Scalable Cross-Architecture Predictions of Memory Hierarchy Response for Scientific Applications

Gabriel Marin
mgabi@cs.rice.edu

John Mellor-Crummey
johnmc@cs.rice.edu

Department of Computer Science
Rice University
6100 Main St., MS 132
Houston, TX 77005

ABSTRACT

The gap between processor and memory speeds has been growing with each new generation of microprocessors. As a result, memory hierarchy response has become a critical factor limiting application performance. For this reason, we have been working to model the impact of memory access latency on program performance. We build upon our prior work on constructing machine-independent characterizations of application behavior [9] by improving instruction-level modeling of the structure and scaling of data access patterns. This paper makes two contributions. First, it describes static analysis techniques that help us build accurate reference-level characterizations of memory reuse patterns in the presence of complex interactions between loop unrolling and multi-word memory blocks. Second, it describes a strategy for combining memory hierarchy response characterizations suitable for predicting behavior for fully associative caches with a probabilistic technique that enables us to predict misses for set-associative caches. We validate our approach by comparing our predictions (at loop, routine and program level) against measurements using hardware performance counters for several benchmarks on two platforms with different memory hierarchy characteristics over a large range of problem sizes.

Keywords

reuse distance, static analysis, symbolic formulae, modeling, set-associative, cache miss predictions

1. INTRODUCTION

For more than a decade, memory latency measured in terms of processor cycles has increased with each new generation of processors. For data-intensive programs, it is widely accepted that memory hierarchy latency and bandwidth are the factors most limiting node performance on microprocessor-based systems. To alleviate this problem, modern architectures include one or more levels of cache between the CPU and the main memory to boost available data bandwidth and help hide memory latency. However, caches are effective only if an application exhibits temporal or spatial data reuse. Therefore, characterizing an application's memory access patterns is important for understanding its performance and identifying opportunities for optimization.

Predicting application behavior on a different architecture for problem sizes not studied is difficult. Successful cross-architecture prediction requires an architecture-independent characterization of application behavior. Simple models that extrapolate from hardware performance counter measurements (e.g. the number of cache or TLB misses) on a particular machine cannot accurately predict application behavior on other architectures. Moreover, since an application's cache miss rate tends to grow in steps rather than smoothly with the size of the working set, extrapolations of hardware counter measurements should be attempted over only a limited range even on the same architecture.

In this paper, we expand upon our earlier work on cross-architecture prediction of application performance [9] with a focus on our techniques for predicting memory hierarchy response. We present a method for characterizing an application's memory access patterns and a modeling strategy that enables predictions of cache miss counts at the instruction level for different architectures and problem sizes.

New contributions of our work include static binary analysis techniques for symbolically characterizing memory reference access patterns, and a strategy for combining memory hierarchy response models that assume full associativity with a probabilistic technique that enables cache miss predictions for set-associative caches. We present a set of experiments to validate our cache and TLB miss predictions at loop, routine and program level. Our experiments compare measured and predicted memory hierarchy responses of several applications on two different architectures over a large range of problem sizes.

To model the memory hierarchy behavior of an application, we characterize the memory reuse distances[1] seen by each reference in the program. Memory reuse distance is a measure of the number of unique memory blocks accessed between a pair of accesses to the same block. Characterizing memory access behavior in this way for programs has two important advantages. First, data reuse distance is an application characteristic that is independent of target architecture. Second, reuse distance is a scalable measure of data reuse, which is the main determinant in cache per-

[1]Memory reuse distance is also known as LRU stack distance.

formance. However, there is one exception to the claim of total architecture independence. To capture spatial reuse in cache lines, we must collect reuse information with a memory block size equal to the size of the cache line on the target architecture. We will address this limitation again in Section 3. In addition, our reuse distance based models capture the memory access pattern of an application, therefore it is only sensible that they are not portable across high level loop optimizations (HLO) such as tiling, loop interchange, unroll & jam, that change the application's memory access pattern.

The rest of this paper is organized as follows. Section 2 presents static analysis that we use to guide both program instrumentation and modeling. Section 3 describes our infrastructure for collecting memory reuse distance (MRD) histograms from instrumented programs. Section 4 explains our algorithm for synthesizing scalable models from the collected data. Section 5 shows how we predict the number of cache misses for fully-associative or set-associative caches. Section 6 presents the results of using our methodology to predict cache and TLB miss counts at loop level for several NAS benchmarks on two different platforms. Section 7 summarizes closely related work. Section 8 presents our conclusions and our plans for future work.

2. STATIC ANALYSIS

We use memory reuse distance as a metric to understand and model memory locality in programs. To collect memory reuse distance data, we instrument an application's binary and insert profiling code before each memory reference in the program. Because we instrument object code instead of source code, our tools are language independent and naturally handle applications with modules written in different languages. In addition, binary instrumentation enables us to study highly-optimized code whereas source code instrumentation may inhibit aggressive compiler optimizations.

In this section, we describe our toolkit's static analysis capabilities. We use static analysis of binaries to guide every step of our modeling and prediction process. Using static analysis, we

- reconstruct the control flow graph (CFG) for each routine,[2]
- identify the natural loops in each CFG using interval analysis [12] and compute the loop nesting structure,
- derive symbolic formulae for each reference's access pattern, and
- identify references that must be profiled (e.g. we do not monitor accesses produced by register spill/unspill code) and customize the profiling code for each memory instruction.

The rest of this section describes a static analysis technique for computing symbolic formulae that characterize the access pattern of each memory reference, and how we use this analysis to improve modeling accuracy.

[2]CFG construction from a binary is performed by EEL [7] on top of which our toolkit's binary analysis capabilities are built.

2.1 Static Analysis of an Access Pattern

For each memory reference, we statically derive several symbolic formulae that describe the pattern of locations it accesses during execution. We perform this analysis intraprocedurally. We compute two types of formulae. For each reference in the program, we compute a formula for the *first location* it accesses. For references inside loops, we also compute formulae that describe how the accessed location changes from one iteration to the next.

We begin by constructing a *first accessed location* formula for each reference in a routine. For each register used in a reference's address computation, we recursively traverse the CFG backwards along use-def chains until either we encounter a load immediate value, we cannot trace any further inside this routine (the traced register is the result of a function call or we reach the top of the routine), or we determine that a formula for the register was already computed while analyzing a previous instruction. During this backward traversal of the CFG, we consider only forward CFG edges. As we unwind each step of our traversal, we compute a symbolic formula at each machine instruction along the traversal by applying the instruction's operator to the formulae computed for its source registers. We cache every intermediate result so that we don't have to traverse the same chain of instructions a second time when analyzing another instruction.

During a use-def chain traversal, if we find that a register is reloaded from the spill area, we trace backward along CFG edges for a corresponding spill to the same location and then resume our use-def chain traversal from the spill. We generalized our mechanism for handling reloaded spill values to work with arbitrary loads after we encountered a binary in which we found that the compiler had saved the value of a register in the data segment and later loaded it to compute the address of a memory reference. If the value of a register is defined by a load instruction, we trace backward along CFG edges for a store with a symbolic formula equal to the load's symbolic formula[3], and we continue tracing back from the register whose value was stored.

Formulae are restricted to sums of general terms including immediate values, loads from a constant address, loads from the caller's stack frame (an argument passed by reference), and registers whose formulae cannot be written as a sum of these terms (e.g. loads from an undetermined address, a product of two non-constant formulae, etc.). Each term of a formula can have integer numerator and denominator coefficients. With these restrictions, any non additive (not an add or a sub) operation of two non-constant formulae will produce a register value. When at least one operand is a constant, several operations can be computed without simplification to a register value: multiply, divide, and left/right shift by a constant value can be performed by updating the coefficients of each term in the non-constant operand. If both operands are constants, all arithmetic and bitwise operators can be computed precisely.

