# Predicting Memory-Access Cost Based on Data-Access Patterns

Surendra Byna Xian-He Sun
*Illinois Institute of Technology*
*{renbyna, sun}@iit.edu*

William Gropp Rajeev Thakur
*Argonne National Laboratory*
*{gropp, thakur}@mcs.anl.gov*

## Abstract

*Improving memory performance at software level is more effective in reducing the rapidly expanding gap between processor and memory performance. Loop transformations (e.g. loop unrolling, loop tiling) and array restructuring optimizations improve the memory performance by increasing the locality of memory accesses. To find the best optimization parameters at runtime, we need a fast and simple analytical model to predict the memory access cost. Most of the existing models are complex and impractical to be integrated in the runtime tuning systems. In this paper, we propose a simple, fast and reasonably accurate model that is capable of predicting the memory access cost based on a wide range of data access patterns that appear in many scientific applications.*

## 1. Introduction

Immense research effort has been spent on reducing the performance gap between processor and memory. Processor speed continues to increase every year. CPU clock frequency is doubling every 18 months complying with Moore's Law. On the other hand main memory (DRAM) speeds haven't increased enough to catch up with the processor speed. This performance gap has been increasing for the last 20 years [13] and the trend appears to continue in the near future.

Cache memories work on the principle of spatial and temporal locality [17]. However, there are many applications that lack locality in accessing the memory. These applications spend a major fraction of execution time waiting for data accesses. Cache memories are exploited better if the cached blocks of data are reused extensively before other cache blocks replace them.

Transforming and reordering the memory accesses improve application performance [10, 7]. As loops are the basic blocks, where most of time is spent in applications, various loop optimization techniques have been developed to enhance the memory hierarchy utilization. Loop transformations (loop unrolling, loop fusion, loop interchange, loop reversal and loop tiling) are some of the most effective loop optimizations.

Some advanced compilers utilize these optimization techniques. But, compilers alone are not sufficient to achieve the best possible optimization [1]. Superior manual optimizations require extensive knowledge of the hardware architecture and data access patterns. The developer needs to be aware of efficient optimization techniques to be applied in the right place. Automating the optimization process is needed to obtain consistent performance.

Performance prediction of memory access cost is required to automate the optimizations. Currently there are a few automatic tuning software tools. One of the most popular tools of optimization is Automatically Tuned Linear Algebra Software (ATLAS) [19]. This tool runs subroutines multiple times to obtain the best optimization parameters by a trial and error method. A prediction model can remove these multiple runs and be extended to optimize more than just linear algebra subroutines. This model should be simple and fast to perform the optimization dynamically, at runtime, based on the data access pattern and available memory hierarchy.

Many researchers worked towards developing accurate cache performance models. But most of these models [2, 18, 6] lack generality. They are complex, and are bounded to a few algorithms or data access patterns. Jacob [6] extracts address traces from the code, which requires execution of the program, and consumes a lot of time if an optimization has to be applied. We base our prediction model on various access patterns, which are parameterized. This helps in predicting the memory cost with very small complexity and skips the costly process of tracing the references every time a loop parameter is changed. Chatterjee et al. [3] studies the exact analysis of cache misses based on the polyhedral model, which is complex. The Cache Miss Equations model (CME) [5] is the least costly performance model to our knowledge. However, this model also requires tracing the references to create the reuse vectors and solve cache miss equations. These

models are accurate but expensive, and are better choices for static analysis of cache behavior. Our model fits better in choosing the optimization parameters dynamically at runtime than CMEs.

Our model also focuses on wide range of data access patterns with multiple array variables. Most of the other cache analysis models hold good results for a specific algorithm [2, 18] and fall short in acquiring generality.

The rest of this paper is organized as follows. Section 2 classifies various data access patterns that are used in most of the scientific applications. Section 3 discusses the parameters of memory hierarchy. In Section 4, we propose the memory access cost analysis and prediction equations. Section 5 provides experimental verification, and section 6 discusses an application of our model and current projects. Section 7 concludes with further objectives.

## 2. Data Access Patterns

Loops and arrays are fundamental structures of most numerical and scientific applications [14]. A major share of the execution time of these applications is spent in loops, accessing data from arrays. Analyzing these access patterns is needed to find out the hotspots and to optimize the performance by reorganizing these memory references.

Data access patterns are classified based on the stride between successive accesses. Modal model of memory [11] categorize data accesses as constant, strided and non-monotonic modes. Yan et al. [15] classify memory access patterns into three types: migratory, group and unpredictable patterns.

We classify data access patterns in scientific applications as constant, contiguous and non-contiguous. Non-contiguous patterns are further divided into four patterns based on the size of data blocks accessed with each reference and their successive strides. Stride is the distance between the previous reference and current reference.

*Constant* accesses are those where the same data block is accessed repeatedly i.e., stride is equal to zero. *Contiguous* access pattern is where the stride between successive accesses is equal to the size of datatype. These are divided further as fixed length block accesses and variable length block accesses. Fixed length block accesses refer to the same datatype in consecutive references.

