# Performance Modeling and Analysis of Cache Blocking in Sparse Matrix Vector Multiply

*Rajesh Nishtala* *Richard W. Vuduc* *James W. Demmel*
*Katherine A. Yelick*

**Report No. UCB/CSD-04-1335**

Computer Science Division (EECS)
University of California
Berkeley, California 94720

**Abstract**

We consider the problem of building high-performance implementations of sparse matrix-vector multiply (SpM×V), or $y = y + A \cdot x$, which is an important and ubiquitous computational kernel. Prior work indicates that cache blocking of SpM×V is extremely important for some matrix and machine combinations, with speedups as high as 3x. In this paper we present a new, more compact data structure for cache blocking for SpM×V and look at the general question of when and why performance improves. Cache blocking appears to be most effective when simultaneously 1) the vector $x$ does not fit in cache 2) the vector $y$ fits in cache 3) the non zeros are distributed throughout the matrix and 4) the non zero density is sufficiently high. In particular we find that cache blocking does not help with band matrices no matter how large $x$ and $y$ are since the matrix structure already lends itself to the optimal access pattern.

Prior work on performance modeling assumed that the matrices were small enough so that $x$ and $y$ fit in the cache. However when this is not the case, the optimal block sizes picked by these models may have poor performance motivating us to update these performance models. In contrast, the optimum block sizes predicted by the new performance models generally match the measured optimum block sizes and therefore the models can be used as a basis for a heuristic to pick the block size.

We conclude with architectural suggestions that would make processor and memory systems more amenable to SpM×V.

# Contents

# 1 Introduction

We consider the problem of building high-performance implementations of sparse matrix-vector multiply (SpM×V), or $y = y + A \cdot x$, which is an important and ubiquitous computational kernel. We call $x$ the *source vector* and $y$ the *destination vector*. Making SpM×V fast is complicated both by modern hardware architectures and by the overhead of manipulating sparse data structures. It is not unusual to see SpM×V run at under 10% of the peak floating point performance of a single processor.

In hardware, the oft-cited performance gap between processor and memory drives the need to exploit locality and the memory hierarchy. Designing locality-aware data structures and algorithms can be a daunting and time-consuming task, because the best implementation will vary from processor to processor, from compiler to compiler, and from matrix to matrix. This need to have a different data structure for each sparse matrix is a major distinction from the problem of tuning dense matrix kernels (dense BLAS), since the information about matrix structure is not available until run-time.

In prior work on the Sparsity system (Version 1.0), [13, 12], Im developed an algorithm generator and search strategy for SpM×V that was quite effective in practice. The Sparsity generators employed a variety of performance optimization techniques, including *register blocking*, *cache blocking*, and multiplication by *multiple vectors*. In this paper, we focus on cache blocking (Section 2) and ask the fundamental questions of what limits exist on such performance tuning, and how close tuned code gets to these limits.

The main difference between cache blocking and register blocking is that register blocking decreases the size of the data structure in order to decrease the overall memory traffic whereas cache blocking reorders memory accesses to increase temporal locality. Both increase the complexity of their data structures used to represent the matrix. Register blocking explicitly fills the matrix with zeros to decrease the number of indices, while cache blocking adds an extra set of row pointers for each block. As we shall see the fundamental trade off we need to make is whether the benefit of the added temporal locality outweighs the costs associated with the added accesses to the data structures.

In related work on dense matrices, cache and memory behavior have been well-studied. A variety of sophisticated static models have been developed, each with the goal of providing a compiler with sufficiently precise models for selecting memory hierarchy transformations and parameters such as tile sizes [5, 7, 15, 4, 24]. However, it is difficult to apply these analyses directly to sparse matrix kernels due to the presence of indirect and irregular memory access patterns.

Despite the difficulty of analysis in the sparse case, there have been a number of notable attempts. Temam and Jalby [20], Heras, *et al.* [11], and Fraguela, *et al.* [6] have developed sophisticated probabilistic cache miss models, but assume uniform distribution of non-zero entries. These models are primarily distinguished from one another by their ability to account for self- and cross-interference misses.

Gropp, *et al.*, use bounds similar to the ones we develop to analyze and tune a computational fluid dynamics code [9]; Heber, *et al.*, present a detailed performance study of a fracture mechanics code on Itanium [10]. However, we are interested in tuning for matrices that come from a variety of domains. Furthermore, we explicitly model execution time (instead of just modeling misses) in order to evaluate the extent to which our tuned implementations achieve optimal performance.

Finally, we mention examples of work in the sparse compiler literature by Bik [1], Pugh and Shpeisman [16], and the Bernoulli compiler [19]. The first analyzes matrices for high-level structure using techniques complementary those that we consider; the latter two consider sparse code specification and generation issues, but do not specialize for specific matrix structures.

First we present two new optimizations for cache blocking in Section 2 and show when they help performance. Next, we develop upper and lower bounds on the execution rate (in Mflop/s) of SpM×V, based on the nonzero pattern in the matrix and the cost of basic memory operations, such as cache hits and misses in Section 3. The two bounds differ only in their assumption about whether conflict misses occur: in the upper bound any value that has been used before is modeled as a cache hit (no conflict misses), whereas the lower bound assumes that all data must be reloaded. These models extend our prior work [22, 21] by accounting for the TLB, enabling accurate selection of optimal *cache block* sizes. We then use detailed hardware counter data collected on 3 different computing platforms (Table 1) over a test set of 14 sparse matrices (Table 2) to validate our models in Section 5. We first analyze the new models across the three platforms and analyze why and where the models breaks down. We then analyze a set of randomly generated matrices, to get more intuition into why and when cache blocking helps in Section 6. Section 7 analyzes when the parameters of the matrix lend themselves to cache blocking in the context of these models. We close with conclusions and future work of this project in Section 8.

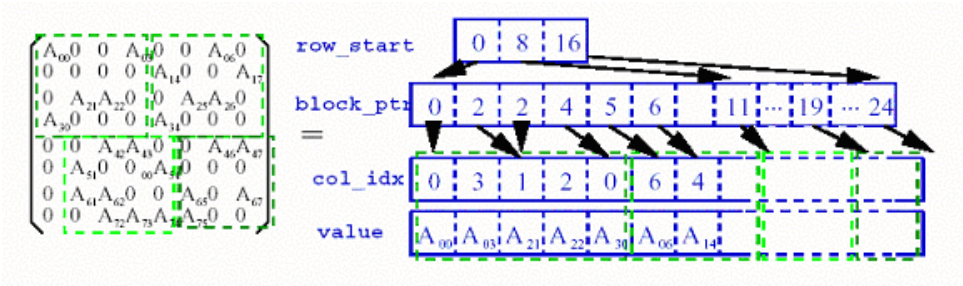



Figure 1: **Cache Blocking Data Structures**. [12]

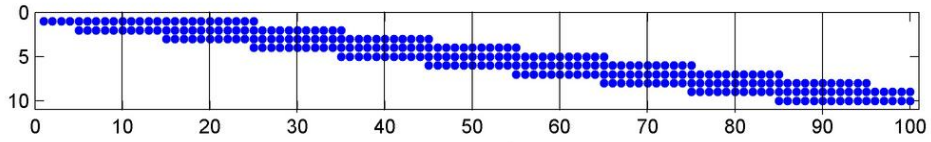



Figure 2: **Band Matrix Example**. Here we notice that when there is no row blocking and the column block size is 10 all the blocks have mostly zero rows thus motivating the need for RSE

# 2 Cache Blocking

## 2.1 Types of Matrices

One of the main goals of this paper is to classify the set of matrices on which cache blocking has any kind of benefit. Our conclusions will be that cache blocking appears to be most effective when simultaneously 1) the vector $x$ does not fit in cache 2) the vector $y$ fits in cache and 3) the non zeros are distributed throughout the matrix. In particular all the test matrices in (Table 2) (except the dense one) are sparse enough (i.e. the density is less than $10^{-3}$) so that register blocking has no significant effect.

## 2.2 Basic Data Structure

The original matrix is represented in compressed sparse row format (CSR)[2]. In CSR, the nonzero elements of each row and their column indices are stored sequentially in memory (in arrays called value and col_idx in Figure 1) with another indexing array (called block_ptr in Figure 1) to identify where the new rows start. In essence cache blocking breaks the CSR matrix into multiple smaller $r_{cache}$ x $c_{cache}$ CSR matrices and then stores these sequentially in memory. In addition a new array (called row_start) is created which points to the start of each row of cache blocks. A diagram of this is shown in Figure 1. The fact that each of the cache blocks is a smaller matrix is in an important observation that is exploited by a software engineering optimization presented in Section 2.3.2. This optimization also allows easy recursion with multiple levels of cache blocking. One of the main goals of this work is to choose the size of these $r_{cache}$ x $c_{cache}$ blocks so that we get the most reuse out of caching of the source and destination vectors when performing SpM×V. We assume that the size of a double is twice that of an integer.

## 2.3 New Optimizations

### 2.3.1 Row Start / End (RSE)

When matrices (especially band matrices) are blocked it is possible that within a cache block non-zeros do not exist on all the rows of that cache block (see Figure 2). The first cache block, for example, might have only non zero elements in the first tenth of the rows and have the rest of the cache block be empty. However the basic cache blocked data structure of Section 2.2 would loop over all zero rows without doing any work. In order to avoid the unnecessary work, a new vector that contains row start(RS) and row end (RE) information for each cache block is also created to point to the first and last nonzero rows in the cache block. This new

```
void bsmvm_1x1_1 (int start_row, int end_row,
               int *row_start, int *col_idx, double *value,
               double *src, double *dest)
{
  int i, j;

1  for (i=start_row; i<=end_row; i++)
  {
2    register double d0;
3    d0 = dest[1*i+0];
4    for (j=row_start[i]; j<row_start[i+1]; j++)
    {
5      d0 += value[j*1+0] * src[col_idx[j]+0];
    }
6    dest[1*i+0] = d0;
  }
}
```

```
void sparse_block_smvp_funcall (int r, int c, int m, int n,
                           int *row_start, int *sparse_ptr, int *row_se,
                           double *val, int *col_idx,
                           double *src, double *dest,
                           void (*rse_sparsity_routine)(int start_row, int end_row, int bm,
                                        int *row_start, int *col_idx,
                                        double *value, double *src,
                                        double *dest))
{
  int i, j;
  int b_i, b_j;
  int b_m, b_n;
  int b_col =0;
  int end_r;

1  b_m = (m+r-1)/r;
2  b_n = (n+c-1)/c;
3  for (b_i=0; b_i<b_m; b_i++){

4    end_r = ((b_i*r+r) < m) ? r : m - b_i*r ;
5    b_col = 0;
6    for (b_j=row_start[b_i]; b_j<row_start[b_i+1]; b_j+=end_r, b_col++)\{
7      register int a = row_se[b_i*2*b_n+2*b_col];
8      register int b = row_se[b_i*2*b_n+2*b_col+1];

9      (*rse_sparseity_routine)(a, b, sparse_ptr+b_j, col_idx,
10               val, src,
11               dest+(b_i*r));
    }
  }
```

Figure 3: **Code to Perform SpM×V**. The top code fragment shows the multiplication of a CSR matrix with the modifications for the row start row end optimizations. The bottom code fragment shows the outer loops from which the top fragment is called. The bottom code fragment iterates over the cache blocks while the top code fragment performs the multiplications inside a given cache block.

indexing information makes the performance less sensitive to the size of the cache block. The top of Figure 3 shows the code that implements this. Performance results have shown that this optimization can only help improve performance as shown later in Section 6 and further illustrated in Appendix B and Appendix M.

### 2.3.2 Exploiting Cache Block Structure

As described in Section 2.2, the cache blocked matrix can be thought of as many smaller sparse matrices stored sequentially in memory. We can exploit this by calling our prior sparse matrix vector multiplication routines on each smaller matrix, passing the appropriate part of the source and destination vectors as arguments. The advantage of handling the multiplication in this fashion is that the inner loops can be generated independently of the code for cache blocking and code previously written for non-cache blocked implementations can be reused. Tests indicate that the function call overhead is negligible since the number of cache blocks for a matrix is usually small compared to the total memory operations. The bottom of Figure 3 shows the actual C functions that implement these routines.

# 3 Analytic Models

In this section we create analytic upper and lower bounds on performance by modeling various levels of the memory hierarchy. After describing these models we evaluate them on the three platforms described in Table 1. We first describe the overall performance model. We then model the different parts of the memory system that contribute to this overall model. We first create a load model and then discuss analytic upper and lower bounds for the number of cache misses at every level. We then examine the upper and lower bounds for TLB misses and a more complex relation between these upper and lower bounds to yield a more accurate estimate of the actual number of TLB misses.

## 3.1 Overall Performance Model

The overall performance model is similar to the one in [21] except that we have added one more latency term to account for the TLB misses.

We model execution time as follows. First, since we want an upper bound on performance (lower bound on time), we assume we can overlap the latencies of computation and memory accesses. Let $h_i$ be the number of hits at cache level $i$, and $m_i$ be the number of misses. Then the execution time $T$ is

$$T = \sum_{i=1}^{\kappa-1} h_i \alpha_i + m_\kappa \alpha_{\text{mem}} + m_{TLB} \alpha_{\text{TLB}}, \tag{1}$$

where $\alpha_i$ is the access time (in cycles or seconds) at cache level $i$, $\kappa$ is the lowest level of cache, and $\alpha_{\text{mem}}$ is the memory access time. The L1 hits $h_1$ are given by $h_1 = \text{Loads}(r,c) - m_1$ where $\text{Loads}(r,c)$ is the number of loads with an $r_{cache} \times c_{cache}$ cache block size (see Section 3.2 below). Assuming a perfect nesting of the caches, so that a miss at level $i$ is an access at level $i+1$, then $h_{i+1} = m_i - m_{i+1}$ for $i \geq 1$. The TLB and the L3 might not be nested, however we account for this by assuming that the TLB misses are not overlapped with the misses at the other levels and that they must be serviced before the cache misses can be serviced. The performance expressed as Mflop/s is $\frac{2k}{T} \cdot 10^{-6}$ because each of the $k$ nonzero matrix entries leads to one floating point multiply and one floating point add.

To get an estimate of the *upper bound on performance*, let $m_i = M^{(i)}_{\text{lower}}$ in Equation (1) (where $M^{(i)}_{\text{lower}}$ is a lower bound on misses at the $i^{th}$ cache level as discussed below), and convert to Mflop/s. Similarly, we can get a lower bound on performance by letting $m_i = M^{(i)}_{\text{upper}}$(where $M^{(i)}_{\text{upper}}$ is a upper bound on misses at the $i^{th}$ cache level as discussed below).

In order to take the TLB effects into account we estimate the number of cycles that are needed to process a TLB miss in order to make Equation (1) match the measured performance. We incorporate it into the upper bound model by setting $m_{TLB}$ equal to $M^{(TLB)}_{\text{model}}$. This is further described in Section 3.4. Our estimated values for $\alpha_{TLB}$ are show in Table 1.

## 3.2 Load Model

We assume the cache block data structure as described in Section 2.2. We can count the number of loads required for SpM×V as follows. Let $A$ be an $m \times n$ matrix with $k$ non-zeros. Henceforth we assume no register blocking is done which is optimal for all our sparse test matrices. We define a new variable, $K_{rc}$, to equal the number of cache blocks that a given cache block size $(r \times c)$ produced. Every matrix entry must be loaded once. The number of accesses to the source vector is exactly $k$. The number of accesses to the destination depends on the cache block size. For each cache block $i$, we must load the destination vector $\delta_i = (\text{RE}_\text{i} - \text{RS}_\text{i}) + 1$ times. The variables $\text{RS}_\text{i}$ and $\text{RE}_\text{i}$ indicate the first and last row respectively on which non-zero elements can be found for the $(i)^{th}$ cache block as defined in Section 2.3.1. In all the cases except for the band matrices these were found to be the first and last rows of the cache block respectively. We must load each of the block_ptr elements twice: once when the value is being used as a pointer to the end of a row and then when it is used as the start of a row. The loads can be counted in the following manner:

$$\text{Loads}(r,c) = \underbrace{2k + 2\sum_{i=1}^{K_{rc}} \delta_i + 2(K_{rc}) + 2\left\lceil \frac{m}{r} \right\rceil}_{\text{matrix}} + \underbrace{\sum_{i=1}^{K_{rc}} \delta_i}_{\text{dest vector}} + \underbrace{k}_{src\ vector} \tag{2}$$

The fewest number of loads would occur if the matrix were not cache blocked; in this case $\sum_{i=1}^{K_{rc}} \delta_i$ equals $m$ and $K_{rc}$ equals 1. Therefore cache blocking doesn't decrease the number of loads. If anything, too many cache blocks would greatly increase the overhead.

### 3.3 Cache Miss Model

Here we develop upper and lower bounds on the number of cache misses, which lead to lower and upper bounds on performance in MFlops, respectively.

We start with the L1 cache. Let $l_1$ be the L1-cache line size, in integers. In order to estimate the minimum number of cache misses that can occur we take the total amount of data that we access and divide by the line size. This will give us the total number of lines the matrix, source, and destination vectors would take assuming all the data was perfectly aligned.

Thus, a lower bound $M^{(1)}_{\text{lower}}$ on L1 misses is

$$M^{(1)}_{\text{lower}}(r,c) = \frac{1}{l_1}\left[\underbrace{2m}_{dest\ vector} + \underbrace{2n}_{src\ vector} + \underbrace{2k + k + \sum_{i=1}^{K_{rc}} \delta_i + \left\lceil \frac{m}{r} \right\rceil + 2(K_{rc})}_{matrix}\right] \tag{3}$$

In order to find the lower bounds for another level of the cache simply replace $l_1$ with the appropriate line size. In order to find the upper bound we still assume that every entry in the matrix is loaded once as in the lower bound, but we assume that every access to the source and every access to the destination vectors miss because of conflict and capacity misses.

Thus, an upper bound $M^{(1)}_{\text{upper}}$ on L1 misses is

$$M^{(1)}_{\text{upper}}(r,c) = \underbrace{k}_{src\ vector} + \underbrace{\sum_{i=1}^{K_{rc}} \delta_i}_{dest\ vector} + \frac{1}{l_1}\underbrace{\left[2k + k + \sum_{i=1}^{K_{rc}} \delta_i + \left\lceil \frac{m}{r} \right\rceil + 2(K_{rc})\right]}_{matrix} \tag{4}$$

The first $k$ indicates that we miss for every access to the source vector. The second term $\sum_{i=1}^{K_{rc}} \delta_i$ is the number of times that we access the destination vector. Since we stream through the matrix entries and access each element once the number of misses does not depend on conflict or capacity. Notice that neither the load model of Section 3.2 nor the cache miss model of this section predict the advantages of cache blocking since they only show an increase in data structure overhead.

### 3.4 TLB Miss Model

According to our simple load and cache miss models, cache blocking has no benefit. It turns out that the main benefit of cache blocking is increased temporal locality in the source vector as reflected in the number of TLB misses, which we model here. Experimental data in Section 5.3.2 do in fact show improvements in cache misses too, though this is not captured by our model. Still, the model will turn out to be adequate for predicting good cache block sizes. In order to estimate the lower bounds on the TLB misses we simply take the total size of the data that we access and divide that by the page size. This will give the minimum number of pages that the data resides in and the minimum number of compulsory misses for the TLB. We assume that the pages are of size $p$ integers. Thus, a lower bound, $M^{(TLB)}_{\text{lower}}$, on TLB misses is

$$M^{(TLB)}_{\text{lower}}(r,c) = \frac{1}{p}\left[\underbrace{2m}_{dest\ vector} + \underbrace{2n}_{src\ vector} + \underbrace{2k + k + \sum_{i=1}^{K_{rc}} \delta_i + \left\lceil \frac{m}{r} \right\rceil + 2(K_{rc})}_{matrix}\right] \tag{5}$$

Notice that the equation is identical to the cache miss model except we replace the line size with the page size. It was found that $M^{(TLB)}_{\text{lower}}$ was at least 1000 on the Itanium 2, the only platform on which we have hardware counters for the number of TLB misses.

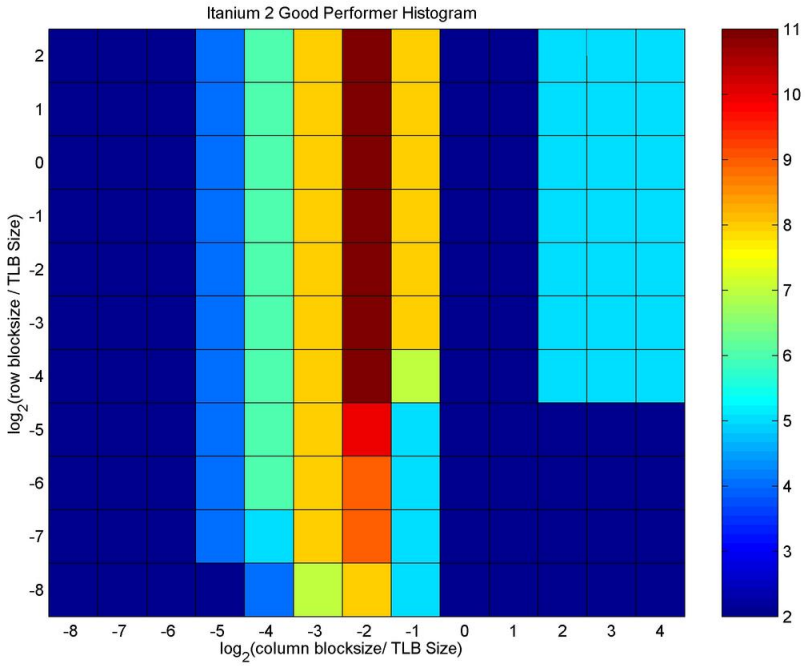



Figure 4: **Good Performer Histogram for Itanium 2**. For each row and column block size shown above, the value in the cell contains the number of matrices whose performance was within 90% of peak if that block size was chosen. Notice that there are two distinct categories for the Itanium 2: (1) when the column blocksize is 65536 (or $\frac{1}{4}$th of the TLB) (Matrices 2–11) and (2) when no blocking is done (Matrices 12–14). TLB size is the number of double-words mappable by the TLB. Thus the column labeled $log_2(columnblocksize/TLBSize) = 0$ means a vector of length $c_{cache}$ exactly fill up the TLB, the row labeled $log_2(rowblocksize/TLBsize) = -2$ means a vector of length $r_{cache}$ fills up $2^{-2} = \frac{1}{4}$the of the TLB, and so on.

The upper bound is also similar to the cache miss equation. For an estimate of the upper bound we assume that we load every matrix page once. We then assume that we take a TLB miss on every access to the source vector and destination vector. Thus the total number of pages that we load is as follows:

$$M_{\text{upper}}^{(TLB)}(r,c) = \underbrace{k}_{src\ vector} + \underbrace{\sum_{i=1}^{K_{rc}} \delta_i}_{dest\ vector} + \frac{1}{p}\left[\underbrace{2k + k + \sum_{i=1}^{K_{rc}} \delta_i + \left\lceil \frac{m}{r} \right\rceil + 2(K_{rc})}_{matrix}\right] \tag{6}$$

Note that this again a very pessimistic model.

Modeling performance based merely on the lower and upper bound models does not take the increased locality of cache blocking into account because the lower bound on cache misses (which is used to calculate the upper bound on performance) only counts the compulsory misses. Since blocking adds overhead to the data storage, the least amount of overhead occurs when there is no blocking. To factor this in, we need to be able to model at least the most important level of the memory system more accurately to expose the advantages of locality. From Appendix F, we notice many of the matrices have a noticeable increase in the number of TLB misses when the source vector occupies a large fraction of the TLB. Because the number of TLB misses is orders of magnitude higher when the incorrect block size is chosen[1], we chose to try to more accurately estimate the number of TLB misses through a combination of the lower and upper bound models in Equation (5) and Equation (6) respectively.

From Figure 4 we see that there are two distinct categories of block sizes that worked on our matrix suite for the Itanium 2. The first category of matrices (Matrices 2–11) showed the best performance when the

[1]This is probably due to early eviction of the source vector with the LRU page replacement policies

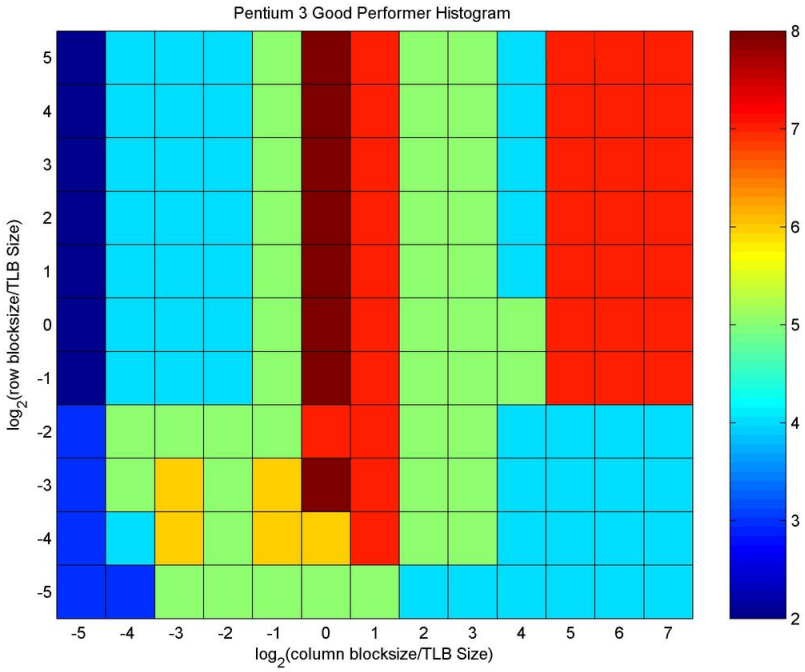



Figure 5: **Good Performer Histogram for Pentium 3**. This plot is similar to Figure 4 for the Pentium 3. Notice that there are two distinct categories for the Pentium 2: (1) when the column block size is equal to 32k or 64k (one half and the full TLB respectively) or (2) when no blocking is done.

column block size equaled $\frac{1}{4}$th of the TLB. In the second category of matrices(Matrices 12–14) the added overhead of blocking hurt performance so the performance was best when the column block size exceeded the number of columns in the matrix (i.e. there was no blocking in the column direction). We also notice that the performance does not depend heavily on the row block size once it is large enough and thus we conclude that no blocking should be done in the row dimension.

From Figure 5 and Figure 6 we see that the same dichotomy of matrices in the Itanium 2 exists on other platforms. The only difference is the number of matrices that fall into each category. A further analysis shows that on all three platforms blocking Matrices 2–11 and not blocking Matrices 12–14 yielded performance that was within 90% of peak. The major difference between these two classes of matrices was their density. Matrices 12–14 are orders of magnitude sparser than Matrices 2–11 as seen by the densities in Table 2. This implies that when the amount of reuse is small, blocking does not help.