For memory references inside loops, we compute additional formulae that indicate how the accessed location differs from one iteration to the next. We compute a *stride formula* for each loop level containing the reference. For each level of a loop nest, we apply a recursive algorithm that tra-

[3]The symbolic formulae for the instructions upstream of the one currently analyzed have been already computed.

```
/* multiply two squared matrices */
void matrix_multiply(int N, double *A,
          double *B, double *C)
{
  int i, j, k;

  for ( i=0 ; i<N ; i+=1 )
    for ( j=0 ; j<N ; j+=1 ) {
      C[i*N+j] = 0;
      for ( k=0 ; k<N ; k+=2 )
        C[i*N+j] +=
            A[i*N+k]*B[k*N+j] +
            A[i*N+k+1]*B[(k+1)*N+j];
    }
}
```

(a) matrix multiply source code

```
                                              First location  k-stride  j-stride  i-stride
          L1: add    %l5, 0x2, %l5
A[i,k]        ldd    [%l3], %f4               %i1+16          16        0         8*%i0
              add    %l3, 0x10, %l3
              cmp    %l5, %o5
              add    %l4, 0x10, %l4
B[k,j]        ldd    [%o3], %f2               16*%i0+%i2      16*%i0    8         0
A[i,k+1]      ldd    [%l4 - 0x10], %f8        %i1+24          16        0         8*%i0
B[k+1,j]      ldd    [%o7], %f6               24*%i0+%i2      16*%i0    8         0
              fmuld  %f4, %f2, %f10
              fmuld  %f8, %f6, %f12
              faddd  %f10, %f12, %f14
              faddd  %f0, %f14, %f0
C[i,j]        std    %f0, [%l2]               %i3             0         8         8*%i0
              add    %o3, %l1, %o3
              bl,pt  %icc,L1
              add    %o7, %l1, %o7
```

(b) assembly for the innermost loop and the derived symbolic formulae

Figure 1: Static analysis example. The left subfigure presents the source code for a naive matrix multiply implementation, and on the right we have the SPARC assembly code for the innermost loop annotated with the symbolic formulae computed for each memory reference.

verses backward along CFG edges, similar to the way we did when computing *first accessed location* formulae. However, when computing stride formulae, we consider only forward CFG edges that are part of the analyzed loop and the loop's back edge. This recursive search terminates when either we encounter a load immediate operation, we traced backwards a complete iteration without finding a definition for this register (this register contains an invariant value with respect to this loop), or we reach a definition for a second time. In this last case, the found instruction is part of a chain of instructions that update an index variable. When the recursion returns, the *stride formulae* are computed by applying the mathematical operators corresponding to each intermediate machine instruction to the non-invariant components of the source formulae.

The *stride formulae* are restricted to the same sums of general terms as the *first accessed locations* formulae. However, a *stride formula* has two additional flags that can be set by the recursive algorithm. The first flag indicates if an access has an irregular stride; this flag is set if the reference's stride is not constant for all iterations of that particular loop. The second flag indicates if an access is indirect and it is set if the formula for the accessed location depends on the value of another load which has a non-zero stride with respect to this same loop.

For example, Figure 1(a) presents the source code for a naive implementation of the matrix multiply algorithm. Because the program is written in C, the three matrices have been allocated as one-dimensional arrays and the rows of each matrix are contiguous in memory. The three matrices are dynamically allocated and their size is passed as an argument to the *matrix_multiply* function to validate the correctness of the computed formulae in the presence of symbolic values.

Figure 1(b) presents the SPARC assembly code for the innermost loop of the *matrix_multiply* algorithm. The five memory references have been annotated with symbolic formulae we derived through static analysis. Each reference has a formula for the *first accessed location*, and three *stride formulae*, one for each level of the loop nest. Each reference is also annotated with the corresponding source code array access to make the code easier to understand. Those familiar with the SPARC assembly language will recall that registers *%i0, ..., %i5* are used for passing the first six input arguments of a routine, and will notice that all symbolic formulae are correctly computed relative to the source code on the left. The compiler has peeled one iteration of the k-loop, therefore the *first location* formulae correspond to $i = 0, j = 0, k = 2$. Although in the assembly code each reference uses distinct address registers, the formulae for references to the same array show they are related. The k-loop was unrolled by hand once[4] and we compiled the binary with unrolling disabled to keep the size of the assembly code small for the purpose of this example, while at the same time having the loop unrolled to show how this type of static analysis can uncover related references.

We say two references r_s, r_t located in the same loop have similar access patterns, if they have equal *stride formulae* relative to each loop containing them. In other words $Stride(r_s, L^k) = Stride(r_t, L^k)$ for every level k loop L^k containing them, $k \geq 1$. For our example in Figure 1(b), references $A[i, k]$ and $A[i, k+1]$, as well as references $B[k, j]$ and $B[k+1, j]$ have equal strides at each loop level, and therefore they have similar access patterns. This is no surprise for somebody looking at the source code, but extracting such information from binaries requires the detailed static analysis we described.

2.2 Applications of Symbolic Formulae

A first use of the statically derived symbolic formulae is at instrumentation time. Previously, we explained that we collect memory reuse data at reference level. Such an approach not only enables detailed predictions at instruction or loop level, but also enables more accurate models than if we collected aggregate reuse information for the entire program. However, as noted in [8, pages 46–53], modeling reuse distance data at instruction level is prone to errors due to the alignment of data in memory when the reuse information is collected for larger than unit size memory blocks. As we describe in Section 3, to account for spatial reuse, we collect reuse information of memory blocks, where the block size is equal to the line size of the target cache.

Consider now the *matrix_multiply* example from Figure 1 where the inner loop is unrolled once. Let's assume that array A is always aligned to the start of a cache line. Because

[4]There must be additional code not included in the figure for the k-loop to process the reminder element when N is odd.

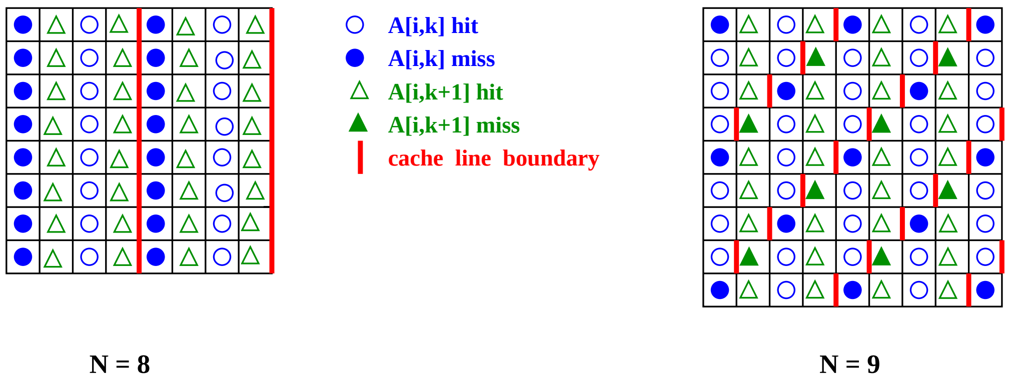

Figure 2: The distribution of cache misses for the two references to matrix A from the matrix multiply code presented in Figure 1, for an even problem size (N=8) and an odd problem size (N=9), assuming an architecture with a cache line that holds four double elements.

the size of a cache line is a power of two, an even number of array elements will fit into a cache line. In our case, for an even value of N, the reference corresponding to $A[i,k+1]$ will always see a small reuse distance due to spatial reuse, because $A[i,k]$ will always perform the first access to a new cache line. However, for an odd value of N, $A[i,k+1]$ will access a new cache line first for odd rows of A, while $A[i,k]$ will access a new cache line first for even rows. Such inconsistencies between the reuse pattern at different problem sizes can cause large modeling errors for the affected references. However, if we consider $A[i,k]$ and $A[i,k+1]$ together, the union of their reuse distance data is consistent and predictable for every problem size.