*Non-contiguous* access pattern is where the stride of next reference is greater than the size of currently accessed datatype. These can be further divided as follows:

a) *Fixed length block, with fixed stride*: Stride is similar through out the access pattern. As shown Figure. 1.a., a block with a size of constant block_size is copied into dest from src. The next block is copied from *src+stride* to *dest+block_size*, i.e. array src is being accessed non-contiguously with a fixed stride and array dest is being accessed contiguously.
b) *Fixed length block, with varying stride*: The stride varies between each access. (Figure. 1.b.)
c) *Variable size block, with fixed stride*: Accessing different or varying size datatypes, where the strides of accesses are similar. (Figure. 1.c.)
d) *Variable size block, with variable stride*: Accessing different or varying size datatypes, where the strides of accesses are varying. (Figure. 1.d.)

In Section 4, we predict the memory access cost for these data access patterns.

```
for (i=0; i < n; i += stride)
{
        memcpy(dest, src+i, block_size);
        dest += block_size;
}
```

**Figure. 1.a**

```
for (i=0, j=0; i < n; i += stride[j])
{
        memcpy(dest, src+i, block_size) ;
        j++ ;
        dest += block_size;
}
```

**Figure. 1.b**

```
for (i=0, j=0; i < n; i += stride)
{
        memcpy(dest, src+i, block_size[j]) ;
        dest += block_size[j];
        j++ ;
}
```

**Figure. 1.c**

```
for (i=0, j=0; i < n; i += stride[j])
{
        memcpy(dest, src+i, block_size[j]) ;
        dest += block_size;
        j++ ;
}
```

**Figure. 1.d**

**Figure. 1 Non-contiguous data access patterns**

## 3. Model parameters

To bridge the gap between processor and memory performance, modern computer architectures include multiple levels of memory hierarchies that consist of cache memory and TLB. In this section, we discuss the details of memory hierarchy parameters, which are used in developing the prediction model.

A Cache memory is characterized by its size, line size and associativity. Cache size ($C$) represents its capacity in bytes. Caches are organized in cache lines. When a cache miss occurs, a block of data of size equal to cache line size ($L$) is fetched from the next level of cache or memory. This property conforms to the spatial locality. Associativity ($A$) of a cache helps in deciding how many places are there to place a cache line.

We treat the TLB as a level of memory hierarchy. Its parameters are page size $P$ (similar to cache line size of a cache) and the capacity. The capacity of a TLB is the amount of memory page mapping it can store and is equal to *number of page entries* multiplied by page size.

Table 1 summarizes the memory hierarchy parameters. Subscript i of a parameter signifies the level of that cache/TLB in the hierarchy of memory. Cache memory at level i has three properties: its size in bytes ($C_i$), cache line size ($L_i$) and its associativity ($A_i$). TLB is represented with the number of page table entries ($T_s$), page size ($P_s$) and it associativity ($A_T$).

**Table 1. Memory hierarchy parameters**

| | |
|---|---|
| $C_k$ | Cache size at $k^{th}$ level cache of memory hierarchy |
| $L_k$ | Cache line size at $k^{th}$ level cache of memory hierarchy |
| $A_k$ | Associativity of $k^{th}$ level cache of memory hierarchy |
| $M_k$ | Number of cache misses at $k^{th}$ level cache of memory hierarchy |
| $M^c_{(k,i)}$ | Number of cache misses at $k^{th}$ level cache of memory hierarchy in accessing $i^{th}$ array, contiguously. |
| $M^n_{(k,i)}$ | Number of cache misses at $k^{th}$ level cache of memory hierarchy in accessing $i^{th}$ variable, non-contiguously. |
| $T_s$ | Number of page table entries (PTE) in TLB |
| $P_s$ | Page size of each PTE |
| $A_T$ | TLB associativity |
| $M$ | Number of cache levels in memory hierarchy |

$M$ refers the total number of cache levels.

Cache misses are classified into three types [9]. They are *compulsory misses, capacity misses and conflict misses.* Cache misses at level i are represented with $M_i$. $M^c_{(k,i)}$ refers to the number of cache misses at level k of memory hierarchy, in accessing $i^{th}$ array (variable) contiguously. If it is being accessed non-contiguously, it is represented by $M^n_{(k,i)}$.

Data access pattern parameters are shown in Table 2. The subscript i represents the $i^{th}$ array being accessed. The parameters $R^c_{(k,i)}$ and $R^n_{(k,i)}$ represent the number of contiguous and non-contiguous references separately. $S_i$ is the fixed stride in accessing the $i^{th}$ array and $S_{(i,j)}$ is the variable stride. $D$ is the working set size and $W_i$ represents the block (word) size of the $i^{th}$ array.

**Table 2. Data access parameters**

| | |
|---|---|
| $R^c_{(k,i)}$ | Number of contiguous references of $i^{th}$ array at cache level $k$ of the memory hierarchy. |
| $R^n_{(k,i)}$ | Number of non-contiguous references of $i^{th}$ array at cache level $k$ of the memory hierarchy. |
| $W_i$ | Fixed size of the data block being accessed in $i^{th}$ array. |
| $S_i$ | Fixed stride of accessing $i^{th}$ array non-contiguously. |
| $W^c_i$ | Variable size of the data block being contiguously accessed in $i^{th}$ array. |
| $W^n_{(i,j)}$ | Variable size of $j^{th}$ data block being non-contiguously accessed in $i^{th}$ array. |
| $S_{(i,j)}$ | Variable stride of the $j^{th}$ data block being contiguously accessed in $i^{th}$ array. ($stride_j$ in stride signature) |
| $D$ | Size of working set |

