Memory Hierarchy Optimizations and Performance Bounds for Sparse $A^T Ax$

Richard Vuduc *Attila Gyulassy* *James W. Demmel*
Katherine A. Yelick

Report No. UCB/CSD-03-1232

February 2003

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

Abstract

This report presents uniprocessor automatic tuning techniques for the sparse matrix operation, $y = A^T Ax$, where A is a sparse matrix and x, y are dense vectors. We describe an implementation of this computational kernel which brings A through the memory hierarchy only once, and which can be combined naturally with the register blocking optimization previously proposed in the Sparsity tuning system for sparse matrix-vector multiply (SpM×V). Extensive experiments, on a benchmark set of 44 matrices and 4 platforms, show that speedups of up to 4.2× are possible compared to a conventional implementation that computes $t = Ax$ and $y = A^T t$ as separate steps. In addition, we develop platform-specific upper-bounds on the performance of our implementations. We analyze how closely our implementations approach these bounds, and show when low-level tuning techniques (*e.g.*, better instruction scheduling) are likely to yield a significant pay-off. Finally, we present a hybrid off-line/run-time heuristic which in practice automatically selects optimal (or near-optimal) values of the key tuning parameters, the register block sizes.

There are at least three implications of this work. First, sparse $A^T Ax$ should be a basic primitive in sparse matrix libraries, based on its utility to applications and the potential pay-off from automatically tuning it. Second, our upper bound analysis shows that there is an opportunity to apply automatic low-level tuning methods, in the spirit of tuning systems such as ATLAS and PHiPAC for dense linear algebra, to further improve the performance of this kernel. Third, the success of our heuristic provides additional validation of the Sparsity tuning methodology.

1 Introduction

This report considers automatic performance tuning of the sparse matrix operation, $y \leftarrow y + A^T Ax$, where A is a sparse matrix and x, y are dense vectors. This computational kernel[1]—Sp$A^T A$ hereafter—is the inner-loop of interior point methods for mathematical programming [28] problems, algorithms for computing the singular value decomposition [9], and Kleinberg's HITS algorithm for finding hubs and authorities in graphs [18], among others. The challenge in tuning this kernel, and sparse kernels in general, is choosing a sparse data structure and algorithm that best exploits the non-zero structure of the matrix for a given memory hierarchy and microarchitecture: this task can be daunting and time-consuming because the best implementation

[1]We restrict our attention to $A^T Ax$ here, though the same ideas apply to the computation of $AA^T x$.

will vary across machines, compilers, and matrices. A purely static (compile-time) approach to this problem is limited since the matrix may not be known until run-time; any run-time approach must balance the cost optimizing at run-time against the potential performance gain.

Our approach to automatic tuning of SpA^TA builds on prior experience with dense linear algebra [4, 29], sparse matrix-vector multiply (SpM$\times$V) [15, 16, 26], and sparse triangular solve (SpTS) [27]. In this study, we apply the specific tuning methodology first proposed for the Sparsity system for SpM$\times$V [15]. We show how SpA^TA can be algorithmically cache-blocked to reuse A in a way that also allows register-level blocking to exploit dense subblocks (Section 2). The set of these implementations, parameterized by block size, defines an *implementation space*. We search this space by first benchmarking these implementations on a synethetic matrix *off-line* (*i.e.*, *once* per platform), and then predicting the best block size by evaluating a heuristic performance model that combines an estimated property of the matrix non-zero structure with the benchmark data.

Our experiments on four hardware platforms (Table 1) and 44 sparse matrices (Table 2) show that we can obtain speedups between 1.5×–4.2× over a reference implementation which computes $t = Ax$ and $y = A^Tt$ as separate steps. Furthermore, if each of these reference steps is tuned by register-level blocking with an optimal choice of block size, our implementation is still up to 1.8× faster. We also show that our search heuristic nearly always chooses optimal or near-optimal tuning parameters (*i.e.*, yielding performance within 5–10% of the best).

We evaluate our SpA^TA performance relative to fundamental limits—or, upper bounds—on performance (Section 3). We have used similar bounds for SpM$\times$V to show that the performance (Mflop/s) of Sparsity-generated code is often within 20% of the upper bound, implying that the benefits of additional low-level tuning (*e.g.*, better instruction scheduling) will be limited [26]. Indeed, this result suggested that one area from which further performance improvements would have to come is higher-level kernels, like SpA^TA, that can reuse the matrix. In this report, we derive upper bounds on the performance of our SpA^TA implementations. We show that our implementations typically achieve between 20%–40% of this bound. Since we rely on the compiler to schedule our fully unrolled code, this finding suggests that the additional cache-reuse due to our algorithmic blocking of SpA^TA exposes new opportunities for low-level tuning; *i.e.*, future work should apply automatic low-level tuning methods, in the spirit of ATLAS/PHiPAC, to further improve the performance of this kernel.

Property	Sun Ultra 2i	Intel Pentium III	IBM Power3	Intel Itanium
Clock rate (MHz)	333	500	375	800
Peak Main Memory Bandwidth (MB/s)	500	680	1600	2100
Peak Flop Rate (Mflop/s)	667	500	1500	3200
DGEMM, $n = 1000$ (Mflop/s)	425	331	1300	2200
DGEMV, $n = 2000$ (Mflop/s)	58	96	260	315
STREAM Triad Bandwidth (MB/s)	250	350	715	1100
No. of FP regs (double)	16	8	32	128
L1 size (KB), line size (B), latency (cy)	16,16,1	16,32,1	64,128,.5	16,32,1 (int)
L2 size (KB), line size (B), latency (cy)	2048,64,7	512,32,18	8192,128,9	96,64,6-9
L3 size (KB), line size (B), latency (cy)	n/a	n/a	n/a	2048,64,21-24
Memory latency (cycles, $\approx$)	36-66 cy	26–60	35-139 cy	36-85 cy
Compiler	Sun C v6.1	Intel C v6.0	IBM C v5.0	Intel C v6.0

Table 1: **Characteristics of the evaluation platforms [17, 1, 24]**. Memory access latencies were measured using the Saavedra microbenchmark [22]. Dense BLAS numbers are reported for ATLAS 3.4.1 [29] on the Ultra 2i and Pentium III, IBM ESSL v3.1 on the Power3, and the Intel Math Kernel Library v5.2 on Itanium.

2 Memory Hierarchy Optimizations for Sparse A^TAx

We assume a baseline implementation of $y = A^TAx$ that first computes $t = Ax$ followed by $y = A^Tt$. For large matrices A, this implementation brings A through the memory hierarchy twice. However, we can compute A^TAx by reading A from main memory only once. Denote the rows of A by $a_1^T, a_2^T, \ldots, a_m^T$. Then, the operation A^TAx can be expressed algorithmically as follows:

$$y = A^TAx = (a_1 \ldots a_m) \begin{pmatrix} a_1^T \\ \ldots \\ a_m^T \end{pmatrix} x = \sum_{i=1}^{m} a_i(a_i^T x). \qquad (1)$$

That is, for each row a_i^T, we can compute the dot product $t_i = a_i^T x$, followed by an accumulation of the scaled vector $t_i a_i$ into y—thus, the row a_i^T is read from memory into cache to compute the dot product, assuming sufficient cache capacity, and then reused on the accumulate step.

Moreover, we can take each a_i^T to be a *block of rows* instead of just a single row. Doing so allows us to combine the cache optimization of Equation

	Name	Application Area	Dimension	Nonzeros
1	dense1000	Dense Matrix	1000	1000000
2	raefsky3	Fluid structure interaction	21200	1488768
3	olafu	Accuracy problem	16146	1015156
4	bcsstk35	Stiff matrix automobile frame	30237	1450163
5	venkat01	Flow simulation	62424	1717792
6	crystk02	FEM Crystal free vibration	13965	968583
7	crystk03	FEM Crystal free vibration	24696	1751178
8	nasasrb	Shuttle rocket booster	54870	2677324
9	3dtube	3-D pressure tube	45330	3213332
10	ct20stif	CT20 Engine block	52329	2698463
11	bai	Airfoil eigenvalue calculation	23560	484256
12	raefsky4	buckling problem	19779	1328611
13	ex11	3D steady flow caculation	16614	1096948
14	rdist1	Chemical process separation	4134	94408
15	vavasis3	2D PDE problem	41092	1683902
16	orani678	Economic modeling	2529	90185
17	rim	FEM fluid mechanics problem	22560	1014951
18	memplus	Circuit Simulation	17758	126150
19	gemat11	Power flow	4929	33185
20	lhr10	Light hydrocarbon recovery	10672	232633
21	goodwin	Fluid mechanics problem	7320	324784
22	bayer02	Chemical process simulation	13935	63679
23	bayer10	Chemical process simulation	13436	94926
24	coater2	Simulation of coating flows	9540	207308
25	finan512	Financial portfolio optimization	74752	596992
26	onetone2	Harmonic balance method	36057	227628
27	pwt	Structural engineering problem	36519	326107
28	vibrobox	Structure of vibroacoustic problem	12328	342828
29	wang4	Semiconductor device simulation	26068	177196
30	lnsp3937	Fluid flow modeling	3937	25407
31	lns3937	Fluid flow modeling	3937	25407
32	sherman5	Oil reservoir modeling	3312	20793
33	sherman3	Oil reservoir modeling	5005	20033
34	orsreg1	Oil reservoir simulation	2205	14133
35	saylr4	Oil reservoir modeling	3564	22316
36	shyy161	Viscous flow calculation	76480	329762
37	wang3	Semiconductor device simulation	26064	177168
38	mcfe	astrophysics	765	24382
39	jpwh991	Circuit physics modeling	991	6027
40	gupta1	Linear programming matrix	31802	2164210
41	lpcreb	Linear Programming problem	9648×77137	260785
42	lpcred	Linear Programming problem	8926×73948	246614
43	lpfit2p	Linear Programming problem	3000×13525	50284
44	lpnug20	Linear Programming problem	15240×72600	304800

Table 2: **Matrix benchmark suite**. These are the same matrices used in the original evaluations of SPARSITY [15].

(1) with previously proposed register-level optimizations that exploit naturally occuring dense block substructure [16]. Below, we review the register blocking optimization (Section 2.1), and describe our heuristic to choose the key tuning parameter, the register block size (Section 2.2). We provide the experimental motivation for this heuristic in Section 4.1.

2.1 Register blocking: improving register reuse

Register blocking, originally proposed in the Sparsity system [16] for SpM$\times$V, improves register reuse by reorganizing the matrix data structure into a sequence of "small" dense blocks, where the block sizes are chosen to keep small blocks of the x and y vectors in registers. An $m\times n$ sparse matrix in $r\times c$ register blocked format is divided logically into $\frac{m}{r}\times\frac{n}{c}$ submatrices, where each submatrix is of size $r\times c$. Only those blocks containing at least one non-zero are stored. Multiplying by A proceeds block-by-block: for each block, we reuse the corresponding r elements of y and c elements of x by keeping them in registers. For simplicity, assume that r divides m and c divides n.

We use the blocked compressed sparse row (BCSR) storage format [21]. Blocks within the same block row are stored consecutively, and the elements of each block are stored in row-major order. A 2$\times$2 example of BCSR is shown in Figure 1, where we assume zero-based array indexing. When $r = c = 1$, BCSR reduces to compressed sparse row (CSR) storage. BCSR can store fewer column indices than CSR (one per block instead of one per non-zero), reducing storage and instruction overhead. We fully unroll the $r\times c$ submatrix computation to reduce loop overhead and expose scheduling opportunities to the compiler. An example of the 2$\times$2 code appears in Figure 2.

Figure 1 also shows that creating blocks may require filling in explicit zeros. We define the *fill ratio* to be the number of stored values (*i.e.*, including zeros) divided by the number of true non-zeros. We may trade-off extra computation (*i.e.*, fill ratio > 1) for improved efficiency from uniform code and memory access.

2.2 Selecting the $r\times c$ register block size

To select the register block size $r\times c$, we adapt the Sparsity v2.0 heuristic for SpM$\times$V [26] to SpA^TA. There are 3 steps. First, we collect a one-time *register profile* to characterize the platform. For SpA^TA, we evaluate the performance (Mflop/s) of the register blocked SpA^TA for all block sizes on

$$A = \begin{pmatrix} a_{00} & a_{01} & 0 & 0 & a_{04} & a_{05} \\ a_{10} & a_{11} & 0 & 0 & a_{14} & a_{15} \\ 0 & 0 & a_{22} & 0 & a_{24} & a_{25} \\ 0 & 0 & a_{32} & a_{33} & a_{34} & a_{35} \end{pmatrix}$$

$\texttt{b\_row\_ptr} = \begin{pmatrix} 0 & 2 & 4 \end{pmatrix}$, $\texttt{b\_col\_idx} = \begin{pmatrix} 0 & 4 & 2 & 4 \end{pmatrix}$

$\texttt{b\_value} = \begin{pmatrix} a_{00} & a_{01} & a_{10} & a_{11} & a_{04} & a_{05} & a_{14} & a_{15} & a_{22} & 0 & a_{32} & a_{33} & a_{24} & a_{25} & a_{34} & a_{35} \end{pmatrix}$

Figure 1: 2×2 **block compressed sparse row (BCSR) storage**. BCSR format uses three arrays. The elements of each block are stored in the `b_value` array. Only the first column index of the (0,0) entry of each block is stored in `b_col_idx` array; the `b_row_ptr` array points to block row starting positions in the `b_col_idx` array. Blocks are stored in row-major order. (Figure taken from Im [15].)

a dense matrix stored in sparse BCSR format. These measurements are independent of the sparse matrix, and therefore only need to be made once per architecture. Second, when the matrix is known at run-time, we estimate the fill ratio for all block sizes. We recently described a sampling scheme for performing this step accurately and efficiently [26]. Third, we select the block size $r\times c$ that maximizes

$$\text{Estimated Mflop/s} = \frac{\text{Mflop/s on a dense matrix in } r\times c \text{ BCSR}}{\text{Estimated fill ratio for } r\times c \text{ blocking}} \quad . \qquad (2)$$

The overhead of picking a register block size and converting into our data structure can all be performed once per matrix and amortized over many uses. This run-time overhead is between 5–20 executions of naïve SpA^TA [26]. Thus, the optimizations we propose are most suitable when SpA^TA must be performed many times (*e.g.*, in sparse iterative methods).