Figure 2 presents graphically this behavior for matrix sizes 8 and 9, assuming an architecture where the cache line size is four times the size of an array element. In both cases, the total number of misses is approximately equal to one quarter of the number of memory accesses because only one miss occurs per four-element cache line. However, the distribution of misses between the two references is different for each problem size. This problem is even more pronounced when the unrolling factor is greater and a larger number of references are affected.

A similar problem occurs in codes working on arrays of records when the cache line size is not a multiple of the record size. In such a case, depending on the record index, different fields can occupy the first position of a cache line. As a result, different references encounter a long reuse distance during the dynamic analysis depending on the record index. For this reason, at instrumentation time we find the sets of references that have similar access patterns and insert code that collects a single reuse distance histogram for every such set.

After reuse distance histograms are collected, we perform additional static analysis to identify object code loops that have their origin in the same source code loop, and we perform additional aggregation between reference groups from these loops that have similar access patterns. We extend the definition presented before to say that two references r_s, r_t located in different loops have similar access patterns, if they have equal *stride formulae* relative to each loop containing both of them, and their *stride formulae* relative to distinct, same level loops containing them have an integer ratio. In other words: $Stride(r_s, L^k_{st}) = Stride(r_t, L^k_{st})$ for every level k loop L^k_{st} containing both references, and $Stride(r_s, L^k_s)/Stride(r_t, L^k_t) = m/n$ for every pair of distinct level k loops L^k_s and L^k_t containing references r_s and r_t respectively, where m and n are integers and either $m = 1$ or $n = 1$.

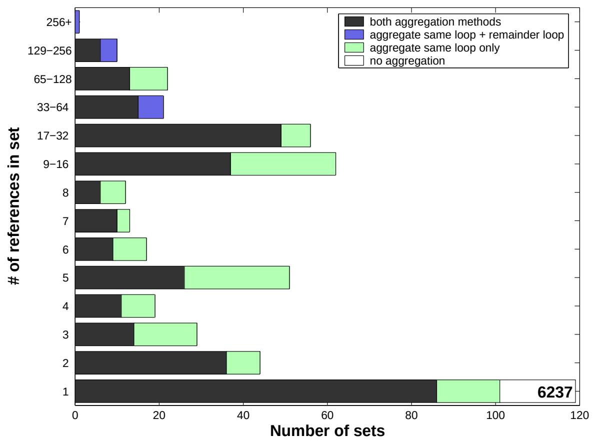

Figure 3: Distribution of the sizes of the instruction groups derived for benchmark NAS BT 3.0 when: (1) we perform no aggregation, (2) only references with similar patterns from the same loop are grouped together, (3) we aggregate across adjacent object code loops.

Loop optimizations such as software pipelining and loop unrolling, split a source loop into multiple object loops: a main loop and a prolog or an epilog loop, which executes the remainder iterations. Compilers use loop unrolling aggressively. In addition, stencil computations found in scientific applications access multiple elements of an array with the same stride. As a result, there are many opportunities to aggregate references into larger sets, both inside the same loop and across adjacent loops. Figure 3 presents the distribution of the sizes of the groups of memory access instructions derived for the NAS 3.0 BT benchmark. On the y axis we have the number of instructions in a group, and on the x axis we see how many groups of that size were produced by aggregation. If we perform no aggregation, there are more

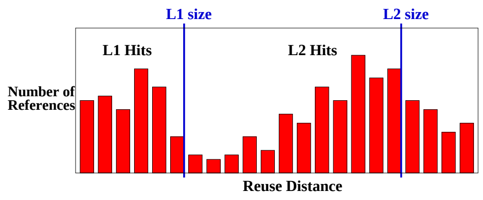

Figure 4: Example of reuse distance histogram. All references with reuse distance less than the cache size are hits.

than six thousand different groups, each with only one reference. During the instrumentation step only references with similar access patterns from the same loop are aggregated, and the number of distinct instruction groups reduces to 447. In the post-processing phase, after additional aggregation is performed across loops, only 329 groups remain for the NAS BT 3.0 benchmark.

Another important use of *symbolic formulae* is to understand memory dependences between instructions within loops. Dependence information is used to construct dependence graphs, which in turn are used to predict the instruction schedule for a different architetcture [9], or to understand the level of memory parallelism within loops.

3. DYNAMIC ANALYSIS

Often, an application's memory access patterns cannot be understood completely using only static analysis. In the previous section we described how to build a static characterization of each reference's access pattern. The statically derived symbolic formulae are useful to determine if two references have the same access pattern, or even to perform dependence analysis for regular array references found in scientific applications. However, symbolic formulae depend upon information that is unknown at compile time, and for irregular applications they are less precise. Moreover, understanding how the distance between two accesses to the same location depends upon input parameters, can only be determined by measuring it at run time.

To understand an application's memory access patterns, we collect histograms of the reuse distance observed by each load or store instruction. For a fully-associative cache, one can predict if a memory access is a hit or a miss by comparing its reuse distance with the size of the cache (see Figure 4). Beyls and D'Hollander [3] show that reuse distance predicts the number of cache misses accurately even for caches with a low associativity level. However, reuse distance alone cannot predict conflict misses. In section 5.2 we show how to estimate the number of conflict misses for a set-associative cache using a probabilistic model in conjunction with our memory reuse distance models.

We collect reuse distance information separately for each reference group as described in section 2.1. Before each memory reference, we invoke a procedure that updates a histogram of reuse distance values for that reference. A detailed description of the algorithm that we use to compute the reuse distance of each memory access can be found in [9].

By using a unit size memory block, we can collect pure temporal reuse distance information. However, using this approach we fail to observe spatial reuse in cache lines. By setting the memory block to a non-unit cache line size, we can also measure spatial reuse because we collect the reuse distance of data blocks rather than data elements. To correctly account for spatial locality, we need to use a memory block size equal to the size of the cache line on the target architecture. Currently, to predict the memory access behavior of an application on arbitrary systems, we need to collect reuse distance data for all cache line sizes that might be encountered in practice. The most common cache line sizes in use today are 32, 64 and 128 bytes. Because of the reuse distance data's dependence on cache line size, our characterizations of application behavior are not entirely architecture independent, but they come close to this goal. The size of the memory block used by our runtime library is defined by an environment variable; therefore collecting data for different cache line sizes does not require re-instrumenting the binary or re-compiling our instrumentation library.

The time complexity for computing the reuse distance seen by one memory access is $O(\log M)$, where M is the number of distinct memory blocks touched by the application. Overall, the overhead of collecting memory reuse distance information for the entire execution is $O(N \log M)$, where N is the number of memory accesses the program executes, and the space required by the data structures for monitoring reuse distance is $O(M)$. Time and space complexities for collecting memory reuse distance histograms are both significant. Therefore, we would like to predict the distribution of the reuse distance histograms for large problem sizes that are of interest in practice, from data collected for several small problem sizes.

4. BUILDING SCALABLE MODELS

To predict an application's memory access behavior for a different problem size, we have to model how each reference's reuse distance scales as a function of problem size. For this, we must first collect MRD data from multiple executions, with different and preferably small data sets.

Modeling memory access behavior is difficult. A single reference in the program may see cold misses and many distinct reuse distances. The simplest possible model of a memory reference's reuse distance would predict its average value for each problem size. However, such a model is almost always useless. Consider a reference performing stride one loads. Its first access to a cache line yields a long reuse distance; accesses to subsequent words yield short reuse distances. An average distance model can predict either all hits or all misses; neither prediction is accurate.