## 4. Memory access cost prediction

Our goal is to predict the memory access cost of a basic block of loop with any type of data access patterns discussed in section 2, and for multiple data array variables. We assume LRU replacement policy for cache and TLB. We assume that the memory hierarchy is following inclusive property. The total cost of accessing memory includes the access time and the miss penalties of these levels in the hierarchy. If there are k levels of cache memory and one level TLB [13],

*Total Memory cost = (Number of TLB hits * Time to access TLB) + (Number of TLB misses * TLB miss penalty) + (Number of L1 hits) * (Time to access L1) + (L1 misses * L1 penalty) + (L2 misses * L2 penalty) + … + (Lk misses * Lk penalty)* (4.1)

To predict this cost, we have to find the number of cache hits/misses at each level and TLB hit rate. We predict the cache and TLB misses based on the access pattern.

Assuming that there are $M$ levels of cache, the total miss penalty due to cache misses is the sum of miss penalty at each level.

$$T_m = \sum_{k=1}^{M} (M_k * T_k) - \alpha \quad (4.2)$$

where $M_k$ is the total number of cache misses and $T_k$ is the miss penalty at level $k$ cache. $\alpha$ is the overlapping the cache misses with prefetching and other OS optimizations.

Consider that there are $m$ array variables accessed contiguously and $n$ array variables accessed non-contiguously, the total number of misses at cache level $k$ is the sum of misses caused in accessing contiguously accessed arrays and those of non-contiguously accessed arrays.

$$M_k = \sum_{i=1}^{m} M^c_{(k,i)} + \sum_{i=1}^{n} M^n_{(k,i)} \quad (4.3)$$

$M^c_{(k,i)}$ is the number of cache misses at $k^{\text{th}}$ level cache of memory hierarchy in accessing $i^{\text{th}}$ variable, contiguously. $M^n_{(k,i)}$ is the number of cache misses at $k^{\text{th}}$ level cache of memory hierarchy in accessing $i^{\text{th}}$ variable, non-contiguously.

Now we count the number of cache misses based on the data access pattern.

**Constant access pattern:** In this type of accesses, once a word is loaded into the cache, the following accesses to the same word cause no extra cache misses. If the word size of a variable is $W_i$ and there are the number of cache misses is equal to $\lceil (W_i / L_k) \rceil$. (4.4)

**Contiguous access pattern:** In this pattern the stride between successive accesses is the same as data size. All the cache misses caused in this pattern are compulsory misses. Each reference fetches a cache line into the cache. Cache line contains more than one word of data. This is to assure spatial locality property of using cache memory. If the cache line size is more than the data type accessed, the next reference utilizes the prefetched data from the cache. Each reference causes $\lceil (W_i / L_k) \rceil$ misses, i.e. if the word size is more that cache line size, then it causes more than one miss, otherwise just one miss occurs for every $\lceil (L_k / W_i) \rceil$ references. If there are $n$ references, the number of cache misses caused at cache level $k$ in accessing variable $i$ is: $M^c_{(k,i)} = \lceil n * (W_i / L_k) \rceil$ (4.5)

If $i^{\text{th}}$ variable has $R^c_{(k,i)}$ references, the number of cache misses is:

$$M^c_{(k,i)} = \left\lceil R^c_{(k,i)} * \left( W_i^c \middle/ L_k \right) \right\rceil \quad (4.6)$$

where $W_i^c$ is size of the data block being contiguously accessed in $i^{\text{th}}$ variable.

The number of cache references at level $k$ ($R^c_{(k,i)}$) is the number of cache misses at the lower level cache, i.e. $R^c_{(k,i)} = M^c_{(k-1,j)}$. (4.7)

**Non-contiguous access patterns:** As described in section 2, there are four main types of access patterns. These patterns are classified based on the variability of stride and data block size. The occurrence of cache misses is categorized into four regions based on the working set size, similar to Saavedra and Smith [16]. First region is the one where all the working set fits in the cache. As long as the working set size is less than the cache size, the total data fits into the cache. All the cache misses are compulsory misses. This number is equal to $M^n_{(k,i)} = \lceil n * (W_i / L_k) \rceil$ at level $k$ of memory hierarchy in accessing $i^{\text{th}}$ variable non-contiguously. This number is the same for all types of non-contiguous access patterns.

When size of the data working set exceeds the cache size, three regions of memory operations are defined. The first region is when the stride ($S$) is between 1 and cache line size ($1 < S \le L_k$). The second region is $L_k < S \le D / A_k$, where $A_k$ is the associativity of $k^{\text{th}}$ level cache of memory hierarchy. The third region is $D / A_k < S \le D / 2$. In this last case, although the $D > C_k$, only $D / S < A_k$ amount of data is needed for access. In the last region the number of references mapping to a single set is less than the set associativity. Thus, only compulsory misses are caused in the third access pattern, i.e.

$$M^n_{(k,i)} = \lceil n * (W_i / L_k) \rceil \quad (4.8)$$

where $n$ is the number of data accesses. Thus, we set our focus on the first two regions to count the number of cache misses.

First we find the cache misses for a fixed size of data block accesses of one variable, with a fixed stride.