In order to capture this behavior in our performance model we present a modified version of the TLB miss model that combines both the upper bound and lower bound to create a reasonable estimate of the number of misses. One of the main aims for the performance model is to expose the penalty when there is not enough temporal locality in accessing the source vector. To account for this our TLB miss model switches to using the upper bound model as an estimate for the number of misses when the column block size is too large[2]. Since the optimal block size as a percentage of the TLB size changes from machine to machine, there will be a different TLB model for each platform. TLB counter data was only available for the Itanium 2, thus we present the model for that platform only. The models for the other platforms will be similar.

$$M^{(TLB)}_{\text{model}}(r,c) = \begin{cases} M^{(TLB)}_{\text{upper}}(r,c) \times min(\frac{\frac{c\times 2}{p}}{\text{TLBEnt}}, 1), & \text{if } \frac{c\times 2}{p} \geq \frac{\text{TLBEnt}}{2} \text{ and } \frac{k}{nnzcols} > 4 \quad (7a) \\ M^{(TLB)}_{\text{lower}}(r,c) & \text{else} \quad (7b) \end{cases}$$

Equation (7) shows the model used to calculate the approximate number of TLB misses for the Itanium 2. The variables are as follows: $p$ is the page size in integers, TLBEnt is the number of TLB entries in

[2]the actual definition of too large varies across different platforms, for the Itanium 2 we set it at $\frac{1}{2}$ of the TLB

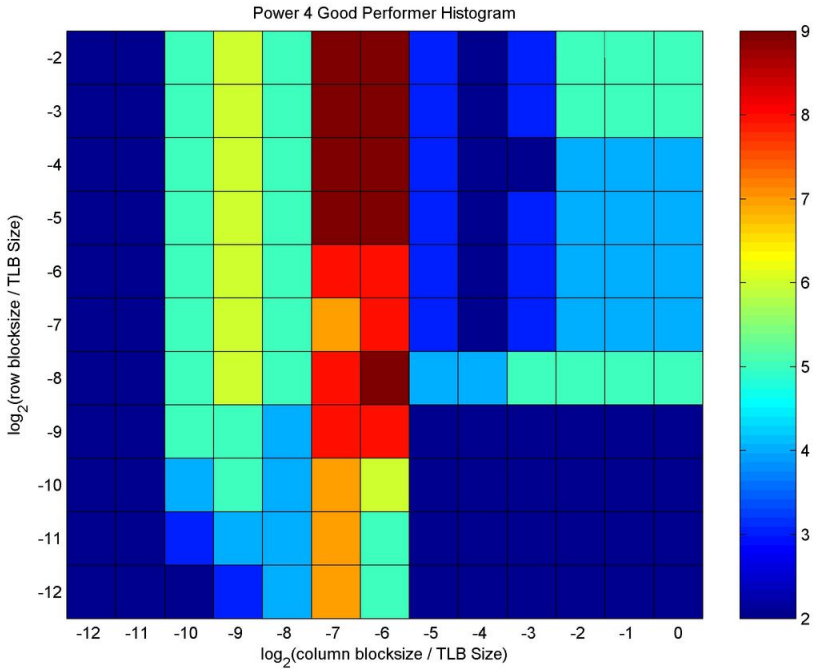



Figure 6: **Good Performer Histogram for Power 4**. This plot is similar to Figure 4 for the Power 4. Notice that there are two distinct categories for the Pentium 2: (1) when the column block size is equal to 32k or 64k ($\frac{1}{128}$th and $\frac{1}{64}$th of the TLB respectively) or (2) when no blocking is done.

the TLB, TLBSz is the number of TLB entries times the page size, and *nnzcols* is the number of non-zero columns. According to our empirical data for the Itanium 2, for Matrices 2–11 the optimal column block size is $\frac{1}{4}$th of the TLB, thus when the column block size is $\frac{1}{2}$ of the TLB we switch to using the upper bound model. The upper bound is scaled by the percentage of the source vector that overflows the TLB. This switch is only performed when the matrix is dense enough ($\frac{k}{nnzcols} > 4$) ensuring us that blocking provides enough reuse. If either of these conditions fail we use the lower bound model on TLB misses. Appendix F shows that when this model is applied to the Itanium 2, it does a good job of predicting the noticeable jump in the number of TLB misses which is at least good enough to predict good cache block sizes. Future work hopes to refine this model further and verify it for the other platforms. The effect of this model can be seen in Appendix B. The peaks of the upper bound performance model correlate better to the peaks of the actual performance in most of the matrices. Without this model the peaks of the upper bound model would show guess that blocking is not a good idea, which is obviously not the case. We will evaluate the models further in Section 5.

# 4 Experimental Setup

## 4.1 Platforms

The experiments were conducted on the platforms shown in Table 1. This table summarizes the various parameters of the processor and the underlying memory system. It also includes the compiler flags that were used to perform the experiments. The cache and memory latencies were derived from published processor manuals, experimental work using the Saavedra-Barrera memory system microbenchmark[17] and MAPS benchmarks [18]. The TLB latencies were found through curve fitting the performance model with the measured performance and are thus rough estimates of the actual latency. The TLB latency is dependent on the memory access pattern because of complicated hardware structures to translate virtual addresses. Therefore a single number representing all the possible access patterns would be difficult to measure. Notice on the Power 4 and the Pentium 3 that with the default page size, the size of amount of memory that can

be mapped by the TLB is a fraction of the size of the largest level of caches. This means that even if the data is in the cache, the cost of a TLB miss is still incurred.

## 4.2 Matrices

We evaluate SpM×V on a set of matrices that are large enough and sparse enough for cache blocking to have a significant effect. The properties of the 14 matrices that were chosen are referenced in Table 2. The spy plots of these matrices are located in the appendix of this paper (Appendix L). On the Pentium 3 and the Power 4, the operating system did not allocate us enough memory to create Matrix 8 so the data set includes only the other 13 matrices on these two platforms.

## 4.3 Timing

We use the PAPI v2.1 library for access to hardware counters on the Itanium 2 and the Pentium 3 [3]; we use the cycle counters as timers on the Power 4 since PAPI's implementation is not complete on this machine. However, the TLB counters were only available for the Itanium 2. By default, PAPI only increments the counters during user level operation, thus kernel crossings are not counted. On the Itanium 2 and the Power 4 the TLB miss is serviced in hardware so the cycle counters reflect the times taken to service these misses. On the Pentium 3 the miss is serviced in software, so the times do not necessarily take the time to process system calls and interrupts (such as the TLB misses) into account. When the PAPI cycle counters were compared with wall clock timers the results were similar, indicating that this discrepancy did not adversely effect the measurements. The counter values reported are the median of 10 consecutive trials. The standard deviation for all the runs across all the platforms was under 5%. Each of the runs was performed with a warm-cache.

| Property | Intel Pentium III | Intel Itanium II | IBM Power 4 |
|---|---|---|---|
| Clock Rate | 500 MHz | 900 MHz | 1.3 GHz |
| Peak Main Memory Bandwidth | 680 MB/s | 6.4 GB/s | 8 GB/s |
| Peak Flop Rate | 500 Mflop/s | 3.6 Gflop/s | 5.2 Gflop/s |
| DGEMM ($n = 1000$) | 330 Mflop/s | 3.5 Gflop/s | 3.5 Gflop/s |
| DGEMV ($n = 1000$) | 58 Mflop/s | 740 Mflop/s | 915 Mflop/s |
| STREAM Triad Bandwidth [14] | 350 MB/s | 3.8 Gflop/s | 2.1 Gflop/s |
| L1 data cache size | 16 KB | 32 KB | 32 KB |
| L1 line size | 32 B | 128 B | 128 B |
| L1 latency | 1 cy | 0.34 cy | 0.7 cy |
| L2 cache size | 512 KB | 256 KB | 1.5 MB |
| L2 line size | 32 B | 128 B | 128 B |
| L2 latency | 18 cy | 0.5 cy | 12 cy |
| L3 cache size | N/A | 1.5 MB | 16 MB |
| L3 line size | | 128 B | 512 B |
| L3 latency | | 3 cy | 45 cy |
| TLB entries | 64 | 32 (L1 TLB)<br>128 (L2 TLB) | 1024 |
| Page size | 4 KB | 16 KB | 4 KB |
| TLB entries × Page size | 256KB | 512KB(L1)<br>2MB (L2) | 4MB |
| TLB latency (est.) | 10 cy | 5 cy | 16 cy |
| Minimum Memory latency ($\approx$) | 25 cy | 11 cy | 60 cy |
| Maximum memory latency ($\approx$) | 60 cy | 60 cy | 10000 cy |
| sizeof(double) | 8 B | 8 B | 8 B |
| sizeof(int) | 4 B | 4 B | 4 B |
| Compiler | Intel C v5.0 | Intel C v7.0 | IBM XLC v6.0 |
| Flags | `-O3`<br>`-xk`<br>`-tpp6`<br>`-unroll` | `-O3` | `-O5, -qhot`<br>`-qalias=allp`<br>`-qcache=auto`<br>`-qarch=pwr4`<br>`-qtune=pwr4`<br>`-qnoipa` |

Table 1: **Evaluation Platforms**. This table summarizes the various platforms used in the experiments. For the BLAS routines (DGEMM, DGEMV, DSYMV, DSPMV, and DSYMM), we show the best performance between hand-tuned and automatically tuned libraries (Intel Math Kernel Library v5.2, Gotos BLAS library [8] , and ATLAS 3.4.1 [23] resp.). The leading dimension (LDA) is set to be equal to the matrix dimension. This table was provided by Ben Lee and Rich Vuduc (http://bebop.cs.berkeley.edu)

| | Application Area | Dimension | Nonzeros | Density | Avg.Number of Nonzeros per Row | Correlation |
|---|---|---|---|---|---|---|
| 1 | Dense Matrix | 2000 x 2000 | 4000000 | 1.00 | 2000 | 0 |
| 2 | Statistical Experimental Design | 231 x 319770 | 8953560 | 1.21e-1 | 38760 | 3.27e-1 |
| 3 | Linear programming (LP) | 52260 x 379350 | 1567800 | 7.91e-5 | 30 | 2.20e-1 |
| 4 | LP | 10280 x 243246 | 1408073 | 5.63e-4 | 137 | 3.74e-2 |
| 5 | Latent Semantic Indexing | 10000 x 255943 | 3712489 | 1.45e-3 | 371 | 2.48e-3 |
| 6 | column wise expansion of LSI | 10000 x 2559430 | 3712489 | 1.45e-4 | 371 | 2.48e-3 |
| 7 | row wise expansion of LSI | 100000 x 255943 | 3712489 | 1.45e-4 | 37.1 | 2.48e-3 |
| 8 | row wise stamping of LSI | 100000 x 255943 | 37124890 | 1.45e-3 | 371 | 2.6e-5 |
| 9 | Queuing model of mutual exclusion | 65535 x 65535 | 1114079 | 2.59e-4 | 17 | 8.72e-1 |
| 10 | Italian Railways scheduling (LP) | 4284 x 1092610 | 11279748 | 2.41e-3 | 2633 | 5.73e-2 |
| 11 | Italian Railways scheduling (LP) | 4284 x 546305 | 5661231 | 2.42e-3 | 1321 | 1.57e-1 |
| 12 | Web connectivity graph (WG) | 1000005 x 1000005 | 3105536 | 3.11e-6 | 3.10 | 9.03e-1 |
| 13 | WG after MMD reordering | 1000005 x 1000005 | 3105536 | 3.11e-6 | 3.10 | 7.62e-1 |
| 14 | WG after RCM reordering | 1000005 x 1000005 | 3105536 | 3.11e-6 | 3.10 | 8.62e-1 |

Table 2: **Matrix Benchmark Suite**. Matrices are listed in alphabetical order. Note that matrices 6, 7, and 8 are just modified versions of matrix 5. As we will see in Section 5, out of all the original matrices(Matrices 1–5 and Matrices 9–14) matrix 5 proved to have the most significant improvement. Therefore to understand this matrix better we manipulated the matrix into the different forms and found that a modified version of matrix 5 (matrix 6) showed the most significant speedup. Matrix 6 is identical to matrix 5 except that the column indices are multiplied by 10. Matrix 7 is the same as matrix 6 except that the row indices are multiplied by 10. Matrix 8 was simply where 5 was stacked 10 times along the row dimension. All the matrices (except matrix 1) have the property that their source vector is much larger than the largest level of cache on the platforms that the matrices were evaluated on. The density is the total number of non zeros divided by the total number of matrix entries. The average number of non zeros per row is the total number of non zeros divided by the number of rows. The last column entitled *correlation* is the statistical correlation between the row and the column indices of the nonzeros in the matrix. If the matrix is dense or the non zeros are uniformly distributed throughout the matrix the correlation is 0. The more the nonzeros get clustered around the diagonal, the closer the correlation is to 1. For the band matrices used in Section 6, the coefficient is less than 0.96 for all the band matrices with a band-width of less than 65536 elements.

# 5 Evaluations of Models

The models shown in Section 3.1 take into account misses at the various levels of the memory hierarchy in order to predict performance. The values for the miss counts can come from three different sources: the lower bound at each level of the memory hierarchy (which should give an upper bound to performance), the actual measured misses from the hardware counters, and lastly the upper bound on the misses (which should yield the lower bound on performance). In this section we will explore how well each of the different models predict the actual performance, as well as the optimal cache block size.

We shall first evaluate the performance model in which we use true hardware counters to predict the performance (henceforth called the PAPI model) and compare it to the model in which we use estimates of lower and upper bound of cache and TLB misses (henceforth termed the analytic model). We shall also evaluate our performance models based on three different metrics. The first is how the well the models predict the absolute performance. The second is how well they predict the best cache block size. The third is how well the models predict a cache block size that yields performance that is within 90% of peak. The third metric is the most important for the purposes of performance tuning. Lastly we will evaluate the miss models on the platforms on which the hardware counters were available.

## 5.1 Evaluation of the PAPI Model

This section explores how well the performance model presented in Section 3.1 predicts the actual performance. We shall evaluate this model with three different metrics. Figure 7 shows a plot of how well the PAPI model predicts performance with respect to the base and the best performance.

### 5.1.1 Metric 1: Comparing Predicted and Actual Performance

The first metric that we will use is to see how well the models predict the actual performance. As seen from Figure 7 there is a lot of room for improvement on the Itanium 2. This implies that there is lot of time spent inside or outside the memory system that is unaccounted for. Our models assume that all the floating point data is cached at every level on the memory system which is not the case on the Itanium 2. To account for this we only factor in the L2 and L3 caches and leave out the time spent in the L1 cache. If we incorporate the L1 misses appropriately, the accuracy of the model will increase. From Appendix B we see that the shape of the PAPI model is roughly the same as the actual performance curves indicating that that if the latencies were fixed the PAPI model would be a good predictor of absolute performance, however they are currently too optimistic for SpM×V.

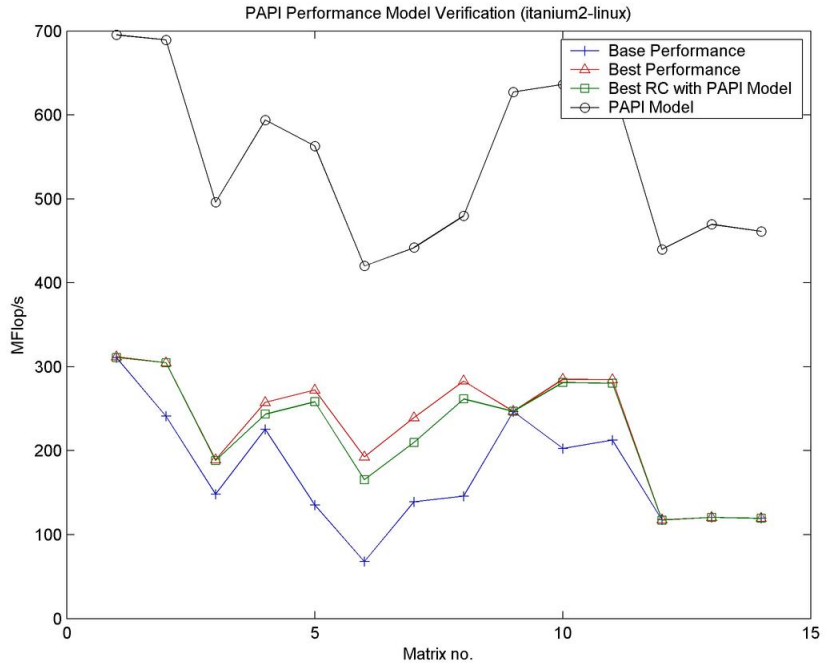
As seen from Figure 7 the PAPI model is a lot closer to predicting the actual performance on the Pentium 3 than on the Itanium 2. From Appendix G we also see a strong correlation between the PAPI model and the measured performance for all block sizes indicating that the latencies chosen for the Pentium 3 are a good fit for SpM×V.

### 5.1.2 Metric 2: Comparing Predicted and Actual Best Cache Block Sizes

The next metric that we will use to evaluate the models is to see how well the PAPI model predicts the best cache block size. Here we see that we do better than before. The cache block sizes chosen by the PAPI model are close to that of the optimum cache block size, however in most cases the PAPI model does not choose the optimum cache block size as seen in Figure 7 and Appendix B. The high accuracy implies that if we were able to plug in the most accurate memory latencies, the PAPI model can predict the performance.

### 5.1.3 Metric 3: Comparing Performance with Predicted Best and Actual Best Cache Block Sizes

The next and perhaps the most interesting metric to validate a performance model is how well the model predicts a good block size (that is a block size that yields 90% of peak performance). Appendix B shows that the block sizes chosen by PAPI for the Itanium 2 are all within 90% of peak, implying that the performance model itself is a good indicator to choose the correct block sizes as long as accurate miss models can be furnished. As seen from Appendix G the PAPI model picks a block size on the Pentium III that is within 90% of peak on 12 out of the 13 matrices chosen. Interestingly enough, the PAPI model chose a block size



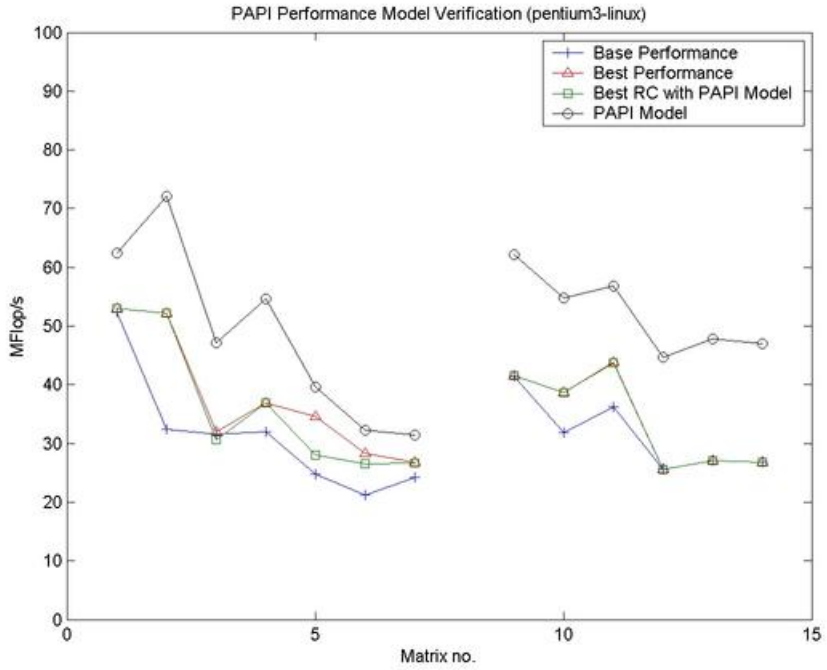




Figure 7: **Itanium2/Pentium 3 PAPI Model Verification**. This plot shows how well the PAPI model performs with respect to the actual performance. The *Base Performance* line shows the performance without cache blocking. The *Best Performance* line shows the best performance across all cache block sizes. The *PAPI Model Mflops* line shows the best predicted performance according to the PAPI model. The *Best RC with PAPI* line shows the actual performance of the cache block size chosen assuming the optimum cache block size was chosen with the PAPI model.

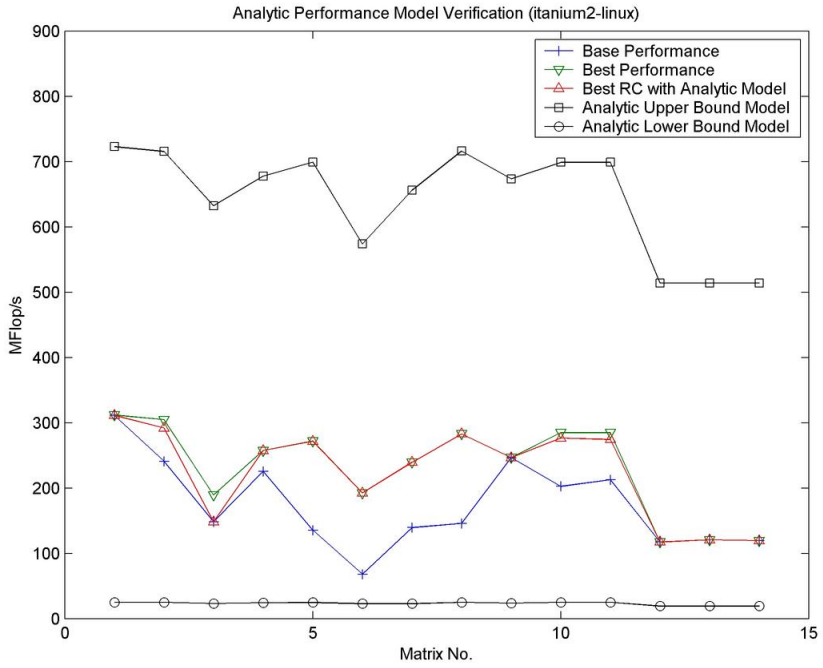



Figure 8: **Itanium 2 Analytic Performance Plot**. This figure shows the analytic model verification on the Itanium 2. The line *Base Performance* shows the performance without cache blocking while the *Best Performance* lines show the performance at the best cache block size. The *Best RC with Analytic Model* shows the performance when the block size is set to be the best block size according to the analytic model. The *Analytic Lower Bound Model* and *Analytic Upper Bound Model* lines show the model's lower and upper bound at the best block size according to the analytic model.

with a low row dimension on the Pentium 3 for the LSI matrix. As we shall see later, the discrepancy came from the cache miss models.

## 5.2 Evaluation of Analytic Model

In this section we will evaluate the performance of the analytic models to the various metrics presented in the sections above. We analyze these models in order to motivate future work on finding a heuristic that can accurately predict the correct block size.

### 5.2.1 Metric 1: Comparing Predicted and Actual Performance

As expected the upper bound performance models do not predict the exact performance very well. They overshoot the actual performance by significant amounts in most cases. This implies that the lower bound cache miss models are too optimistic. The full performance data can be found in Appendices B, G, and K. The points in which the analytical upper bound drops below the actual performance are points in which the TLB model severely over predicted the number of TLB misses leading to a very pessimistic performance prediction.

### 5.2.2 Metric 2: Comparing Predicted and Actual Best Cache Block Sizes

In this section we evaluate how well the block size chosen by the analytic upper bound compares to the best block size. According to Figures 8–10, we do a good job of predicting this performance on the Itanium 2. On 10 out of 14 matrices, the analytic model chooses the cache block size that gives the best performance. As the models indicate, the shapes of the various miss models mirror the actual performance, even if the actual values do not. The TLB miss models as shown above accurately predict the dramatic rises in the

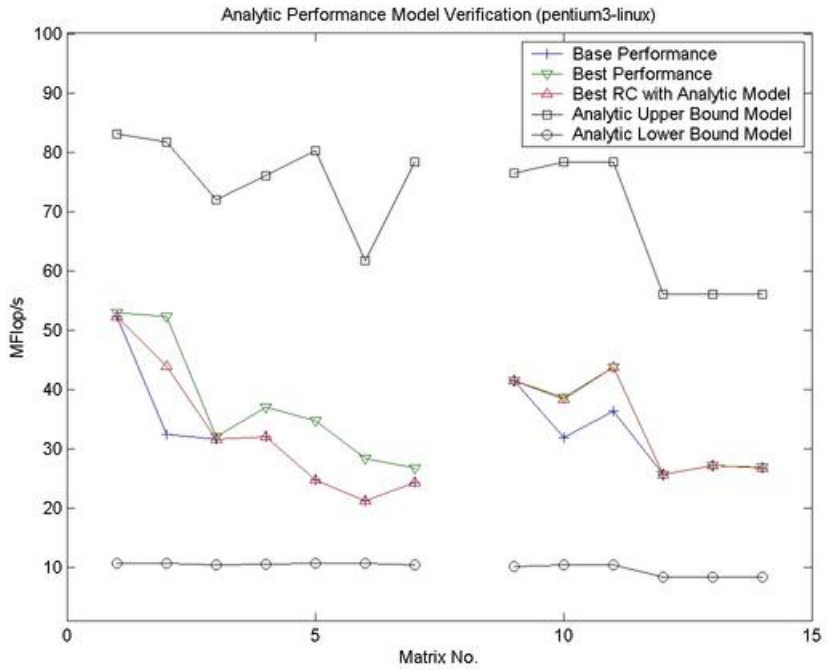



Figure 9: **Pentium 3 Analytic Performance Plot**. This figure shows the analytic model verification on the Pentium 3.

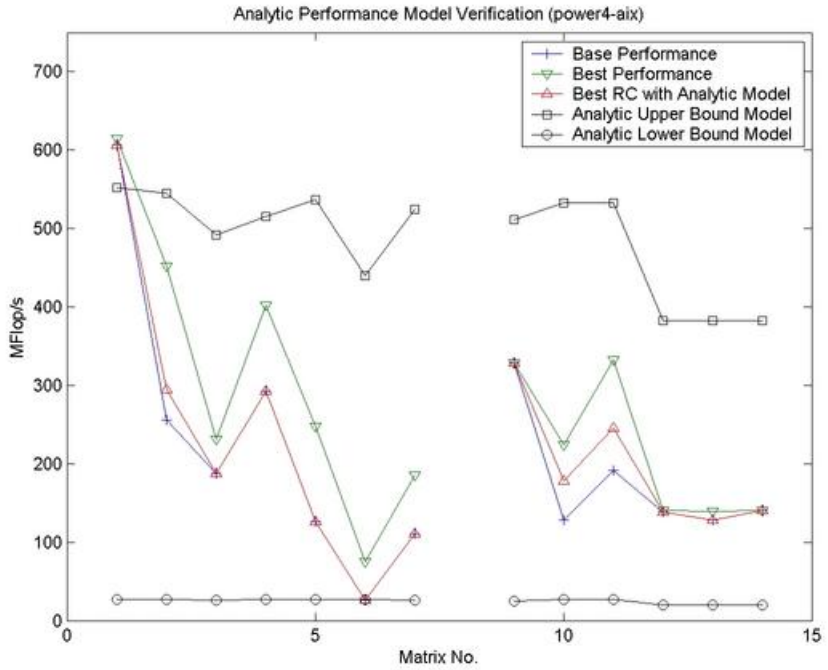



Figure 10: **Power 4 Analytic Performance Plot**. This figure shows the analytic model verification on the Power 4.

TLB misses, giving an accurate estimate of the performance on this machine.

On the Power 4, the model chooses the correct block size only on 4 out of the 13 matrices. This implies that the models on this machine are not accurate, however it is hard to verify this hypothesis without the actual hardware counters.

On the Pentium 3, the analytical model predicts the correct block size on 8 out of the 13 matrices tested. The models on this machine can not be fully verified because it is impossible to verify the TLB miss models without the hardware counters. On this platform it was found that the models predicted that not blocking the matrix (except for Matrices 10 and 11) was optimal, however in the cases that yielded the largest speedups, the analytic models were unable to pick the best block size. This implies that the TLB model, which most influenced the overall performance model, was not good enough further validating the need for platform specific TLB models

#### 5.2.3 Metric 3: Comparing Performance with Predicted Best and Actual Best Cache Block Sizes

On the Itanium 2 we find that the analytical model does really well at picking a block size that yields within 90% of peak performance. On all the matrices except Matrix 3, the analytic model picks a blocksize that is within 90% of peak. As Figure 8 shows, the block size chosen with the analytic model matches the performance of the best block size.