We need to model the behavior using histograms. A reuse distance histogram for a reference contains a separate bin for each distinct distance encountered. We must model the structure and scaling of these histograms to understand the distribution of reuse distances as a function of problem size. Building meaningful models for histograms of reuse distance from executions with different problem sizes is challenging. Executions using different problem sizes result in histograms that each have a different number of bins and frequency counts; the varying number of bins complicates modeling.

One possible modeling approach is to divide each histogram for any reference or problem size into an identical number of bins using a fixed strategy regardless of the distribution of the data. How many bins to consider has an important impact on the size and accuracy of the models. A small number of bins will yield a compact model, but the model may lack precision. A large number of bins will im-

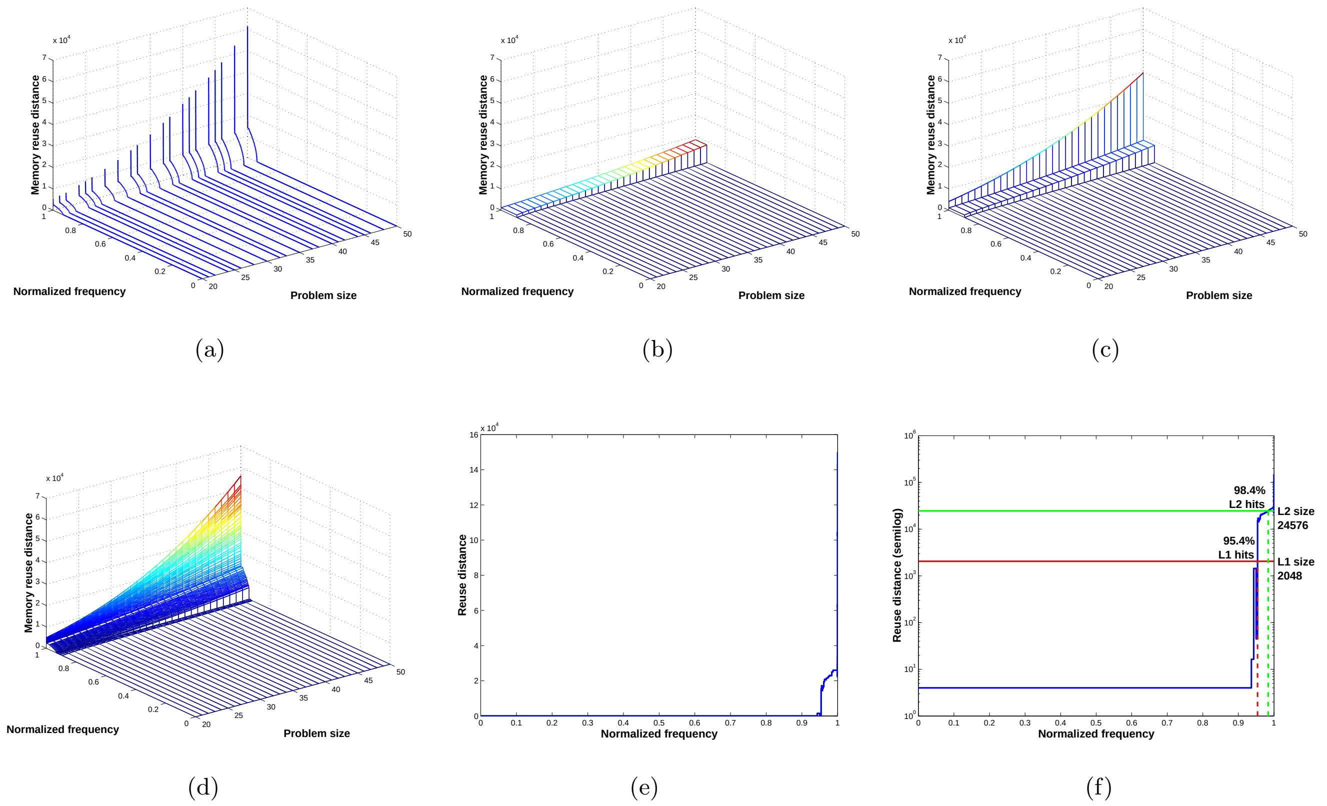

Figure 5: (a) MRD data collected for one reference in Sweep3D; (b) Model constant distance first, and lump remaining data in one bin; (c) First split of the non-constant data; (d) Final model for the data in (a); (e) Model evaluation at problem size 70; (f) Model evaluation at problem size 70 on a logarithmic y axis, and predictions for a 2048 blocks level 1 and 24576 blocks level 2 cache.

prove model accuracy, but will add unnecessary complexity and cost to modeling for many references that use only a few different reuse distances. To avoid this problem, we examine a reference's collected data and pick an appropriate number of bins and their boundaries to adequately represent its histogram data across the range of problem sizes.

4.1 Modeling MRD Histograms

We sort the bins in each reference's MRD histogram by increasing distance. The first bin in a reference's histogram holds the smallest reuse distance for the reference. Figure 5(a) shows MRD histograms collected by our tool for different problem sizes for one of the most frequently executed memory accesses in the ASCI Sweep3D benchmark [1]. To illustrate our modeling algorithm, we plotted our models of MRD histograms in 3D Cartesian coordinates for each step of the algorithm. The `x` axis represents the problem size (from 20 to 50 in this case); the `y` axis represents the additive normalized execution frequency of the bins for each problem size, such that the total execution frequency of each histogram is one; and the `z` axis represents the reuse distance.

We begin our analysis by examining the leading bins of a reference's histograms for each problem size. If the leading bins have the same reuse distance across all problem sizes, it means they contain the fraction of accesses that have experienced temporal or spatial reuse in the innermost loop enclosing that reference. The values of the constants depend upon the shape of the code in each particular loop, namely, how many accesses to other data structures are executed between two accesses to the same data structure during one iteration. These small distances are constant across all problem sizes because the shape of the code is invariant across problem sizes.

If a reference's histograms have such leading bins with a small constant distance, we model them separately (see Figure 5(b)), and remaining data is lumped together in one bin represented in the figure by a model of its average distance. Since the reuse distance of the leading bins is constant, we need to model only the execution frequency of these bins. The fraction of accesses that experience spatial reuse is usually not constant across all problem sizes. The explanation for this is that for mesh like data structures, if the dimensions are padded to align the data or to minimize conflict misses, some of the memory blocks are not completely filled with useful data. Therefore, accesses to these incomplete lines experience less spatial reuse than accesses to cache lines that are completely filled. As problem size increases, the significance of the padding relative to the size of the data structure diminishes. As a result, the fraction of accesses with spatial reuse increases asymptotically towards $1 - reference\_stride/cache\_line\_size$, and we are able to capture this behavior with our technique.

The remaining bins and their parameters are determined using a recursive algorithm. We start by computing an average distance for all of the references that were not modeled in the first step, and we build a model for their average distance as seen in Figure 5(b). Next, we recursively split the set of accesses in two and compute a model for each subset. The recursion stops when the models of the two resulting subsets are sufficiently close. We apply this algorithm to determine a partitioning of the data into an appropriate number of bins by considering the data for all problem sizes at once. At each step, we use a heuristic to determine how to partition the accesses. Its decisions influence the convergence speed, the accuracy, and the stability of the final model.