If the stride (fixed) is less than the cache line size, one cache miss occurs for ( $L_k / S$ ) references. If there are $n$ references, the number of cache misses is: $n*(S/L_k)$, where $n$ is the number of data accesses.

If the stride (fixed) is more than the cache line size, each reference causes $(\max(\lceil W_i / L_k \rceil,1)$ cache misses, i.e. each access causes one miss when the word size is less than $L_k$. If word size is more than $L_k$, each reference causes $\lceil W_i / L_k \rceil$ misses. If there are $n$ references, the number of cache misses is equal to $n*(\max(\lceil W_i / L_k \rceil,1)$.

$$M^{n}_{(k,i)} = R^{n}_{(k,i)} * (\max(\lceil W_i / L_k \rceil,1) \qquad (4.9)$$

For variable stride with fixed size block accesses, the cache misses have to be counted for each stride. If the stride is less than $L_k$, it does not cause a cache miss as the pre-fetched line of data is reused. The number of cache misses is:

$$M^{n}_{(k,i)} = (\sum_{j=1}^{R^{n}_{(k,i)}} \lfloor \min((S_{(i,j)} / L_k),0) \rfloor) * (\max(\lceil W^{n}_{(i,j)} / L_k \rceil,1)) \qquad (4.10)$$

$M^{c}_{(k,i)}$ is the number of cache misses at $k^{\text{th}}$ level cache of memory hierarchy in accessing $i^{\text{th}}$ variable, non-contiguously, $S_{(i,j)}$ is variable stride of the data block being contiguously accessed in $i^{\text{th}}$ variable. $R^{n}_{(k,i)}$ is the number of non-contiguous references of i$^{\text{th}}$ array at cache level $k$ of the memory hierarchy. $W^{n}_{(i,j)}$ is the size of $j^{\text{th}}$ data block being non-contiguously accessed in $i^{\text{th}}$ variable. In this pattern when the stride $S_{(i,j)}$ is less than the cache line size, we assume that the cache line has already been fetched into the cache. However when this stride is causing to fetch a new cache line, then this formula misses to count that cache miss. This can be corrected by maintaining the history of cache line that has been fetched recently.

The number of cache references at level $k$ ( $R^{n}_{(k,i)}$ ) is the number of cache misses at the lower level cache, i.e. $R^{n}_{(k,i)} = M^{n}_{(k-1,j)}$. (4.11)

For fixed or variable stride with variable size block accesses, the cache misses have to be counted for each block size. In this case, we assume that the stride is always more than $L_k$. If the data block size is less than $L_k$, it does not cause a cache miss as the pre-fetched line of data is reused. The number of cache misses is:

$$M^{n}_{(k,i)} = \sum_{j=1}^{R^{n}_{(k,i)}} (\max(\lceil W^{n}_{(i,j)} / L_k \rceil,1) \qquad (4.12)$$

Refer Table 3 and Table 4 (at the end of this document) for a summary of formulae to calculate the cache misses for all data access patterns. Using (4.3) total number of cache misses in accessing contiguous and non-contiguous data is calculated. Formula 4.2 gives the total memory access cost.

## 5. Performance verification

This section presents performance measurements to verify the predicted memory access cost with the measured cost on various architectures. We measure the performance of loops with all the data access patterns mentioned above and compare that performance with the predicted performance.

We took the measurements on a Sun Solaris based cluster called Sunwulf, which is located at the Scalable Computer Software Lab of Illinois Institute of Technology. Sunwulf is composed of a four-processor E450 server and 63 high-end workstations. We run our experiments on one of the nodes. Each node is a SUN Blade-100 workstation with one UltraSparc-IIe, 500MHz CPU. The L1 cache is 16KB, with a 16-byte cache line size. The L2 cache has a capacity of 8MB and its line size is 64 bytes. It also has a TLB with 4KB page size and 48 entries. We used a microbenchmark to find the average access time and miss penalty of each level of memory hierarchy. This is similar to the microbenchmark proposed by Saavedra and Smith [16].

Another platform we used for experiments is a 32-node Beowulf, located at University of South Carolina. Each node consists of 933MHz, Pentium III processor. It has 16 KB L1 cache and 256KB L2 cache. Both these caches are on the die, and the average penalty for load misses is measured as 7 cycles and 70 cycles for L1 and L2 respectively. We chose these processors, as they apply inclusive property in the memory hierarchy with less aggressive pre-fetching.

We used the loops similar to Figure.1 and measured the time to execute those loops. In all these loops, two array variables are accessed with different access patterns. We chose these loops since many applications

contain loop blocks where the data accesses are similar to the access patterns discussed above in section 2. We can apply the same prediction model for any number of arrays. Execution time of these loops contains only the data access cost, without any computation cost. We used pointer-to-pointer copy to avoid the cost of memcpy. In these experiments we ran many iterations of the program to find the minimum cost. We also flushed the cache after measuring the time for an iteration to replace any cache blocks that are reusable. We compiled these programs using gcc 3.0 and padded the arrays to avoid any cache thrashing. The comparison of predicted cost and measured memory access cost is presented in the following paragraphs. The memory access cost is presented as a ratio of execution time to the number of memory references. This normalization is done to fit all the data into the graph. The performance is better for lower values.