On the Power 4, we choose a good block size on 5 out of the 13 matrices. Again the reasons are that the models on the Power 4 aren't as accurate as on the other platforms. Access to the hardware counters are needed to further analyze this problem. Future work will revisit this platform to collect exhaustive data.

On the Pentium 3 the correct block size was predicted on 10 out of the 13 matrices. However the block sizes that are chosen, except for the rail matrices (matrices 10 and 11), reflect no column blocking. This implies that for any matrix (except matrices 10 and 11) that requires column blocking for good performance, the analytic model does not do a good job of predicting the correct block size. Matrices 10 and 11 show that the analytic model does a good job of picking good performers because the TLB model is able to indicate when the switch can happen. Again the only way to verify the TLB model is to have a hardware counters.

### 5.3 Evaluation of Memory Hierarchy Models

The deviations in the previous section between the actual performance, the PAPI models, and the analytic performance models indicate that our models predict the actual performance, and in some cases, by significant amounts. In this section we will analyze how well we model the different parts of the memory system. Since the PAPI counters were unavailable on the Power 4 we only present performance data from the Itanium 2 and the Pentium 3.

#### 5.3.1 Evaluation of the Load Model

Figure 11 shows the number of load instructions issued in the machine to perform the SpM×V. Appendices C and H show the measured and modeled for for all the block sizes on all the matrices for the Itanium 2 and the Pentium 3 respectively. The data indicate that our models always undercount the number of loads by 20-30%. This is probably due to register spilling which is unaccounted for. In all the cases the Itanium 2 model undercounts the actual loads. Matrix 4 on the Pentium 3 shows that the model can overcount the number of loads. Further analysis of the block size chosen shows that the optimum row blocksize was 1024 elements. This implies that the row start/row end optimization reduced the number of loads to the matrix, however this is unaccounted for in the diagram since we assumed that all the values of $\delta_i$ were equal to r. When this assumption was removed, the model correctly predicted the number of loads (which is not reflected in the load plot for the Pentium 3 in Figure 11). Notice that in all the cases the block size that led to the best performance had more loads than the unblocked case. The increased locality advantage gained must outweigh the cost of these extra loads to yield performance improvements.

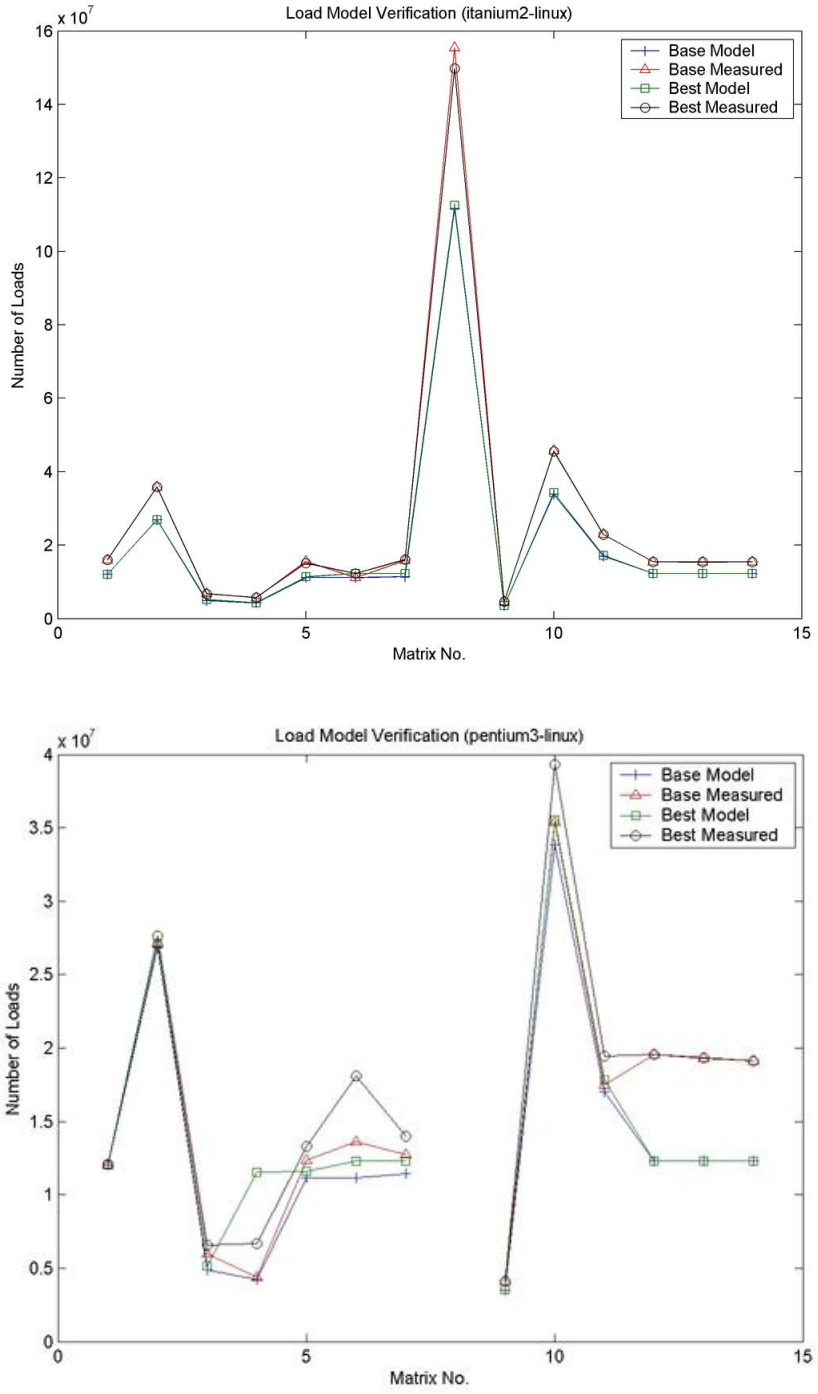



Figure 11: **Loads Model Verification**. This plot shows the accuracy of the loads model on the Itanium 2 and Pentium 3. The *Base* lines show the unblocked loads model and measurements. The *Best* lines shows the optimum blocked loads model and measurements

.

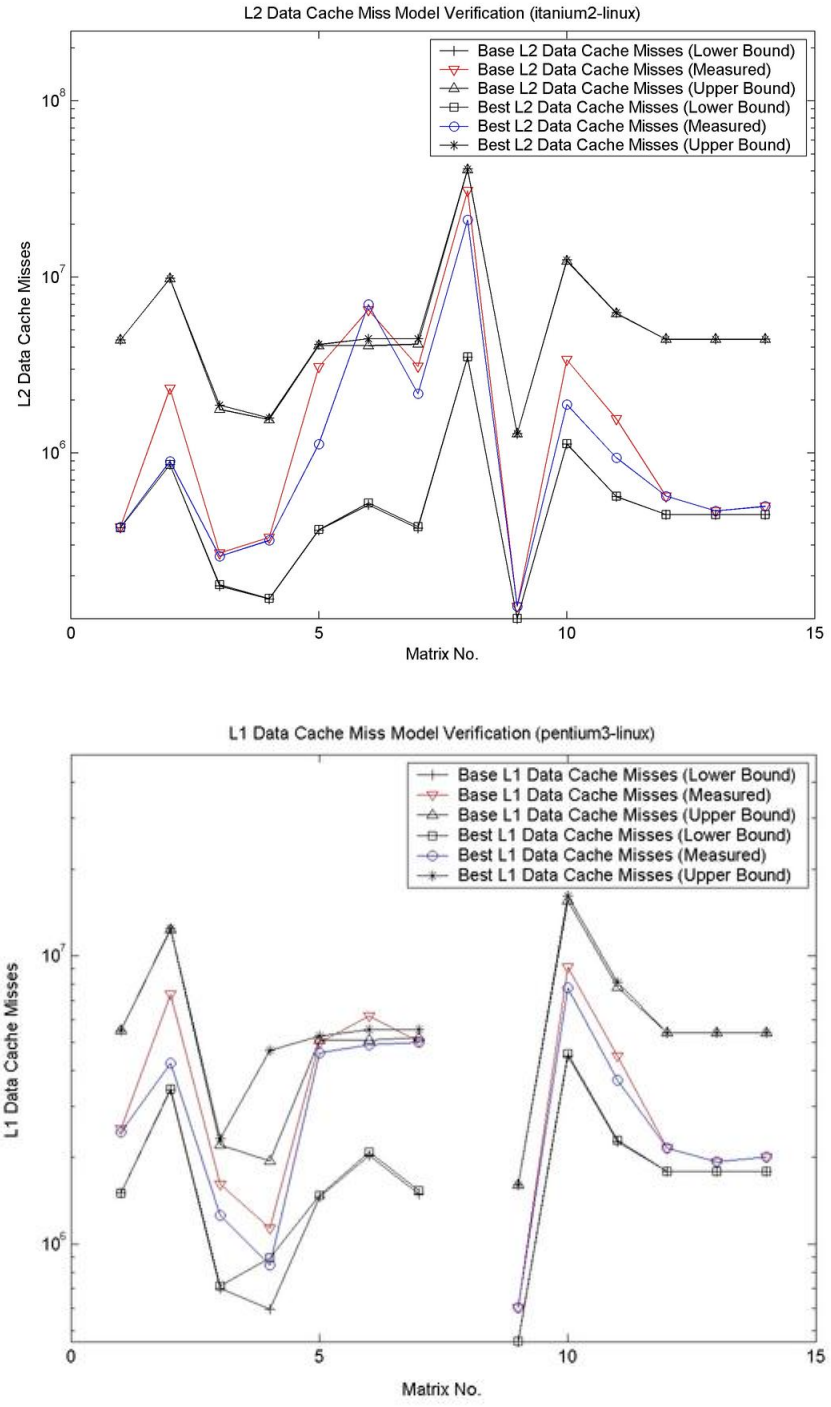



Figure 12: **Pentium 3 L1 / Itanium 2 L2 Misses**. This plot shows the accuracy of the models at the level of the memory system closest to the processor hierarchy where doubles can be stored. In the Itanium 2 this is the L2 and in the Pentium 3 it is the L1. The *Best* lines show the misses if the matrix is optimally blocked while the *Base* lines shows the case in which the matrix is unblocked.

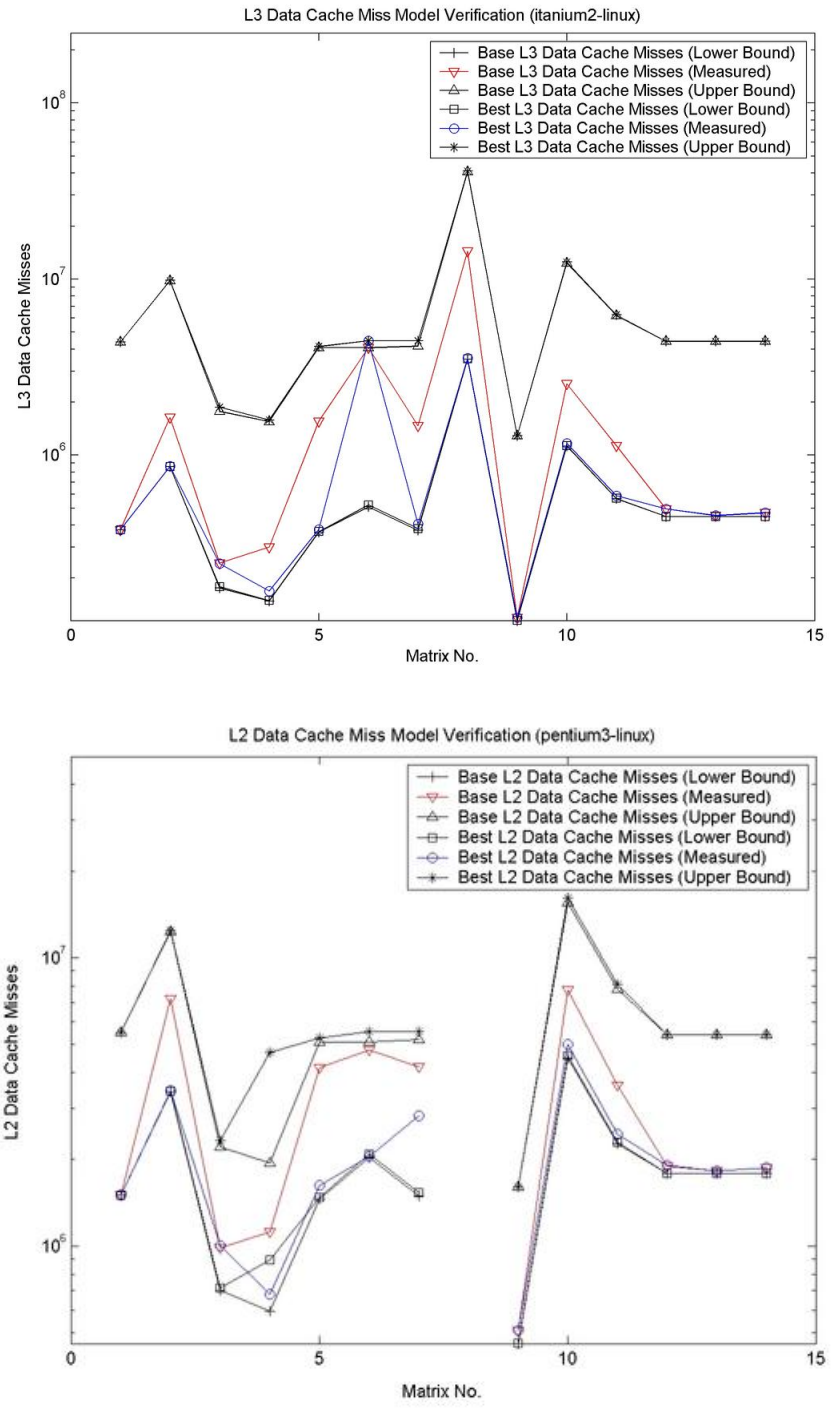



Figure 13: **Itanium 2 L3 /Pentium 3 L2 Misses**. This plot shows the same information as in Figure 12 however it shows the data at the cache level closest to the main memory; in the case of the Pentium this is the L2 and in the case of the Itanium this is the L3.

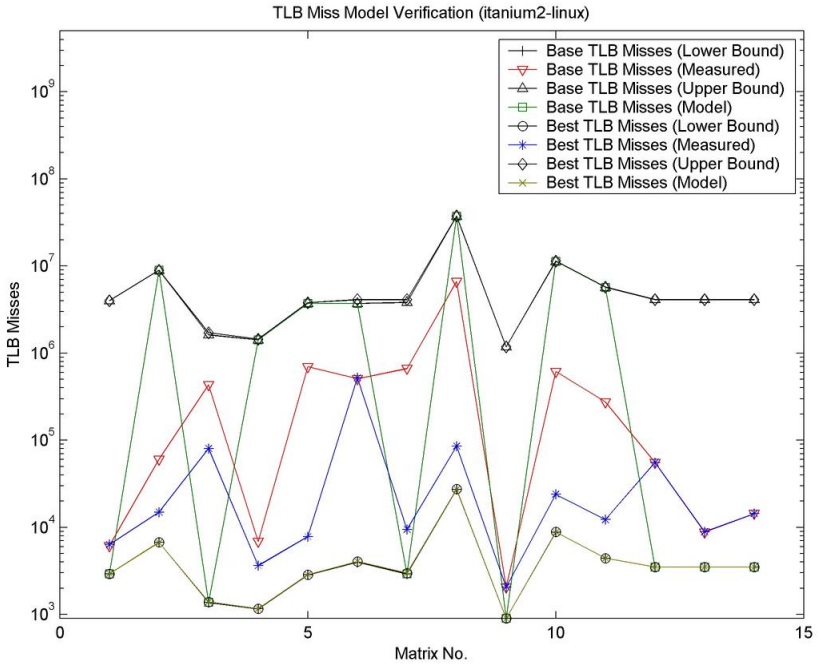



Figure 14: **Itanium2 TLB Misses**. The Base TLB Miss lines show the TLB misses when there is no blocking while the Best TLB Miss lines show the TLB misses at the optimal block size. The lower bound was calculated using Equation (5). The upper bound was calculated using Equation (6) while the model was calculated using Equation (7)

#### 5.3.2 Evaluation of Cache Miss Model

The main advantage of cache blocking is that we expect the total number of cache misses to drop since we improve the locality of access to the source and destination vectors. Note that because the Itanium 2 only caches doubles in the L2 and L3 we only present these two levels of cache in our models for this platform. As Figure 12 shows cache blocking significantly reduces the number of cache misses for those matrices that showed the greatest performance improvement. This difference is especially clear in Figure 13. The unblocked implementation has almost a factor of 10 more misses in some cases than the blocked implementations. As the data show, those matrices that showed little improvement in their number of cache misses had the worst speedups.

An interesting point to note is that the web matrices (Matrices 12–14) show that their base implementation runs at nearly the lower bound in cache misses. This implies that the unblocked versions of the matrix experience few conflict misses in their accesses, most likely because they are so sparse. This validates our assumption that when the matrix is too sparse it is hard to exploit locality.

On Matrix 6 (which also yields the best speedup) on both platforms, the L2 cache misses go from the near the upper bound to the near the lower bound when the matrix is cache blocked. Since matrix 6 has little spatial locality when accessing the source vector, cache blocking really helps decrease the number of misses since we can create locality where it does not already exists.

The model validation for each of the matrices on the Itanium 2 can be found in Appendix D and Appendix E. The same data for the Pentium 3 can be found in Appendix I and Appendix J. In some cases the measured number of misses drops below the lower bound on the cache misses. This is probably due to cache misses being hidden by prefetching within the memory system.

#### 5.3.3 Evaluation of the TLB Miss Models

One more interesting effect of cache blocking is that it also helps reduce the number of TLB misses that the processor has to handle as shown in Figure 14. Again, the matrices that have the greatest speedups also see the greatest decreases in the number of TLB misses and in most cases we are seeing the TLB misses

drop to near the lower bounds which indicates that cache blocking also helps reduce TLB misses. As seen in Appendix F the TLB model does a good job of switching at the right time on the Itanium 2 to yield the best performance, however the actual number of misses deviates from the predicted value.

# 6 Band Matrices

In this section we will analyze the performance of randomly generated banded matrices. The main focus of this section is to validate the *RSE* optimizations and provide more intuitions into when cache blocking helps.

## 6.1 Varying Bandwidth

In this section we present how the *RSE* optimization helps performance. We best illustrate this idea with banded matrices, that is matrices in which all the non zeros are clustered around a narrow band along the diagonal. In all our experiments, the matrices had the same number of non zeros($\approx$ 3.2 million non zeros) and the same dimension(8k $\times$ 256k). The only difference was the width of the diagonal band that varied from 1k to 256k by powers of 2. Spy plots of these matrices with full performance summaries are presented in Appendix M. A summary of this data is presented in table Table 3.

As the summary table and Appendix M show, the *RSE* optimizations do not improve the optimal performance. However they do erase the drastic differences in performances amongst the various block sizes, especially with narrow banded matrices. As the width of the band grows to the full matrix, the *RSE* performance starts to look the same as the non-*RSE* performance. The data shown here is an experimental verification of the theories presented by Temam *et al.* [20].

### 6.1.1 Cache Blocking with Band Matrices

As the data show, when the system is presented with band matrices, cache blocking does not help. This agrees with our intuitions because we no longer need to worry about blocking because the diagonal structure of the matrix will automatically lend itself to the optimal access pattern. We can smooth out differences in performance across block sizes by applying the *RSE* optimization. As the diagonal band grows to the full matrix, blocking becomes important since the structure of the matrix is not enough to exploit locality. This will be further explored in the next section.

## 6.2 Effects of Randomly Permuting Rows and Columns of Band Matrices

In this section we explore the effect of randomly permuting the rows and the columns of the narrow banded matrices in order to show the performance degradation when the optimal access pattern of the banded matrices are lost. The first permutation we analyze is randomly permuting the rows of the matrix. The expectation is that the performance should drop since we lose the locality gained with band matrices. Indeed as Figure 15 shows, the performance of the banded matrix in which the rows are randomly permuted is the same same as the performance of a matrix with the same dimension and density in which the non zeros are spread uniformly throughout the matrix. The second experiment we analyze is randomly permuting the columns in cache line sized chunks. We expect that the performance of this should be nearly as good as not permuting the matrix because we keep the locality when we access the source vector across columns. The data in Figure 15 shows that the best performance with this permutation is still close to the peak performance of the unpermuted banded matrix, however the performance is a lot more sensitive to the block size. The performance across row block sizes for the same column block size is roughly constant across all the different permutations. However the performance is a lot more sensitive to the column block size since the access patterns to the source vectors are no longer sequential.

| Diag. Width | w/o Row Start Row End Best MFLOPS | Performance Variation (%) | Good Performers | w/ Row Start Row End Best MFLOPS | Performance Variation (%) | Good Performers |
|---|---|---|---|---|---|---|
| 1 k | 295.45 | 51.83 | 26/36 | 295.92 | 0.48 | 36/36 |
| 2 k | 292.42 | 51.43 | 26/36 | 292.96 | 0.45 | 36/36 |
| 4 k | 291.51 | 51.34 | 26/36 | 291.75 | 1.27 | 36/36 |
| 8 k | 290.45 | 51.20 | 26/36 | 291.04 | 3.04 | 36/36 |
| 16 k | 286.68 | 50.59 | 26/36 | 289.42 | 5.99 | 32/36 |
| 32 k | 283.27 | 50.90 | 25/36 | 287.14 | 14.73 | 29/36 |
| 64 k | 276.33 | 53.29 | 26/36 | 282.65 | 32.31 | 19/36 |
| 128 k | 271.21 | 57.03 | 17/36 | 278.05 | 50.12 | 14/36 |
| 256 k | 272.94 | 65.09 | 11/36 | 274.18 | 69.14 | 8/36 |

Table 3: **Band Matrices Performance Summary**. This table shows a summary of the the diagonally banded matrices and their performance on the Itanium 2. The *Best MFLOPS* columns show the performance of the optimal cache block size in each of the respective categories. The *Performance Variation* columns show the the percentage difference between the performance of the best cache block size and the worst cache block size. The *Good Performers* columns show the number of cache block sizes out of 36 unique block sizes that yield performance within 90% of peak performance.

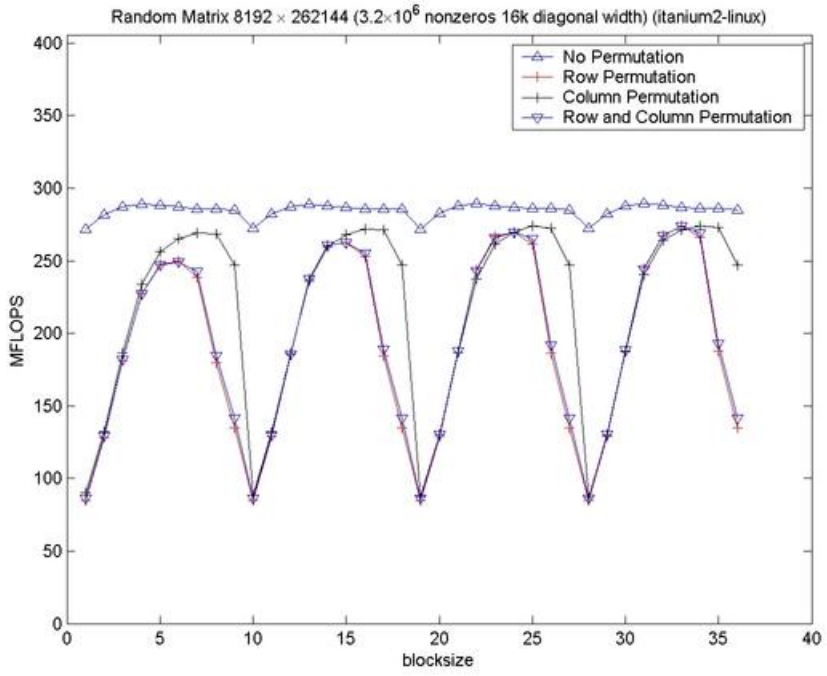



Figure 15: **Itanium 2 Diagonal Permutations**. This figure shows the performance of the various random permutations of a random banded matrix to see the effect on performance. The original matrix is 8192 x 262144 with $3.2\times10^6$ non zeros. The width of the band is 16384. The experiment were run on the Itanium 2. The *Row Permutation* are the results when the rows are randomly permuted. The *Column Permutation* are the results when the columns are randomly permuted. The columns were permuted in blocks of 16 columns. The *Row and Column Permutation* data shows the case in which the rows were randomly permuted, followed by the columns. The x-axis shows the block size unrolled along the row dimension. Thus block size of index 1 corresponds to 1024x1024 while a block size index of 2 corresponds to 1024x2048. A block size index of 10 corresponds to 2048x1024 in this case.

# 7 Evaluation of Cache Blocking Across Matrices and Platforms

As the data shows in Figure 16 the speedups for each of the matrices varied across the machines, however the matrices with the best speedups were the same across all the platforms suggesting that these matrices exhibited a certain characteristics that made them amenable to cache blocking. We also suggest architectural features to speedup SpM×V operations.

## 7.1 Matrix Structure

The matrices that experienced the most speedup were Matrices 5–8, Matrix 2, and Matrices 10–11. All of these matrices except Matrix 7 and Matrix 8 have a small row dimension and a very large column dimension and the non zeros are scattered uniformly throughout the matrix. According to the plots in Appendices D and E and Appendices I and J, the matrices that experience the most significant speedups had the largest jumps in the number of cache misses as the column block size increased. The large jumps imply that cache blocking created locality of access to the source vector that did not exist in the unblocked case.

We expect the matrices with the largest number of columns to see significant speedups, however this is not always the case since Matrices 12–14, which have very high column dimensions, exhibit little or no speedup. These matrices are so sparse that there is effectively no reuse when accessing the source vector and thus blocking does not help. When the matrices are dense enough so that there is reuse of the source vector, we see matrices with high column block sizes helped by cache blocking. The sparsest matrix in our suite that gets helped by cache blocking is Matrix 3. Thus matrices in our test suite with densities higher than $10^{-5}$ were helped with cache blocking, provided that their column block size is large enough (greater than 200,000 elements in our suite). It is important to understand the structural aspects of these matrices that lend themselves to this property and the aspects of the other matrices do not.

We also find that in general matrices in which the row dimension is much less than the column dimension benefit the most from cache blocking. The smaller row dimension implies the overhead added by cache blocking is small since the number of rows themselves are limited. The larger column dimension implies that the unblocked implementations lack locality. Even though Matrix 3 has a large column dimension, blocking did not yield much performance improvement, especially on the Pentium 3 because the number of rows was too high.

The difference between the structure of Matrix 7 and Matrix 8 illustrates the costs associated with having too many zero rows. Since the base performance of these two matrices was roughly the same on this platform, we can theorize that the increase in the row dimension is what caused the decreased speedup in Matrix 7 in relation to Matrix 5. Because Matrix 7 is an expanded version of Matrix 5 it will experience the same locality as Matrix 5 however the overhead indexing information for cache blocking that must be loaded will be larger. On Matrix 8 the locality will also increase by the same factor as the size of the data structure since the matrix is merely replicated rather than expanded. Thus we expect the same performance from Matrix 5 as Matrix 8 since we are doing the same operation 10 times over. The results match these intuitions. The difference in performance between Matrices 7 and 8 is 14%. This is the cost of accessing the extra indexing information for zero rows on the Itanium 2.