In our experiments, the partitioning heuristic that yielded the most stable and accurate results was one that selects partition boundaries such that the ratio between the number of accesses in the two partitions resulting from a split is the same across all problem sizes. With such an approach we need to model only the average reuse distance of each bin; execution frequency is easily computed by dividing the frequency of the parent bin proportionally with the splitting ratio. To compute the splitting ratio of a bin, we apply the following algorithm:

- Determine midpoint of the reuse distance range for each problem size, and compute the ratio between the number of accesses with reuse distance less than and greater than the midpoint value for each problem size. Using the reuse distance midpoint speeds up model convergence by favoring creation of narrow bins where reuse distance vary abruptly along the y axis, and wide bins where large fractions of accesses have similar reuse distances, with a minimal recursion depth.

- Select median ratio across all problem sizes, and use this median value as the splitting ratio for all problem sizes. We opted to use the median ratio to increase modeling stability in case the midpoint ratios for some problem sizes are significantly different.

Figure 5(c) presents a snapshot of the model after the first splitting step. If we look at the reuse distance histogram for problem size 50 in Figure 5(a), we see that the largest observed reuse distance is around 6×10^4. Then, we can approximate the midpoint reuse distance for this problem size at around 3×10^4. We notice that many more accesses have reuse distance under the midpoint value than above it. As a result, the two bins produced by the first split contain very different fractions of accesses, but they cover approximately equal ranges of reuse distance.

After partitioning, we perform a (rarely needed) coalescing step that examines adjacent bins and aggregates them together if they have similar polynomials describing their reuse distance. Our approach produces a minimal number of bins with almost no loss in accuracy. If a large fraction of accesses have comparable reuse distances across all problem sizes, all those accesses go into one bin. However, if part of a reference's histogram is composed of many small fractions of accesses with different reuse distances, our approach produces a large number of bins for that part of the histogram and successfully captures the instruction's complex behavior. Figure 5(d) presents the final model computed for the data in Figure 5(a).

Our modeling strategy is currently implemented in Matlab. We use quadratic programming [10] to determine the function that best approximates the input data we collected. Each approximation function is written as a linear combination of a set of basis functions. The program uses either a default monomial base or a set of user-provided bases in symbolic form such that logarithmic or other non-linear contributions to the model can be considered. The modeling program computes the coefficients of the basis functions in the linear combination that closest approximates the collected data. We include restrictions to reduce or remove oscillations of the resulting fit and to ensure that the computed function is either convex or concave depending upon the program characteristic that is modeled. Our approach works best with scientific codes that have predictable execution patterns, namely, ones that do not use adaptive algorithms.

5. MODEL EVALUATION

The problem of determining the ratio of hits and misses for a given cache size $\mathcal{C}$ is equivalent to determining the intersection of the model with the plane defined by $z = \mathcal{C}$. Similarly, the problem of computing the expected behavior for one instruction at a given problem size $\mathcal{P}$ is equivalent to determining the intersection of the surface and the plane defined by $x = \mathcal{P}$. We can also determine the minimum cache size such that the hit-ratio is $\mathcal{H}$. The solution to this problem is the intersection of the model and the plane defined by $y = \mathcal{H}$. Any two of these three problems can be combined and the solution is the intersection of the surface with the corresponding two orthogonal planes.

5.1 Predictions for Fully-associative Caches

For a fully-associative cache, we can use this approach to predict the ratio of misses for a given problem size and cache size. Figure 5(e) presents the expected behavior of the instruction modeled in Figure 5(d) at problem size 70. Assume that we want to predict the hit ratios for an architecture with two fully-associative levels of cache, where level one has 2048 blocks and level two has 24576 blocks. For this we must determine the number of accesses that have a reuse distance less than the specified cache sizes. Because the maximum reuse distance predicted for this reference is three orders of magnitude larger than the size of the target L1 cache, Figure 5(f) presents the predicted MRD histogram for problem size 70 on a logarithmic y-axis. The hit ratio is determined by the intersection of the predicted curve with the cuts corresponding to the sizes of the two cache levels. For this instruction and the considered target architecture, the model predicts a hit ratio of 95.4% for the L1 cache and 98.4% for the L2 cache.

5.2 Predictions for Set-Associative Caches

In section 3 we defined memory reuse distance as the number of **distinct** memory blocks referenced between two consecutive accesses to the same memory block. If an access has reuse distance n, it means that we referenced n distinct other blocks since the previous access to the block currently accessed. For a fully-associative cache, any memory block can map to any cache block. Therefore, if the cache uses LRU replacement policy and has less than or equal to n blocks, we know that current access will be a miss because the n distinct blocks accessed since the previous access to this block have caused it to be evicted from the cache. Similarly, if the cache has more than n blocks, the current access is a hit because the accessed block was not evicted yet.

For a set-associative cache with s sets and associativity level k, a memory block can map only to one of the k blocks of a single set, where the set is uniquely determined by the block's location in memory. As a result, an access with reuse distance n is a hit if less than k out of the n accessed blocks map to this same set. The mapping of memory blocks to cache sets depends upon how data structures are laid out in memory. However, we do not collect information about the location of accessed blocks. As Hill and Smith noted in [6], we can estimate set-associative LRU distance from fully-associative LRU distance using a statistical model. This model is based on the simplifying assumption that accessed blocks are uniformly distributed in memory. In other words, the probability that two blocks map to the same set is $1/s$ and independent of where other blocks map.

With this assumption, we first compute the probability that exactly i blocks out of n distinct blocks map to a given set. We first notice that for $i > n$, the probability is zero because we cannot have more than n blocks map to a single set when there are n blocks overall. The mapping probability can be written as:

$$P_{mapping}(s,n,i) = \begin{cases} \left(\frac{1}{s}\right)^i \left(\frac{s-1}{s}\right)^{n-i} \binom{n}{i} & \text{if } i \le n \\ 0 & \text{if } i > n \end{cases}$$

The probability formula for $i \le n$ has three terms:

- $\left(\frac{1}{s}\right)^i$ because i blocks must map onto a specific set (the set of the currently accessed block)
- $\left(\frac{s-1}{s}\right)^{n-i}$ because the other $n-i$ blocks must map onto the other $s-1$ sets.
- $\binom{n}{i}$ because any combination of i blocks out of the total number of n blocks can map onto our set.

The probability that an access with reuse distance n hits in a set-associative cache with s sets and associativity k can be written as:

$$P_{hit}(s,k,n) = \sum_{i=0}^{min(k-1,n)} \left(\frac{1}{s}\right)^i \left(\frac{s-1}{s}\right)^{n-i} \binom{n}{i}$$

and the probability of that access being a cache miss is 1 minus the previous formula:

$$P_{miss}(s,k,n) = 1 - \sum_{i=0}^{min(k-1,n)} \left(\frac{1}{s}\right)^i \left(\frac{s-1}{s}\right)^{n-i} \binom{n}{i}$$

This model fits very well with our MRD model, because we do not predict just an average distance for a reference, but a histogram of how many times each distance is encountered. For each bin of a reference's histogram we compute a miss probability as a function of the bin's reuse distance. The resulting probability represents the fraction of accesses in that bin that should be expected as cache misses.

In the case of a fully-associative cache we have only one set ($s = 1$) and k represents the number of blocks in the cache. If $n > k-1$, probability to hit in the cache is zero because $\left(\frac{s-1}{s}\right)^{n-i} = 0$ for any $i \le k-1 < n$. If $n \le k-1$, probability to hit in the cache is one because the sum reduces to a single term, $\binom{n}{i}$, where $i = n \le k-1$. Thus, the formula is valid also in the special case of a fully-associative cache, although it is more efficient to use the direct method presented in Section 5.1 to compute the number of cache misses for fully-associative caches. However, we observe that while for a fully-associative cache each bin counts as either all hits or all misses, in the case of a set-associative cache a bin can have a dual behavior.

We can approximate the number of misses for a set-associative cache from the histogram of reuse distances predicted by our MRD model, with the following formula:

$$Num_{misses}(Hist,s,k) = \sum_{bin_i \in Hist} (P_{miss}(s,k,D_{bin_i})F_{bin_i})$$

where D_{bin_i} and F_{bin_i} are the average MRD of bin_i and the execution frequency of bin_i respectively.