Figure 2 and Figure 3 compare the predicted memory access cost with measured cost in running the loops in various data access patterns explained in section 2 (Figure.1) on Sunwulf cluster. For contiguous data accesses (Figure 2.a.), the predicted cost is constant per reference. The prediction error reduced as the number of references increased. The error was mainly due to the approach of counting cache misses

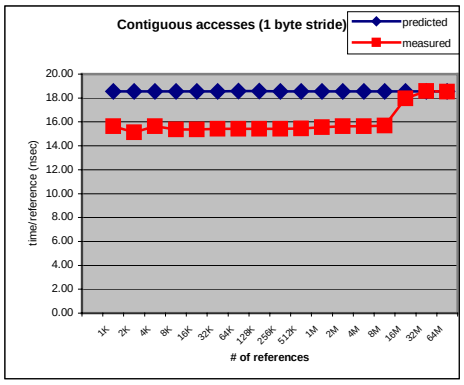


Figure 2.a.

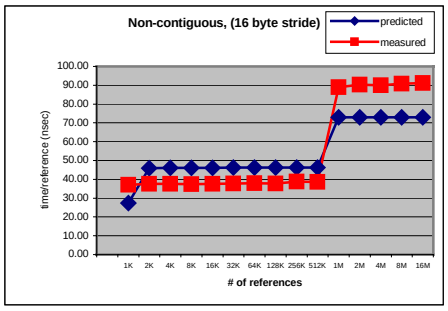


Figure 2.b.

**Fig.2. Comparison of measured and predicted memory access cost. The access patterns are: 2.a. Contiguous data access (word size: 1byte, stride: 1byte). 2.b. Non-contiguous data access with fixed word size and stride (word size: 8 bytes, stride: 16 bytes)**

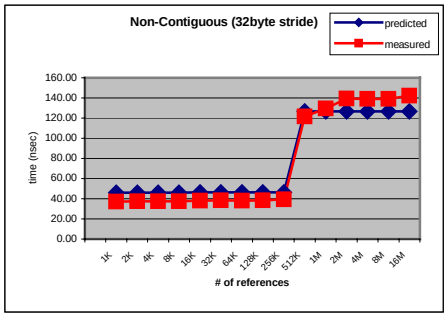


Figure. 3.a.

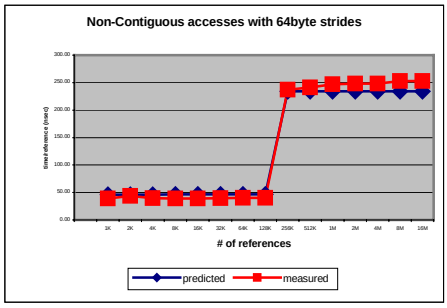


Figure. 3.b.

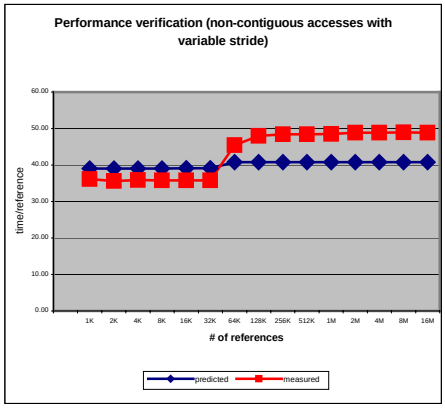


Figure. 3.c.

**Figure.3. Comparison of measured and predicted memory access cost. The access patterns are: 3.a. Non-contiguous data access with fixed word size and stride (word size: 8 bytes, stride: 32 bytes) 3.b. Non-contiguous data access with fixed word size and stride (word size: 8 bytes, stride: 64 bytes) , 3.c. non-contiguous data access with fixed word size and variable strides (word size: 8 bytes, stride varies from 1 to 128 bytes periodically)**

pessimistically without taking prefetching into consideration. The prediction error was below 20% for small data and below 4% for large data with this data access pattern.

To test the non-contiguous access pattern performance we used three sizes of fixed strides (16bytes, 32 bytes and 64 bytes) that are equal to L1 cache line size, more than L1 line size and that of equal to L2 line size. For non-contiguous accesses, with stride equal to L1 cache line size, the prediction error reduced as the data size increase. It can be seen from Figure 2.b and Figure 3.a, that the utilization of caches are more effective when the data size is less than L2 cache size. Overall the error is below 20% in most of the cases. For the remaining two non-contiguous access patterns with fixed strides, the prediction error is below 10% for larger data sizes.

For non-contiguous access pattern with variable strides, we initialized an array that contains strides of accesses. Prediction cost of this access pattern contains the cost of accessing non-contiguous arrays as well as the cost of accessing the array of strides. The prediction error is below 15% (Figure 3.c). This error is caused by missing some of the cache misses in non-contiguous accesses, which requires maintaining the history of the length of cache lines that are already been fetched into the cache. Another reason for

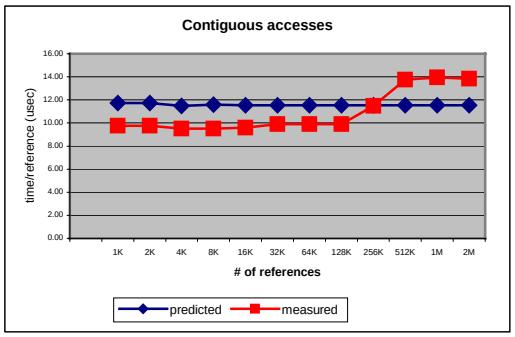


Figure 4.a.