Matrix 6 is an interesting case since it has the highest speedup on the Itanium 2. However the reason for this is that the base performance itself was rather low. This implies that the decrease in density hurt the raw performance, but the large column dimension made the matrix a lot more amenable to cache blocking. The main exception to this rule is Matrix 4. From the spy plot of this matrix we see that matrix has many diagonals. This implies in an unblocked implementation, the structure of the matrix already lends itself to generating the proper locality.

Matrix 4 leads us into more intuitions into the structure of the matrix. As seen from Section 6 narrow band matrices experienced little or no speedup via cache blocking, because the access pattern to the source vector already had the optimum locality. However as the bandwidth of the matrix increased, the importance of the locality created by column blocking grew until the full matrix where it was clear that there was one optimal column block size. This effect can also be seen in the permutations of the matrix. As the rows of a narrow banded band matrix were permuted, the performance degraded to that of a non-banded matrix, however when the columns were permuted the performance at the best block sizes was better than that of the non-diagonal matrix because some of the locality was maintained. We can also see this effect if we look at the correlation numbers presented in Table 2. The matrices that had correlations closer to 1 did not benefit from cache blocking nearly as much as those matrices that had low correlations. This shows that the correlation coefficient can be used as part of a heuristic to pick decide when it is important to cache block.

| Matrix No. | Platform | | |
|---|---|---|---|
| | Itanium 2 | Pentium 3 | Power 4 |
| 1 | 1.00 | 1.01 | 1.01 |
| 2 | 1.27 | 1.61 | 1.77 |
| 3 | 1.28 | 1.02 | 1.24 |
| 4 | 1.14 | 1.15 | 1.37 |
| 5 | 2.00 | 1.40 | 1.97 |
| 6 | 2.84 | 1.33 | 2.93 |
| 7 | 1.72 | 1.10 | 1.68 |
| 8 | 1.94 | N/A | N/A |
| 9 | 1.00 | 1.00 | 1.00 |
| 10 | 1.40 | 1.21 | 1.75 |
| 11 | 1.34 | 1.21 | 1.73 |
| 12 | 1.00 | 1.00 | 1.01 |
| 13 | 1.00 | 1.00 | 1.09 |
| 14 | 1.00 | 1.00 | 1.01 |

Table 4: **Speedups across Matrices and Across Platforms**. This table shows the performance of the optimum cache block divided by the performance of the non-blocked implementation on that platform for that matrix.

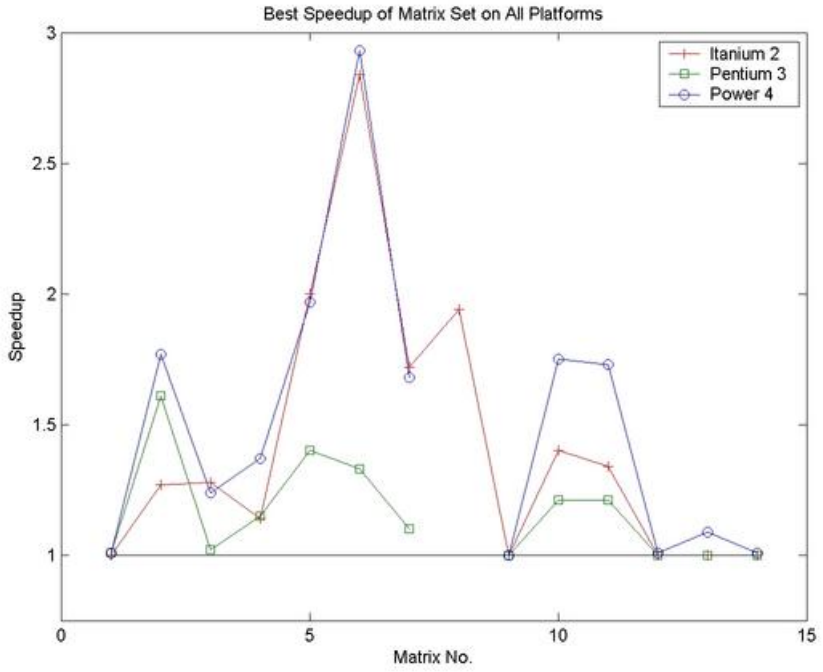



Figure 16: **Speedup**. This plot shows the best speedup of the different matrices in Table 2 across the platforms in Table 1

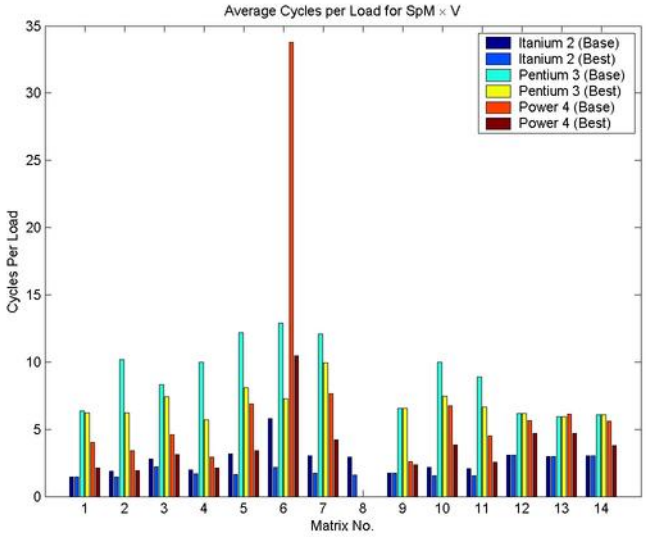



Figure 17: **Average Cycles per Load**. This plot shows the average cycles per load taken to complete the SpM×V for the different matrices across different platforms. The load counts are taken from the hardware counters on the Pentium 3 and the Itanium 2 and from the model on the Power 4. The plot shows the number of cycles taken to do the multiplication divided by the number of loads needed for the multiplication.

### 7.2 Platform Evaluation

In this section we will analyze what the important components of the memory system are for matrices of this suite to be aided by cache blocking. Certain matrices such as Matrix 5 experienced significant performance gains through cache blocking on the Itanium 2 and the Power 4, but the speedup was less drastic on the Pentium 3. Our expectations are that as the average number of cycles to access the memory grows, cache blocking will provide a good improvement in performance since cache blocking allows us to create temporal locality.

Figure 17 shows the average number of cycles per load taken to complete the SpM×V across different platforms and matrices. As this plot shows, the matrices that experienced the best speedups across all the architectures also saw the most drastic drops in the average time taken to perform a load. This suggests that platforms with high degree of variability in the average memory access time, such as the Power 4, will be helped by cache blocking.

We first notice that we experience a significant number of TLB misses and that cache blocking greatly reduces these TLB misses. However all of these TLB misses can easily be avoided in SpM×V by creating large page sizes. There are aspects of SpM×V that take advantage of the caches and aspects that are hurt by them. The reuse of the source vector elements can be helped by large caches however the access to the matrix is not helped by caching since there is no reuse of the matrix elements. Thus this suggests that two paths to the memory would be ideal. One would use a cache-based memory system and the second a vector-based memory system. With this addition we should see the reuse of our source vectors dramatically increase since we no longer need to worry about source vector elements getting evicted by matrix elements. The associativity of the caches starts to solve the problem, however it is an incomplete solution since the old source vector elements are still prone to premature eviction by matrix elements. In an ideal case, source vector elements should only get evicted by access to other source vector elements. In future work it would be nice to verify this theory on the Cray X1 which implements a portion of these ideas and how the combination of a vector and scalar architecture would aid in cache blocking of SpM×V.

## 8 Conclusions and Future Work

In this paper we analyzed the effects of cache blocking SpM×V and found that for cache blocking to work the benefits from the added locality must outweigh the costs to access the extra overhead in the data structure

due to blocking. We first described the data structures and presented two new optimizations that helped performance. We then presented performance models and argued that the performance models have to find a way to take into account the added locality of cache blocking. We then analyzed these models on our three different platforms and found that they did well at predicting a good block size on the Itanium 2 and to some extent the Pentium 3. This shows that they can be used as the basis for a heuristic to pick the correct cache block size given the matrix dimension and density. We found that the best speedups from cache blocking occur when the matrices have a relatively small row dimension and a very large column dimension and are generally on the denser end of the matrices tested. These matrices had large enough column dimensions so that creating temporal locality in access to the source vector helps. We see that with narrow-band matrices of the same dimension and density, the cache blocking doesn't help since the matrix structure itself yields the optimal access patterns to the source vector.

We also find that our analytic models predict the performance well on the Itanium 2. On the Pentium 3 these analytic models are good but are not as accurate as the Itanium 2. On the Power 4, the analytic model has a difficult time finding the optimal block size. In contrast, the PAPI model was better at picking the best block size on the Pentium 3 than on the Itanium 2. We have seen that the models are extremely optimistic in some cases and extremely pessimistic in others. One way to improve the performance is to create more accurate models of the memory system to take the structure of the matrix into account. Another drawback with the models is they don't take the sizes of the cache into account when predicting performance. When we added the size of the TLB into the model, it got a lot better at picking the correct block size. The cache miss models are unable to account for the increase in capacity misses as the column block size gets too large. The cache miss models alone predict the optimal thing to do is not to block the matrix since it adds the least data structure overhead, and thus resulting in the least amount of misses. Thus as future work more accurate cache miss models would help predict performance more on the different platforms.

The next goals of this work are to create more accurate memory system models to better predict performance, find heuristics to pick the correct block size, and analyze the problem on novel architectures.

**Acknowledgements**

This work was supported in part by the National Science Foundation under ACI-9619020 ACI-0090127 and gifts from: Intel Corporation, Hewlett Packard, and Sun Microsystems.

The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

# References


[1] A. J. C. Bik and H. A. G. Wijshoff. Automatic nonzero structure analysis. *SIAM Journal on Computing*, 28(5):1576–1587, 1999.

[2] S. Blackford, G. Corliss, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, C. Hu, W. Kahan, L. Kaufman, B. Kearfott, F. Krogh, X. Li, Z. Maany, A. Petitet, R. Pozo, K. Remington, W. Walster, C. Whaley, and J. W. von Gudenberg. Document for the Basic Linear Algebra Subprograms (BLAS) standard: BLAS Technical Forum, 2001. `www.netlib.org/blast`.

[3] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proceedings of Supercomputing*, November 2000.

[4] S. Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Proceedings of Supercomputing*, pages 114–124, 1992.

[5] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck. Exact analysis of the cache behavior of nested loops. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 286–297, Snowbird, UT, USA, June 2001.

[6] B. B. Fraguela, R. Doallo, and E. L. Zapata. Memory hierarchy performance prediction for sparse blocked algorithms. *Parallel Processing Letters*, 9(3), March 1999.

[7] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, 1999.

[8] K. Goto and R. van de Geijn. On reducing TLB misses in matrix multiplication. Technical Report TR-2002-55, The University of Texas at Austin, Department of Computer Sciences, 2002. FLAME Working Note #9.

[9] W. D. Gropp, D. K. Kasushik, D. E. Keyes, and B. F. Smith. Towards realistic bounds for implicit CFD codes. In *Proceedings of Parallel Computational Fluid Dynamics*, pages 241–248, 1999.

[10] G. Heber, A. J. Dolgert, M. Alt, K. A. Mazurkiewicz, and L. Stringer. Fracture mechanics on the Intel Itanium architecture: A case study. In *Workshop on EPIC Architectures and Compiler Technology (ACM MICRO 34)*, Austin, TX, December 2001.

[11] D. B. Heras, V. B. Perez, J. C. C. Dominguez, and F. F. Rivera. Modeling and improving locality for irregular problems: sparse matrix-vector product on cache memories as a case study. In *HPCN Europe*, pages 201–210, 1999.

[12] E.-J. Im. *Optimizing the performance of sparse matrix-vector multiplication*. PhD thesis, University of California, Berkeley, May 2000.

[13] E.-J. Im and K. A. Yelick. Optimizing sparse matrix computations for register reuse in SPARSITY. In *Proceedings of the International Conference on Computational Science*, volume 2073 of *LNCS*, pages 127–136. Springer, May 2001.

[14] J. D. McCalpin. STREAM: Measuring sustainable memory bandwidth in high performance computers, 1995. `http://www.cs.virginia.edu/stream`.

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

[16] W. Pugh and T. Shpeisman. Generation of efficient code for sparse matrix computations. In *Proceedings of the 11th Workshop on Languages and Compilers for Parallel Computing*, LNCS, August 1998.

[17] R. H. Saavedra-Barrera. *CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking*. PhD thesis, University of California, Berkeley, February 1992.

[18] A. Snavely, L. Carrington, and N. Wolter. Modeling application performance by convolving machine signatures with application profiles. 2001.

[19] P. Stodghill. *A Relational Approach to the Automatic Generation of Sequential Sparse Matrix Codes.* PhD thesis, Cornell University, August 1997.

[20] O. Temam and W. Jalby. Characterizing the behavior of sparse algorithms on caches. In *Proceedings of Supercomputing '92*, 1992.

[21] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance optimizations and bounds for sparse matrix-vector multiply. In *Proceedings of Supercomputing*, Baltimore, MD, USA, November 2002.

[22] R. W. Vuduc. *Automatic performance tuning of sparse matrix kernels.* PhD thesis, University of California, Berkeley, 2003.

[23] C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Proc. of Supercomp.*, 1998.

[24] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 1991.

# A Model Verification

| Matrix No. | Base Performance | Best R | Best C | Best Performance |
|---|---|---|---|---|
| 1 | 311.1 | 1024 | 1024 | 311.7 |
| 2 | 241.1 | 231 | 4096 | 304.9 |
| 3 | 148.3 | 32768 | 131072 | 189.2 |
| 4 | 225.6 | 10280 | 65536 | 257.5 |
| 5 | 135.5 | 10000 | 32768 | 272.3 |
| 6 | 67.9 | 10000 | 65536 | 192.6 |
| 7 | 139.3 | 100000 | 65536 | 239.4 |
| 8 | 145.8 | 100000 | 65536 | 283.0 |
| 9 | 246.4 | 16384 | 65535 | 246.8 |
| 10 | 202.7 | 4284 | 32768 | 285.0 |
| 11 | 212.7 | 4284 | 32768 | 284.7 |
| 12 | 117.3 | 1000005 | 1000005 | 117.3 |
| 13 | 120.6 | 262144 | 1000005 | 120.7 |
| 14 | 119.3 | 1000005 | 1000005 | 119.3 |

Table 5: **Itanium 2 Performance**. The *Base Performance* column shows the performance of the unblocked implementation (in MFLOP/s) while the *Best Performance* shows the performance of the best blocked implementation. The *Best R* and *Best C* column show the cache block sizes that produced the best performance.

| Matrix No. | Base Performance | Best R | Best C | Best Performance |
|---|---|---|---|---|
| 1 | 52.3 | 2000 | 1024 | 53.0 |
| 2 | 32.4 | 231 | 1024 | 52.2 |
| 3 | 31.5 | 1024 | 131072 | 32.1 |
| 4 | 32.0 | 4096 | 1024 | 36.9 |
| 5 | 24.7 | 10000 | 16384 | 34.6 |
| 6 | 21.2 | 10000 | 65536 | 28.3 |
| 7 | 24.2 | 65536 | 65536 | 26.7 |
| 8 | N/A | | | |
| 9 | 41.5 | 32768 | 65535 | 41.5 |
| 10 | 31.9 | 4284 | 8192 | 38.6 |
| 11 | 36.3 | 4096 | 8192 | 43.8 |
| 12 | 25.6 | 32768 | 1000005 | 25.6 |
| 13 | 27.1 | 32768 | 1000005 | 27.1 |
| 14 | 26.8 | 16384 | 1000005 | 26.8 |

Table 6: **Pentium 3 Performance**.

| Matrix No. | Base Performance | Best R | Best C | Best Performance |
|---|---|---|---|---|
| 1 | 255.8 | 231 | 2048 | 452.2 |
| 2 | 606.0 | 2000 | 1024 | 615.3 |
| 3 | 187.0 | 52260 | 65536 | 231.0 |
| 4 | 293.0 | 10280 | 65536 | 402.0 |
| 5 | 126.0 | 10000 | 32768 | 248.0 |
| 6 | 25.6 | 10000 | 65536 | 75.0 |
| 7 | 110.5 | 100000 | 65536 | 185.6 |
| 8 | N/A | | | |
| 9 | 327.7 | 65535 | 65535 | 327.7 |
| 10 | 128.0 | 4284 | 32768 | 224.0 |
| 11 | 192.0 | 4284 | 16384 | 333.0 |
| 12 | 128.0 | 16384 | 1000005 | 139.0 |
| 13 | 140.0 | 16384 | 524288 | 141.0 |
| 14 | 138.0 | 262144 | 1000005 | 140.0 |

Table 7: **Power 4 Performance**.

| | Base | | Best | |
|---|---|---|---|---|
| Matrix No. | Model Loads | Measured Loads | Model Loads | Measured Loads |
| 1 | 1.20E+07 | 1.60E+07 | 1.20E+07 | 1.60E+07 |
| 2 | 2.69E+07 | 3.59E+07 | 2.69E+07 | 3.59E+07 |
| 3 | 4.86E+06 | 6.86E+06 | 5.16E+06 | 6.75E+06 |
| 4 | 4.26E+06 | 5.67E+06 | 4.34E+06 | 5.76E+06 |
| 5 | 1.12E+07 | 1.56E+07 | 1.14E+07 | 1.51E+07 |
| 6 | 1.12E+07 | 1.69E+07 | 1.23E+07 | 1.61E+07 |
| 7 | 1.14E+07 | 1.58E+07 | 1.23E+07 | 1.61E+07 |
| 8 | 1.12E+08 | 1.55E+08 | 1.13E+08 | 1.50E+08 |
| 9 | 3.54E+06 | 4.66E+06 | 3.54E+06 | 4.66E+06 |
| 10 | 3.39E+07 | 4.57E+07 | 3.43E+07 | 4.56E+07 |
| 11 | 1.70E+07 | 2.29E+07 | 1.72E+07 | 2.29E+07 |
| 12 | 1.23E+07 | 1.55E+07 | 1.23E+07 | 1.55E+07 |
| 13 | 1.23E+07 | 1.54E+07 | 1.23E+07 | 1.54E+07 |
| 14 | 1.23E+07 | 1.54E+07 | 1.23E+07 | 1.54E+07 |

Table 8: **Itanium 2 Loads**. This table shows how well the load model predicts the actual amount of loads. The *Base* columns contain the number of loads when cache blocking is not used while the *Best* column shows the data with the optimum cache block size.

| | Base | | Best | |
|---|---|---|---|---|
| Matrix No. | Model Loads | Measured Loads | Model Loads | Measured Loads |
| 1 | 1.20E+007 | 1.20E+007 | 1.20E+007 | 1.21E+007 |
| 2 | 2.69E+007 | 2.72E+007 | 2.71E+007 | 2.76E+007 |
| 3 | 4.86E+006 | 5.98E+006 | 5.16E+006 | 6.59E+006 |
| 4 | 4.26E+006 | 4.39E+006 | 1.16E+007 | 6.67E+006 |
| 5 | 1.12E+007 | 1.23E+007 | 1.16E+007 | 1.33E+007 |
| 6 | 1.12E+007 | 1.36E+007 | 1.23E+007 | 1.81E+007 |
| 7 | 1.14E+007 | 1.27E+007 | 1.23E+007 | 1.40E+007 |
| 8 | N/A | | | |
| 9 | 3.54E+006 | 4.10E+006 | 3.54E+006 | 4.10E+006 |
| 10 | 3.39E+007 | 3.54E+007 | 3.56E+007 | 3.93E+007 |
| 11 | 1.70E+007 | 1.75E+007 | 1.78E+007 | 1.94E+007 |
| 12 | 1.23E+007 | 1.96E+007 | 1.23E+007 | 1.96E+007 |
| 13 | 1.23E+007 | 1.93E+007 | 1.23E+007 | 1.93E+007 |
| 14 | 1.23E+007 | 1.91E+007 | 1.23E+007 | 1.91E+007 |

Table 9: **Pentium 3 Loads**. This table shows how well the load model predicts the actual amount of loads. The *Base* columns contain the number of loads when cache blocking is not used while the *Best* column shows the data with the optimum cache block size.

| | Base | | | Best | | |
|---|---|---|---|---|---|---|
| Marix No. | Lower Bound | Measured | Upper Bound | Lower Bound | Measured | Upper Bound |
| 1 | 3.75E+05 | 3.80E+05 | 4.38E+06 | 3.75E+05 | 3.79E+05 | 4.38E+06 |
| 2 | 8.59E+05 | 2.33E+06 | 9.79E+06 | 8.60E+05 | 8.98E+05 | 9.81E+06 |
| 3 | 1.76E+05 | 2.71E+05 | 1.77E+06 | 1.79E+05 | 2.60E+05 | 1.87E+06 |
| 4 | 1.48E+05 | 3.32E+05 | 1.55E+06 | 1.49E+05 | 3.19E+05 | 1.58E+06 |
| 5 | 3.65E+05 | 3.08E+06 | 4.07E+06 | 3.67E+05 | 1.13E+06 | 4.14E+06 |
| 6 | 5.09E+05 | 4.36E+06 | 4.07E+06 | 5.21E+05 | 1.84E+06 | 4.46E+06 |
| 7 | 3.73E+05 | 3.10E+06 | 4.16E+06 | 3.82E+05 | 2.17E+06 | 4.46E+06 |
| 8 | 3.51E+06 | 3.09E+07 | 4.07E+07 | 3.51E+06 | 2.12E+07 | 4.10E+07 |
| 9 | 1.15E+05 | 1.34E+05 | 1.29E+06 | 1.15E+05 | 1.34E+05 | 1.29E+06 |
| 10 | 1.13E+06 | 3.39E+06 | 1.23E+07 | 1.13E+06 | 1.89E+06 | 1.25E+07 |
| 11 | 5.65E+05 | 1.56E+06 | 6.20E+06 | 5.67E+05 | 9.40E+05 | 6.27E+06 |
| 12 | 4.47E+05 | 5.70E+05 | 4.43E+06 | 4.47E+05 | 5.70E+05 | 4.43E+06 |
| 13 | 4.47E+05 | 4.69E+05 | 4.43E+06 | 4.47E+05 | 4.69E+05 | 4.43E+06 |
| 14 | 4.47E+05 | 4.98E+05 | 4.43E+06 | 4.47E+05 | 4.98E+05 | 4.43E+06 |

Table 10: **Itanium 2 L2 Data Cache Misses**.

| | Base | | | Best | | |
|---|---|---|---|---|---|---|
| Marix No. | Lower Bound | Measured | Upper Bound | Lower Bound | Measured | Upper Bound |
| 1 | 3.75E+05 | 3.76E+05 | 4.38E+06 | 4.38E+06 | 3.75E+05 | 3.76E+05 |
| 2 | 8.59E+05 | 1.64E+06 | 9.79E+06 | 9.81E+06 | 8.60E+05 | 8.63E+05 |
| 3 | 1.76E+05 | 2.44E+05 | 1.77E+06 | 1.87E+06 | 1.79E+05 | 2.41E+05 |
| 4 | 1.48E+05 | 3.00E+05 | 1.55E+06 | 1.58E+06 | 1.49E+05 | 1.69E+05 |
| 5 | 3.65E+05 | 1.56E+06 | 4.07E+06 | 4.14E+06 | 3.67E+05 | 3.79E+05 |
| 6 | 5.09E+05 | 3.56E+06 | 4.07E+06 | 4.46E+06 | 5.21E+05 | 5.59E+05 |
| 7 | 3.73E+05 | 1.47E+06 | 4.16E+06 | 4.46E+06 | 3.82E+05 | 4.07E+05 |
| 8 | 3.51E+06 | 1.44E+07 | 4.07E+07 | 4.10E+07 | 3.51E+06 | 3.56E+06 |
| 9 | 1.15E+05 | 1.19E+05 | 1.29E+06 | 1.29E+06 | 1.15E+05 | 1.19E+05 |
| 10 | 1.13E+06 | 2.54E+06 | 1.23E+07 | 1.25E+07 | 1.13E+06 | 1.17E+06 |
| 11 | 5.65E+05 | 1.13E+06 | 6.20E+06 | 6.27E+06 | 5.67E+05 | 5.87E+05 |
| 12 | 4.47E+05 | 4.95E+05 | 4.43E+06 | 4.43E+06 | 4.47E+05 | 4.95E+05 |
| 13 | 4.47E+05 | 4.53E+05 | 4.43E+06 | 4.43E+06 | 4.47E+05 | 4.53E+05 |
| 14 | 4.47E+05 | 4.71E+05 | 4.43E+06 | 4.43E+06 | 4.47E+05 | 4.71E+05 |

Table 11: **Itanium 2 L3 Data Cache Misses**.

| Marix No. | Base | | | Best | | |
|---|---|---|---|---|---|---|
| | Lower Bound | Measured | Upper Bound | Lower Bound | Measured | Upper Bound |
| 1 | 1.50E+06 | 2.52E+06 | 5.50E+06 | 1.50E+06 | 2.44E+06 | 5.50E+06 |
| 2 | 3.44E+06 | 7.35E+06 | 1.23E+07 | 3.45E+06 | 4.25E+06 | 1.24E+07 |
| 3 | 7.02E+05 | 1.61E+06 | 2.21E+06 | 7.15E+05 | 1.26E+06 | 2.33E+06 |
| 4 | 5.93E+05 | 1.13E+06 | 1.95E+06 | 8.97E+05 | 8.46E+05 | 4.68E+06 |
| 5 | 1.46E+06 | 4.99E+06 | 5.12E+06 | 1.48E+06 | 4.62E+06 | 5.28E+06 |
| 6 | 2.04E+06 | 6.18E+06 | 5.12E+06 | 2.08E+06 | 4.92E+06 | 5.54E+06 |
| 7 | 1.49E+06 | 5.03E+06 | 5.22E+06 | 1.53E+06 | 5.02E+06 | 5.54E+06 |
| 8 | N/A | | | | | |
| 9 | 4.59E+05 | 6.03E+05 | 1.61E+06 | 4.59E+05 | 6.03E+05 | 1.61E+06 |
| 10 | 4.50E+06 | 9.15E+06 | 1.55E+07 | 4.58E+06 | 7.73E+06 | 1.62E+07 |
| 11 | 2.26E+06 | 4.48E+06 | 7.79E+06 | 2.30E+06 | 3.72E+06 | 8.11E+06 |
| 12 | 1.79E+06 | 2.15E+06 | 5.40E+06 | 1.79E+06 | 2.15E+06 | 5.40E+06 |
| 13 | 1.79E+06 | 1.93E+06 | 5.40E+06 | 1.79E+06 | 1.93E+06 | 5.40E+06 |
| 14 | 1.79E+06 | 2.00E+06 | 5.40E+06 | 1.79E+06 | 2.00E+06 | 5.40E+06 |

Table 12: **Pentium3 L1 Data Cache Misses**.