3 Upper Bounds on Performance

We use performance upper bounds to estimate the best possible performance given a matrix and data structure but *independent* of any particular instruction mix or ordering. Code generators in automatic tuning systems for dense linear algebra, such as ATLAS or PHiPAC [29, 4], vary the instruction schedule during the tuning process. In our work on sparse kernel tuning, we have focused on data structure transformations, relying on the compiler

```
void spmv_bcsr_2x2_ata( int mb, const int* ptr, const int* ind,
                        const double* val,
                        const double* x, double* y, double* t )
{
    int i;

    /* for each block row i of A */
    for( i = 0; i < mb; i++, t += 2 )
    {
        int j;
        register double t0 = 0, t1 = 0;
        const int* ind_t = ind;
        const double* val_t = val;

        /* compute (block row of A) times x */
        for( j = ptr[i]; j < ptr[i+1]; j++, ind_t++, val_t += 2*2 )
        {
            t0 += val_t[0*2+0] * x[ind_t[0]+0];
            t1 += val_t[1*2+0] * x[ind_t[0]+0];
            t0 += val_t[0*2+1] * x[ind_t[0]+1];
            t1 += val_t[1*2+1] * x[ind_t[0]+1];
        }

        t[0] = t0;
        t[1] = t1;

        /* compute y <-- (block row of A)^T times t */
        for( j = ptr[i]; j < ptr[i+1]; j++, ind++, val += 2*2 )
        {
            double* yp = y + ind[0];
            register double y0 = 0, y1 = 0;

            y0 += val[0*2+0] * t0;
            y1 += val[0*2+1] * t0;
            y0 += val[1*2+0] * t1;
            y1 += val[1*2+1] * t1;

            yp[0] += y0;
            yp[1] += y1;
        }
    }
}
```

Figure 2: **Cache-optimized,** $2{\times}2$ **SpA^TA implementation**. Here, A is stored in $2{\times}2$ BCSR format (see Figure 1), where A has `2*mb` rows.

to produce efficient schedules. An upper bound allows us to estimate the likely pay-off from low-level tuning.

Our bounds for the cache-optimized, register blocked implementations of SpA^TA described in Section 2 are based on bounds we developed previously for SpM$\times$V [26]. In particular, we make the following assumptions to derive upper bounds:

1. SpA^TA is memory bound since most of the time is spent streaming through matrix data. Thus, we bound time from below by considering only the cost of *memory* operations. Furthermore, we assume write-back caches (true of the platforms considered in Table 1) and sufficient store buffer capacity so that we can consider only loads and ignore the cost of stores.

2. Our model of execution time charges for cache and memory *latency*, as opposed to assuming that data can be retrieved from memory at the manufacturer's reported peak main memory bandwidth. When data resides in the internal cache (L1 on these machines), we assume that all accesses to this data can be fully pipelined, and therefore commit at the maximum load/store commit rate. Table 1 shows this effective L1 access latency.[2] We refer the reader to Section 5 of our prior paper [26] for a more careful analysis of the STREAM benchmarks that justifies this assumption.

3. As shown below in Equation (5), we can get a lower bound on memory costs by computing a lower bound on cache misses. Therefore, we consider only compulsory and capacity misses, *i.e.*, we ignore conflict misses. Also, we account for cache capacity and cache line size, but assume full associativity.

4. We do not consider the cost of TLB misses. Since operations like SpA^TA, SpM$\times$V, and SpTS essentially spend most of their time streaming through the matrix using stride 1 accesses, there are always very few TLB misses.[3]

Let the total time of SpA^TA be T seconds. Then, the performance P in

[2]For example, on the Sun Ultra 2i platform, the L1 load latency is really 2 cycles [24]. However, this processor can commit 1 load per cycle, so we charge a 1 cycle latency for L1 accesses in the bound. The IBM Power3 has a 1 cycle access latency but can commit 2 loads per cycle; we therefore show the effective L1 latency in Table 1 as .5 cycles.

[3]We have verified this experimentally using hardware counters.

Mflop/s is

$$P = \frac{4k}{T} \times 10^{-6} \qquad (3)$$

where k is the number of non-zeros in the $m \times n$ sparse matrix A, *excluding* explicitly filled in zeros.[4] To get an *upper bound on performance*, we need a *lower bound* on T. We present our lower bound on T, which incorporates Assumptions 1 and 2, in Section 3.1, below. Our expression for T in turn uses lower bounds on cache misses (Assumption 3) described in Section 3.2.

3.1 A latency-based execution time model

We model execution time by counting only the cost of memory accesses. Consider a machine with κ cache levels, where the access latency at cache level i is α_i seconds, and the memory access latency is α_{mem}. Suppose SpA^TA executes H_i cache accesses (or cache hits) and M_i cache misses at each level i, and that the total number of loads is L. We charge α_i for each access to cache level i; thus, the execution time T, ignoring the cost of non-memory operations, is

$$\begin{aligned} T &= \sum_{i=1}^{\kappa} \alpha_i H_i + \alpha_{\text{mem}} M_\kappa && (4) \\ &= \alpha_1 L + \sum_{i=1}^{\kappa-1} (\alpha_{i+1} - \alpha_i) M_i + \alpha_{\text{mem}} M_\kappa && (5) \end{aligned}$$

where we use $H_1 = L - M_1$ and $H_i = M_{i-1} - M_i$ for $2 \leq i \leq \kappa$. According to Equation (5), we can minimize T by minimizing M_i, assuming $\alpha_{i+1} \geq \alpha_i$. In Section 3.2, we give expressions for L, M_i to evaluate Equation (5).

3.2 A lower bound on cache misses

Following Equation (5), we obtain a lower bound on M_i for SpA^TA by counting compulsory and capacity misses but ignoring conflict misses. The bound is a function of the cache configuration and matrix data structure.

Let C_i be the size of each cache i in double-precision words, and let l_i be the line size, in doubles, with $C_1 \leq \ldots \leq C_\kappa$, and $l_1 \leq \ldots \leq l_\kappa$. Suppose γ integer indices use the same storage as 1 double.[5] To get lower bounds,

[4] That is, T is a function of the machine architecture and data structure, so we can fairly compare different values of P for fixed A and machine.

[5] For all the machines in this study, we use 32-bit integers; thus, $\gamma = 2$.

assume full associativity and complete user-control over how data is placed in cache.

We describe the BCSR data structure as follows. Let $\hat{k} = \hat{k}(r, c)$ be the number of stored values, so the fill ratio is $\hat{k}/k$, and the number of stored blocks is $\frac{\hat{k}}{rc}$. Then, the total number of loads L is $L = L_A + L_x + L_y$, where

$$L_A = 2\left(\hat{k} + \frac{\hat{k}}{rc}\right) + \frac{m}{r} \qquad L_x = \frac{\hat{k}}{r} \qquad L_y = \frac{\hat{k}}{r}. \tag{6}$$

L_A contains terms for the values, block column indices, and row pointers; the factor of 2 accounts for reading A twice (once to compute Ax, and once for A^T times the result). L_x and L_y are the total number of loads required to read x and y, where we load c elements of each vector for each of the $\frac{\hat{k}}{rc}$ blocks.

We must account for the amount of data, or *working set*, required to multiply by a block row and its transpose in order to model capacity misses correctly. For the moment, assume that all block rows have the same number of $r{\times}c$ blocks; then, each block row has $\frac{\hat{k}}{rc} \times \frac{r}{m} = \frac{\hat{k}}{cm}$ blocks. We define the *matrix working set*, $\hat{W}$, to be the size of matrix data for a block row:

$$\hat{W} = \frac{\hat{k}}{m} r + \frac{1}{\gamma}\frac{\hat{k}}{cm} + \frac{1}{\gamma} \tag{7}$$

The total size of the matrix data in doubles is $\frac{m}{r}\hat{W}$. We define the *vector working set*, $\hat{V}$, to be the size of the corresponding vector elements for x and y:

$$\hat{V} = 2\hat{k}/m \tag{8}$$

i.e., there are $\frac{\hat{k}}{m}$ non-zeros per row, each of which corresponds to a vector element to be reused within a block row; the factor of 2 counts both x and y elements.

For each cache level i, we compute a lower bound on the misses M_i according to one of the following 2 cases, depending on the relative values of C_i, $\hat{W}$, and $\hat{V}$.

1. $\hat{W} + \hat{V} \leq C_i$: *Entire working set fits in cache.*
   Since there is sufficient cache capacity, we incur only compulsory misses on the matrix and vector elements:

$$M_i \geq \frac{1}{l_i}\left(\frac{m}{r}\hat{W} + 2n\right) \quad . \tag{9}$$

2. $\hat{W} + \hat{V} > C_i$: *The working set exceeds the maximum cache capacity.* In addition to the compulsory misses shown in Equation (9), we incur a capacity miss for each element of the total working set that exceeds the cache capacity:

$$M_i \geq \frac{1}{l_i}\left[\frac{m}{r}\hat{W} + 2n + \frac{m}{r}(\hat{W} + \hat{V} - C_i)\right] \quad . \tag{10}$$

We refer the reader to Appendix A for detailed derivations. Note that Equations (9)–(10) have a factor of $\frac{1}{l_i}$, *i.e.*, our lower bound optimistically assumes we will incur only 1 miss per cache line in the best case. To mitigate the effect of this assumption, we could refine these bounds by taking $\hat{W}$ and $\hat{V}$ to be functions of the non-zero structure of each block row. Finally, we remark that we do account for architecture-specific features, such as the fact that on the Itanium platform, only integer data is cached in L1.

4 Experimental Results and Analysis

Below, we present (1) an experimental example of SpA^TA in practice which justifies our approach to search, and motivates our register block size selection heuristic, (2) an experimental validation of the cache miss bounds model described in Section 3.2, and (3) an experimental evaluation of our cache-optimized, register-blocked implementations of SpA^TA with respect to the upper bounds described in Section 3.1. We conducted our experiments on the 4 platforms shown in Table 1 for the 44 matrices listed in Table 2. These matrices are numbered consistently with prior work on Sparsity, and can be categorized as follows.

- Matrix 1: A dense matrix in sparse format, shown for reference.
- Matrices 2–17: Matrices from finite element method (FEM) applications. Matrices 2–9 have a predominantly uniform block structure (a single block size aligned uniformly), while matrices 10–17 have a more irregular block structure (multiple block sizes, or a single block size with blocks not aligned on a fixed grid).
- Matrices 18–39: Matrices from various non-FEM applications, including chemical process simulation, oil reservoir modeling, and macroeconomic modeling applications, among others.
- Matrices 40–44: Matrices from linear programming problems.

We instrumented our code with the PAPI hardware counter library (v2.1) to measure cycle and cache miss counts, among other metrics [6].

Note that when we present results for a particular platform, we omit matrices which fit within the largest cache level to avoid reporting inflated performance results. Furthermore, when reporting performance in Mflop/s, we do *not* count the extra flops due to the explicit zeros caused by fill.

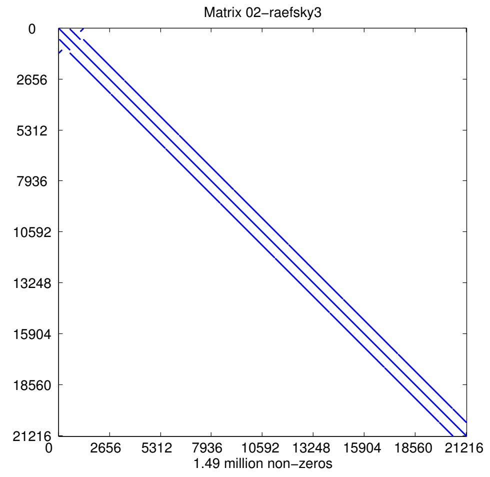
4.1 Experimental motivation for our approach

To motivate our search-based approach to tuning, we consider an example which illustrates thtat (1) even in the simplest example, non-zero structure alone is insufficient to dictate the optimal choice of block size in practice, (2) performance can be an irregular function of block size, and (3) performance is a strong function of the architecture.

Figure 3 (*left*) shows a spy plot of matrix #2 (`raefsky3`) from our benchmark suite, and Figure 3 (`right`) shows the upper leftmost 80×80 submatrix. The non-zero structure of this matrix consists entirely of dense 8×8 subblocks, uniformly aligned on an 8×8 grid as shown. Therefore, an 8×8 blocking does not require filling in explicit zeros. Furthermore, our performance upper bounds predict that the 8×8 format would be the best choice since that block size leads to the smallest index overhead. An application developer is likely to choose this block size, or perhaps 4×4, the next smallest, square factor. (Note that the popular library, PETSc, supports only square block sizes at the time of this writing [2].)

We ran an experiment in which we tried the cache-optimized, register blocked implementation for all $r \times c$ where $r, c \in \{1, 2, 4, 8\}$. We would expect that increasing the block size should lead to uniform improvements in performance, according to our bounds model. Figure 4 shows our experimental results on each of the four evaluation platforms. Specifically, we show the 16 $r \times c$ implementations, each color-coded by its performance in Mflop/s and labeled by its speedup relative to the unblocked (1×1) cache-optimized code (lower-left square in each plot).

On the Ultra 2i, performance increases with block size as we expect, and 8×8 has the best performance (just over 100 Mflop/s and a 2.3× speedup). However, choosing 8×8 or even 4×4 block sizes does not yield the optimal performance on all platforms: on the Pentium III, the 4×2 implementation is the fastest. On the Power3, 8×8 is better than no blocking, but much worse than the optimal 4×4; the Power3 has twice as many double-precision floating point registers as the Ultra 2i, so register pressure does not explain why 8×8 is relatively slower than 4×4 on the Power3. On the Itanium, the



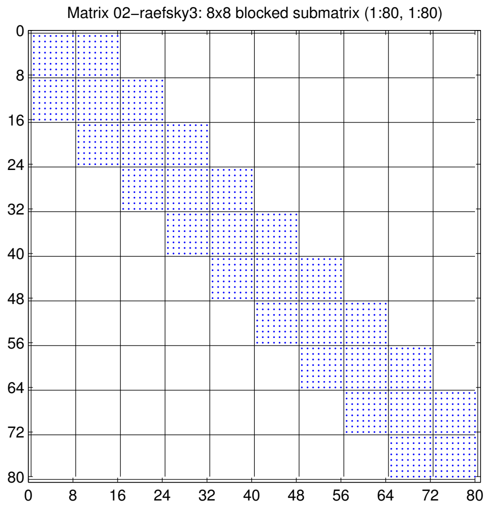


Figure 3: **Spy plots of Matrix #2 (`raefsky3`).** (*Left*) The matrix consists primarily of three bands. (*Right*) These bands consist entirely of 8×8 blocks, aligned uniformly as shown by this 80×80 submatrix.

4×4 code is anomalously slower than all of the "neighboring" implementations (*e.g.*, 2×2, 8×8, 4×4, 4×2, ...). In short, performance is a strong function of the architecture, and the optimal block size cannot necessarily be predicted by considering only non-zero structure or anticipated register usage.

Indeed, the purpose of collecting the register profiles (off-line benchmarking data) as described in Section 2.2 is to capture the machine-dependent structure in performance, independent of any particular sparse matrix. Figure 5 shows the profile data on our evaluation platforms—we see a dramatic variation in performance as a function of the platform.