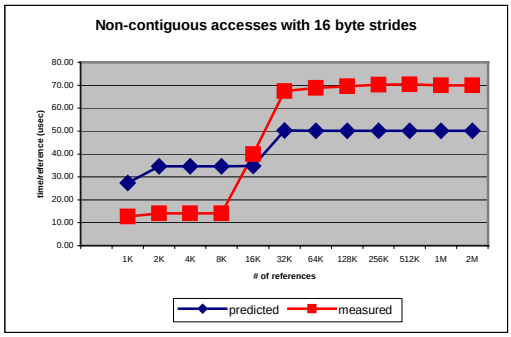


Figure. 4.b.

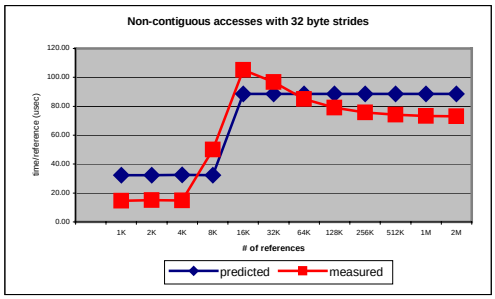


Figure 4.c.

**Figure 4. Comparison of measured and predicted memory access cost on Pentium III processor. The access patterns are: 4.a. Contiguous data access (word size: 1byte, stride: 1byte). 4.b. Non-contiguous data access with fixed word size and stride (word size: 8 bytes, stride: 16 bytes), 4.c. Non-contiguous data access with fixed word size and stride (word size: 8 bytes, stride: 32 bytes)**

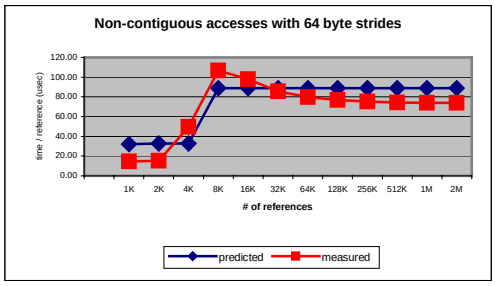


Figure. 5.a.

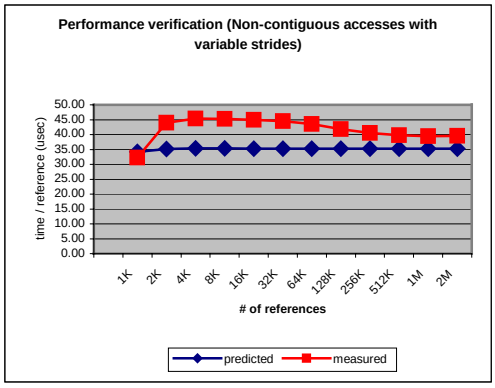


Figure 5.b.

**Figure.5. Comparison of measured and predicted memory access cost on Pentium III processor. The access patterns are: 5.a. Non-contiguous data access with fixed word size and stride (word size: 8 bytes, stride: 64 bytes), 5.b. for non-contiguous data access with fixed word size and variable strides (word size: 8 bytes, stride varies from 1 to 128 bytes periodically)**

prediction error for all these access patterns is that we are using average miss penalties, which may not be accurate.

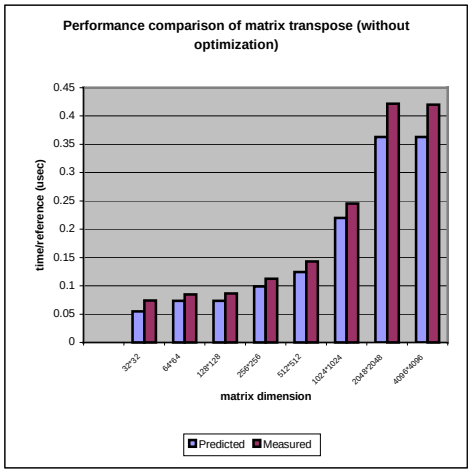


**Fig.6. Comparison of measured and predicted memory access cost Matrix transpose algorithm without cache blocking optimization**

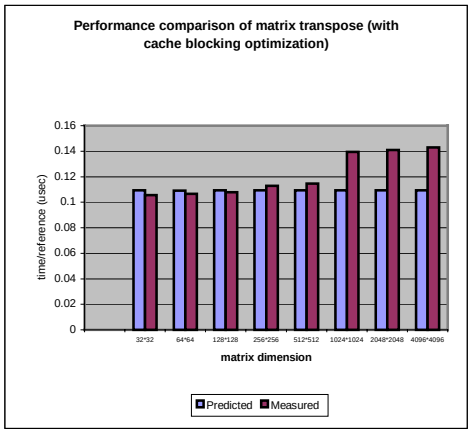


**Fig.7. Comparison of measured and predicted memory access cost Matrix transpose algorithm with cache blocking optimization**

We observe the similar results on Pentium III processor on Beowulf cluster (Fig 4 and 5). The prediction error is slightly high for small data sizes where the prefetching of this processor is effective. As the working set size increase, the L2 misses increase and the prediction error is below 20% in these cases.