| Marix No. | Base | | | Best | | |
|---|---|---|---|---|---|---|
| | Lower Bound | Measured | Upper Bound | Lower Bound | Measured | Upper Bound |
| 1 | 1.50E+06 | 1.50E+06 | 5.50E+06 | 1.50E+06 | 1.50E+06 | 5.50E+06 |
| 2 | 3.44E+06 | 7.22E+06 | 1.23E+07 | 3.45E+06 | 3.46E+06 | 1.24E+07 |
| 3 | 7.02E+05 | 9.85E+05 | 2.21E+06 | 7.15E+05 | 1.00E+06 | 2.33E+06 |
| 4 | 5.93E+05 | 1.12E+06 | 1.95E+06 | 8.97E+05 | 6.79E+05 | 4.68E+06 |
| 5 | 1.46E+06 | 4.17E+06 | 5.12E+06 | 1.48E+06 | 1.62E+06 | 5.28E+06 |
| 6 | 2.04E+06 | 4.77E+06 | 5.12E+06 | 2.08E+06 | 2.04E+06 | 5.54E+06 |
| 7 | 1.49E+06 | 4.20E+06 | 5.22E+06 | 1.53E+06 | 2.84E+06 | 5.54E+06 |
| 8 | N/A | | | | | |
| 9 | 4.59E+05 | 5.11E+05 | 1.61E+06 | 4.59E+05 | 5.10E+05 | 1.61E+06 |
| 10 | 4.50E+06 | 7.73E+06 | 1.55E+07 | 4.58E+06 | 5.00E+06 | 1.62E+07 |
| 11 | 2.26E+06 | 3.62E+06 | 7.79E+06 | 2.30E+06 | 2.45E+06 | 8.11E+06 |
| 12 | 1.79E+06 | 1.90E+06 | 5.40E+06 | 1.79E+06 | 1.90E+06 | 5.40E+06 |
| 13 | 1.79E+06 | 1.82E+06 | 5.40E+06 | 1.79E+06 | 1.82E+06 | 5.40E+06 |
| 14 | 1.79E+06 | 1.86E+06 | 5.40E+06 | 1.79E+06 | 1.86E+06 | 5.40E+06 |

Table 13: **Pentium3 L2 Data Cache Misses**.

| Marix No. | Base | | | |
|---|---|---|---|---|
| | Lower Bound | Measured | Upper Bound | Model |
| 1 | 2.93E+03 | 6.15E+03 | 4.00E+06 | 2.93E+03 |
| 2 | 6.71E+03 | 6.05E+04 | 8.96E+06 | 8.96E+06 |
| 3 | 1.37E+03 | 4.32E+05 | 1.62E+06 | 1.37E+03 |
| 4 | 1.16E+03 | 6.87E+03 | 1.42E+06 | 1.42E+06 |
| 5 | 2.85E+03 | 6.97E+05 | 3.73E+06 | 3.73E+06 |
| 6 | 3.98E+03 | 2.07E+06 | 3.73E+06 | 3.71E+06 |
| 7 | 2.92E+03 | 6.66E+05 | 3.82E+06 | 3.71E+06 |
| 8 | 2.74E+04 | 6.65E+06 | 3.73E+07 | 3.73E+07 |
| 9 | 8.96E+02 | 2.05E+03 | 1.18E+06 | 8.96E+02 |
| 10 | 8.80E+03 | 6.16E+05 | 1.13E+07 | 1.13E+07 |
| 11 | 4.42E+03 | 2.74E+05 | 5.67E+06 | 5.67E+06 |
| 12 | 3.50E+03 | 5.55E+04 | 4.11E+06 | 3.50E+03 |
| 13 | 3.50E+03 | 8.82E+03 | 4.11E+06 | 3.50E+03 |
| 14 | 3.50E+03 | 1.44E+04 | 4.11E+06 | 3.50E+03 |

| Matrix No. | Best | | | |
|---|---|---|---|---|
| | Lower Bound | Measured | Upper Bound | Model |
| 1 | 2.93E+03 | 6.36E+03 | 4.01E+06 | 2.93E+03 |
| 2 | 6.72E+03 | 1.50E+04 | 8.98E+06 | 6.72E+03 |
| 3 | 1.40E+03 | 8.04E+04 | 1.72E+06 | 1.40E+03 |
| 4 | 1.16E+03 | 3.64E+03 | 1.45E+06 | 1.16E+03 |
| 5 | 2.87E+03 | 7.81E+03 | 3.79E+06 | 2.87E+03 |
| 6 | 4.07E+03 | 1.17E+04 | 4.11E+06 | 4.07E+03 |
| 7 | 2.99E+03 | 9.49E+03 | 4.11E+06 | 2.99E+03 |
| 8 | 2.75E+04 | 8.55E+04 | 3.75E+07 | 2.75E+04 |
| 9 | 8.96E+02 | 2.06E+03 | 1.18E+06 | 8.96E+02 |
| 10 | 8.83E+03 | 2.40E+04 | 1.14E+07 | 8.83E+03 |
| 11 | 4.43E+03 | 1.23E+04 | 5.74E+06 | 4.43E+03 |
| 12 | 3.50E+03 | 5.55E+04 | 4.11E+06 | 3.50E+03 |
| 13 | 3.50E+03 | 8.88E+03 | 4.11E+06 | 3.50E+03 |
| 14 | 3.50E+03 | 1.44E+04 | 4.11E+06 | 3.50E+03 |

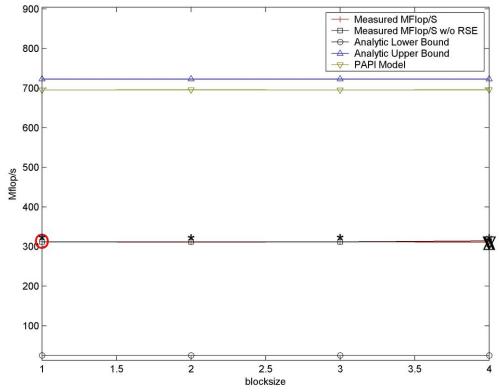
Table 14: **Itanium2 TLB Misses**.

# B Itanium 2 Performance Plots

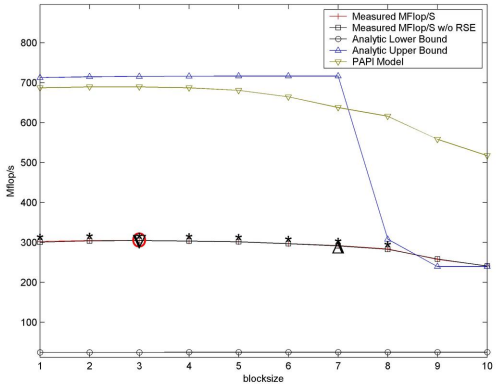
The following plots show the performance at all the various block sizes.The upper bound and lower bound models are derived from the models as described in Section 3.

The *Measured MFLOPS* line is the actual measurement of the megaflop rate with the row start/end and function call optimizations as described in Section 2.3. The *Measured MFLOPS w/o RSE* line is the measured performance without the new optimizations. The *Analytic Upper Bound* line is the Analytic Upper Bound on Performance while the *Analytic Lower Bound* shows the analytic lower bound on performance. The *PAPI Model* shows the performance model in which actual counter data has been plugged in. An important observation to make on these plots is that the shape of the PAPI model and the shape of the measured data coincide very well indicating that if our models on the misses were perfect, then our intuitions on how to measure performance are correct.

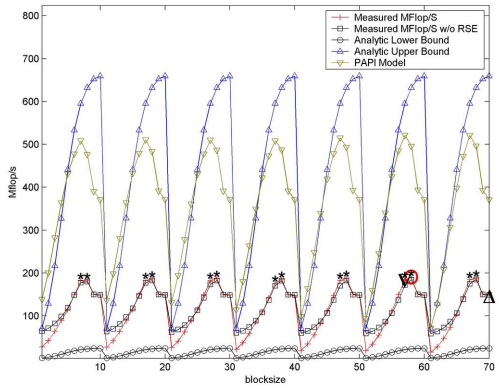
We have also annotated the plots with markers to show the best block sizes and good block sizes according to each of the models. The Best Measured Blocksize is circled in red. All the blocksizes that yield a performance result within 90% of the best block size are annotated with a *. The best block size according to the upperbound analytic model is marked by an up-ward pointing triangle. The best block size according to the PAPI model is annotated with a downward pointing triangle. The more overlap the markers have, the better our models did at predicting the optimal performance on that matrix. The x-axis shows the block size unrolled along the row dimension. Thus block size of index 1 corresponds to 1024x1024 while a block size index of 2 corresponds to 1024x2048 and an index of 3 represents 1024x4096 and so on. In the case of Matrix 5 there are 9 unique column block sizes that can be chosen, therefore an index of 10 corresponds to the next row block size (i.e. 2048x1024). Due to the size of Matrices 12-14, the blocksize exploration started at 16k x 16k and went to the full size of the matrix.