4.2 Validation of the cache miss model

Figure 6 compares the load and cache miss counts given by our model, Equations (6)–(10), to those observed using PAPI. We measured the performance (Mflop/s) for all block sizes to determine empirically the best block size, $r_{\text{opt}} \times c_{\text{opt}}$, for each matrix and platform. Figure 6 shows, at the matrix- and machine-dependent block size $r_{\text{opt}} \times c_{\text{opt}}$, (1) the ratio of measured load operations to the loads predicted by Equation (6) (shown as solid squares), and (2) the ratio of measured L1, L2, and L3 cache misses to the lower bound, Equations (9)–(10) (shown as circles, asterisks, and ×s, respectively). Fur-

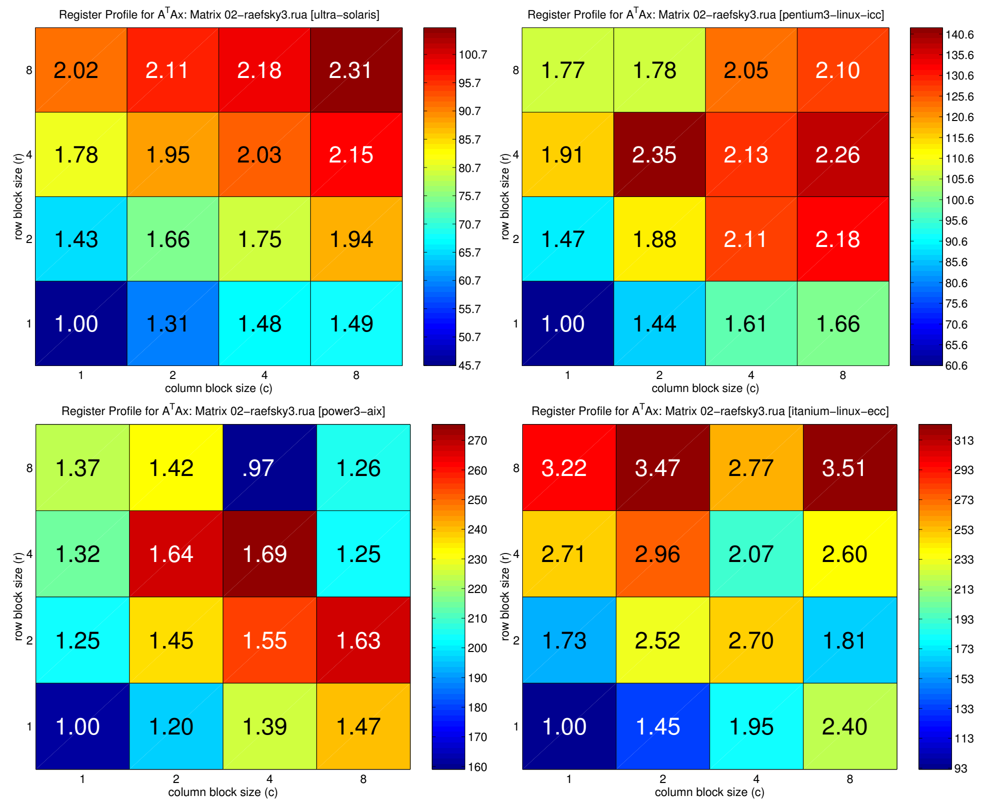


Figure 4: **Variations in performance on different architectures: matrix #2 (`raefsky3`).** For each of the four evaluation platforms, we show the performance of the cache-optimized, register blocked code on Matrix #2, for all $r \times c$ block sizes where $r, c \in \{1, 2, 4, 8\}$. Each square is an implementation, color-coded by its performance (Mflop/s) and labeled by its speedup over the unblocked (1×1), cache-optimized code. The best implementations are 8×8 on the Ultra 2i (*top-left*), 4×2 on the Pentium III (*top-right*), 4×4 on the Power3 (*bottom-left*), and 8×8 on the Itanium (*bottom-right*).

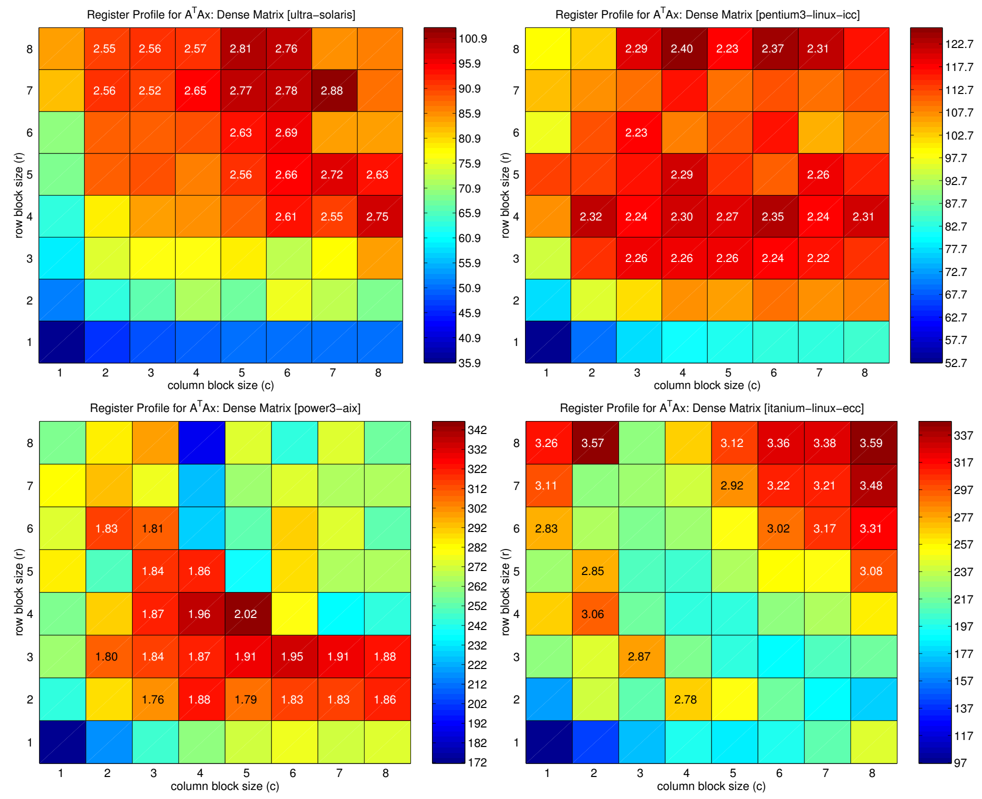


Figure 5: **Register profiles (off-line benchmarks) capture machine-dependent structure**. For each of the four evaluation platforms, we show the performance of the cache-optimized, register blocked code on a dense matrix stored in sparse $r \times c$ format, for all $r \times c$ up to 8×8. Each square is an implementation, shaded by its performance (Mflop/s) and labeled by its speedup over the unblocked (1×1), cache-optimized code. Profiles are shown for the Ultra 2i (*top-left*), Pentium III (*top-right*), Power3 (*bottom-left*), and Itanium (*bottom-right*).

thermore, for each category (*i.e.*, loads, L1 misses, L2 misses, ...), we show the median ratio as a dashed horizontal line. Since our model is indeed a lower bound, all ratios are at least 1; if our model exactly predicted reality, then all ratios would equal 1. We make the following observations:

- L2 and L3 cache miss counts tend to be very accurate: the observed counts are typically within 5–10% of the lower bound, indicating that cache capacity is sufficient to justify ignore conflict misses at these levels.

- The ratio of observed L1 miss counts to the model is relatively high on the Ultra 2i (median ratio of 1.34×) and the Pentium III (1.23×), compared to the Power3 (1.16×) and Itanium (1.00×). One explanation is the lack of L1 cache capacity, which causes more misses than predicted by our model. Recall that the model assumes the full cache capacity is available and that there are no conflicts. On the Itanium, although the cache size is the same as that on the Ultra 2i, less capacity is needed relative to the Ultra 2i because only integer data is cached in L1.

- On the Pentium III and Itanium, the observed load counts are high relative to the model. On the Pentium III, separate load and store counters were not available, so stores are included in the counts. Manually accounting for these stores yields the expected number of loads to within 10% when spilling does not occur (not shown). A secondary reason is that spilling occurs with a few of the implementations (confirmed by inspection of the assembly code).

  On the Itanium, prefetch instructions (inserted by the compiler) are counted as loads by the hardware counter for load instructions. (By contrast, prefetches are also inserted by the IBM compiler, but are not counted as loads.)

- On matrices 15 and 40–44 (linear programming), observed miss counts (particularly L1 misses) tend to be much higher than for the other matrices. These matrices tend to have a much more random distribution of non-zeros than the others, and therefore our assumption of being able to exploit spatial locality fully (the $\frac{1}{l_i}$ factor in Equations (9)–(10)) does not hold. Thus, we expect the upper bound to be optimistic for these matrices.

In summary, we claim that the data show our lower bound cache miss estimates are reasonable, and that we are able to account for discrepancies based

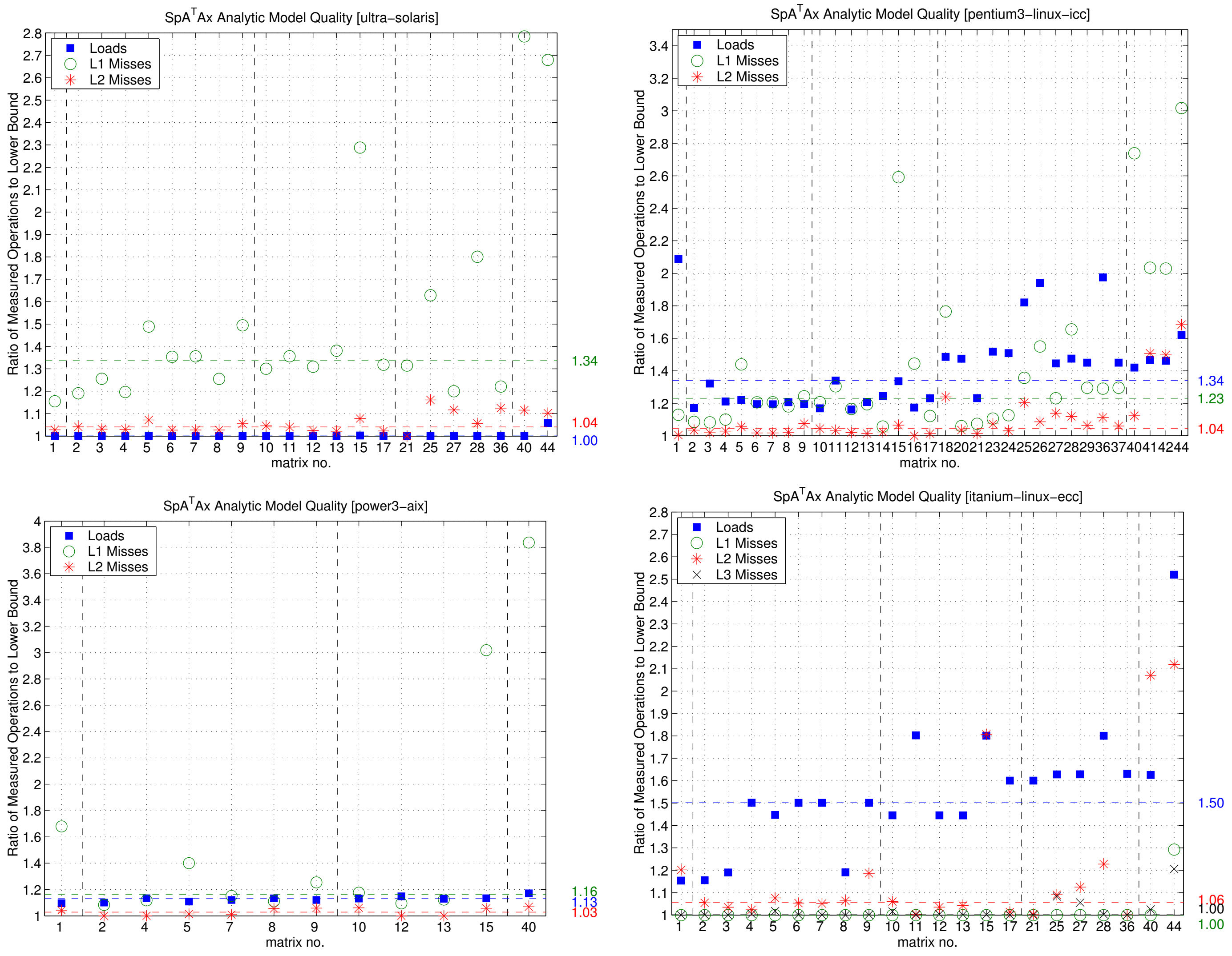


Figure 6: **Cache miss model validation**. We show the ratio (y-axis) of measured loads and cache misses to the counts predicted by our lower bound model, Equations (6)–(10), for each matrix (x-axis). We show data for each of the following four platforms: Sun Ultra 2i (*top-left*), Intel Pentium III (*top-right*), IBM Power3 (*bottom-left*), and Intel Itanium (*bottom-right*). Of these platforms, only the Itanium has three levels of cache. The median of the ratios is shown as a dotted horizontal line, with its value labeled to the right of each plot.

on both our modeling assumptions and our knowledge of each architecture. For the interested reader, we include raw load/store counts in Appendix B.

4.3 Performance evaluation of our SpA^TA implementations

Our performance evaluation results are summarized in Figures 7–10, which compare the performance (Mflop/s; y-axis) of the following bounds and implementations for each matrix (x-axis):

- **Upper bound**, or analytic upper bound (shown as a solid line): This line shows the fastest (highest) value of our performance upper bound, Equations (3)–(10), over all $r\times c$ block sizes up to 8×8. We denote the block size shown by $r_{\mathrm{up}}\times c_{\mathrm{up}}$.

- **PAPI upper bound** (shown by triangles): The "PAPI upper bound" is also an upper bound, except that we substitute true loads and misses as measured by PAPI for L and M_i in Equation (5). In some sense, the PAPI bound is the true bound since misses are "modeled" exactly; the gap between the PAPI bound and the upper bound indicates how well Equations (6)–(10) reflect reality. The data points shown are for the same block size $r_{\mathrm{up}}\times c_{\mathrm{up}}$ used in the analytic upper bound.

  Note that the block sizes ($r_{\mathrm{up}}\times c_{\mathrm{up}}$) used in the analytic and PAPI upper bounds are not necessarily the same as those used in Section 4.2. Nevertheless, the observations of Section 4.2 are qualitatively the same. We chose to use the best model bound in order to show the best possible performance expected, assuming ideal scheduling.

- **Best cache optimized, register blocked** implementation (squares): We implemented the optimization described in Section 2, These points show the best observed performance over all block sizes up to 8×8. We denote the block size shown by $r_{\mathrm{opt}}\times c_{\mathrm{opt}}$, which may differ from $r_{\mathrm{up}}\times c_{\mathrm{up}}$.

- **Heuristic cache optimized, register blocked** implementation (solid circles): These points show the performance of the cache optimized implementation using a register block size, $r_{\mathrm{heur}}\times c_{\mathrm{heur}}$, chosen by the heuristic.

- **Register blocking only** (diamonds): This implementation computes $t = Ax$ and $y = A^Tt$ as separate steps but with register blocking. The same block size, $r_{\mathrm{reg}}\times c_{\mathrm{reg}}$, is used in both steps, and the best performance over all block sizes up to 8×8 is shown.

- **Cache optimization only** (shown by asterisks): This code implements the algorithmically cache optimized version of SpA^TA shown in Equation (1), but without any register-level blocking (*i.e.*, with $r = c = 1$).
- **Reference** implementation (×'s): The reference computes $t = Ax$ and $y = A^Tt$ as separate steps, with no register-level blocking.