Although the assumption that accessed memory blocks are uniformly distributed in memory is not always true, the miss predictions for set-associative caches produced by this model (see section 6) are quite accurate.

6. RESULTS

To validate our approach, in this section we compute cache and TLB miss predictions at the loop level for the ASCI Sweep3D benchmark and several of the NPB 2.3-serial and NPB 3.0 benchmarks, for mesh sizes ranging from 10^3 to 200^3. We compare our predictions against measurements using hardware performance counters on two different platform: an Itanium2 based machine and an Origin 2000 system based on the MIPS R12000 processor. The memory hierarchy characteristics for the two testbed machines are presented in table 1. On the Itanium, floating point loads

Level	# blocks/associativity/block size	
	Itanium2	R12000
L1D	256/4-way/64 B	1024/2-way/32 B
L2	2048/8-way/128 B	65536/2-way/128 B
L3	12288/6-way/128 B	–
L1 TLB[5]	32/fully/16 KB	64/fully/32 KB[6]
L2 TLB[5]	128/fully/16 KB	–

Table 1: Memory hierarchy characteristics for the testbed machines.

and stores bypass the small L1D cache and its associated L1 TLB. Because the benchmarks used in this test suite are all floating point intensive, the L1D cache and the L1 TLB of the Itanium2 machine have very little impact on their performance, and we do not present predictions for these two memory levels.

Our memory reuse distance models for an application are a function of cache line size, are parameterized by one of the application's input parameters as described in Section 4, while size and associativity level of the target cache are used during evaluation (see Section 5) to predict the number of

[5]A TLB behaves exactly like an LRU cache with a number of blocks equal to the number of entries in the TLB, and the size of each block equal to the size of the memory mapped by each entry.

[6]On the R12000, each TLB entry maps two consecutive pages, therefore the size of the memory mapped by an entry is 32 KB.

cache misses. Nowhere in this process we make use of information such as the CPU's frequency or its number of functional units. Our cache miss predictions have nothing to do with the architecture of the CPU core. Therefore, while we consider only two platforms, we present predictions for six different cache configurations (two cache levels and one TLB level on each platform). From table 1 we see that the testbed machines cover a diverse set of cache configurations, including capacity, block size and associativity.

To compute the predictions, we compiled the benchmarks on a Sun UltraSPARC-II system using the Sun WorkShop 6 update 2 FORTRAN 77 5.3 compiler, and the optimizations: *-xarch=v8plus -xO4 -depend -dalign -xtypemap=real:64*. Measurements on the Itanium2 machine were performed on binaries compiled with the Intel Fortran Itanium Compiler 8.0, and the optimization flags: *-O2 -tpp2 -fno-alias*. On the Origin 2000 system we compiled the binaries with the SGI Fortran compiler version 7.3.1.3m and the optimization flags: *-O3 -r10000 -64 -LNO:opt=0*. We used the highest optimization level but we disabled high-level loop optimizations, because the sets of loop nest transformations implemented in the Sun, Intel and SGI compilers are different. Loop nest transformations change the execution order of the iterations of a loop nest, effectively altering an application's memory access pattern.

We present results for three benchmarks on each of the two machines, including results at routine and loop level for one benchmark on each architecture. We present results for ASCI Sweep3D and the BT benchmark from NPB 2.3-serial on both architectures. In addition, on the Itanium2 machine we analyze in more detail the behavior of the hyper-plane 2D implementation of LU from NPB 3.0 , and on the Origin 2000 we look at the SP benchmark from NPB 3.0. The NAS benchmarks use statically allocated data structures, with the maximum size of the working mesh specified at compile time. The benchmarks can be compiled in several standard classes named A, B and C, which have a maximum mesh size of 64, 102 and 162 respectively. We created an extra class L with a maximum mesh size of 200. We used static and dynamic analysis of the class A binaries to construct the models. The measurements on the Itanium2 and R12000 machines were performed on the binary of minimum class that accommodates that particular size.

To compute the predictions, we collected MRD data for block sizes 32, 128, 16 KB and 32 KB, for a set of problem sizes randomly selected between 20 and 50. We collected data on relatively small input problems to limit the cost of executing the instrumented binaries. Next, we built models of MRD parameterized by problem size for each of the applications, as described in section 4. Finally, to predict the cache and TLB miss counts, we evaluated the models at each problem size of interest. For each memory hierarchy level on each of the two machines, we predict a miss count for a fully-associative cache of the same size as the actual cache on the machine using only the MRD models, and a miss count that takes associativity into account using the probabilistic model described in section 5.2.

6.1 Predictions for Itanium2

Figure 6 presents the results for the Itanium2 machine. Each graph on the top row presents the measurements and the predictions aggregated at the entire program level for one application. For the L2 and L3 caches we present both fully-associative and set-associative predictions as explained above. Because the TLB is fully associative, there are no set-associative predictions for it. For all graphs, the x axis represents the mesh size and the y axis represents the number of misses per cell, per iteration, where number of cells is equal to $mesh\_size^3$ for all applications considered. This normalized view of the data enables us to understand how the application's characteristics scale with the amount of useful work, and at the same time makes the graphs more readable by bringing the counts for all mesh sizes to comparable levels. The range of problem sizes for which we measured reuse distance histograms to build the models is indicated on each graph by the two vertical lines.

Wile the predictions are in general accurate, we notice that we under-predict the number of L2 misses for the LU benchmark at large problem sizes. Looking at the routine level predictions, we noticed that the entire L2 prediction error came from routine *rhs*. The second row in figure 6 presents the predictions for three routines of the LU application, including routine *rhs*. Moreover, we have noticed that our models under predict the number of L2 misses for all implementations of LU in NPB 3.0 and in NPB 2.3, and in all cases the error was occurring in routine *rhs*. In all these cases, we noticed a correlation between the higher number of measured L2 misses and a high rate of TLB misses. As you can see from the graph for routine *rhs* in figure 6, almost all TLB misses measured for the LU application are produced by this routine. On a TLB miss, the OS needs to find the entry for the offending page in the virtual page table which is stored in memory. On the Itanium, the page table is accessed through the L2 cache. This has the advantage that on a TLB miss, in addition to the offending entry being brought into the TLB, an entire cache line of page entries is brought into the L2 cache. Thus, successive TLB misses to neighboring pages are serviced much faster from the L2 cache instead of going all the way to memory. However, if an application accesses memory with a large stride (larger than the size of a memory page times the number of page entries that fit in a cache line), each TLB miss will have to go to L3 or to memory causing an L2 cache miss. This is what happens in routine *rhs* of the LU benchmark. We cannot predict these cache misses because they are not produced by the application explicitly, but are the result of an interaction between the architectural design and the application's access stride.

On the bottom row of figure 6, we present the predictions and the measurements for two level 3 loops from routine *rhs*. For these loops, we notice that the number of L2 misses predicted by the model is zero for all problem sizes, but the measurements on the Itanium2 show the code experiences L2 misses once it starts missing in the TLB.