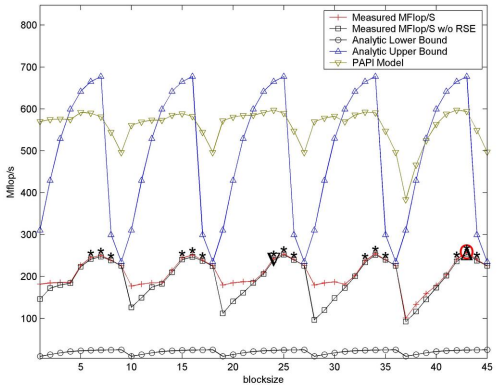

Matrix 1: dense2



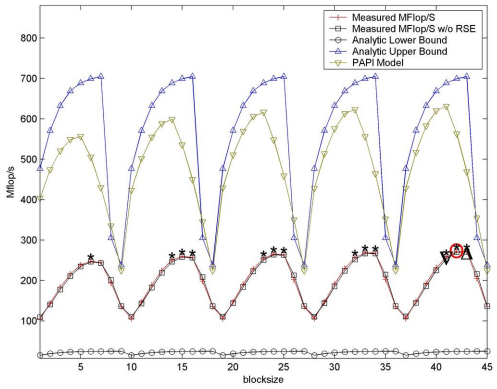

Matrix 2: bibd 22 8



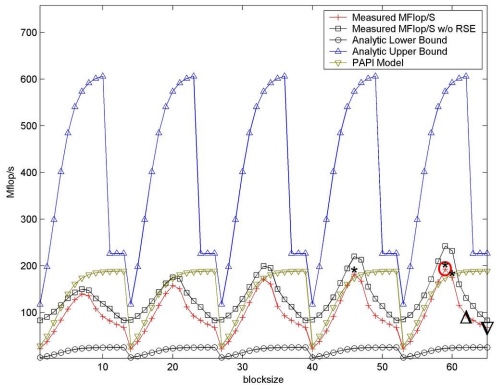

Matrix 3: lp nug30



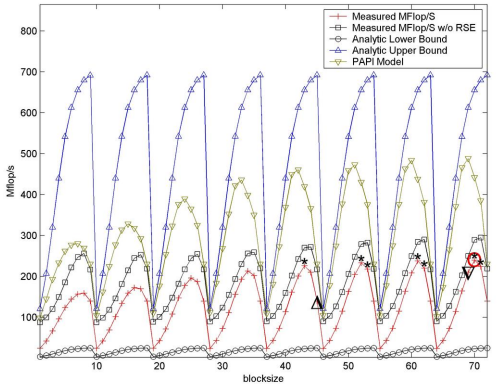

Matrix 4: lp osa 60



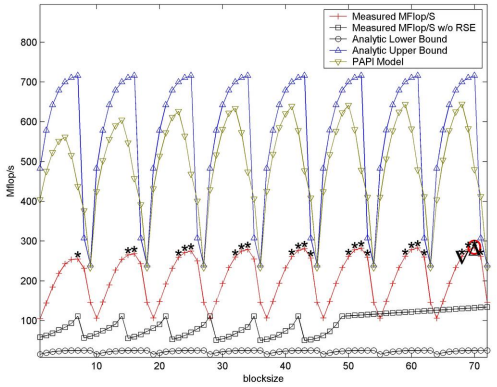

Matrix 5: lsi small



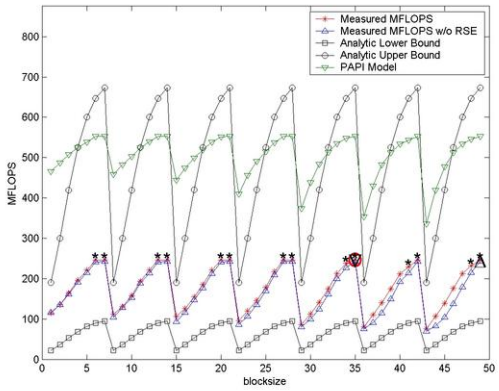

Matrix 6: lsi spread c



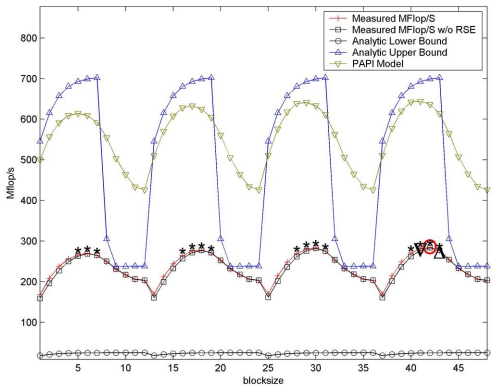

Matrix 7: lsi spread r



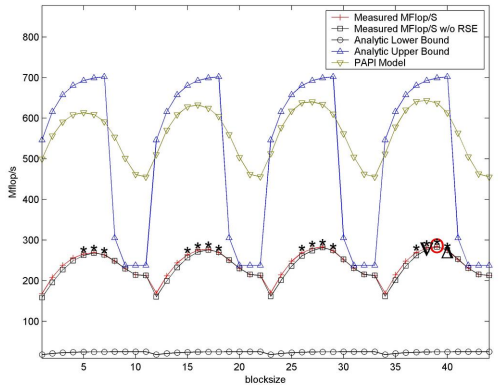

Matrix 8: lsi stamp r



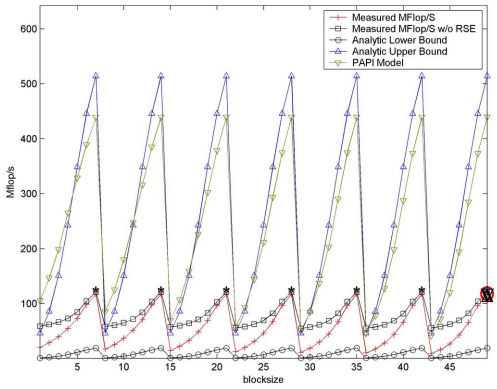

Matrix 9: marca mutex



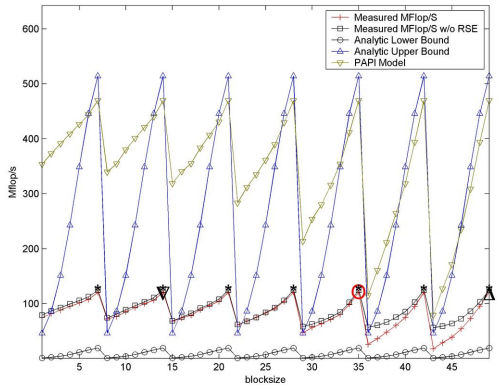

Matrix 10: rail4284



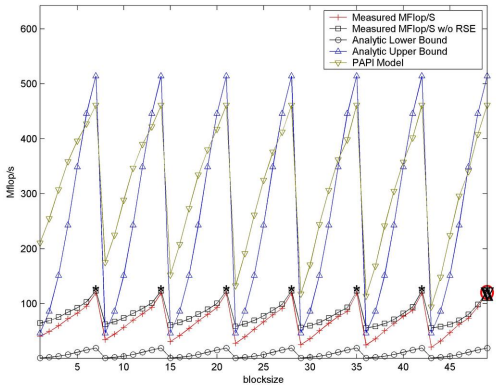

Matrix 11: rail4284s




Matrix 12: webbase-1m

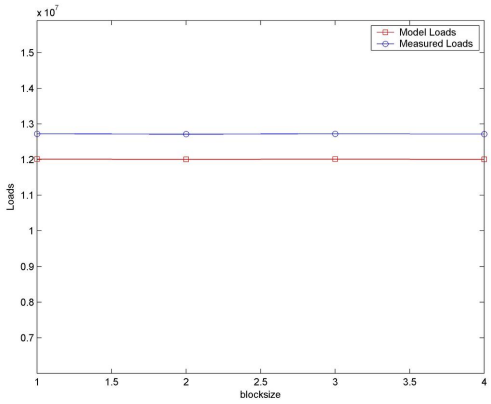



Matrix 13: webbase-1m-mmd

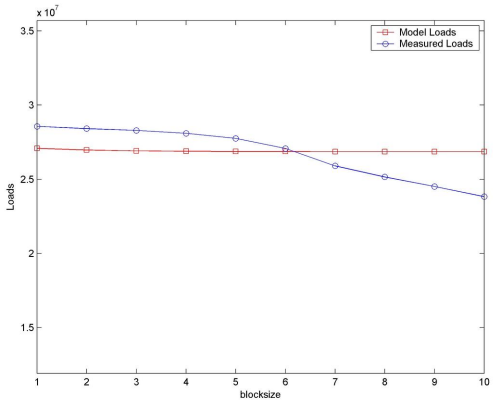



Matrix 14: webbase-1m-rcm

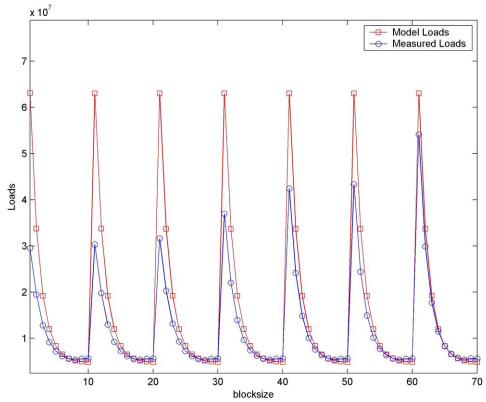
# C Itanium 2 Loads Plots



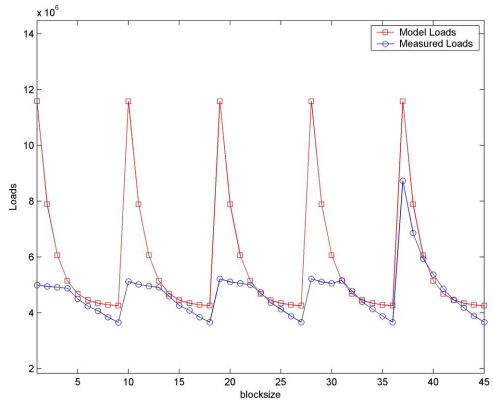

Matrix 1: dense2



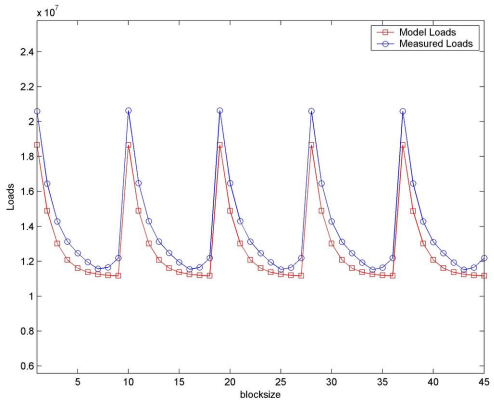

Matrix 2: bibd 22 8



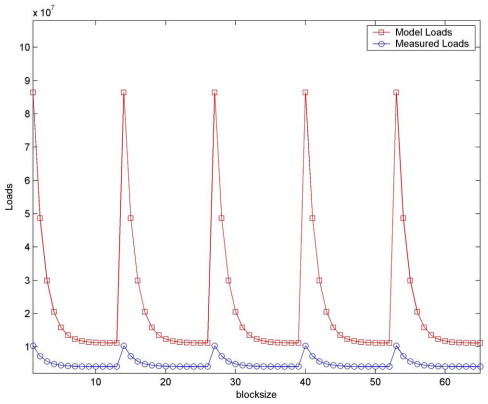

Matrix 3: lp nug30



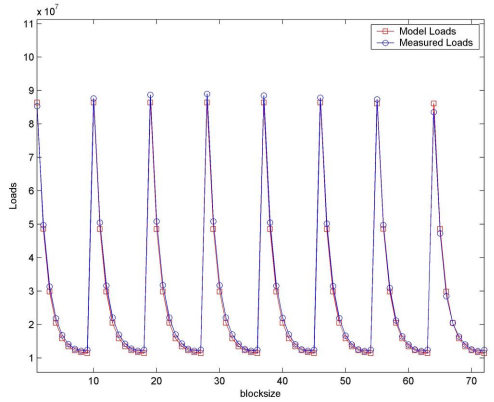

Matrix 4: lp osa 60



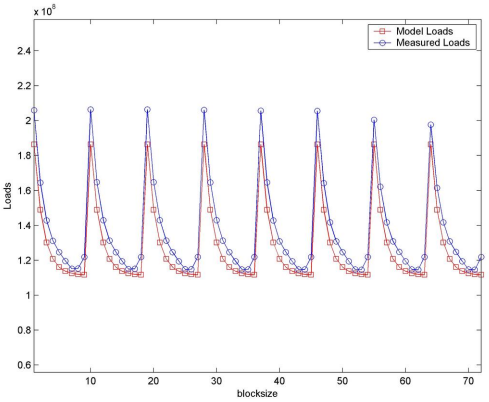

Matrix 5: lsi small



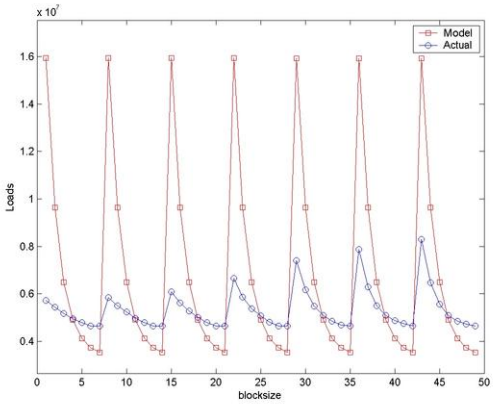

Matrix 6: lsi spread c



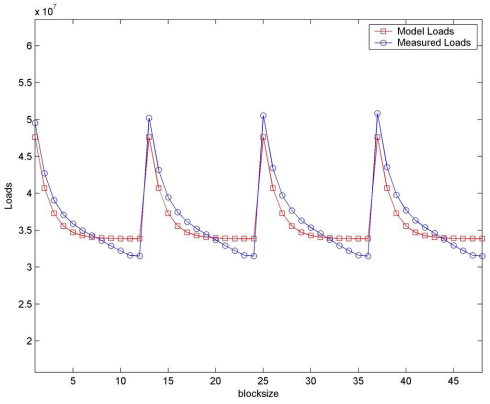

Matrix 7: lsi spread r



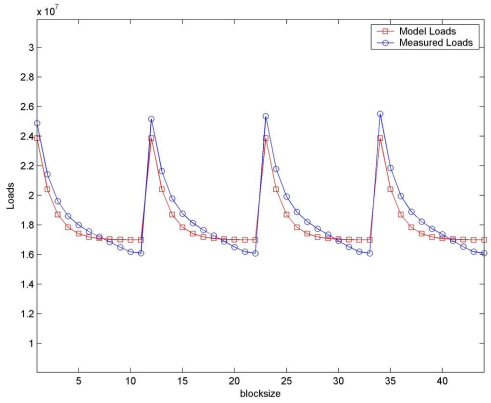

Matrix 8: lsi stamp r



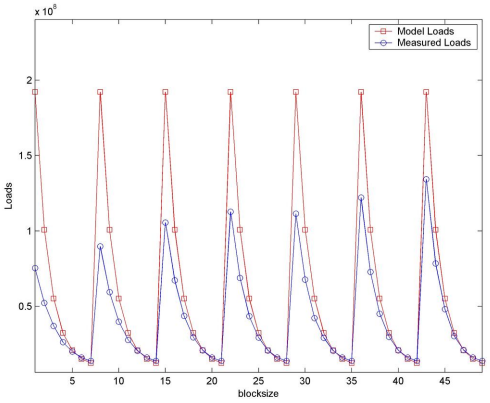

Matrix 9: marca mutex



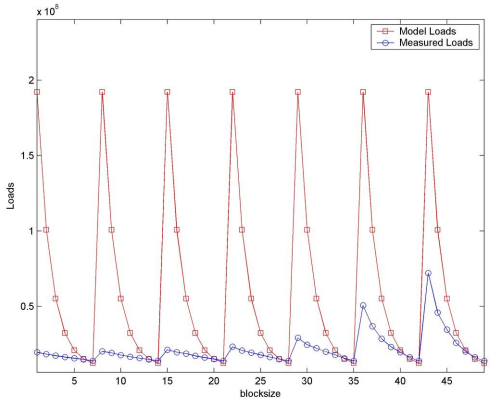

Matrix 10: rail4284



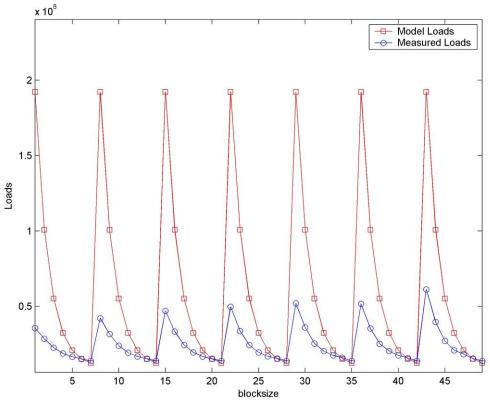

Matrix 11: rail4284s




Matrix 12: webbase-1m

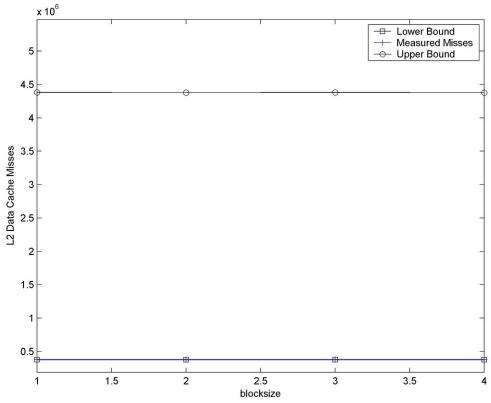



Matrix 13: webbase-1m-mmd

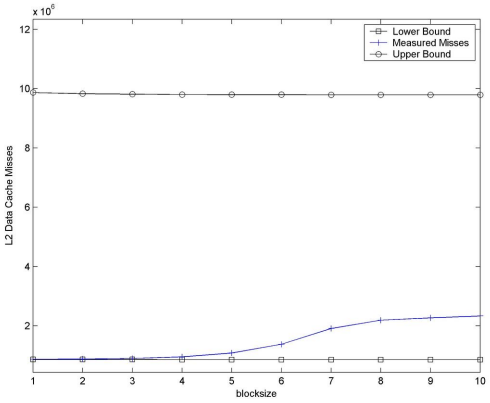



Matrix 14: webbase-1m-rcm

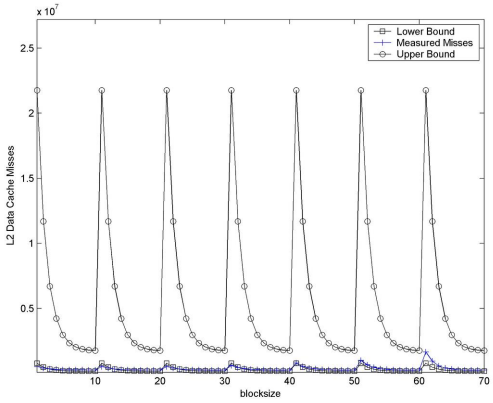
# D Itanium 2 L2 Cache Miss Plots



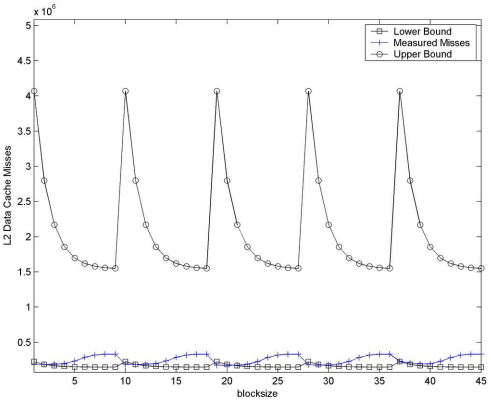

Matrix 1: dense2



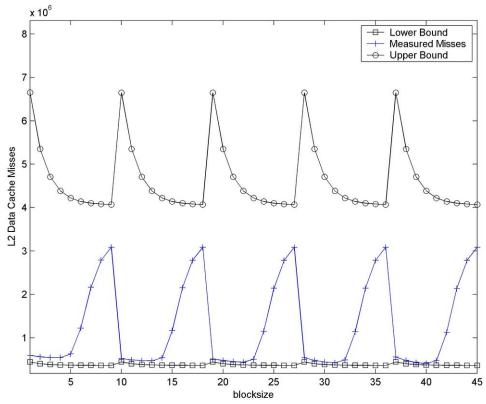

Matrix 2: bibd 22 8



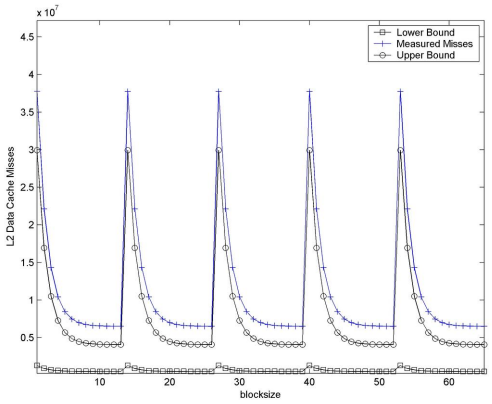

Matrix 3: lp nug30



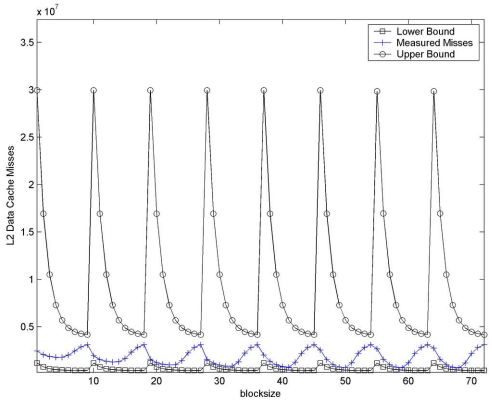

Matrix 4: lp osa 60



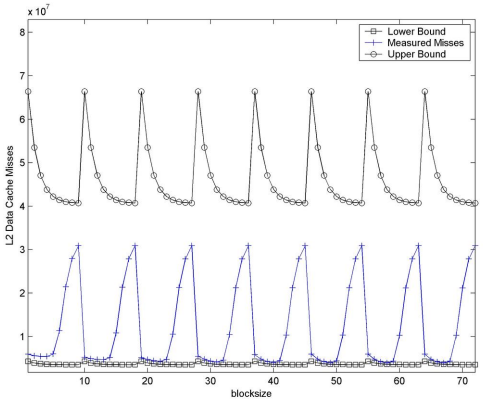

Matrix 5: lsi small



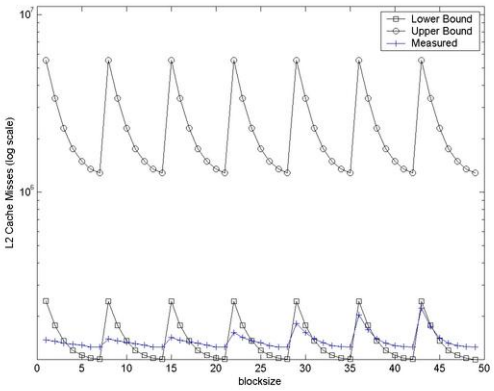

Matrix 6: lsi spread c



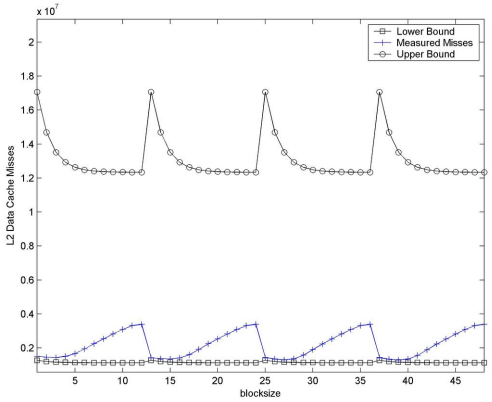

Matrix 7: lsi spread r



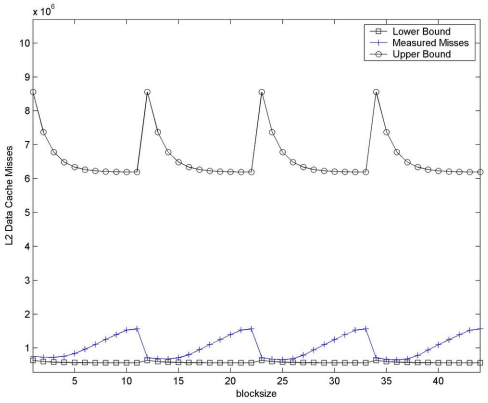

Matrix 8: lsi stamp r



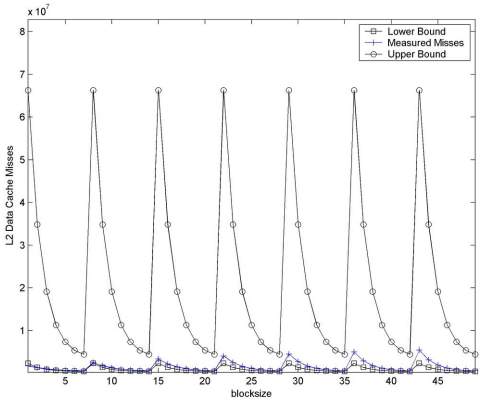

Matrix 9: marca mutex



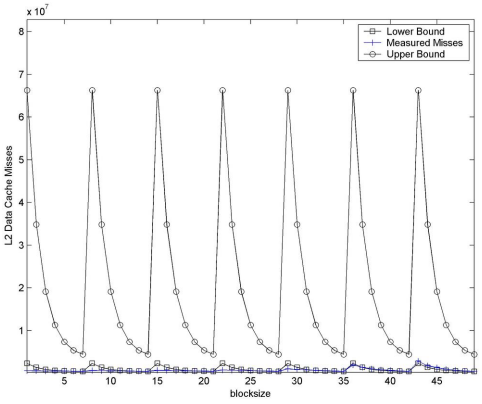

Matrix 10: rail4284



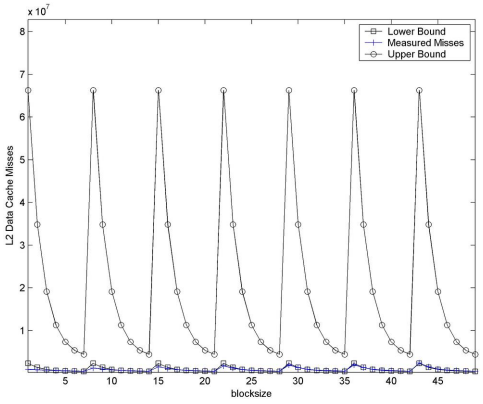

Matrix 11: rail4284s




Matrix 12: webbase-1m

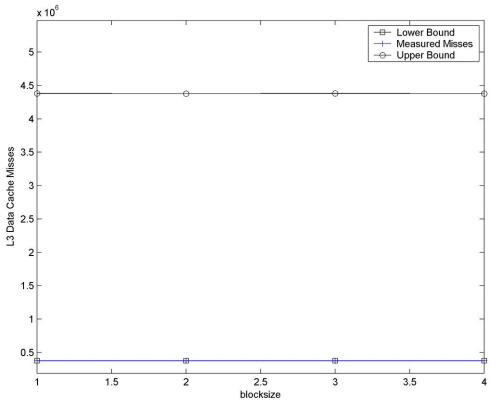



Matrix 13: webbase-1m-mmd

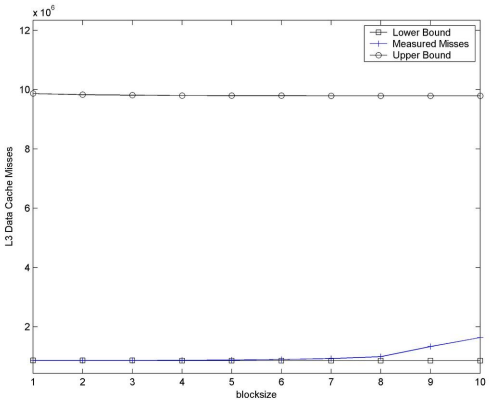



Matrix 14: webbase-1m-rcm

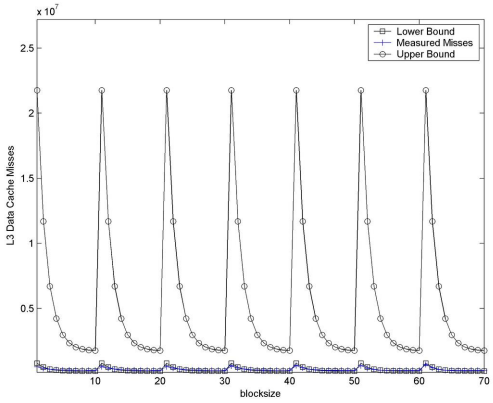
# E Itanium 2 L3 Cache Plots



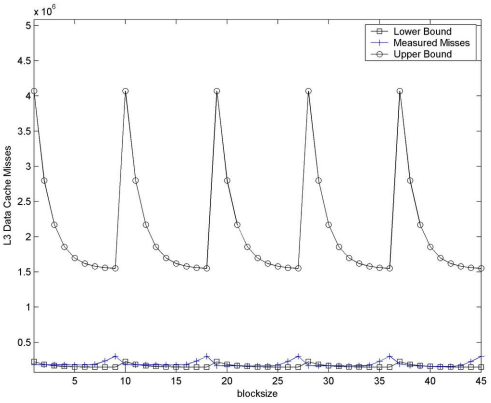

Matrix 1: dense2



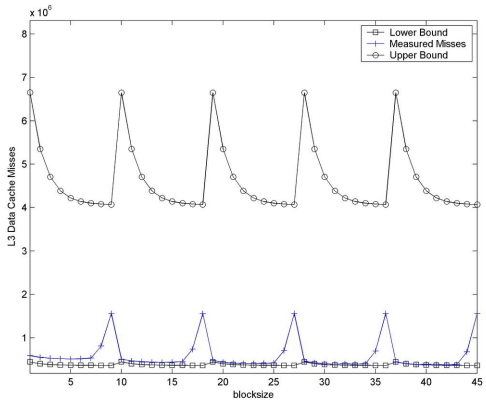

Matrix 2: bibd 22 8



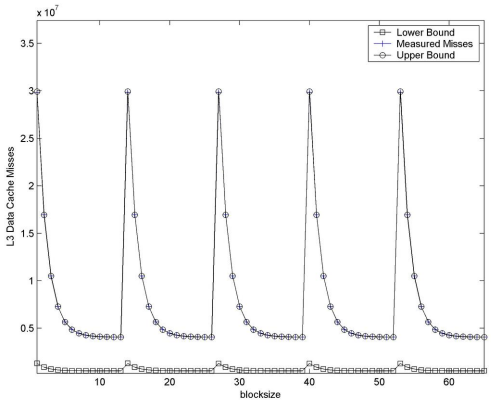

Matrix 3: lp nug30



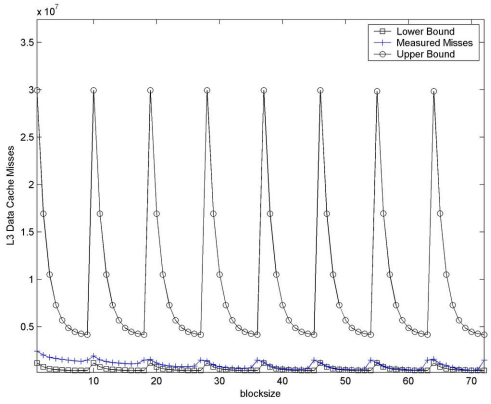

Matrix 4: lp osa 60



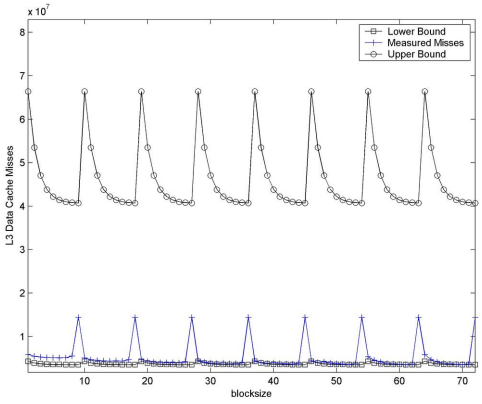

Matrix 5: lsi small



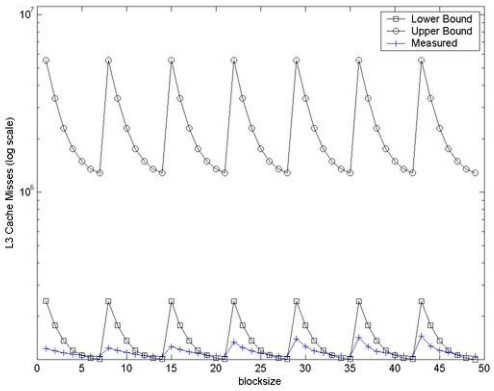

Matrix 6: lsi spread c



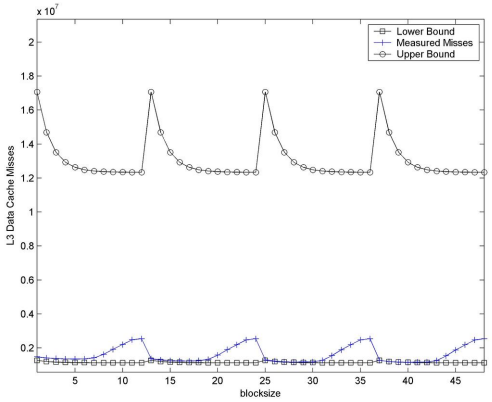

Matrix 7: lsi spread r



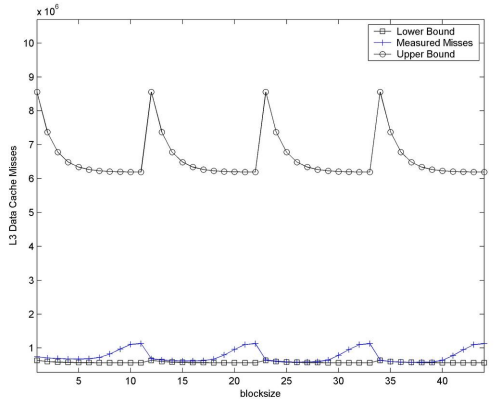

Matrix 8: lsi stamp r



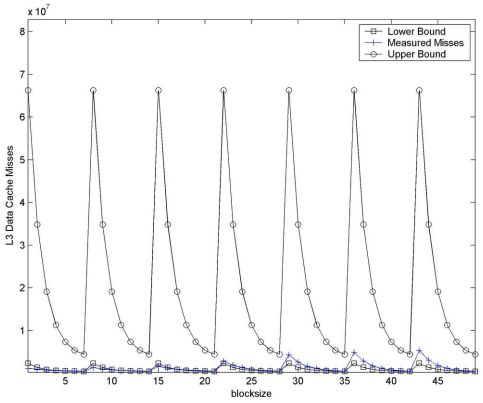

Matrix 9: marca mutex



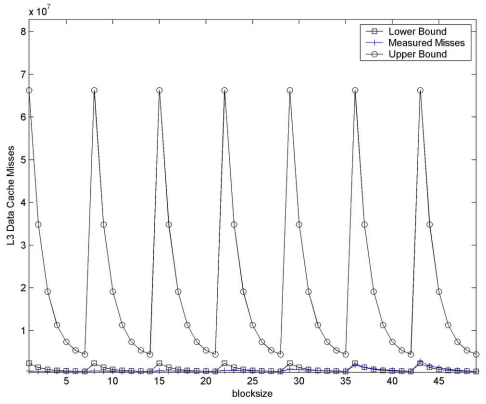

Matrix 10: rail4284



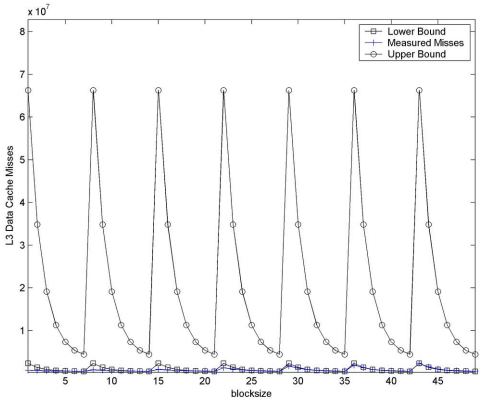

Matrix 11: rail4284s




Matrix 12: webbase-1m

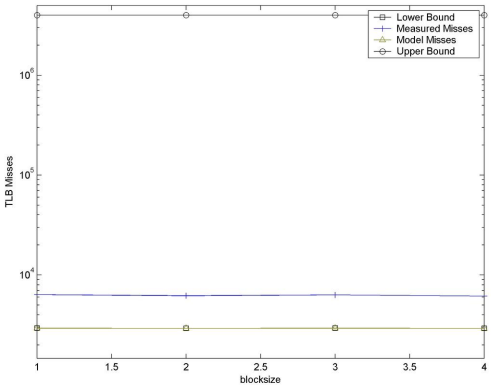



Matrix 13: webbase-1m-mmd

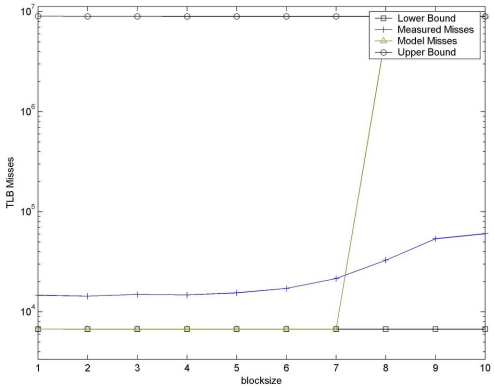



Matrix 14: webbase-1m-rcm

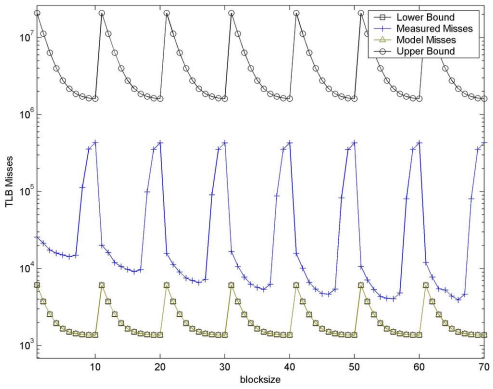
# F Itanium 2 TLB Plots



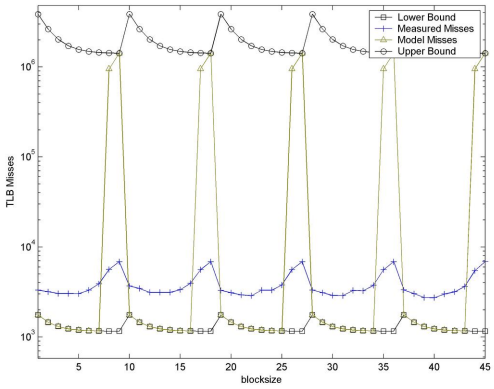

Matrix 1: dense2



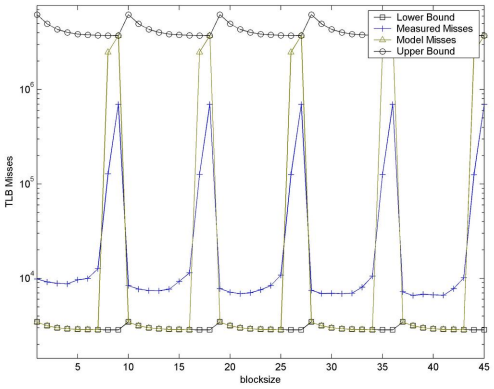

Matrix 2: bibd 22 8



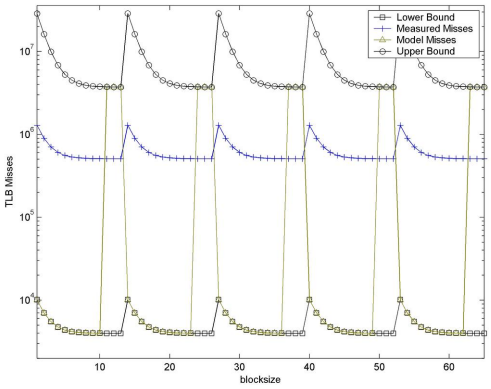

Matrix 3: lp nug30



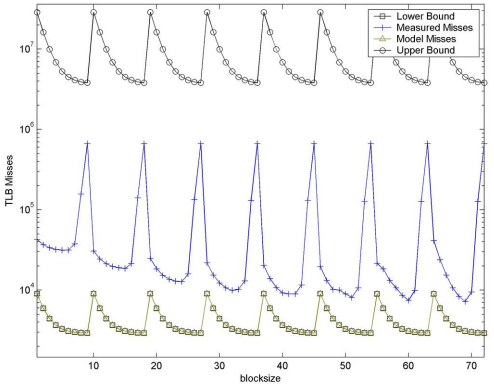

Matrix 4: lp osa 60



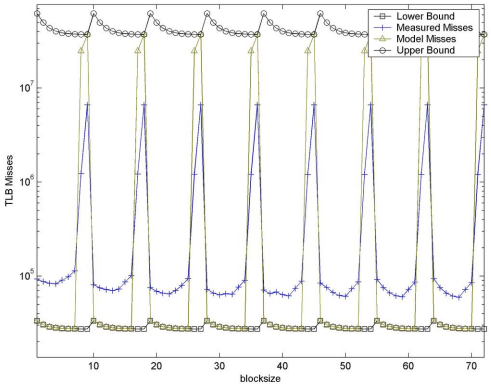

Matrix 5: lsi small



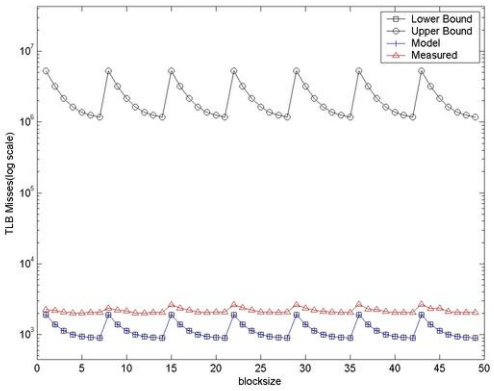

Matrix 6: lsi spread c



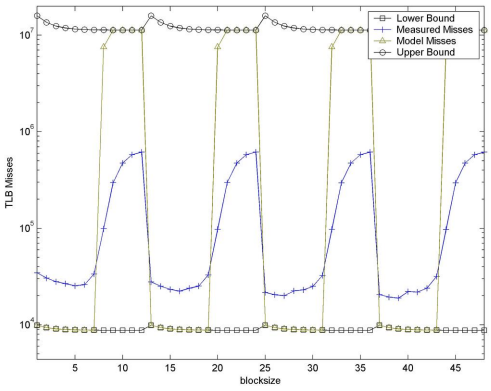

Matrix 7: lsi spread r



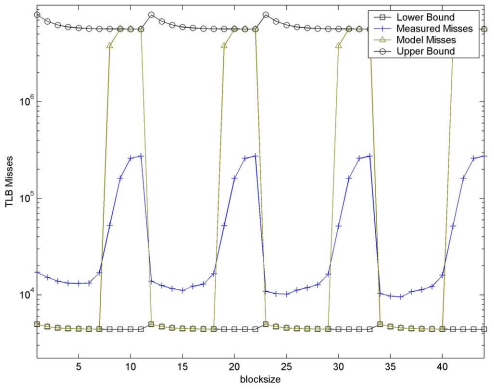

Matrix 8: lsi stamp r



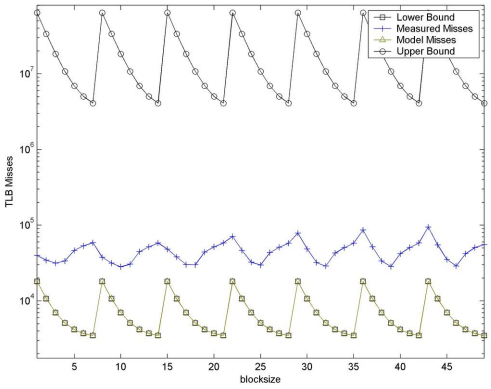

Matrix 9: marca mutex



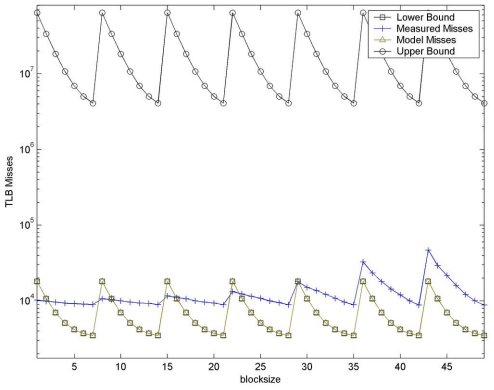

Matrix 10: rail4284



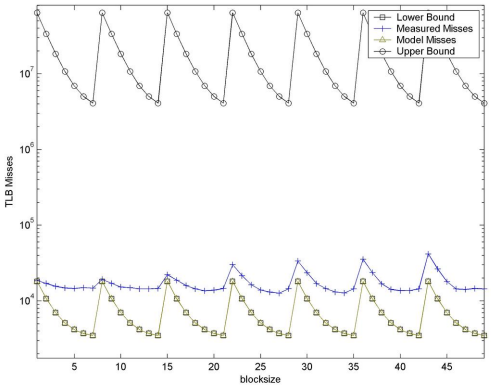

Matrix 11: rail4284s




Matrix 12: webbase-1m



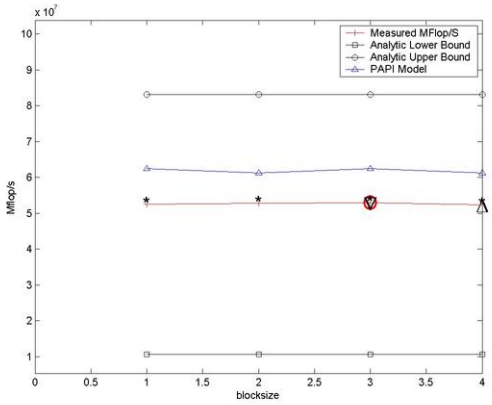

Matrix 13: webbase-1m-mmd



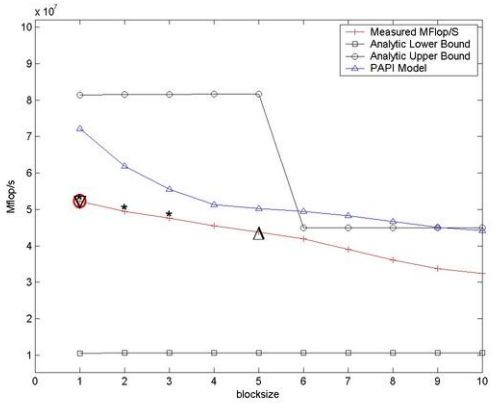

Matrix 14: webbase-1m-rcm

# G Pentium 3 Performance Plots

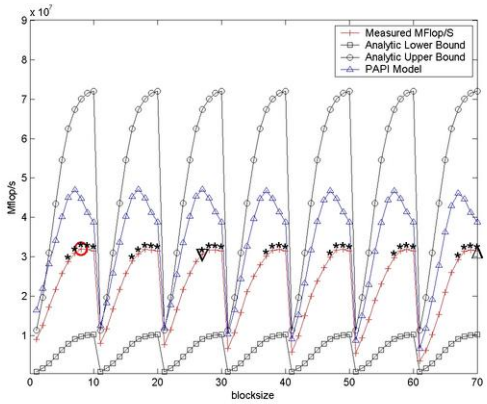
The following section shows the performacne at all the various blocksizes of all the different matrices. The description of the various data points is provided in Section B. Since the Row Start / Row End optimizations didn't prove to be that helpful with the matrices in our benchmark suite we do not present the data for the non RSE optimized cases on this machine.