Note that on each platform, we have omitted matrices which are small relative to the cache size to avoid reporting inflated performance data. Appendix C show the values of $r_{\text{opt}} \times c_{\text{opt}}$, $r_{\text{heur}} \times c_{\text{heur}}$, and $r_{\text{reg}} \times c_{\text{reg}}$ used in Figures 7–10. We draw the following 5 high-level conclusions based on Figures 7–10.

1. *The cache optimization leads to uniformly good performance improvements.* Applying the cache optimization, even without register blocking, leads to speedups ranging from up to 1.2× on the Itanium and Power3 platforms, to just over 1.6× on the Ultra 2i and Pentium III platforms. These speedups do not vary significantly across matrices, suggesting that this optimization is always worth trying.

2. *Register blocking and the cache optimization can be combined to good effect.* When the algorithmic cache blocking and register blocking are combined, we observe speedups from 1.2× up to 4.2× over the reference code. Furthermore, if we compare the best, combined implementation to the register blocking only code, we see speedups of up to 1.8×.

Moreover, the effect of combining the register blocking and the cache optimization is synergistic: the observed, combined speedup is *at least* the product (the register blocking only speedup) × (the cache-optimization only speedup), when $r_{\text{reg}} \times c_{\text{reg}}$ and $r_{\text{opt}} \times c_{\text{opt}}$ match. Indeed, the combined speedup is greater than this ratio on the Ultra 2i, Power3, and Itanium platforms. In Appendix D, we show speedup versions of Figures 7–10 in order to make the claim of synergy explicit. One possible explanation is that the compilers on these three platforms schedule instructions for in-cache workloads better than out-of-cache workloads. Some indirect evidence of this claim is shown in Appendix C, where the best implementations have surprisingly high fill ratios (1.5 or more).

3. *Our heuristic always chooses a near-optimal block size.* Indeed, the performance of the block size selected by the heuristic is within 10% of the exhaustive best in all but four instances—in those four cases, the heuristic performance is within 15% of the best.

In Appendix C, we summarize this data in detail, showing the optimal block sizes for SpA^TA, both with and without the cache and register blocking optimizations. We also consider the case in which we use the optimal register blocking only block size, $r_{\text{reg}}\times c_{\text{reg}}$, with the cache optimization. On all platforms except the Power3, we find a number of cases in which the choice of $r_{\text{reg}}\times c_{\text{reg}}$ with the cache optimization is more than 10% worse than choosing the $r_{\text{opt}}\times c_{\text{opt}}$ block size predicted by our heuristic. Therefore, using a SpA^TA-specific heuristic leads to more robust block size selection.

4. *Our implementations are within 20–30% of the PAPI upper bound for FEM matrices, but within only about 40–50% on other matrices.* The gap between actual performance and the upper bound is larger than what we observed previously for SpM×V and SpTS [26, 27]. This result suggests that a larger pay-off is expected from low-level tuning by, for instance, applying tuning techniques used in systems such as ATLAS/PHiPAC to further improve performance.

5. *Our analytic model of misses is accurate for FEM matrices, but less accurate for the others.* For the FEM matrices 1–17, the PAPI upper bound is typically within 10–15% of the analytic upper bound, indicating that our analytic model of misses is accurate in these cases. For the matrices 18–44, the gap between the analytic upper bound and the PAPI upper bound increases with increasing matrix number because our cache miss lower bounds assume maximum spatial locality in the accesses to x, indicated by the factor of $\frac{1}{l_i}$ in Equations (9)–(10). We discuss this effect in Section 4.2. FEM matrices have naturally dense block structure and can benefit from spatial locality; matrices with more random structure (*e.g.*, linear programming matrices 40–44) cannot. In principle, we can refine our lower bounds to account for this by a more detailed examination of the non-zero structure.

The gap between the analytic and PAPI upper bounds is larger (as a fraction of the analytic upper bound) on the Pentium III than on the other three platforms. As discussed in Section 4.2, this is due to two factors: (1) we did not have separate counters for load and store operations, so we are charging for stores as well in the PAPI upper bound, and (2) in some cases, the limited number of registers on the Pentium III (8 registers) led to spilling in some implementations (confirmed by inspection of the load operation counts and inspection of the assembly code).

For additional analysis of the performance bound model, we refer the reader to Appendix E. There, we breakdown execution time according to the time spent at each cache and memory level. We discuss what the model implies about the design of cache architectures for SpA^TA in particular, and streaming applications more generally.

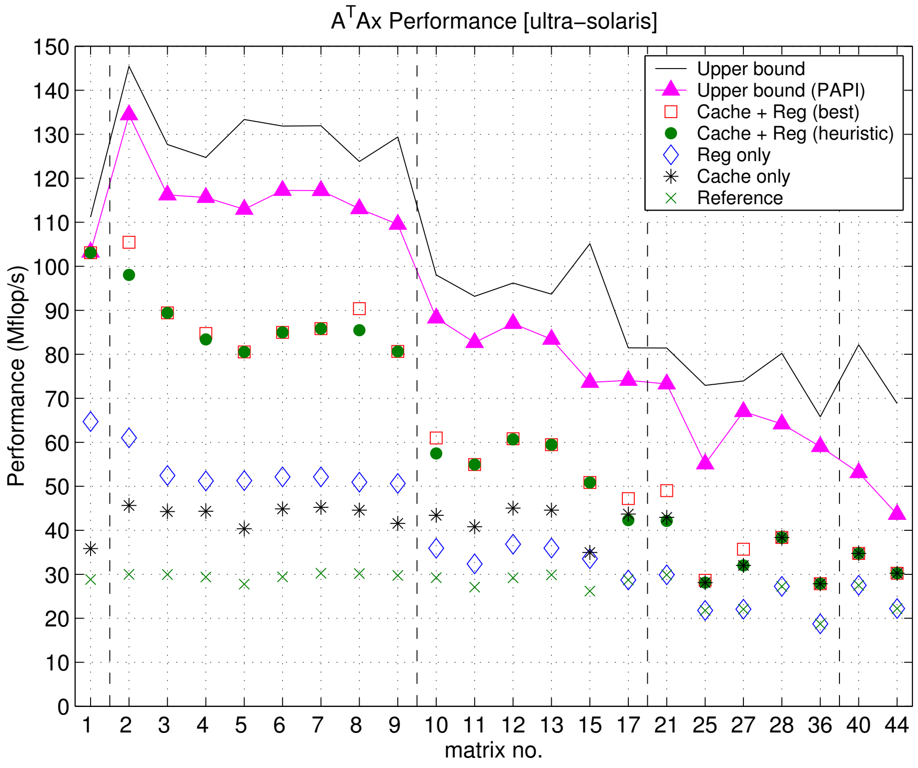


Figure 7: **SpA^TA performance on the Sun Ultra 2i platform**. A speedup version of this plot appears in Appendix D.

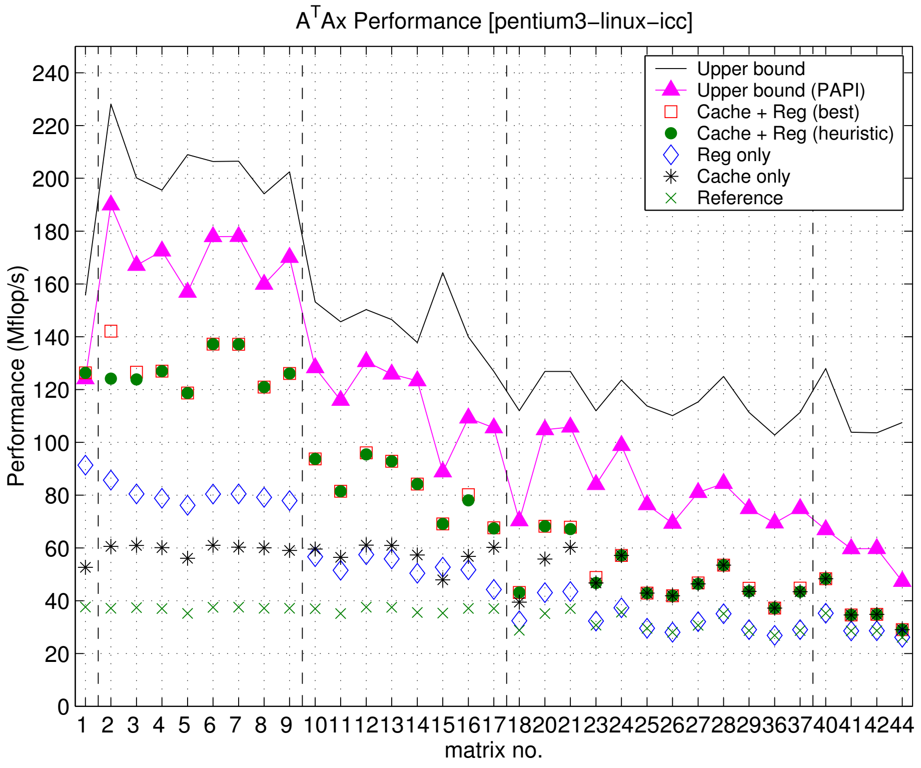


Figure 8: **SpA^TA performance on the Intel Pentium III platform**. A speedup version of this plot appears in Appendix D.

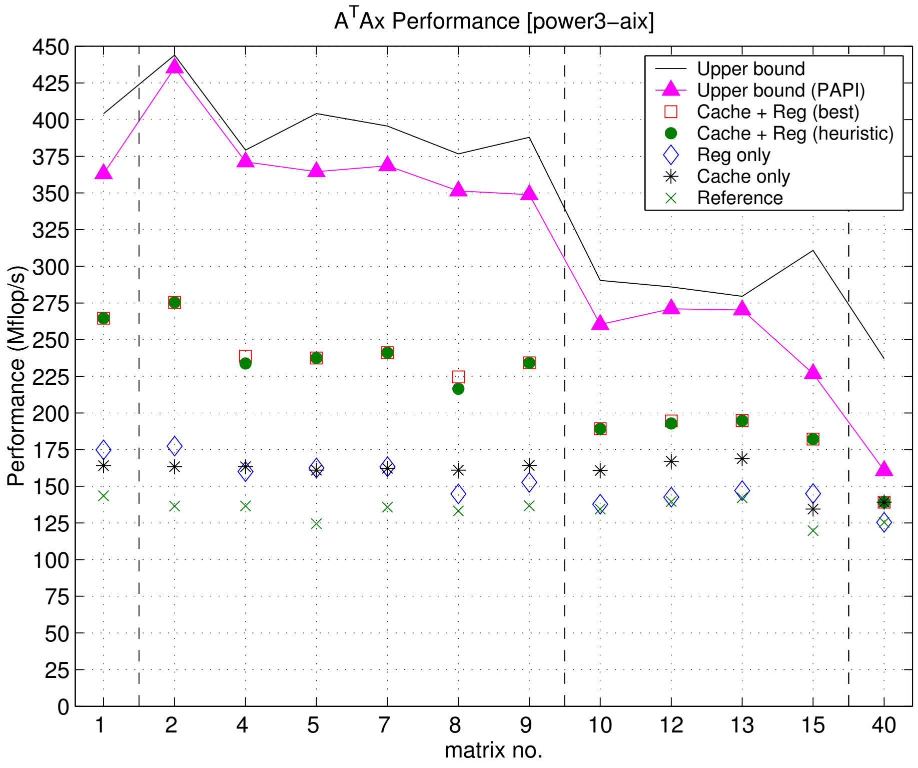


Figure 9: **SpA^TA performance on the IBM Power3 platform**. A speedup version of this plot appears in Appendix D.

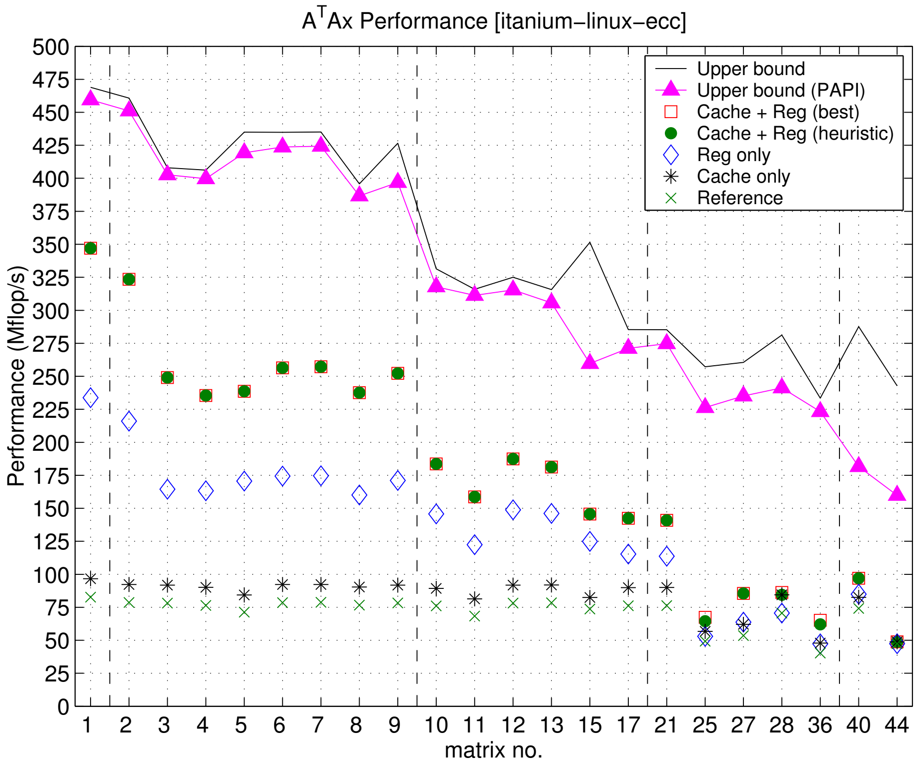


Figure 10: **SpA^TA performance on the Intel Itanium platform**. A speedup version of this plot appears in Appendix D.

5 Related Work

For dense algorithms, a variety of sophisticated static models for selecting transformations and tuning parameters 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 [8, 11, 19, 7, 30]. However, it is difficult to apply these analyses directly to sparse matrix kernels due to the presence of indirect and irregular memory access patterns. Nevertheless, there have been a number of notable modeling attempts in the sparse case for SpM$\times$V. Temam and Jalby [25], Heras, *et al.* [14], and Fraguela, *et al.* [10] have developed sophisticated probabilistic cache miss models for SpM$\times$V, but assume uniform distribution of non-zero entries. These models vary in their ability to account for self- and cross-interference misses. To obtain lower bounds, we account only for conflict and capacity misses, though refinements are possible (see Section 4).

Gropp, *et al.*, use bounds like the ones we develop to analyze and tune a computational fluid dynamics code [12]; Heber, *et al.*, develop, study, and tune a fracture mechanics code [13] on Itanium. However, we are interested in tuning for matrices that come from a variety of domains and machine architectures. Furthermore, in our bounds we explicitly model execution time (instead of just modeling misses) in order to evaluate the extent to which our tuned implementations achieve optimal performance.