6.2 Predictions for MIPS R12000

Figure 7 presents the results for the Origin 2000 system. As with the Itanium results, each graph presents the normalized counts of cache and TLB misses. All L2 and TLB miss counts are scaled by a factor of 5 to bring them into the same range with the L1 miss counts and make the graphs easier to read. We present both the fully-associative and the set-associative predictions for the two cache levels, and only fully-associative predictions for TLB. While for the Itanium machine the difference between the fully-associative and the set-associative predictions is quite small due to the high as-

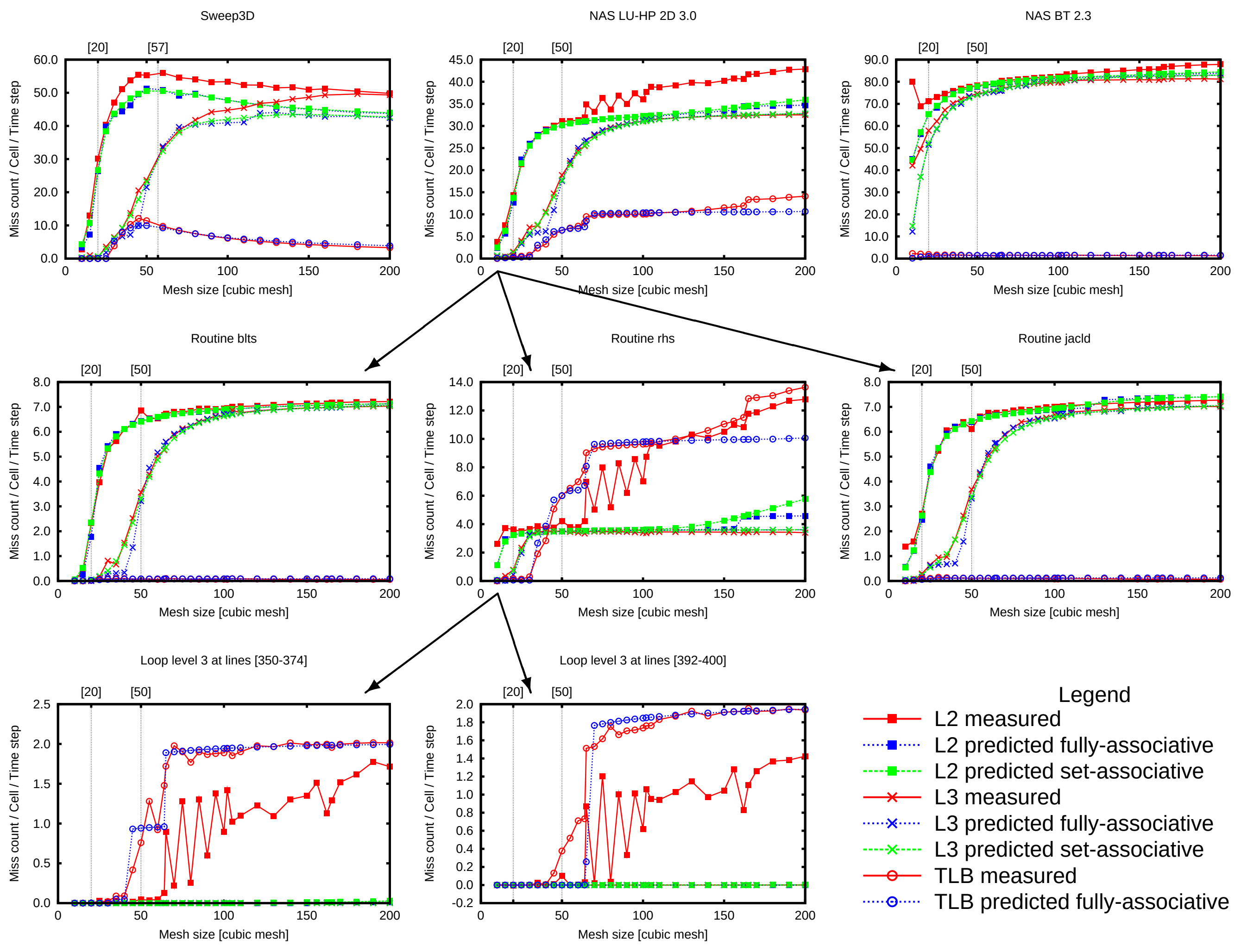

Figure 6: Predictions of L2, L3 and TLB misses for ASCI Sweep3D and two NAS benchmarks on an Itanium2 based machine with a 256KB 8-way set-associative L2 cache, 1.5MB 6-way set-associative L3 cache, and 128 entries fully-associative L2 TLB. For the LU application we present more detailed predictions for three of its routines and two level 3 loops.

sociativity level of its L2 and L3 caches, on the R12000 with its 2-way set-associative caches we can notice a significant difference. Although based on the simplifying assumption that accessed memory blocks are uniformly distributed, the set-associative predictions approximate well the measured counts. We cannot estimate precisely the conflict misses at each problem size (see the graph for Sweep3D in figure 7), but the set-associative predictions capture the actual trend.

We'll analyze in more detail the SP benchmark from NPB 3.0. We selected this benchmark on the R12000 because of the large difference between its fully-associative and set-associative L1 predictions. Second row in figure 7 presents the results for three routines from benchmark SP 3.0. We selected routines that show different memory utilization profiles, to demonstrate the accuracy of the models with various memory access patterns. We notice that most of the SP's L2 miss prediction error is produced by routine *compute_rhs*, and that almost all its TLB misses are produced by routine *z_solve*. While on the Itanium the 256 KB L2 cache seems too small for caching page table entries in the presence of large stride accesses, on the R12000 with its large 8 MB L2 cache we did not notice an increase in the number of L2 misses due to a high rate of TLB misses. Last row in figure 7 shows two level 3 loops from routine *z_solve*. The L1 set-associative predictions approximate well the measured values for all problem sizes, and we can see the number of capacity and conflict misses at each problem size. The accuracy of the set-associative predictions validate in turn the accuracy of our MRD models which predict the fully-associative distances used by the probabilistic model.

7. RELATED WORK

Over the years, memory reuse distance has been studied by many researchers investigating memory hierarchy management techniques [2, 11] or trying to understand data locality in program executions for individual program inputs [3, 4]. Recently, two other research groups have explored using memory reuse distance data from a few training runs to compute cache miss rate predictions for other program inputs. Zhong et al. [13] describe using two mem-