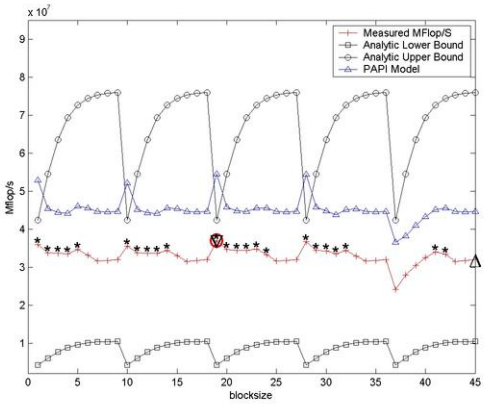

Matrix 1: dense2



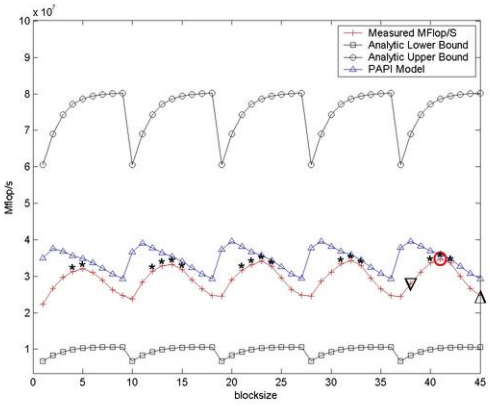

Matrix 2: bibd 22 8



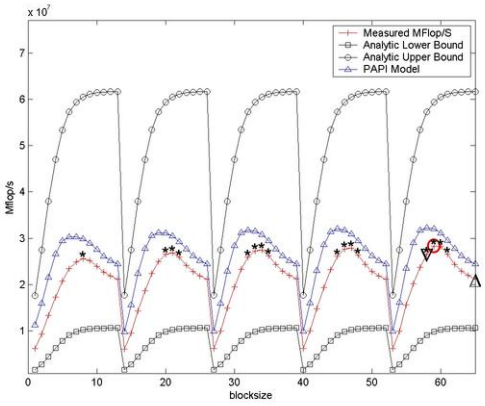

Matrix 3: lp nug30



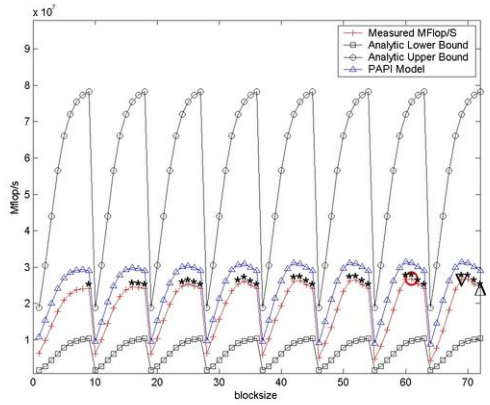

Matrix 4: lp osa 60




Matrix 5: lsi small

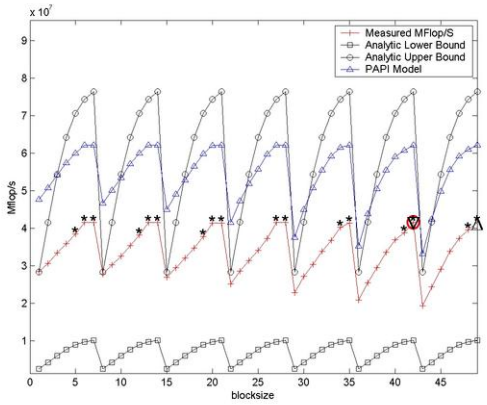



Matrix 6: lsi spread c

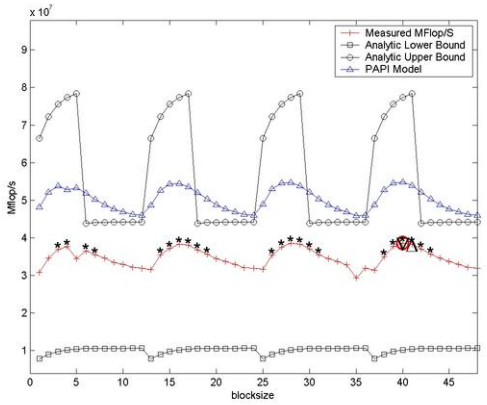



Matrix 7: lsi spread r

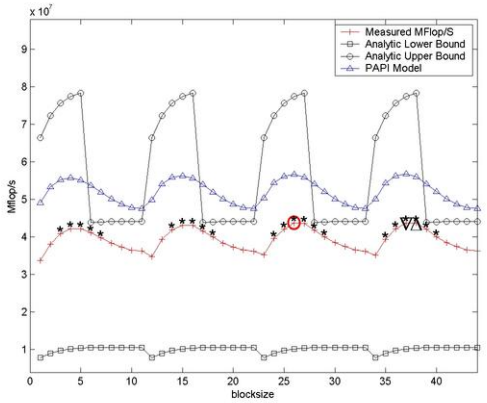
Matrix 8: lsi stamp r (not available)



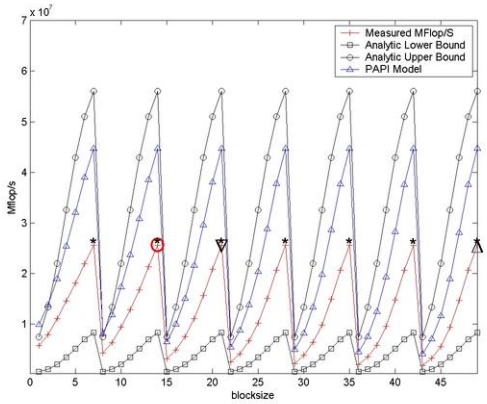

Matrix 9: marca mutex



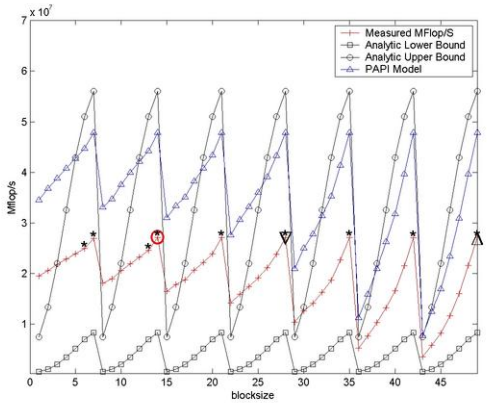

Matrix 10: rail4284



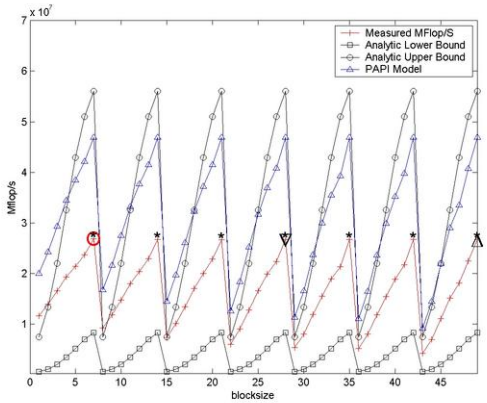

Matrix 11: rail4284s




Matrix 12: webbase-1m

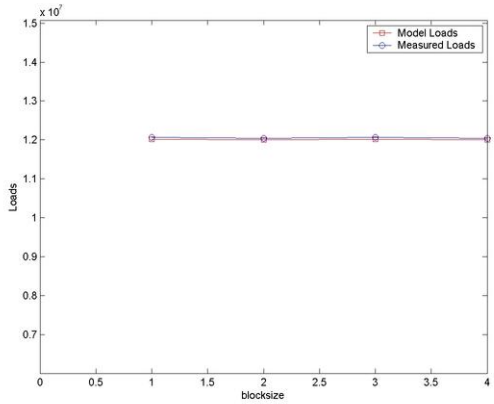



Matrix 13: webbase-1m-mmd

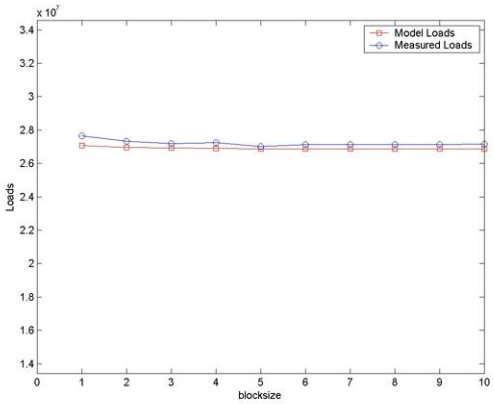



Matrix 14: webbase-1m-rcm

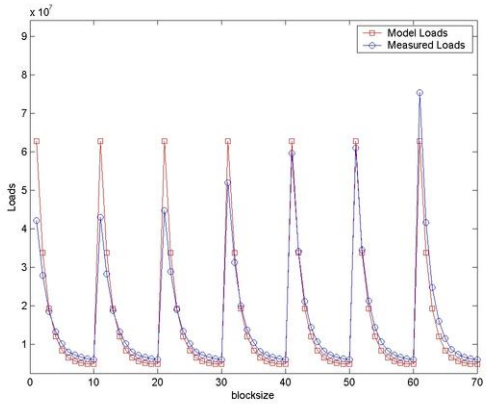
# H Pentium 3 Loads Plots



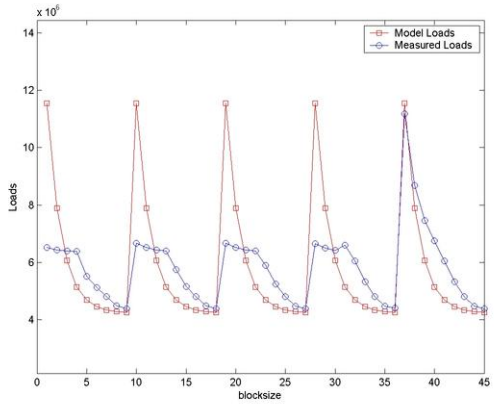

Matrix 1: dense2



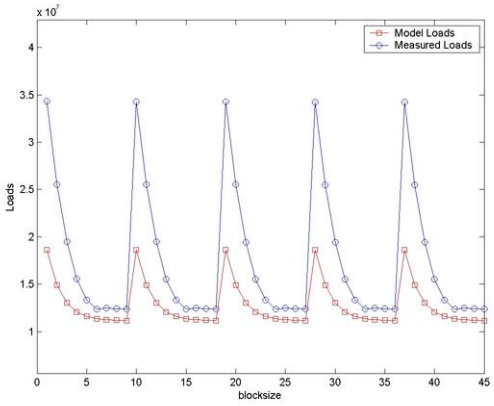

Matrix 2: bibd 22 8



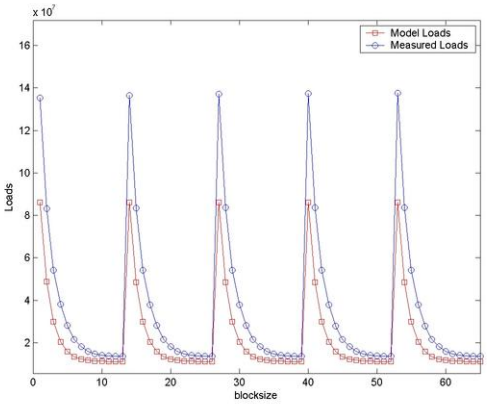

Matrix 3: lp nug30



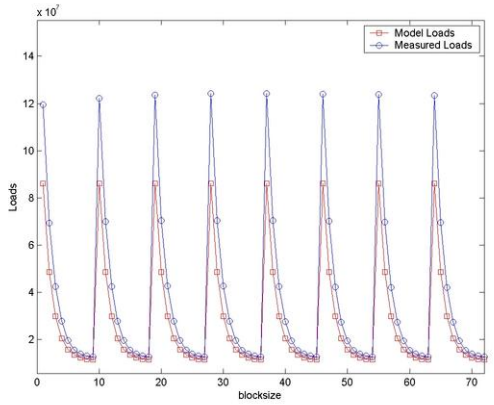

Matrix 4: lp osa 60




Matrix 5: lsi small

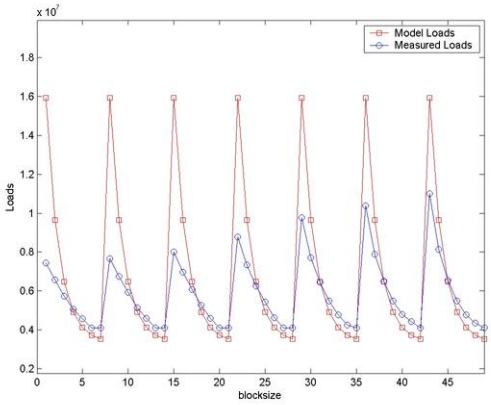



Matrix 6: lsi spread c

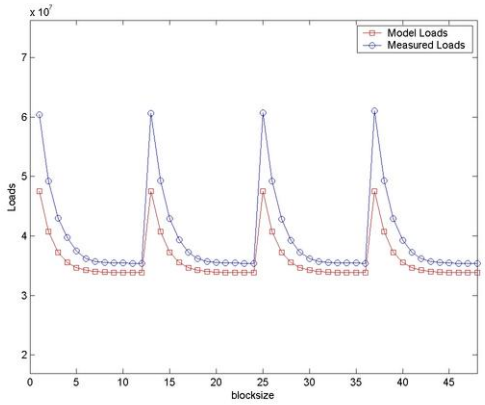



Matrix 7: lsi spread r

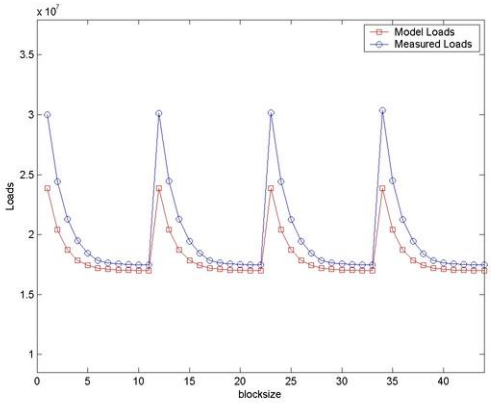
Matrix 8: lsi stamp r (not available)



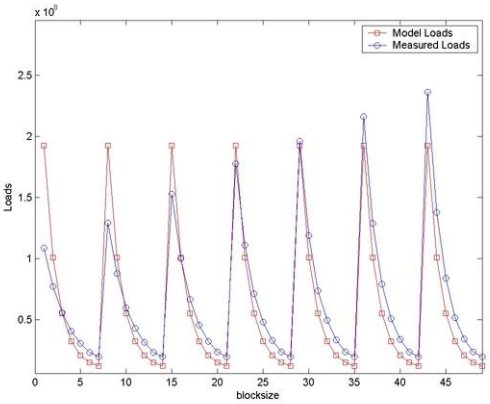

Matrix 9: marca mutex



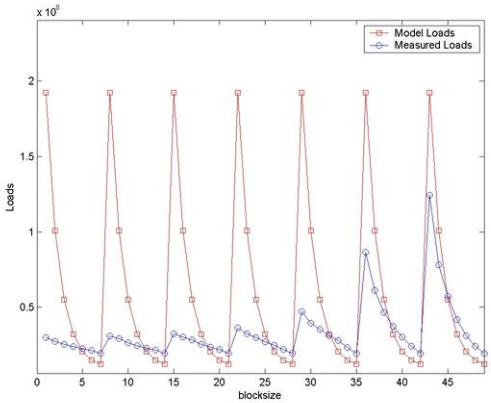

Matrix 10: rail4284



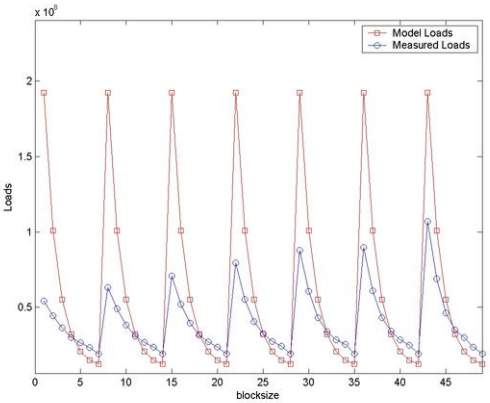

Matrix 11: rail4284s




Matrix 12: webbase-1m

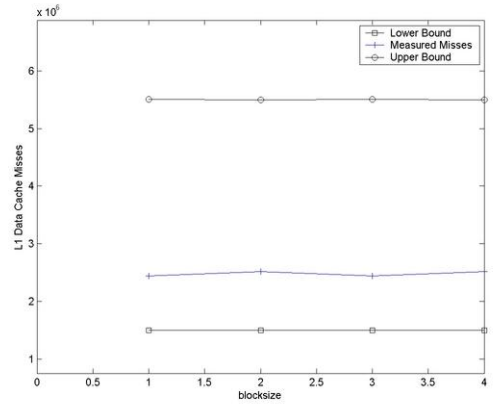



Matrix 13: webbase-1m-mmd

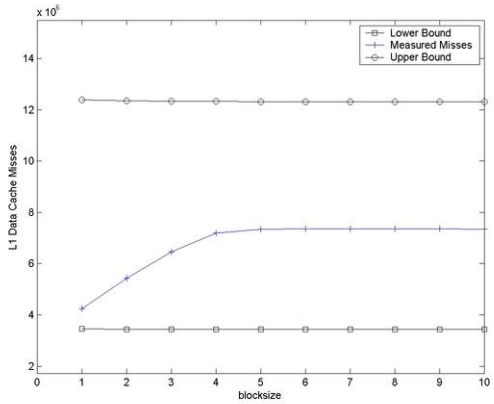



Matrix 14: webbase-1m-rcm

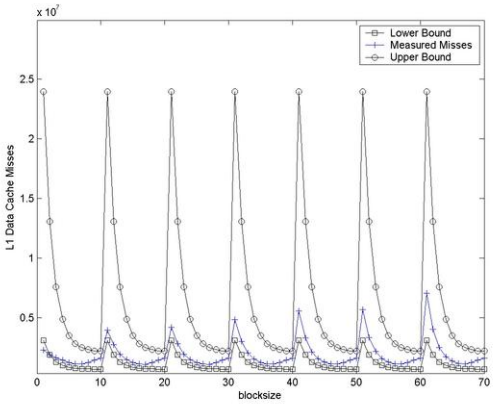
# I Pentium 3 L1 Cache Miss Plots



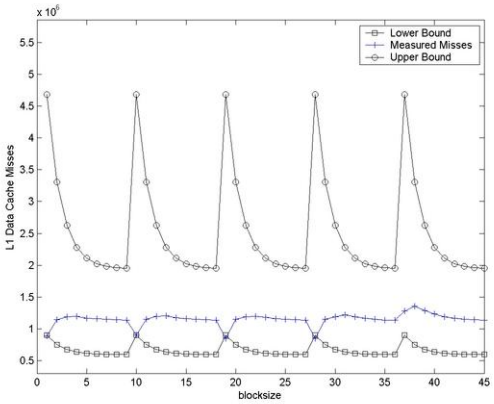

Matrix 1: dense2



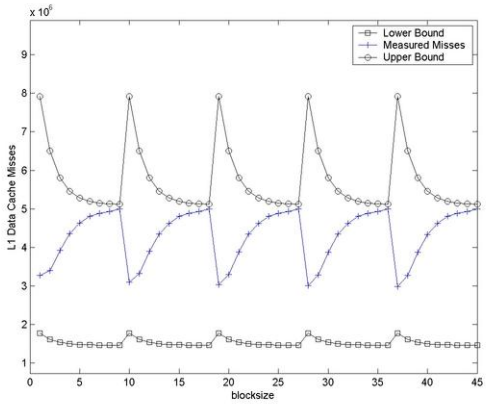

Matrix 2: bibd 22 8



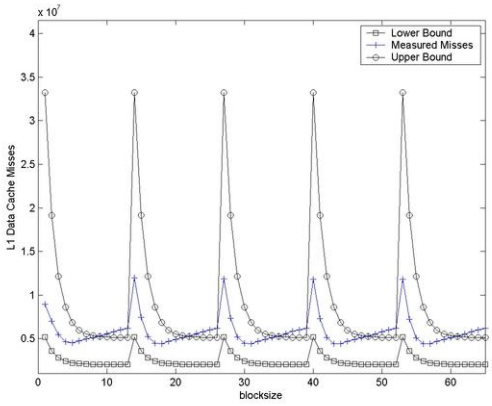

Matrix 3: lp nug30



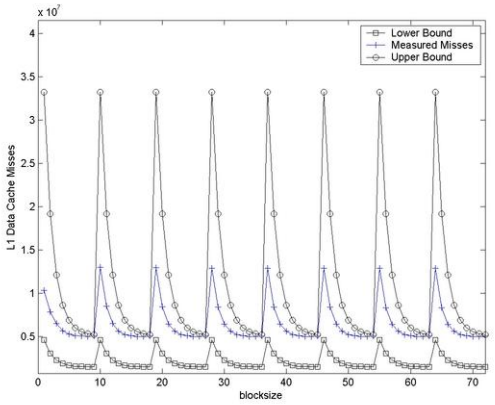

Matrix 4: lp osa 60




Matrix 5: lsi small

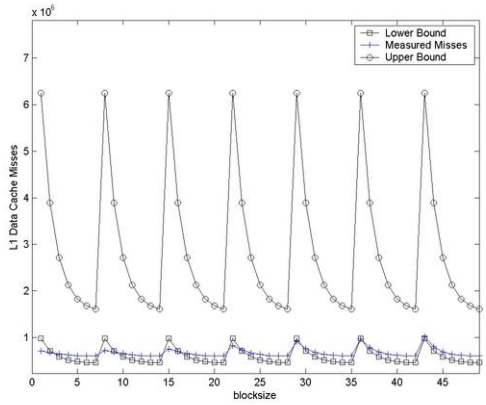



Matrix 6: lsi spread c

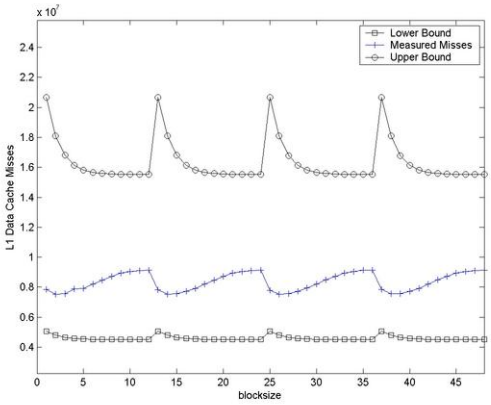



Matrix 7: lsi spread r

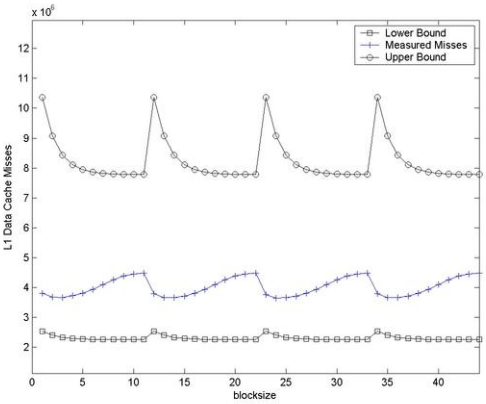
Matrix 8: lsi stamp r (not available)