Work in sparse compilers, *e.g.*, Bik *et al.* [3], Pugh and Shpeisman [20], and the Bernoulli compiler [23], complements our own work. These projects focus on the expression of sparse kernels and data structures for code generation, and will likely prove valuable to generating our implementations. One distinction of our work is our use of a hybrid off-line, on-line, architecture-specific model for selecting transformations (tuning parameters).

6 Conclusions and Future Directions

The speedups of up to 4.2$\times$ that we have observed indicate that there is tremendous potential to boost performance in applications dominated by SpA^TA. The implementation of our heuristic and its accuracy in choosing a block size helps to validate the approach to tuning parameter selection originally proposed in the SPARSITY. We are incorporating this kernel and these optimizations in an automatically tuned sparse library based on the Sparse BLAS standard [5].

Our upper bounds indicate that there is a more room for improvement using low-level tuning techniques than with prior work on SpM×V and SpTS. Applying automated search techniques to improve scheduling, as developed in ATLAS and PHiPAC, is a natural extension of this work. We have also identified refinements to the bounds that make explicit use of matrix nonzero structure (*e.g.*, making the working set size block row structure dependent, and accounting for the degree of actual spatial locality in source vector accesses). Such models could be used to study how performance varies with architectural parameters, in the spirit of SpM×V modeling work by Temam and Jalby [25].

Additional reuse is possible when multiplying by multiple vectors instead of a single vector. Preliminary results on Itanium for sparse matrix-multiple-vector mltiplication show speedups of 6.5 to 9 over single-vector code [26]. This is a natural opportunity for future work with SpA^TA as well. We are exploring this optimization and other higher-level kernels with matrix reuse (*e.g.*, A^kx, matrix triple products).

Acknowledgements

We thank Michael de Lorimier for writing the SpA^TA initial implementations. This research was supported in part by the National Science Foundation under NSF Cooperative Agreement No. ACI-9813362, the Department of Energy under DOE Grant No. DE-FC02-01ER25478, and a gift from Intel. 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] S. Andersson, R. Bell, J. Hague, H. Holthoff, P. Mayes, J. Nakano, D. Shieh, and J. Tuccillo. *RS/6000 Scientific and Technical Computing: Power3 Introduction and Tuning.* International Business Machines, Austin, TX, USA, 1998. `www.redbooks.ibm.com`.

[2] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.

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

[4] J. Bilmes, K. Asanović, C. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the International Conference*

on Supercomputing, Vienna, Austria, July 1997. ACM SIGARC. see `http://www.icsi.berkeley.edu/~bilmes/phipac`.

[5] 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. Chapter 3: `www.netlib.org/blast`.

[6] 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.

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

[8] 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.

[9] J. W. Demmel. *Applied Numerical Linear Algebra.* SIAM, 1997.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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, San Francisco, CA, May 2001. Springer.

[17] Intel. Intel Itanium Processor Reference Manual for Software Optimization, November 2001.

[18] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.

[19] 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.

[20] 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.

[21] Y. Saad. SPARSKIT: A basic toolkit for sparse matrix computations, 1994. `www.cs.umn.edu/Research/arpa/SPARSKIT/sparskit.html`.

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

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

[24] Sun-Microsystems. UltraSPARC IIi: User's Manual, 1999.

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

[26] 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.

[27] R. Vuduc, S. Kamil, J. Hsu, R. Nishtala, J. W. Demmel, and K. A. Yelick. Automatic performance tuning and analysis of sparse triangular solve. In *ICS 2002: Workshop on Performance Optimization via High-Level Languages and Libraries*, New York, USA, June 2002.

[28] W. Wang and D. P. O'Leary. Adaptive use of iterative methods in interior point methods for linear programming. Technical Report UMIACS-95-111, University of Maryland at College Park, College Park, MD, USA, 1995.

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

[30] 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 Cache Miss Lower Bounds for SpA^TA

Below, we derive each of the 2 cases given in Section 3.2. We assume the same notation. To simplify the discussion, let $l_i = 1$; the case of $l_i > 1$ reduces each of the miss counts below by a factor of $\frac{1}{l_i}$, as shown in Equations (9)–(10).

1. $\hat{W} + \hat{V} \leq C_i$: *The total working set fits in cache.*
   In this case, there is sufficient cache capacity to hold both the matrix and vector working sets in cache. Therefore, we incur only compulsory misses: 1 miss for each of the $\frac{m}{r}\hat{W}$ matrix data words, and 1 miss for each of the $2n$ vector elements (x and y).

2. $\hat{W} + \hat{V} > C_i$: *The total working set exceeds the cache size.*
   To obtain a lower bound in this case, suppose (1) the cache is fully associative, and (2) we have complete control over how data is placed in the cache. Suppose we choose to devote a fraction α of the cache to the matrix elements, and a fraction $1 - \alpha$ of the cache to the vector elements. The following condition ensures that α lies in a valid subset of the interval $[0, 1]$:

   $$\max\left\{0, 1 - \frac{\hat{V}}{C_i}\right\} \leq \alpha \leq \min\left\{\frac{\hat{W}}{C_i}, 1\right\} \quad .$$

   (The case of α at the lower bound means that we devote the maximum possible number of elements to the vector working set, and as few as possible to the matrix working set. When α meets the upper bound, we devote as many cache lines as possible to the matrix and as few as possible to the vector.)

   First consider misses on the matrix. In addition to the $\frac{m}{r}\hat{W}$ compulsory misses, we incur capacity misses: for each block row of A, each of the $\hat{W} - \alpha C_i$ words exceeding the alotted capacity for the matrix will miss. Summing the capacity misses over all $\frac{m}{r}$ block rows and adding the compulsory misses, we find $\frac{m}{r}\hat{W} + \frac{m}{r}\left(\hat{W} - \alpha C_i\right)$ misses to the matrix elements.

   A similar argument applies to the x and y vectors. There are $2n$ compulsory misses and, for each block row, $\hat{V} - (1 - \alpha)C_i$ capacity misses, or $2n + \frac{m}{r}\left[\hat{V} - (1 - \alpha)C_i\right]$ misses in all.

   Thus, a lower bound on cache misses in this case is

   $$\begin{aligned} M_i &\geq \frac{m}{r}\hat{W} + \frac{m}{r}(\hat{W} - \alpha C_i) + 2n + \frac{m}{r}\left[\hat{V} - (1 - \alpha)C_i\right] \\ &= \frac{m}{r}\hat{W} + 2n + \frac{m}{r}(\hat{W} + \hat{V} - C_i) \end{aligned}$$

   which is independent of how cache capacity is allocated among the matrix and vector data, *i.e.*, independent of α.

Note that we can further refine the bounds by considering each block row individually, *i.e.*, taking $\hat{W}$ and $\hat{V}$ to be functions of the non-zero structure of the actual block row. For the particular matrices used in this study, this refinement of the bounds would have produced tighter upper bounds.

Finally, we note that in this study we also account for architecture-specific features. For instance, on the Itanium platform, only integer data is cached in L1.

B Raw PAPI Load and Miss Count Data

Figures 11–14 shown the raw load and cache miss count data reported by PAPI.

C Tabulated Performance Data

Tables 3–6 list the block sizes and corresponding performance values and measurements for Figures 7–10. In particular, each table shows the following data:

- **Best cache optimized, register blocked** block size ($r_{\text{opt}} \times c_{\text{opt}}$) and performance: Best block size and corresponding performance based on an exhaustive search over block sizes.
- **Heuristic cache optimized, register blocked** block size ($r_{\text{heur}} \times c_{\text{heur}}$) and performance: Block size chosen by the heuristic and its corresponding performance. Items in this column marked with a $*$ show when this choice of block size yields performance that is more than 10% worse than the optimal block size, $r_{\text{opt}} \times c_{\text{opt}}$.
- **Register blocking only** block size ($r_{\text{reg}} \times c_{\text{reg}}$) and performance.
- **Cache optimized, register blocked** implementation using the same block size, $r_{\text{reg}} \times c_{\text{reg}}$, as in the register blocking only case. Items in this column marked with a $*$ show when this choice of block size yields performance that is more than 10% worse than the optimal block size, $r_{\text{opt}} \times c_{\text{opt}}$. That is, marked items show when the Sp$A^T A$-specific heuristic makes a better choice than using the optimal block size based only on SpM$\times$V performance.

D Speedup Plots

Figures 15–18 compare the observed speedup when register blocking and the cache optimization are combined with the product (register blocking only speedup) $\times$ (cache optimization only speedup). When the former exceeds the latter, we say there is a synergistic effect from combining the two optimizations. This effect occurs on all the platforms but the Pentium III, where the observed speedup and the product of individual speedups are nearly equal.

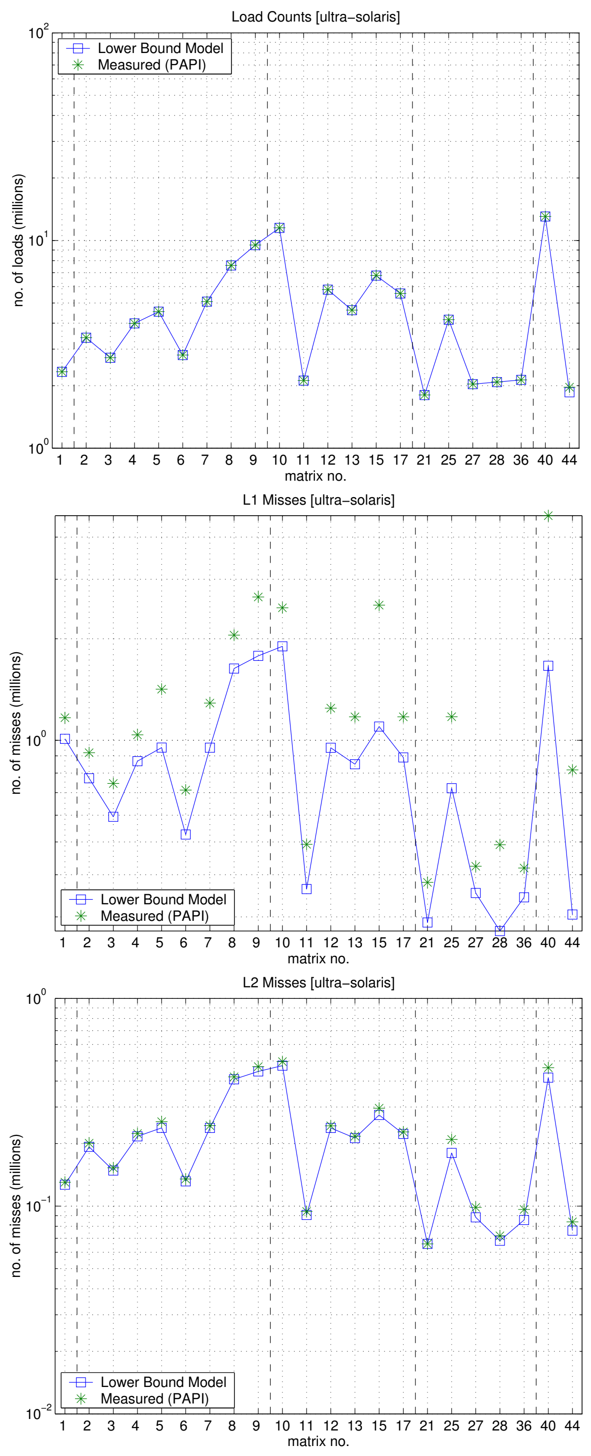
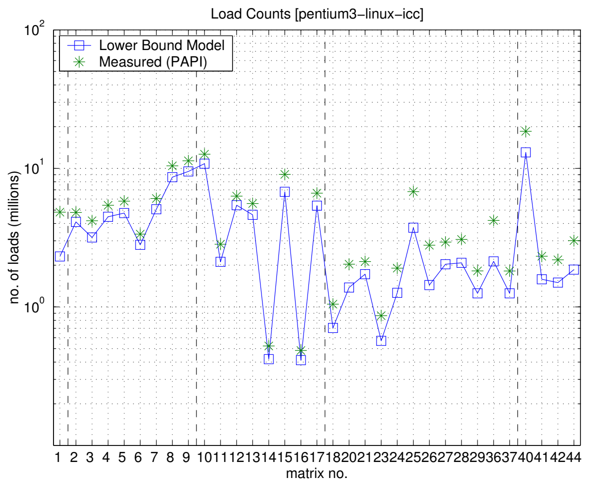
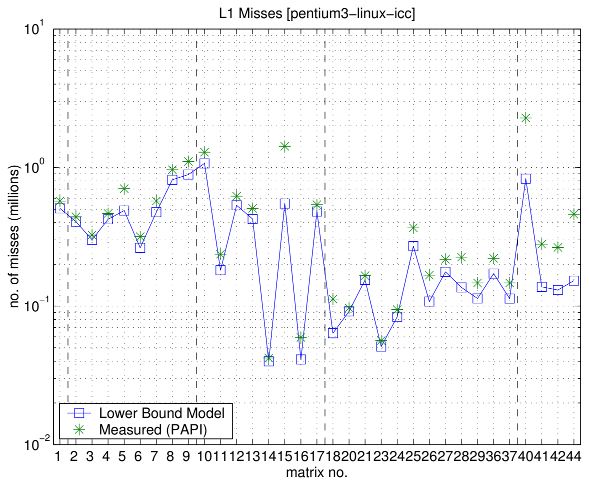


Figure 11: **Raw load and cache miss count data: Sun Ultra 2i**. Counts reported by PAPI for load instructions (*top*), L1 misses (*middle*), and L2 misses (*bottom*). Note the logarithmic scale on the y-axis.





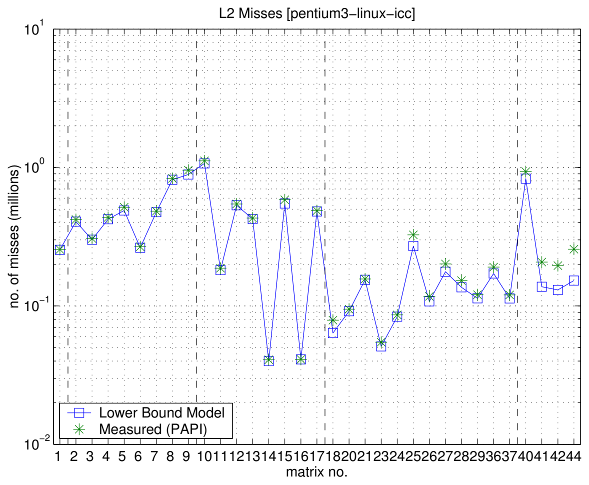


Figure 12: **Raw load and cache miss count data: Intel Pentium III**. Counts reported by PAPI for load instructions (*top*), L1 misses (*middle*), and L2 misses (*bottom*). Note the logarithmic scale on the y-axis.