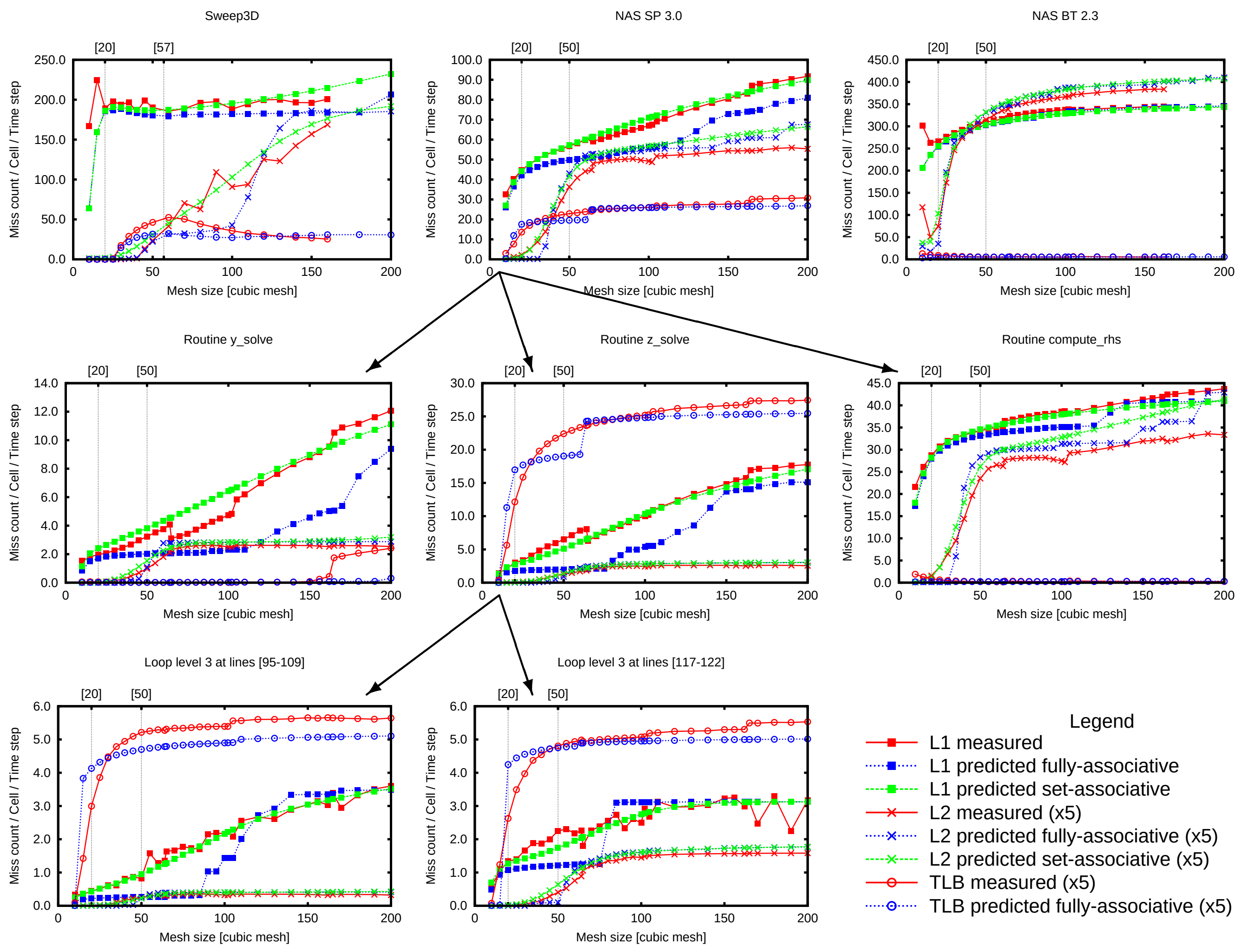

Figure 7: Predictions of L1, L2 and TLB misses for ASCI Sweep3D and two NAS benchmarks on a MIPS R12000 based machine with a 32KB 2-way set-associative L1 cache, 8MB 2-way set-associative L2 cache, and 64 double entries fully-associative TLB. For the SP application we present more detailed predictions for three of its routines and two level 3 loops.

ory reuse distance histograms that are an aggregation of all accesses executed by a program as the basis for modeling. Fang et al. [5] use a similar modeling strategy but they collect data and build models on a per-instruction basis.

Our work differs from that of Zhong et al. and Fang et al. in six important ways. First, we characterize memory access patterns at the level of references groups determined through static analysis while the other two groups, respectively, build their models for the entire program or at the level of single instructions. Although we have never directly compared our models against those produced by either of the other two approaches, we have experimented with different levels of aggregation using our implementation. In those experiments, we found that building models from histograms constructed at the program or routine level for non trivial programs results in significant errors with our automated method. Similarly, our first implementation of fine-grain modeling (at the instruction-level) performed no aggregation and its accuracy was hurt by complex interactions between multi-word memory blocks and loop unrolling [8, pages 46–53]. Second, our modeling tool adaptively determines an appropriate partitioning of reuse distance histograms into bins while the other two groups use a fixed strategy based either on a constant number of bins (e.g. 1000) for every histogram, or on a logarithmic distribution of distances into bins. Third, we discover the appropriate modeling polynomials for each bin automatically and our models are linear combinations of a set of basis functions with a dynamically determined number of terms in each model. Zhong et al. and Fang et al. use combinations of only two terms where one is selected from a small set of pre-determined functions and the other is a free term. Fourth, we predict the actual number of cache misses for different input sizes rather than just a miss rate. Fifth, we predict cache miss counts for both fully-associative and set-associative caches. Finally, our models can be used to directly predict memory hierarchy responses for problem sizes not measured; the other aforementioned techniques require partial execution of using the problem size for which a prediction is desired to experimentally determine data sizes.

8. CONCLUSIONS

This paper describes a technique for constructing machine-independent models that can be used to predict the memory hierarchy response for an application on architectures and problem sizes that have not been studied. By combining models of application memory access patterns based on data reuse distance with probabilistic models that capture the essence of set-associativity in architectures, we are able to accurately predict cache miss counts for a diverse set of cache configurations over a large range of problem sizes. In validating the fidelity of our models, we found that architectural quirks can cause differences between measured and predicted performance. For instance, on Itanium2, caching page table entries in a relatively small L2 cache can produce a significant number of additional L2 misses when the TLB miss rate is high. Our memory reuse distance based models cannot predict these additional L2 misses, as they are not caused explicitly by memory references in the application.

Our reuse distance based models capture the memory access pattern of an application, therefore they are not portable across HLO[7] optimizations that change the application's memory access pattern. We can predict the memory hierarchy behavior of an application in the presence of HLO transformations by constructing models from measurements on a binary optimized with the same set of transformations. Currently, neither our models nor any of the other application-centric models described in Section 7 can predict the number of cache misses in the presence of hardware or software prefetching. However, prefetching algorithms implemented in hardware are usually not very complex and we plan to explore predicting their effects through a combination of static and dynamic analysis.

Our immediate plans for research in this area include analyzing the dependence graph of each loop to understand the level of memory parallelism at loop level. Such information would enable us to predict the exposed latency for each cache miss, which we can then use to refine our predictions of execution time for scientific applications [9].

9. ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under Grant No. ACI 0103759 and the Department of Energy under Contract Nos. 03891-001-99-4G, 74837-001-03 49, and/or 86192-001-04 49 from the Los Alamos National Laboratory. We thank the Performance and Architecture Laboratory at Los Alamos National Laboratory for hosting Gabriel as an intern during the summer of 2003. We thank Dick Hanson for his suggestions on modeling approaches with Matlab.

10. REFERENCES

[1] The ASCI Sweep3D Benchmark Code. DOE Accelerated Strategic Computing Initiative. `http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/asci_sweep3d.html`.

[2] B. Bennett and V. Kruskal. LRU stack processing. *IBM Journal of Research and Development*, 19(4):353–357, July 1975.

[3] K. Beyls and E. D'Hollander. Reuse distance as a metric for cache behavior. In *IASTED conference on Parallel and Distributed Computing and Systems 2001 (PDCS01)*, pages 617–662, 2001.

[4] C. Ding and Y. Zhong. Reuse distance analysis. Technical Report TR741, Dept. of Computer Science, University of Rochester, 2001.

[5] C. Fang, S. Carr, S. Onder, and Z. Wang. Reuse-distance-based Miss-rate Prediction on a Per Instruction Basis. In *The Second ACM SIGPLAN Workshop on Memory System Performance*, Washington, DC, USA, June 2004.

[6] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. *IEEE Trans. Comput.*, 38(12):1612–1630, 1989.

[7] J. Larus and E. Schnarr. EEL: Machine-Independent Executable Editing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–300, June 1995.

[8] G. Marin. Semi-Automatic Synthesis of Parameterized Performance Models for Scientific Programs. Master's thesis, Dept. of Computer Science, Rice University, Houston, TX, Apr. 2003.

[9] G. Marin and J. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 2–13. ACM Press, 2004.

[10] MathWorks. *Optimization Toolbox: Function quadprog*. `http://www.mathworks.com/access/helpdesk/help/toolbox/optim/quadprog.shtml`.

[11] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[12] R. E. Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9:355–365, 1974.

[13] Y. Zhong, S. G. Dropsho, and C. Ding. Miss Rate Prediction across All Program Inputs. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, New Orleans, Louisiana, Sept. 2003.

[7]HLO = high level loop optimizations such as tiling, loop interchange, unroll & jam, prefetching, etc.