of the set field are to be replaced with 4 bits of the tag field. We copy bits from the tag field to the set field starting from that tag bit whose mask is 1, and on to the left of it. The procedure is carried out until we have copied 4 bits. In the case of overlapping copy fields (i.e. the distance between a pair of 1's in the mask is less than 4), the procedure is done until all the bits contained in the union of the fields and 3 more to the left of such union are copied. Thus, for the mask 0000101011, we copy all but the highest-order tag bit to the set field. The state diagram of the permutation logic is given by Fig. 15. Note that the clocking mechanism has to be faster than that of the system clocking since the permutation must not take more two cycles. We do not see this as a problem due to the simplicity and the parallelism of the operations in each state.

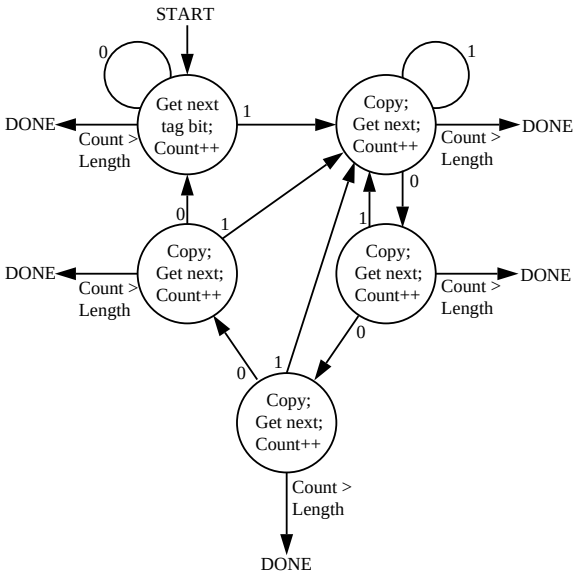


Figure 15: States of the sample permutation logic

The life of a remapping point is determined by means of an aging mechanism. *Aging* is a counter which decrements the value stored in *AGE*. Each entry in AGE is initialized to the intended age of each remapping point. This age corresponds to the value $w*l$ of the interference symbolized by the existence of the remapping point. Thus, during a cache miss on a remapping point, the cache control logic sees if the age of the remapping point is greater than 0. If it is, it tells Aging to decrement the age of the accessed remapping point. If it is 0, the type of the set, currently a remapping point, is converted to that of the arriving data, either regular or remapped. The set continues to be a non-remapping point until a new constantly-strided access stream arrives, in which case the set is converted back to a remapping point.

To generate a remapping ID (RID), we recommend selecting several highest-order bits and some lowest-order bits of the remapping set. The principle of data locality suggests the possibility of adjacent remapping points. Selecting the lowest order bits ensures that neighboring remapping points are assigned different RID. However, our experience with numerical applications suggests that most of the time a remapping point corresponds to the starting address of a matrix. In this case, the distance between remapping points are far enough to merit selecting some highest-order bits. By covering both cases, we believe that this simple RID-generating scheme can perform satisfactorily. *IDGen* is the hardware that selects the appropriate set bits and produces the RID for a remapping point. It contains a register that stores the generated RID in case the next cycle is a remapping and consequently the controller needs to install the proper RID in the remapping set. The RID itself acts as a control signal. In the mandatory StaBitS access, a register inside the cache controller stores the RID of the accessed set. In the optional SelBiP access, the stored RID is compared against the new RID that was just read. When trying to read a remapped set, the result of the comparison is used to determine cache hit or miss.

Fig. 16 gives the behavioral description of the cache control logic.

```
/* Note: follows C language convention, all numbers are in binary, */
/*       | denotes concatenation of signals                        */
Start:
switch (WR)
  case 1: switch (Hit | CType | Comp)
            case 0000:
            case 100x:
            case 110x: TagWR = 1; /* write to set & memory */
                       MemWR = 1;
                       goto Start;
            case x01x: TagWR = 1; /* write to set & memory, */
                       MemWR = 1;
                       UpdateType = 1; /* and change to regular set */
                       NType = 00;
                       goto Start;
            case 0001: UpdateType = 1; /* change to remapping point */
                       NType = 10;
                       InstallRID = 1;
                       InstallAge = 01;
                       InstallMask = 1;
                       Remap = 1; /* 2nd access */
                       switch (CType)
                         case 00: UpdateType = 1;
                                  NType = 01;
                                  InstallRID = 1;
                         case 01: TagWR = 1;
                                  MemWR = 1;
                                  goto Start;
                         case 10: switch (Age)
                                    case 0: UpdateType = 1;
                                            NType = 01;
                                            InstallRID = 1;
                                    other : TagWR = 1;
                                            MemWR = 1;
                                            goto Start;
            case 010x: switch (Age)
                         case 0: TagWR = 1;
                                 MemWR = 1;
                                 UpdateType = 1;
                                 NType = 00;
                                 goto Start;
                         other : InstallAge = 10; /* Decrement Age */
                                 Remap = 1; /* 2nd access */
                                 switch (CType)
                                   case 00: UpdateType = 1;
                                            NType = 01;
                                            InstallRID = 1;
                                   case 01: TagWR = 1;
                                            MemWR = 1;
                                            goto Start;
```

Figure 16: Behavioral description of the cache control logic

```
                                case 10: switch (Age)
                                           case 0: UpdateType = 1;
                                                   NType = 01;
                                                   InstallRID = 1;
                                           other : TagWR = 1;
                                                   MemWR = 1;
                                                   goto Start;

case 0: switch (Hit | CType | Comp)
          case 0000:
          case 100x:
          case 110x: goto Start;
          case x01x: MemRD = 1; /* read from memory */
                     TagWR = 1; /* write to cache */
                     UpdateType = 1; /* change to regular set */
                     NType = 00;
                     goto Start;
          case 0001: UpdateType = 1; /* change to remapping point */
                     NType = 10;
                     InstallRID = 1;
                     InstallAge = 01;
                     InstallMask = 1;
                     Remap = 1; /* 2nd access */
                     switch (CType)
                       case 00: UpdateType = 1;
                                NType = 01;
                                InstallRID = 1;
                       case 01: MemRD = 1;
                                TagWR = 1;
                                goto Start;
                       case 10: switch (Age)
                                  case 0: UpdateType = 1;
                                          NType = 01;
                                          InstallRID = 1;
                                  other : MemRD = 1;
                                          TagWR = 1;
                                          goto Start;
          case 010x: switch (Age)
                       case 0: MemRD = 1;
                               TagWR = 1;
                               UpdateType = 1;
                               NType = 00;
                               goto Start;
                       other : InstallAge = 10; /* Decrement Age */
                               store RID;
                               Remap = 1; /* 2nd access */
                               if (CType=01 AND storedRID=currentRID)
                                 goto Start;
                               else
```

Figure 17: Behavioral description (continued)

```
switch (CType)
  case 00: UpdateType = 1;
           NType = 01;
  case 01: InstallRID = 1;
           MemRD = 1;
           TagWR = 1;
           goto Start;
  case 10: switch (Age)
             case 0: UpdateType = 1;
                     NType = 01;
                     InstallRID = 1;
             other : MemRD = 1;
                     TagWR = 1;
                     goto Start;
```

Figure 18: Behavioral description (continued)