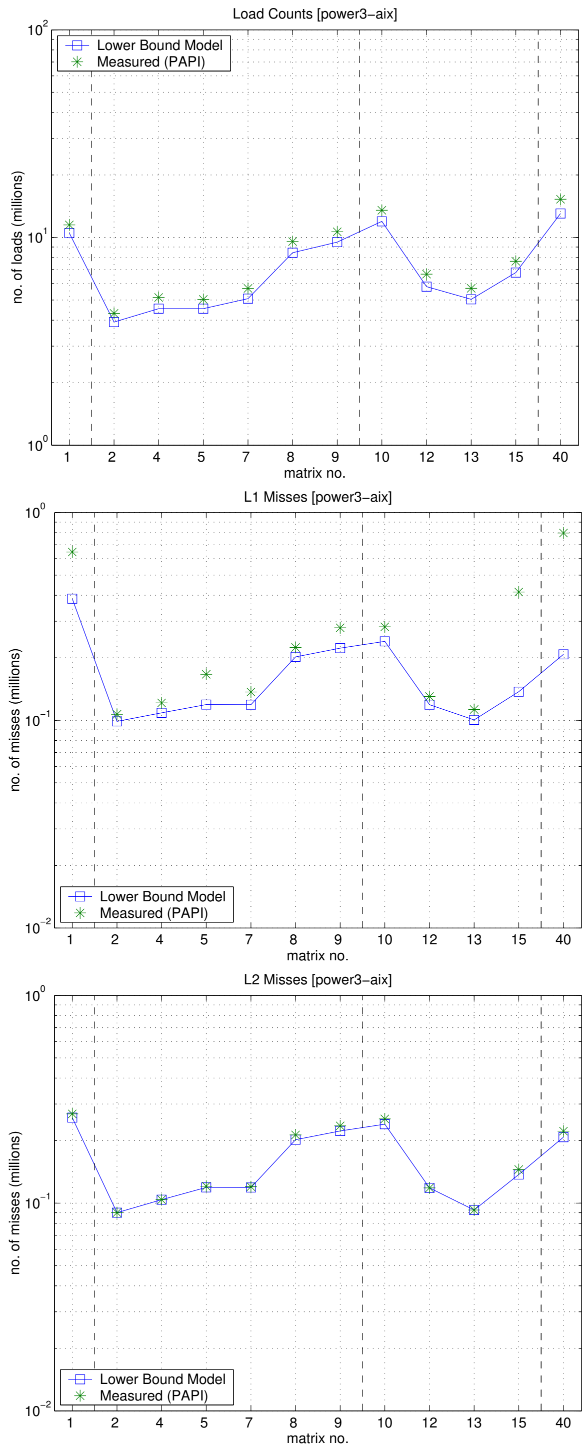


Figure 13: **Raw load and cache miss count data: IBM Power3**. Counts reported by PAPI for load instructions (*top*), L1 misses (*middle*), and L2 misses (*bottom*). Note the logarithmic scale on the y-axis.

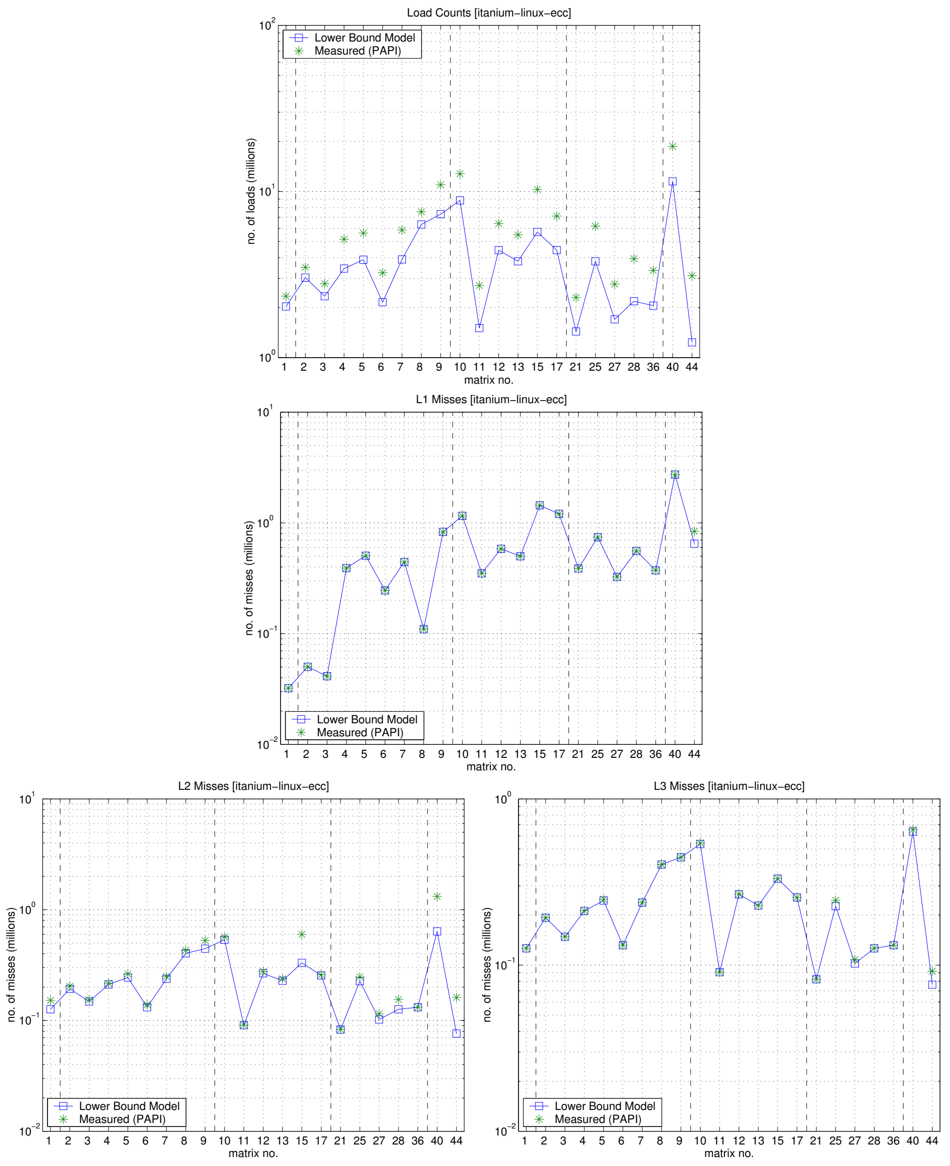


Figure 14: **Raw load and cache miss count data: Intel Itanium**. Counts reported by PAPI for load instructions (*top*), L1 misses (*middle*), L2 misses (*bottom-left*), and L3 misses (*bottom-right*). Note the logarithmic scale on the y-axis.

No.	Best cache-opt. + reg. blocking $r_{\text{opt}} \times c_{\text{opt}}$	Fill	Mflop/s	Heuristic cache-opt. + reg. blocking $r_{\text{heur}} \times c_{\text{heur}}$	Fill	Mflop/s	Reg. blocking only $r_{\text{reg}} \times c_{\text{reg}}$	Fill	Mflop/s	cache-opt. + reg. block $r_{\text{reg}} \times c_{\text{reg}}$ Mflop/s
1	7×7	1.00	103.1	7×7	1.00	103.1	8×5	1.00	64.7	100.5
2	8×8	1.00	105.5	4×8	1.00	98.0	8×2	1.00	61.0	96.2
3	6×6	1.12	89.4	6×6	1.12	89.4	6×6	1.12	52.5	89.4
4	6×3	1.12	84.7	6×6	1.19	83.4	6×2	1.13	51.2	82.7
5	4×4	1.00	80.6	4×4	1.00	80.6	4×4	1.00	51.3	80.6
6	3×3	1.00	85.0	3×3	1.00	85.0	3×3	1.00	52.1	85.0
7	3×3	1.00	85.8	3×3	1.00	85.8	3×3	1.00	52.1	85.8
8	6×2	1.13	90.4	6×6	1.15	85.5	6×6	1.15	50.9	85.5
9	3×3	1.02	80.7	3×3	1.02	80.7	3×3	1.02	50.6	80.7
10	2×2	1.21	61.0	5×2	1.58	57.5	2×2	1.21	35.9	61.0
11	2×2	1.23	54.9	2×2	1.23	54.9	2×2	1.23	32.3	54.9
12	2×2	1.24	60.8	3×2	1.36	60.7	3×2	1.36	36.8	60.7
13	3×2	1.40	59.5	3×2	1.40	59.5	3×2	1.40	35.9	59.5
15	2×1	1.00	50.9	2×1	1.00	50.9	2×1	1.00	33.5	50.9
17	2×1	1.36	47.2	7×1	2.09	42.3∗	1×1	1.00	28.7	43.7
21	2×1	1.38	49.0	7×1	2.10	42.1∗	1×1	1.00	29.9	42.9∗
25	2×1	1.71	28.6	1×1	1.00	28.1	1×1	1.00	21.8	28.1
27	2×1	1.53	35.7	1×1	1.00	32.0∗	1×1	1.00	22.0	32.0∗
28	1×1	1.00	38.4	1×1	1.00	38.4	1×1	1.00	27.3	38.4
36	1×1	1.00	27.9	1×1	1.00	27.9	1×1	1.00	18.7	27.9
40	1×1	1.00	34.7	1×1	1.00	34.7	1×1	1.00	27.5	34.7
44	1×1	1.00	30.2	1×1	1.00	30.2	1×1	1.00	22.2	30.2

Table 3: **Block size summary data for the Sun Ultra 2i platform**. An asterisk (∗) by a heuristic performance value indicates that this performance was less than 90% of the best performance.

No.	Best cache-opt. + reg. blocking $r_{\text{opt}}\times c_{\text{opt}}$	Fill	Mflop/s	Heuristic cache-opt. + reg. blocking $r_{\text{heur}}\times c_{\text{heur}}$	Fill	Mflop/s	Reg. blocking only $r_{\text{reg}}\times c_{\text{reg}}$	Fill	Mflop/s	cache-opt. + reg. block $r_{\text{reg}}\times c_{\text{reg}}$ Mflop/s
1	8×4	1.00	126.3	8×4	1.00	126.3	6×2	1.00	91.3	111.4∗
2	4×2	1.00	142.1	8×4	1.00	124.1∗	4×8	1.00	85.7	137.0
3	3×6	1.12	126.5	3×3	1.12	123.8	6×2	1.12	80.5	105.5∗
4	3×3	1.06	127.0	3×3	1.06	127.0	6×2	1.13	78.8	102.0∗
5	4×2	1.00	118.6	4×2	1.00	118.6	4×2	1.00	76.2	118.6
6	3×3	1.00	137.2	3×3	1.00	137.2	3×3	1.00	80.4	137.2
7	3×3	1.00	137.1	3×3	1.00	137.1	3×3	1.00	80.4	137.1
8	3×3	1.11	120.9	3×3	1.11	120.9	6×2	1.13	79.1	102.6∗
9	3×3	1.02	126.1	3×3	1.02	126.1	3×3	1.02	77.9	126.1
10	4×2	1.45	93.7	4×2	1.45	93.7	4×2	1.45	56.6	93.7
11	2×2	1.23	81.4	2×2	1.23	81.4	2×2	1.23	51.5	81.4
12	4×2	1.48	96.0	3×2	1.36	95.4	3×2	1.36	57.4	95.4
13	3×2	1.40	92.8	3×2	1.40	92.8	3×2	1.40	55.8	92.8
14	3×2	1.47	84.2	3×2	1.47	84.2	3×2	1.47	50.4	84.2
15	2×1	1.00	69.1	2×1	1.00	69.1	2×1	1.00	52.7	69.1
16	4×2	1.66	80.2	4×1	1.43	78.0	4×1	1.43	51.7	78.0
17	3×1	1.59	67.6	4×1	1.75	67.4	6×1	1.98	44.2	54.5∗
18	2×1	1.36	43.1	2×1	1.36	43.1	2×1	1.36	32.4	43.1
20	1×2	1.17	68.2	1×2	1.17	68.2	1×2	1.17	43.1	68.2
21	3×1	1.59	67.9	4×1	1.77	67.1	5×1	1.88	43.5	65.7
23	2×1	1.46	48.8	1×1	1.00	46.7	2×1	1.46	32.3	48.8
24	1×1	1.00	57.2	1×1	1.00	57.2	2×1	1.52	37.3	57.1
25	1×1	1.00	42.8	1×1	1.00	42.8	1×1	1.00	29.5	42.8
26	1×1	1.00	41.9	1×1	1.00	41.9	1×1	1.00	28.0	41.9
27	2×1	1.53	46.7	1×1	1.00	46.4	2×1	1.53	32.1	46.7
28	1×1	1.00	53.5	1×1	1.00	53.5	1×1	1.00	35.1	53.5
29	2×2	1.98	44.7	1×1	1.00	43.5	2×2	1.98	29.0	44.7
36	1×1	1.00	37.2	1×1	1.00	37.2	1×1	1.00	26.9	37.2
37	2×2	1.98	44.8	1×1	1.00	43.4	2×2	1.98	29.0	44.8
40	1×1	1.00	48.3	1×1	1.00	48.3	1×1	1.00	35.2	48.3
41	1×1	1.00	34.6	1×1	1.00	34.6	1×1	1.00	28.6	34.6
42	1×1	1.00	34.8	1×1	1.00	34.8	1×1	1.00	28.6	34.8
44	1×1	1.00	29.1	1×1	1.00	29.1	1×1	1.00	26.1	29.1

Table 4: **Block size summary data for the Intel Pentium III platform**. An asterisk (∗) by a heuristic performance value indicates that this performance was less than 90% of the best performance.

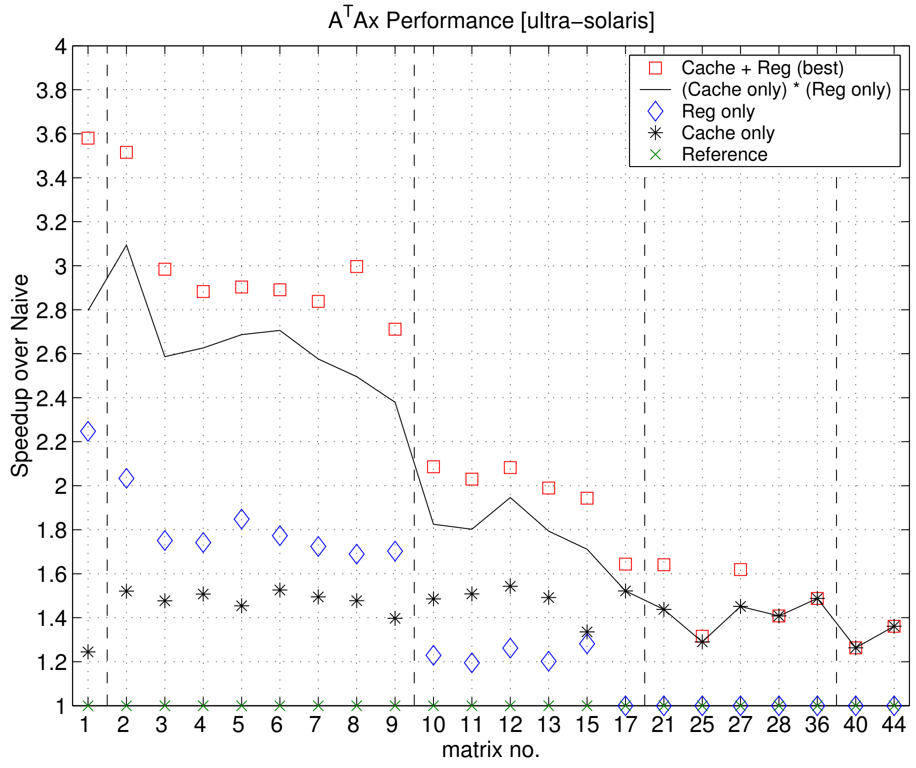


Figure 15: **Combined effect of register blocking and the cache optimization on the Sun Ultra 2i platform**. Here, we see the synergistic effect of combining register blocking and cache blocking—the observed speedup of combining these optimizations (squares) is greater than the product of (cache optimization only speedup) and (register blocking only speedup), shown as a solid line.