We also verified the performance of the loops in NAS Parallel benchmarks that are performing matrix transpose operation. We have measured the performance two variations of matrix transpose algorithms from NAS Parallel benchmarks' Fast Fourier Transform program. The first algorithm is a simple matrix transpose of copying rows of one matrix to columns of another matrix. The second algorithm uses cache-blocking optimization to improve the performance. Both algorithms fit into the data access patterns explained in section 2. The data working set of the first algorithm increases with the dimension of the matrices. Due to the row major ordering of arrays (in C or column major ordering in Fortran), one matrix is accessed contiguously and the other is accessed non-contiguously with fixed stride. The second algorithm makes sure that a block of data is fully utilized before replacing it from the cache. In this algorithm, the two matrices are accessed non-contiguously with fixed strides. However, as the whole data block is reused before it is being replaced, and we chose the block size such that it fits into the cache, the number of cache misses is very less compared to the unoptimized version of matrix transpose. These experiments are performed on Sun UltraSparc IIe processor node.

As expected, the performance (time/reference) increases as the data size increases for the unoptimized transpose algorithm (Fig 6). Predicted values of performance are slightly different from the measured values. The error is around 13%. In the second algorithm, the performance is improved for the transpose algorithm due to the cache-blocking optimization (Fig 7). The performance error was below 5% for most of the data sizes, but increased for large data sizes. This is mainly due the increase in average time per memory reference for the large data sizes.

## 6. An application of the model

Parallel communication models such as LogP [4] focus on network communication, with limited consideration of memory communication. Recently, the LogP model was extended to incorporate memory-communication cost. The memory-LogP model formally characterizes the memory-communication cost under four parameters: l: the effective latency, defined as the length of time the processor is engaged in transmission or reception of a message due to the influence of data size (D) and distribution also called as strides (S), l=f(D,S); o: the overhead, defined as the length of time the processor is engaged in transmission or reception of an ideally distributed (contiguous) message (during this time, the processor cannot perform other operations); g: the gap, defined as the minimum time interval between consecutive message receptions at the processor (the reciprocal of g corresponds to the available per processor bandwidth for a given implementation of data transfer on a given system); and P: the number of processor/memory modules (point-to-point communication in the memory

hierarchy implies P=1). Detailed information about the memory-LogP model can be found in [8].

The memory-communication cost for sending a data segment depends on architectural parameters, such as cache capacity, and code characteristics, such as data distribution, as explained in the memory-LogP model. In general, the overall communication cost includes data-collection overhead, the cost of data copying to the network buffer, the cost of data forwarding to the receiver (network-communication cost), and other costs added by the middleware implementation. When data distribution in memory is noncontiguous, the data is typically collected into a contiguous buffer before being copied to the network buffer. This process adds extra buffering overhead to the overall communication cost and is implementation dependent. The memory access cost predicted in this paper is a part of the latency (l) parameter of the memory-logP model.

Currently we apply this model in improving the performance of MPI derived datatypes by optimizing the memory access cost [1]. The MPI Standard [12] supports derived datatypes, which allow users to describe noncontiguous memory layout and communicate noncontiguous data with a single communication function. This feature enables an MPI implementation to optimize the transfer of noncontiguous data. In practice, however, few MPI implementations provide derived datatypes in a way that performs better than what the user can achieve by manually packing data into a contiguous buffer and then calling an MPI function. Memory access cost has been the reason for this performance bottleneck. We use memory-logP model with the help of prediction formulae to predict this cost and apply memory access optimization techniques to improve the performance. Due to space restriction, we cannot explain the optimization method here. Refer to [1] for full details.

## 7. Conclusion

Loop transformations and loop access reordering techniques improve the memory access performance. To obtain these loop optimization parameters, a simple, fast and accurate memory access cost prediction model is necessary. This improves the standard of application level optimizations and reduces the burden on the programmers to learn the rapidly improving processor and computer architecture technology. Towards achieving this goal, in this paper we proposed an analytical model to predict the memory access cost based on the data access patterns. We first classified the most common data access patterns in scientific computing applications. We then proposed a model to predict the memory access cost. We verified this model with measurements and showed that this model is practical. The accuracy of our model is reasonable given its simplicity. We also applied this model to matrix transpose routines in Fast Fourier Transform program of NAS benchmarks, which was implemented in different memory access patterns.

Our model is simple, effective, and easy to be incorporated into memory cost tuning tools, where optimization parameters are to be found at runtime. The prediction errors of 10% to 20% exist, they are reasonably accurate in making optimization decisions. We are currently utilizing this model to improve the performance of MPI derived datatypes, by optimizing the memory access cost. This cost prediction is a part of our memory-logP model, which emphasizes the importance of memory communication performance in point-to-point communication. Our model is practical because of its simplicity. We are able to fit this easily into any optimization library to choose optimization parameters dynamically at runtime. This is not possible with the existing models due to their complexity.

We plan to extend this work in various aspects. We will extend this model to include external and internal conflict misses. We will broaden this model for replacement policies other than LRU, such as FIFO, LFU, MRU, MFU etc. We plan to incorporate this model in an automatic performance tuning system that improves the application performance by optimizing the memory access cost.

## 8. Acknowledgements

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38, and in part by a grant from the Office of Advanced Simulation and Computing, National Nuclear Security Administration, U.S. Department of Energy. The U.S. Government retains for itself, and others acting on its behalf, a paid-up, nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.