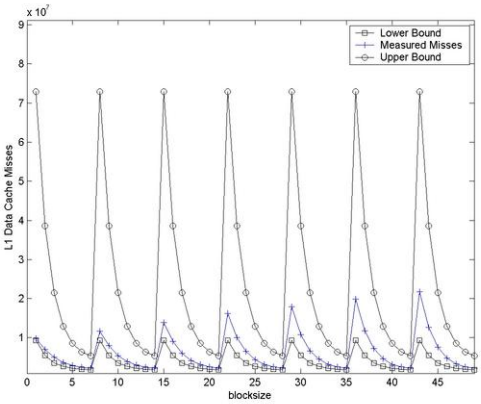

Matrix 9: marca mutex



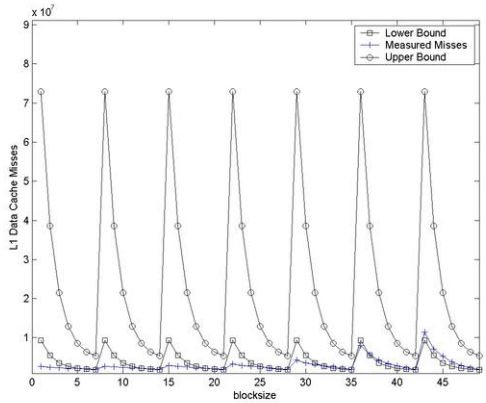

Matrix 10: rail4284



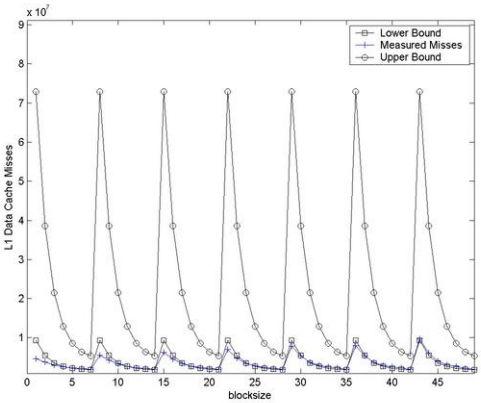

Matrix 11: rail4284s




Matrix 12: webbase-1m

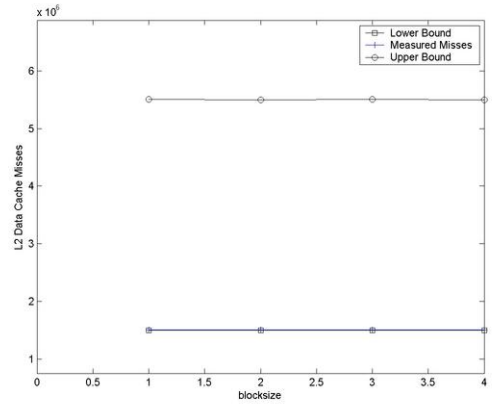



Matrix 13: webbase-1m-mmd

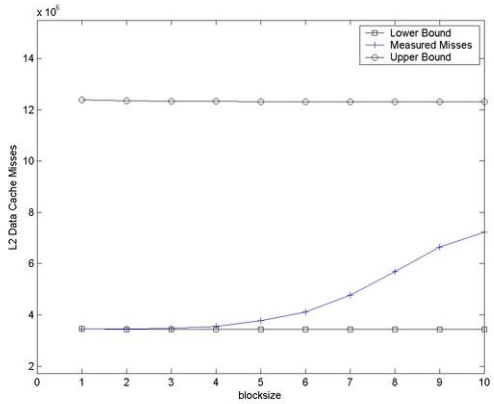



Matrix 14: webbase-1m-rcm

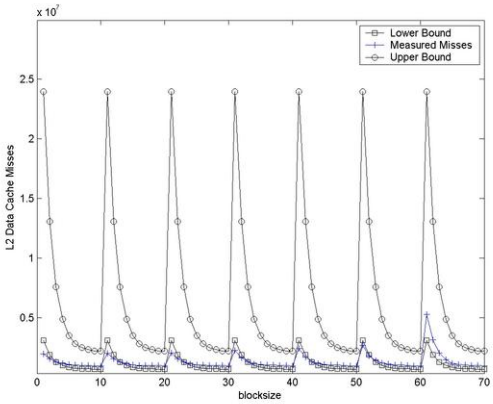
# J Pentium 3 L2 Cache Miss Plots



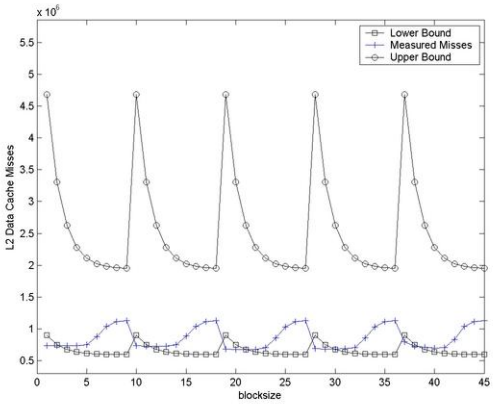

Matrix 1: dense2



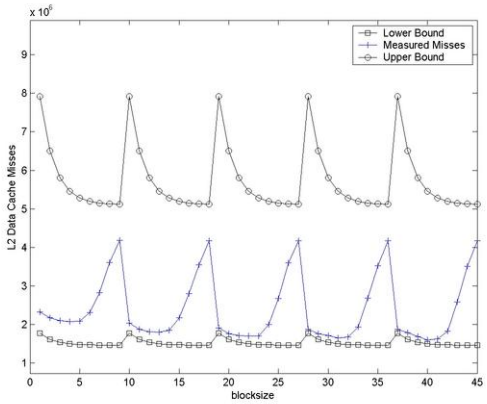

Matrix 2: bibd 22 8



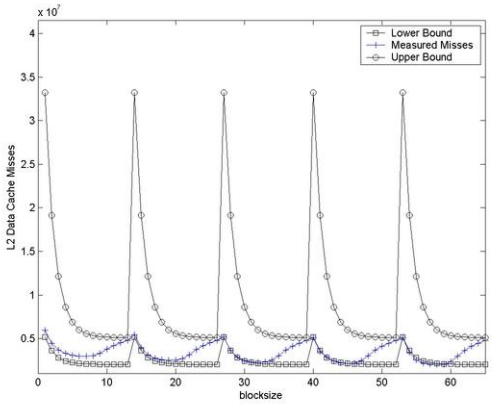

Matrix 3: lp nug30



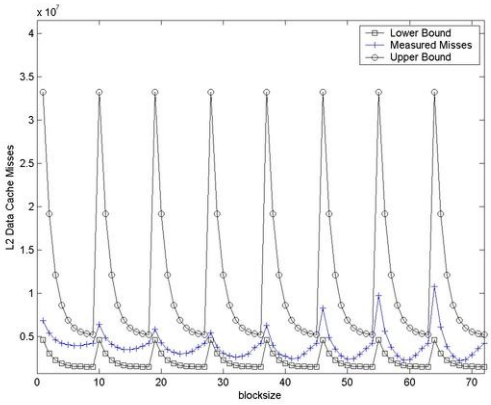

Matrix 4: lp osa 60




Matrix 5: lsi small

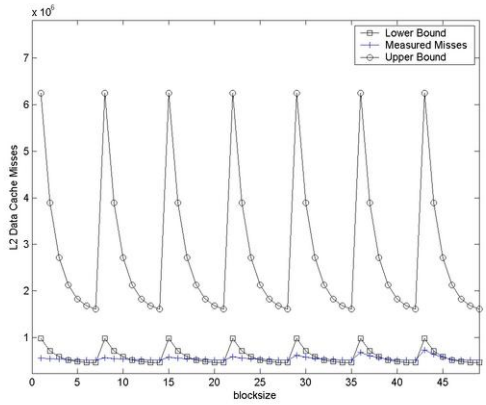



Matrix 6: lsi spread c

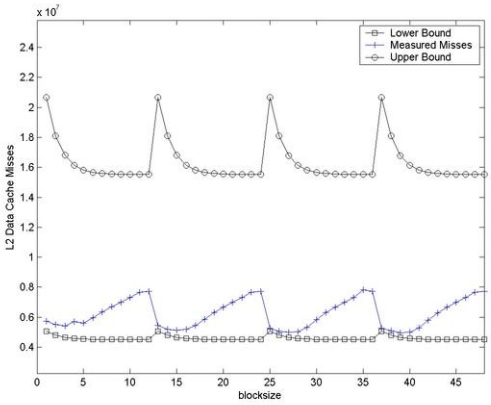



Matrix 7: lsi spread r

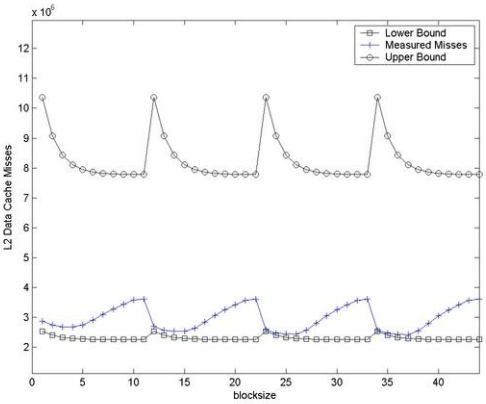
Matrix 8: lsi stamp r (not available)



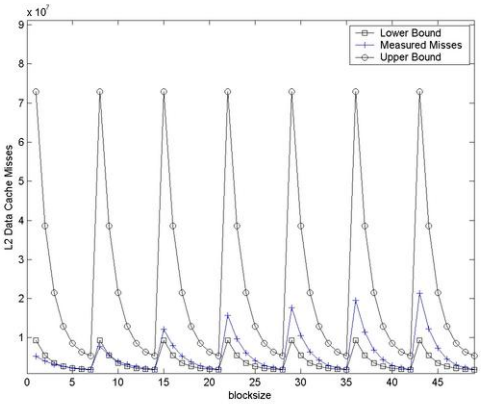

Matrix 9: marca mutex



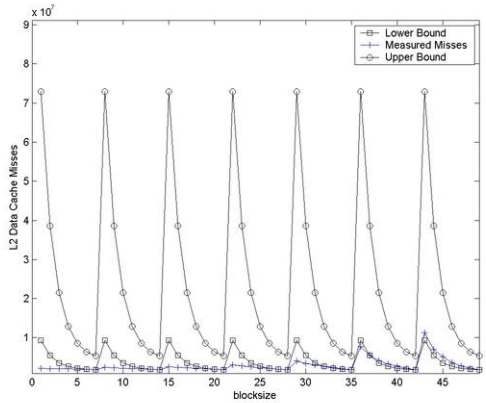

Matrix 10: rail4284



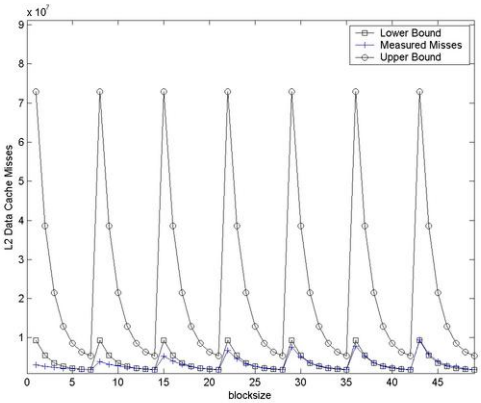

Matrix 11: rail4284s




Matrix 12: webbase-1m



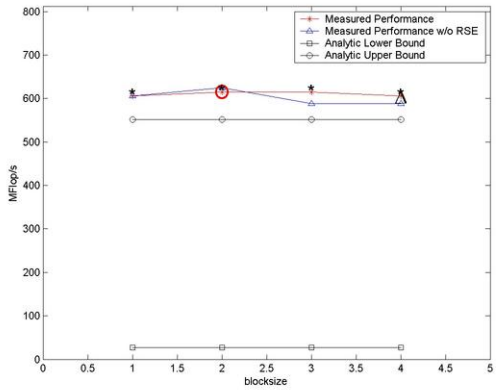

Matrix 13: webbase-1m-mmd



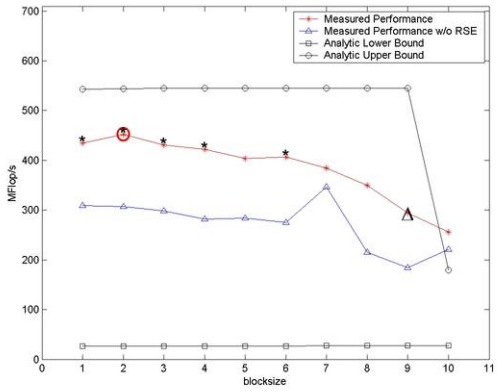

Matrix 14: webbase-1m-rcm

# K Power 4 Performance Plots

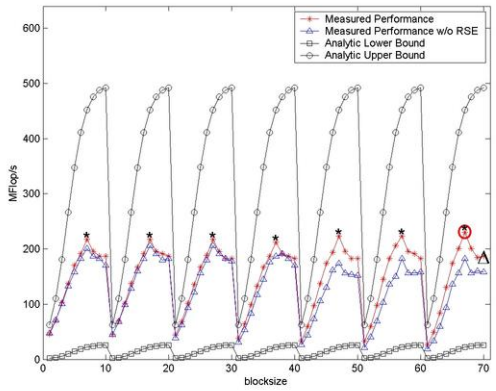
The labels on the plots are the same as descirbed in Section B.



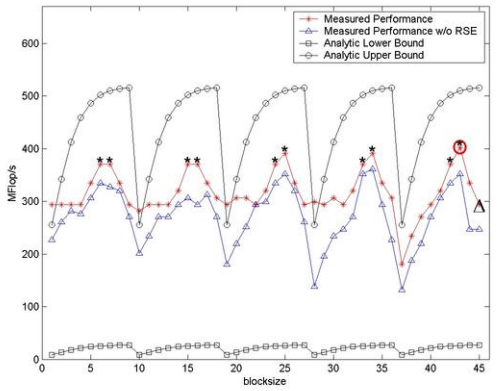

Matrix 1: dense2



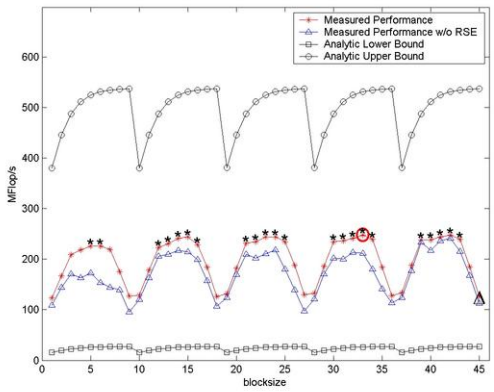

Matrix 2: bibd 22 8



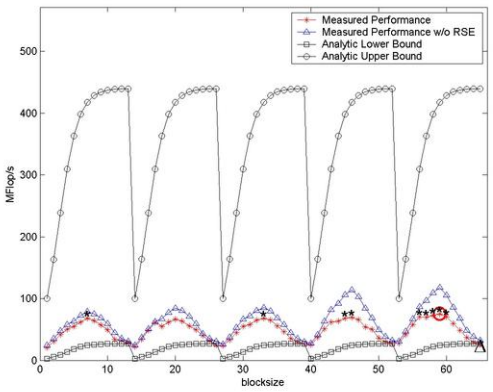

Matrix 3: lp nug30



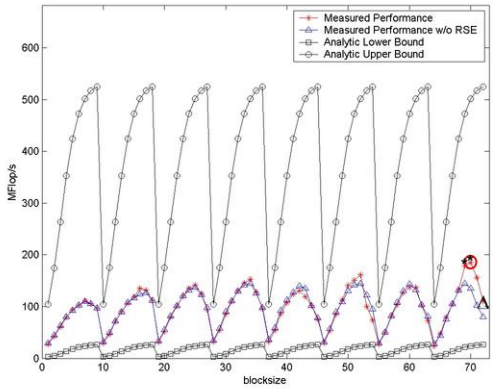

Matrix 4: lp osa 60




Matrix 5: lsi small

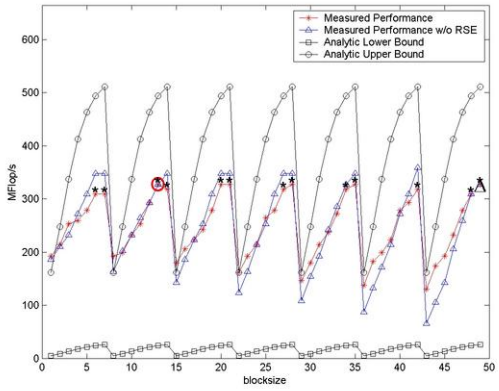



Matrix 6: lsi spread c

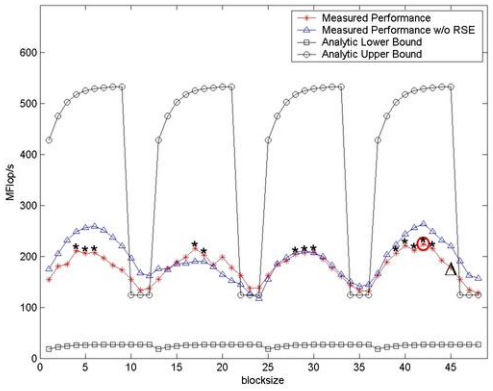



Matrix 7: lsi spread r

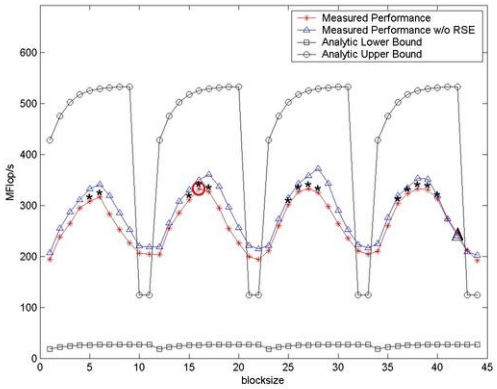
Matrix 8: lsi stamp r (not available)



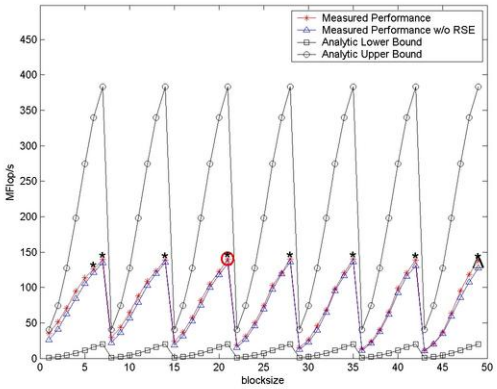

Matrix 9: marca mutex



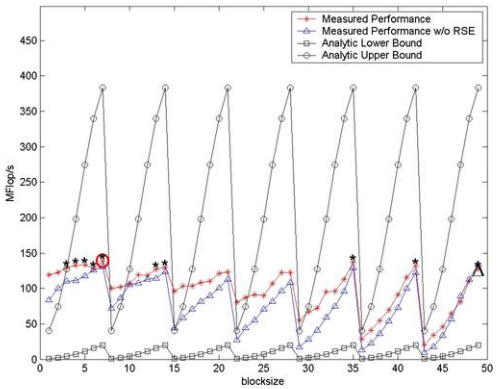

Matrix 10: rail4284



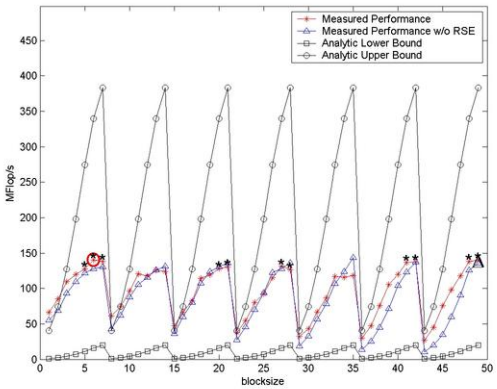

Matrix 11: rail4284s




Matrix 12: webbase-1m

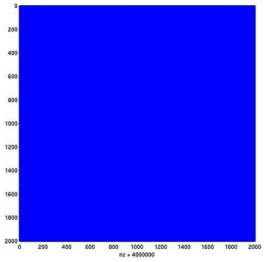



Matrix 13: webbase-1m-mmd



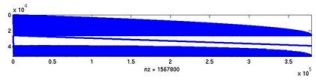

Matrix 14: webbase-1m-rcm

# L Spy Plots

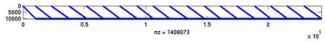


Matrix 1: dense2

Matrix 2: bibd 22 8



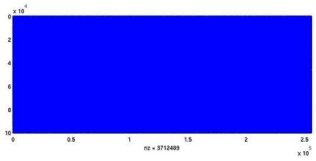
Matrix 3: lp nug30



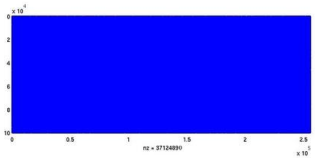
Matrix 4: lp osa 60

Matrix 5: lsi small

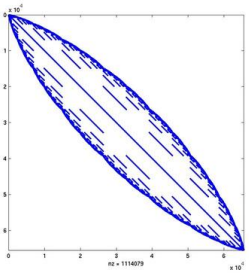
Matrix 6: lsi spread c



Matrix 7: lsi spread r

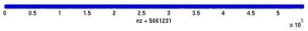


Matrix 8: lsi stamp r

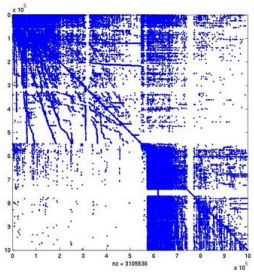


Matrix 9: marca mutex

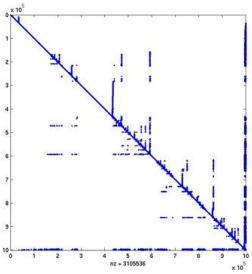
Matrix 10: rail4284



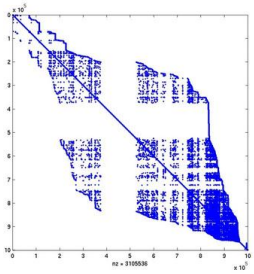
Matrix 11: rail4284s



Matrix 12: webbase-1m

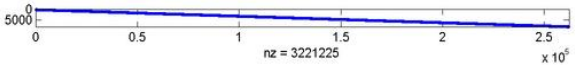


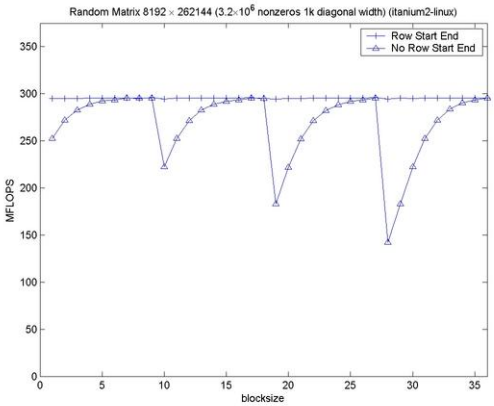
Matrix 13: webbase-1m-mmd



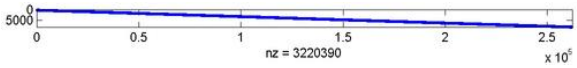
Matrix 14: webbase-1m-rcm

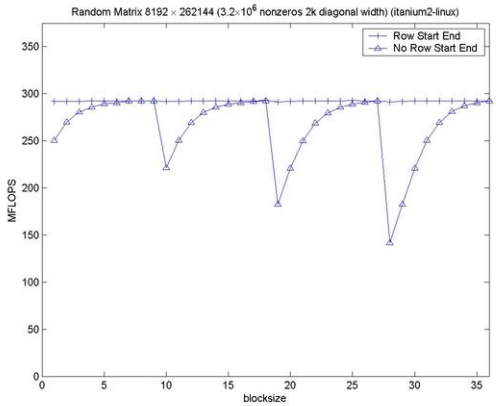
# M Random Diagonal Matrices



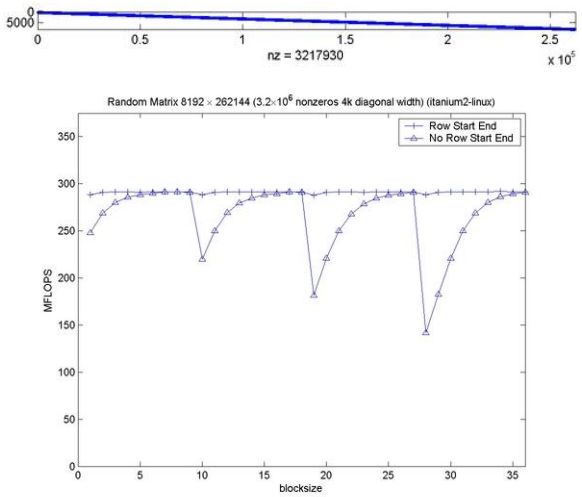




Diagonal Width = 1024

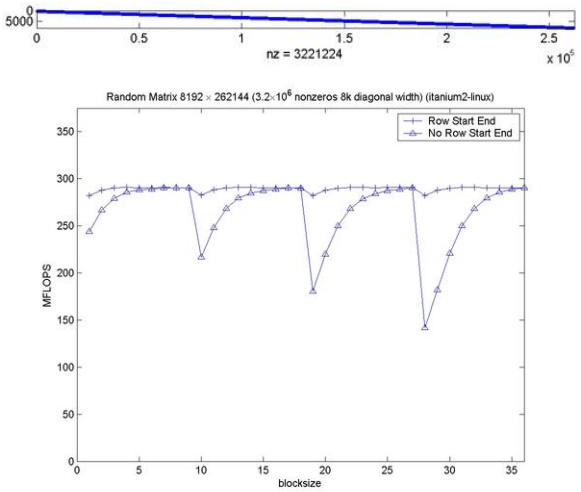


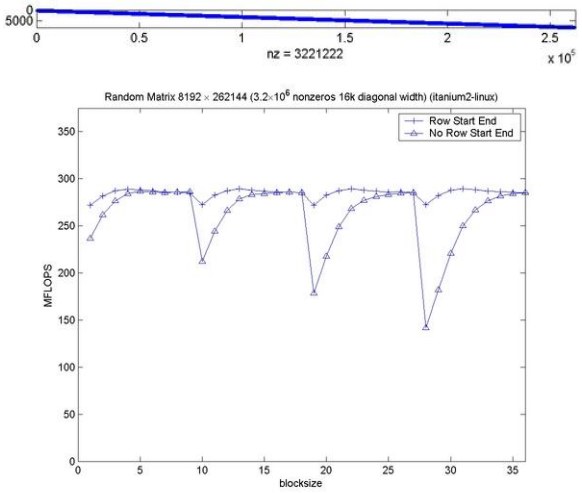

Diagonal Width = 2048



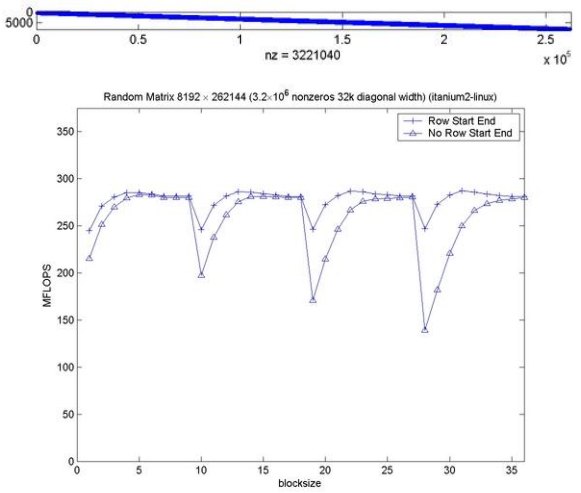

Diagonal Width = 4096



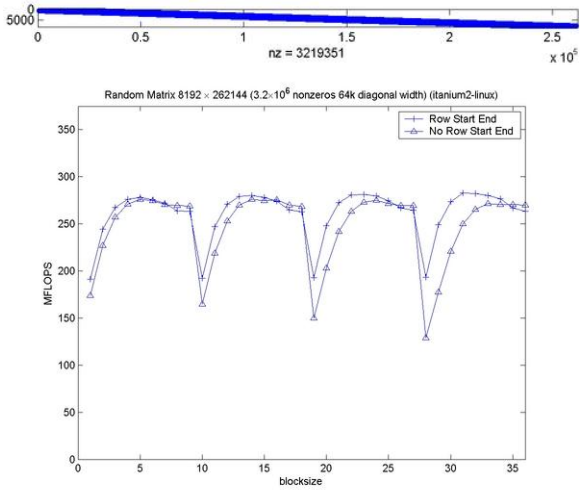

Diagonal Width = 8192



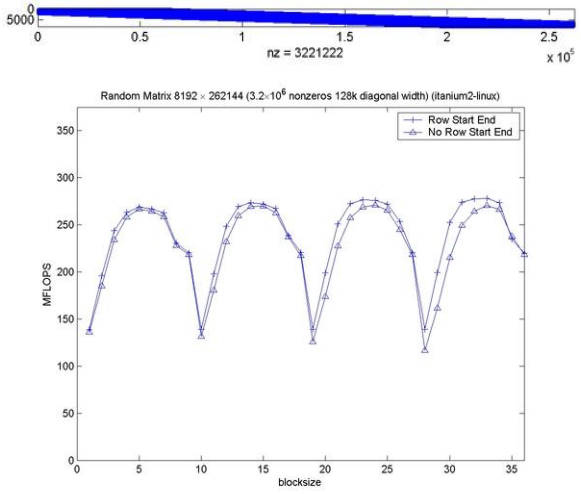

Diagonal Width = 16384



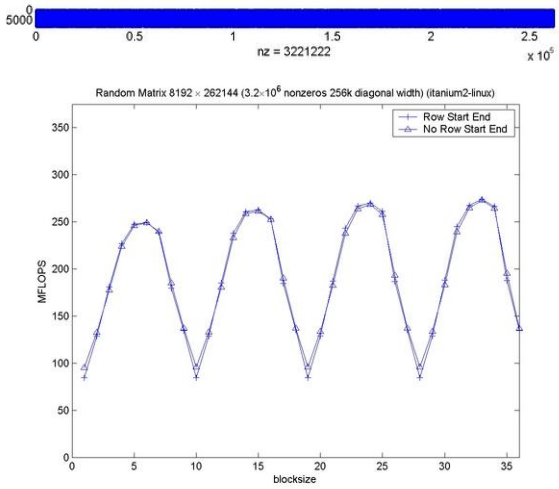

Diagonal Width = 32786




Diagonal Width = 65536




Diagonal Width = 131072




Diagonal Width = 262144