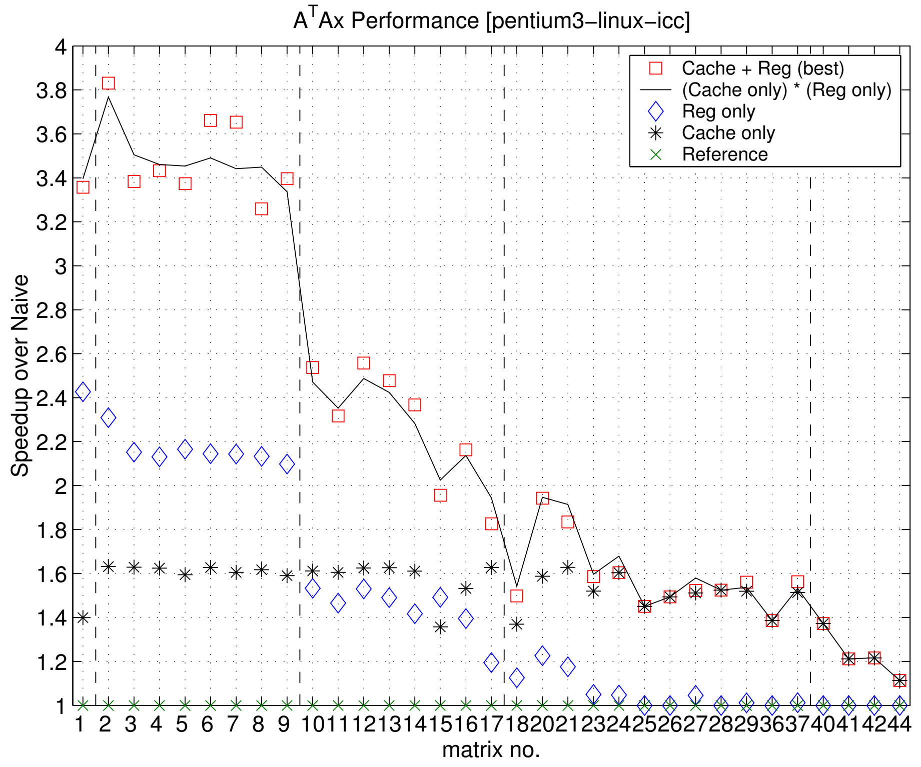


Figure 16: **Combined effect of register blocking and the cache optimization on the Intel Pentium III platform**. The observed speedup of combining register and cache optimizations equals the product of (cache optimization only speedup) and (register blocking only speedup), shown as a solid line.

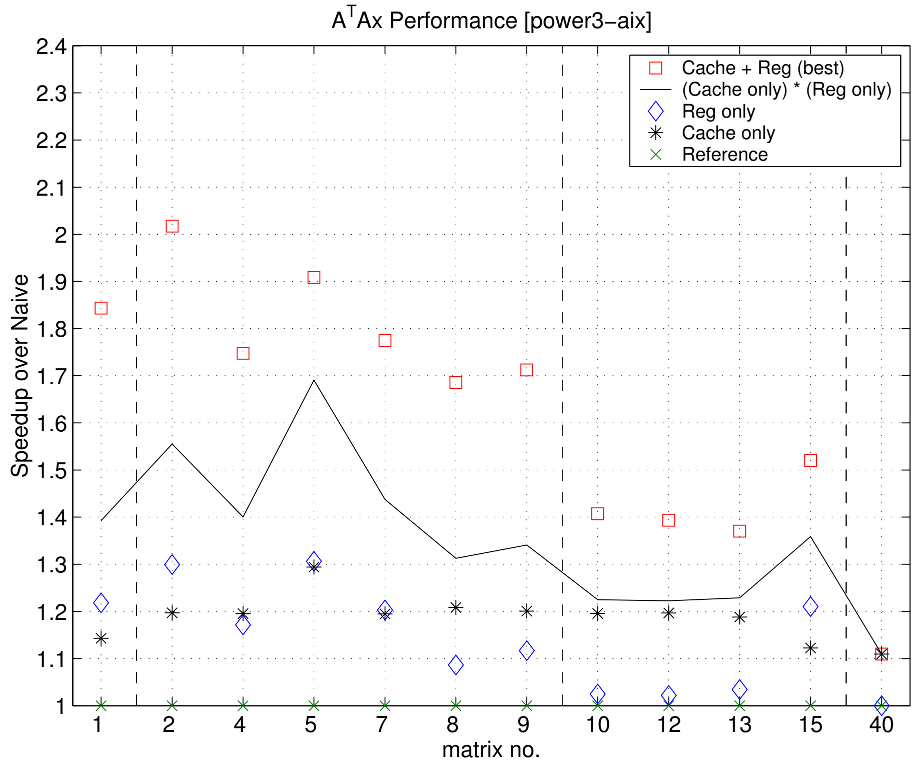


Figure 17: **Combined effect of register blocking and the cache optimization on the IBM Power3 platform**. Here, we see the synergistic effect of combining register blocking and cache blocking—the observed speedup of combining these optimizations (squares) is greater than the product of (cache optimization only speedup) and (register blocking only speedup), shown as a solid line.

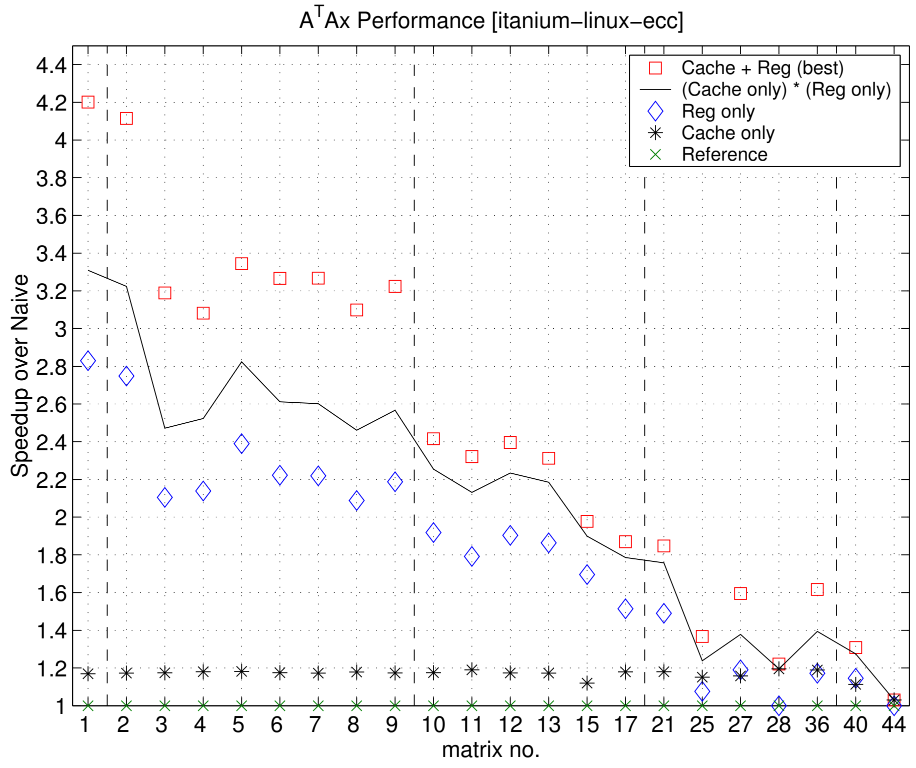


Figure 18: **Combined effect of register blocking and the cache optimization on the Intel Itanium platform**. Here, we see the synergistic effect of combining register blocking and cache blocking—the observed speedup of combining these optimizations (squares) is greater than the product of (cache optimization only speedup) and (register blocking only speedup), shown as a solid line.

No.	Best cache-opt. + reg. blocking $r_{\text{opt}} \times c_{\text{opt}}$	Fill	Mflop/s	Heuristic cache-opt. + reg. blocking $r_{\text{heur}} \times c_{\text{heur}}$	Fill	Mflop/s	Reg. blocking only $r_{\text{reg}} \times c_{\text{reg}}$	Fill	Mflop/s	cache-opt. + reg. block $r_{\text{reg}} \times c_{\text{reg}}$ Mflop/s
1	4×4	1.00	264.6	4×4	1.00	264.6	2×4	1.00	174.9	254.8
2	4×4	1.00	275.3	4×4	1.00	275.3	4×2	1.00	177.3	267.3
4	3×6	1.12	238.7	3×3	1.06	233.8	3×2	1.07	160.0	232.7
5	4×4	1.00	237.4	4×4	1.00	237.4	4×2	1.00	162.5	232.7
7	3×3	1.00	241.0	3×3	1.00	241.0	3×3	1.00	163.4	241.0
8	3×6	1.13	224.6	6×2	1.13	216.5	2×2	1.10	144.7	207.5
9	3×3	1.02	234.1	3×3	1.02	234.1	3×3	1.02	152.7	234.1
10	2×1	1.10	189.2	2×1	1.10	189.2	2×1	1.10	137.8	189.2
12	2×2	1.24	194.5	2×1	1.13	192.8	2×1	1.13	142.6	192.8
13	2×1	1.14	194.7	2×1	1.14	194.7	2×1	1.14	147.0	194.7
15	2×1	1.00	182.2	2×1	1.00	182.2	2×1	1.00	145.0	182.2
40	1×1	1.00	139.1	1×1	1.00	139.1	1×1	1.00	125.4	139.1

Table 5: **Block size summary data for the IBM Power3 platform**. An asterisk (∗) by a heuristic performance value indicates that this performance was less than 90% of the best performance.

E Execution Time Breakdown

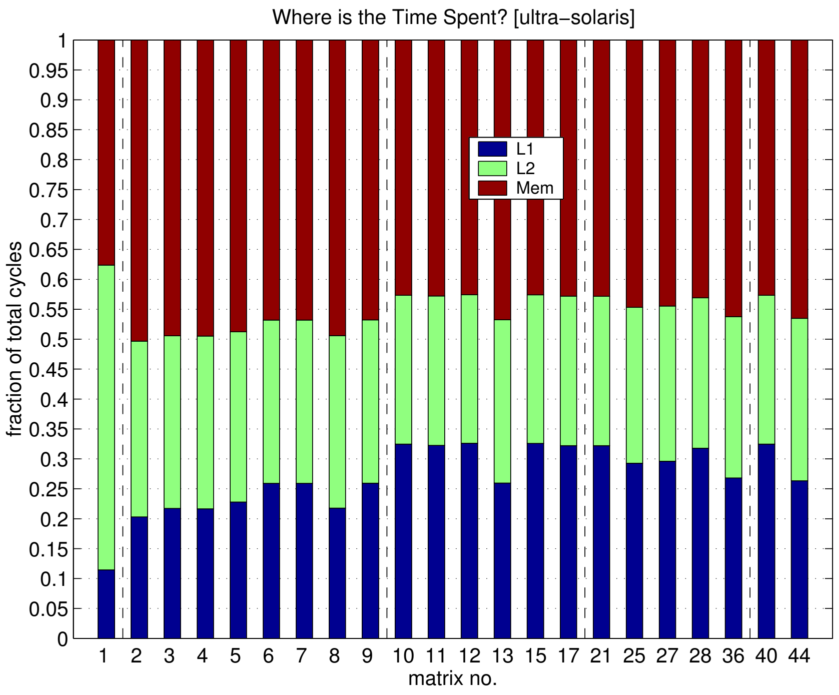
Equation (4) includes a term at each level of the memory hierarchy, from the L1 cache to main memory. Figures 19–22 shows the contribution of each term to the total execution time, according to (1) our analytic counts, Equations (6)–(10), and (2) the observed PAPI counts. Data for the $r_{\text{up}} \times c_{\text{up}}$ block size is shown.

In Figure 19 (*top*), observe that our bounds model charges 40–50% of the execution time on the Ultra 2i to memory accesses (Figure 19, *top*). However, on the other three platforms, the model evidently charges 50–75% of the execution time to memory accesses. Furthermore, on the Pentium III, Power3, and Itanium machines, the time spent in the largest level cache is practically negligible according to the model—*i.e.*, the external cache is essentially transparent, failing to reduce the overall cost of memory operations. Note that (1) SpA^TA spends most of its time streaming through the matrix, and (2) the cache line sizes of the on-chip and off-chip caches are the same on the Pentium III, Power3, and Itanium. Thus, a compulsory miss on an element of the matrix in the on-chip cache is also a miss in the off-chip cache. In hardware, this effect could have been alleviated by ensuring that the line sizes are strictly increasing with increasing cache level.

In Figures 19–22 (*bottom*), we evaluate the execution time model using true cache miss counts. Since the true counts agree with the model for the external caches (see Section 4.2), we see the top and bottom plots of each figure match well.

No.	Best cache-opt. + reg. blocking $r_{\mathrm{opt}} \times c_{\mathrm{opt}}$	Fill	Mflop/s	Heuristic cache-opt. + reg. blocking $r_{\mathrm{heur}} \times c_{\mathrm{heur}}$	Fill	Mflop/s	Reg. blocking only $r_{\mathrm{reg}} \times c_{\mathrm{reg}}$	Fill	Mflop/s	cache-opt. + reg. block $r_{\mathrm{reg}} \times c_{\mathrm{reg}}$ Mflop/s
1	8×8	1.00	347.1	8×8	1.00	347.1	8×1	1.00	233.7	314.9
2	8×8	1.00	323.4	8×8	1.00	323.4	8×1	1.00	216.1	297.3
3	6×6	1.12	249.1	6×6	1.12	249.1	3×1	1.06	164.4	200.1∗
4	3×3	1.06	235.4	3×3	1.06	235.4	2×2	1.07	163.3	207.7∗
5	4×2	1.00	238.6	4×2	1.00	238.6	4×2	1.00	170.5	238.6
6	3×3	1.00	256.4	3×3	1.00	256.4	3×1	1.00	174.4	212.7∗
7	3×3	1.00	257.2	3×3	1.00	257.2	3×1	1.00	174.7	213.0∗
8	6×6	1.15	237.6	6×6	1.15	237.6	3×1	1.06	160.1	197.9∗
9	3×3	1.02	252.2	3×3	1.02	252.2	3×1	1.01	171.1	208.2∗
10	4×2	1.45	183.6	4×2	1.45	183.6	4×1	1.33	145.8	180.5
11	2×2	1.23	158.7	2×2	1.23	158.7	2×2	1.23	122.5	158.7
12	4×2	1.48	187.4	4×2	1.48	187.4	4×1	1.37	148.8	182.5
13	4×2	1.54	181.2	4×2	1.54	181.2	4×1	1.40	146.0	179.2
15	2×2	1.35	145.7	2×2	1.35	145.7	2×2	1.35	124.9	145.7
17	4×1	1.75	142.4	4×1	1.75	142.4	4×1	1.75	115.3	142.4
21	4×1	1.77	140.9	4×1	1.77	140.9	4×1	1.77	113.7	140.9
25	3×1	2.37	67.4	2×1	1.71	64.5	2×1	1.71	53.0	64.5
27	3×1	1.94	85.4	3×1	1.94	85.4	3×1	1.94	63.8	85.4
28	2×2	2.54	86.2	1×1	1.00	84.4	1×1	1.00	70.6	84.4
36	3×1	2.31	65.1	2×2	2.31	62.1	3×1	2.31	47.2	65.1
40	3×1	1.99	97.0	3×1	1.99	97.0	3×1	1.99	84.9	97.0
44	1×1	1.00	48.9	1×1	1.00	48.9	1×1	1.00	47.4	48.9

Table 6: **Block size summary data for the Intel Itanium platform**. An asterisk (∗) by a heuristic performance value indicates that this performance was less than 90% of the best performance.