## 9. References

[1] Surendra Byna, William Gropp, Xian-He Sun, and Rajeev Thakur, "Improving the Performance of MPI Derived Datatypes by Optimizing Memory-Access Cost," *IEEE International Conference on Cluster Computing*, Hong Kong, December 2003

[2] S. Chatterjee and S. Sen, "Cache-Efficient Matrix Transposition", *Proceedings of the 6th International Symposium on High-Performance Computer Architecture*, Toulouse, France, January 2000, pages 195-205.
[3] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck, "Exact Analysis of the Cache Behavior of Nested Loops", *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, Snowbird, UT, June 2001
[4] D.E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken "LogP: A Practical Model of parallel computation", *Communications of the ACM*, vol. 39, pp. 78-85, 1996
[5] Somnath Ghosh , Margaret Martonosi , Sharad Malik, "Cache miss equations: a compiler framework for analyzing and tuning memory behavior", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, v.21 n.4, p.703-746, July 1999
[6] B.L. Jacob, "An analytical model for designing memory hierarchies", *IEEE Transaction on Computers*, volume 45, pp. 83-105, 1996.
[7] M. Kandemir, J. Ramanujam and A. Choudhary, "Cache Locality by a Combination of Loop and Data Transformations," *IEEE Transactions on Computers (TC)* 48(2): 159–167, February 1999.
[8] Kirk W. Cameron, Xian-He Sun, "Quantifying Locality Effect in Data Access Delay: Memory logP," i*n Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS '03)*, April 2003.
[9] Hill, M.D. "Aspects of cache memory and instruction buffer performance", *Ph.D. Thesis*, University of California, Berkeley, 1987.
[10] K.S. McKinley, S.Carr, and C.W. Tseng, "Improving data locality with loop transformations", *ACM TOPLAS*, 18(4): 424-453. July 1996
[11] N. Mitchell, L. Carter, and J. Ferrante, "A modal model of memory". *In V.N.Alexandrov, J.J. Dongarra, Computer Science. Springer*, May 28-30, 2001
[12] Message Passing Interface Forum, "MPI-2: A message passing interface standard", *High Performance Computing Applications*, 12(1-2):1-299, 1998
[13] D. A. Patterson and J. L. Hennessy, "*Computer Architecture: A quantitative approach*", 2nd edition. San Fancisco, CA: Morgan Kaufmann Publishers, 1996.
[14] Y. Paek, J. Hoeflinger, and D. Padua, "Simplification of Array Access Patterns for Compiler Optimizations". *In Proceedings of the ACM SIGPLAN 98 Conference on Programming Language Design and Implementation*, June 1998. 23
[15] Rong Yan and Seth C, "Goldstein Mobile Memory: Improving Memory Locality in Very Large Reconfigurable Fabrics", *FCCM '02*, Napa Valley, CA, April 2002
[16] Rafael H. Saavedra and Alan Jay Smith, "Measuring Cache and {TLB} Performance and Their Effect on Benchmark Runtimes", *IEEE Transactions on Computers*, Volume: 44, number: 10, p1223-1235, 1995.
[17] A.J. Smith, "Cache Memories", *Computing Surveys*, 14(3), p.473, September 1982
[18] S. Sen and S. Chatterjee, "Towards a theory of Cache efficient algorithms", *SODA*, 2000
[19] R. Clint Whaley, Antoine Petitet, and Jack Dongarra, "Automated Empirical Optimizations of Software and the ATLAS Project", *Parallel Computing*, Volume 27, Numbers 1-2, pp 3-25, 2001

**Table 3. Number of cache misses for data access patterns**

| **Data access pattern** | **Number of cache misses** |
|---|---|
| Constant | $M^c_{(k,i)} = \left\lceil (W^c_i / L_k) \right\rceil$ |
| Contiguous | $M^c_{(k,i)} = \left\lceil R^c_{(k,i)} * (W^c_i / L_k) \right\rceil$ |
| Non-contiguous ($D < C_k$) | $M^c_{(k,i)} = \left\lceil R^c_{(k,i)} * (W^c_i / L_k) \right\rceil$ |

**Table 4. Number of cache misses for non-contiguous data access patterns with varying stride and data size**

| **Stride** | **Number of cache misses** |
|---|---|
| $1 < S \le L_k$ | $M^n_{(k,i)} = \left\lceil R^n_{(k,i)} * (S / L_k) \right\rceil$ |
| $L_k < S \le D/A_k$ | $M^n_{(k,i)} = R^n_{(k,i)} * (\max(\lceil W^n_i / L_k \rceil, 1))$ |
| Variable stride, fixed data block size | $M^n_{(k,i)} = (\sum_{j=1}^{R^n_{(k,i)}} \lfloor \min((S_{(i,j)} / L_k), 0) \rfloor) * (\max(\lceil W^n_{(i,j)} / L_k \rceil, 1))$ |
| Variable stride $L_k < S \le D/A_k$, variable data block size | $M^n_{(k,i)} = \sum_{j=1}^{R^n_{(k,i)}} (\max(\lceil W^n_{(i,j)} / L_k \rceil, 1))$ |
| $D/A_k < S < D/2$ | $M^n_{(k,i)} = \left\lceil R^n_{(k,i)} * (W^n_i / L_k) \right\rceil$ |