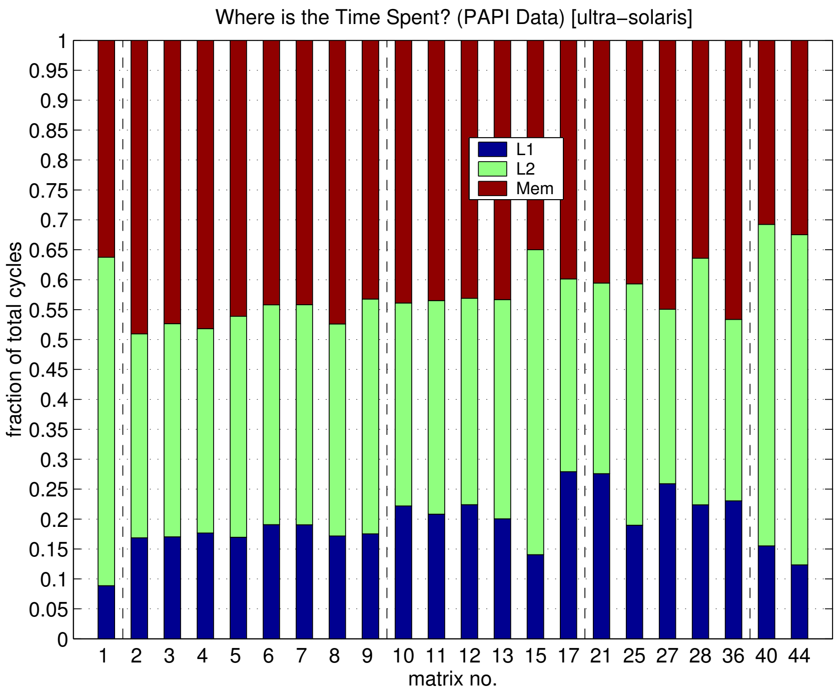
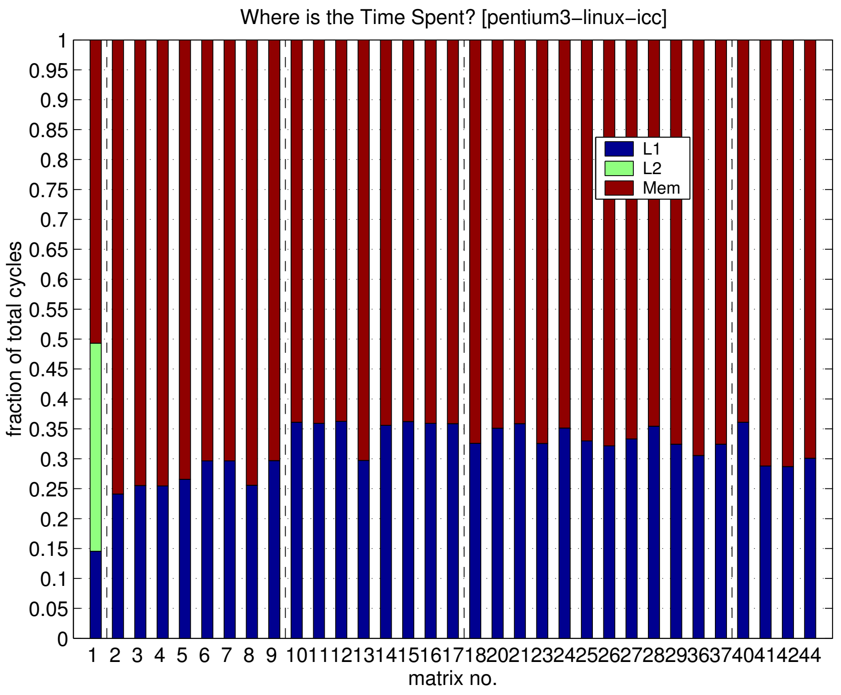


Figure 19: **Execution time breakdown on the Sun Ultra 2i platform**. (*Top*) Fraction of time spent in the L1 cache, L2 cache, and main memory for each matrix, according to the terms of Equation (4) and our analytic load and miss counts, Equations (6)–(10). (*Bottom*) Fraction of time spent in each level of the memory hierarchy, where we evaluate Equation (4) by substituting the true, measured cache hits as reported by PAPI.



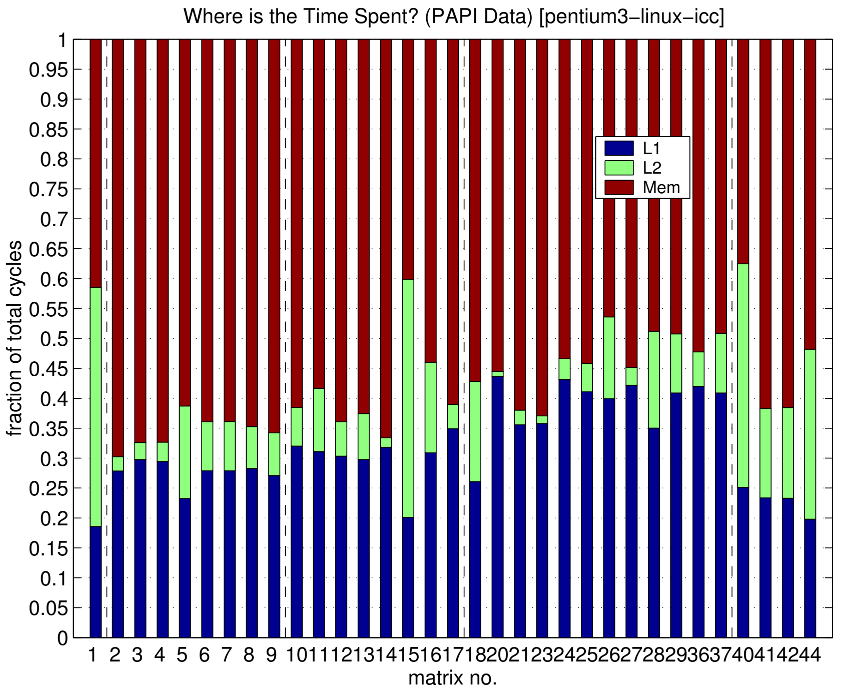
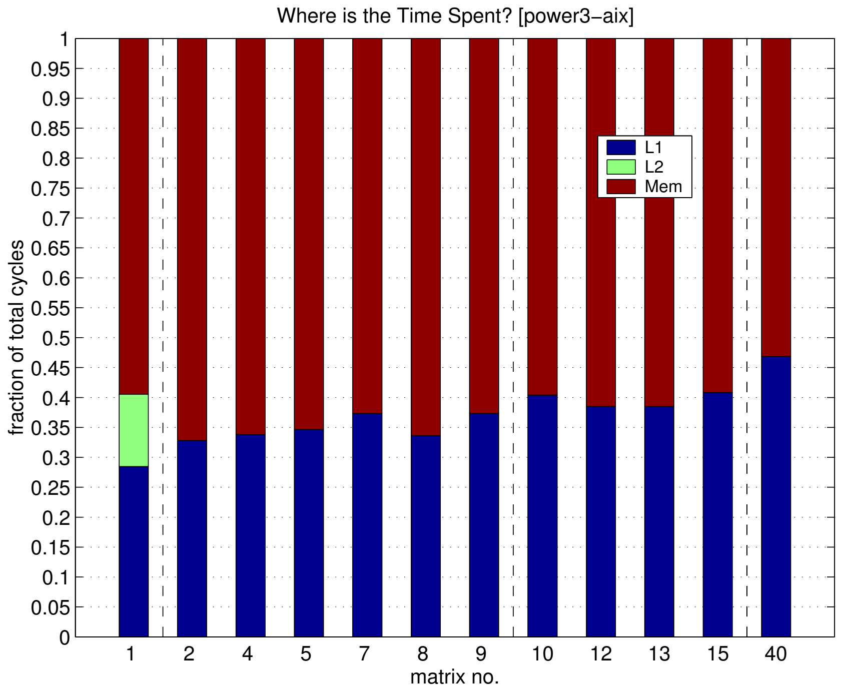


Figure 20: **Execution time breakdown on the Intel Pentium III platform**.



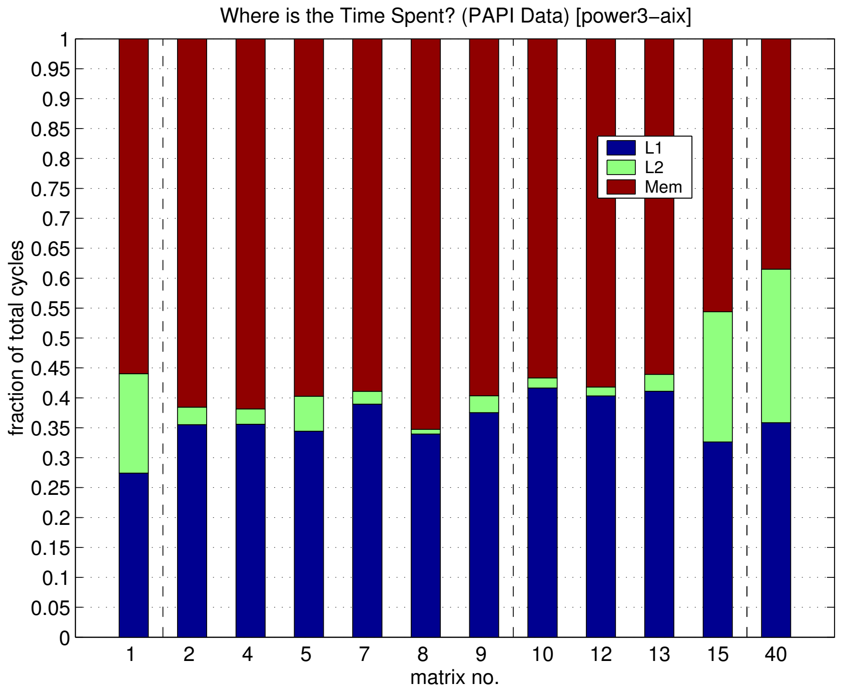


Figure 21: **Execution time breakdown on the IBM Power3 platform**.

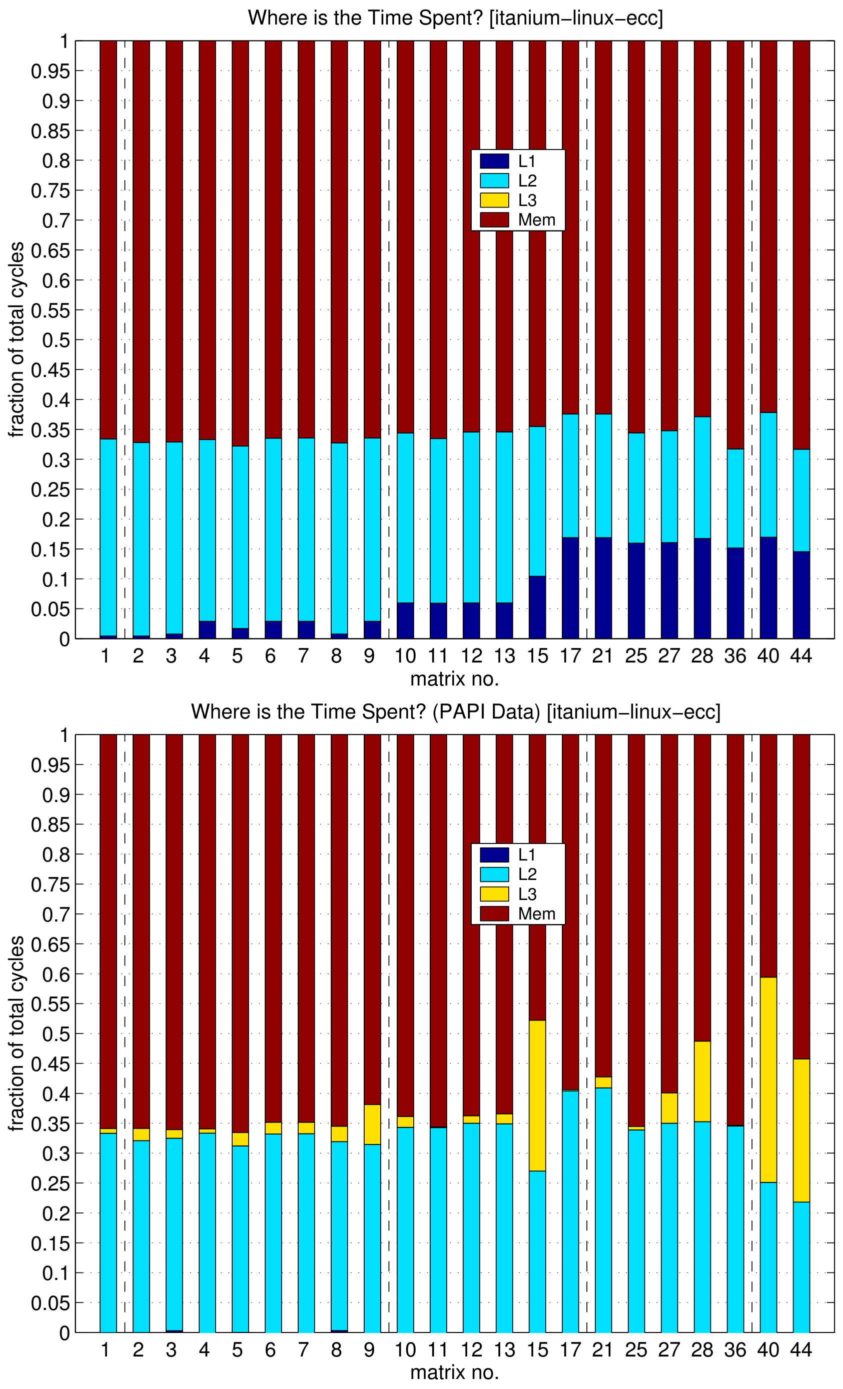


Figure 22: **Execution time breakdown on the Intel Itanium platform.**