Improving Locality of Nonserial Polyadic Dynamic Programming

Guangming Tan[1,2], Ninghui Sun[1], Dongbo Bu[1]

1. Institute of Computing Technology, Chinese Academy of Sciences
2. Graduate School of Chinese Academy of Sciences
{tgm, snh, bdb}@ncic.ac.cn

Abstract

Dynamic programming (DP) is a commonly used technique for solving a wide variety of discrete optimization problems, which have different variants of dynamic programming formulation. This paper investigated one important DP formulation, which called nonserial polyadic dynamic programming formulation and time complexity is $O(n^3)$. We exploit the property of the algorithm to develop a high performance implementation using the combination of cache-oblivious and cache-conscious strategy. The efficiency in our improved algorithm comes from two sources: reducing the number of cache misses and TLB misses. Experiments on three modern computing platforms show a performance improvement of 2-10 times over a standard implementation of DP formulation.

1 Introduction

The processor-memory gap has been extensively investigated for scientific computing such as linear algebra [1] and FFT [2]. However, optimizing cache performance to achieve better overall performance is a difficult problem. Whaley and Dongarra discuss optimizing the widely used Basic Linear Algebra Subroutines(BLAS) using cache conscious strategy [3]. Modern microprocessors are including deeper and deeper memory hierarchies to hide the cost of cache misses. Different miss penalties for each level of the memory hierarchy as well as the TLB also play an important role in the effectiveness of cache friendly optimizations. These miss penalties vary from processor to processor and can cause large variations in experimental results. Another approach to improving the performance of the cache is to design cache oblivious algorithms, which is explored by Frigo, et al. [4]in to discusses the cache performance of cache oblivious algorithms for matrix multiplication/transpose, FFT, and sorting. Using cache oblivious approach, the algorithms do not ignore the presence of a cache, but rather they use recursion to improve performance regardless of the size or organization of the cache. By doing this, they can improve the performance of the algorithm without tuning the application to specifics of the host machine. Although much of the focus of cache optimization has been on dense linear algebra problems, there has also been some work that focuses on irregular computation problem such as graph algorithm and optimization problems [5] [6][7].

Supported by Youth Fund of Institute of Computing Technology, Chinese Academy of Sciences

Dynamic programming (DP) is a commonly use technique for solving a wide variety of discrete optimization problems such as scheduling, string-editing, packaging and inventory management. More recently, it has found applications in bioinformatics in the Smith-Waterman algorithm [8] for matching sequences of amino-acids and necleotides and Zuker [9] algorithm for predicting RNA secondary structures. Grama, et al. model the dependencies in DP formulation as a directed graph and classify them into four classes of DP formulation [10]: serial monadic (single source shortest-path problem, 0/1 knapsack problem), serial polyadic (Floyd all pairs shortest paths algorithm), nonserial monadic (longest common subsequence problem, Smith-Waterman algorithm) and nonserial polyadic (Optimal matrix parenthesization problem and Zuker algorithm). Venkataraman et al. [6] present a blocked implementation of the Floyed-Warshall algorithm to improve the cache performance. Park et, al. [7] proposed another recursive implementation and consider data layouts to avoid conflict misses in the cache. The improved DP implementations in their works belong to serial polyadic. Our work focus on nonserial polyadic problem. In this paper, we develop a high performance implementation of nonserial polyadic dynamic programming algorithm through the combination of cache conscious and oblivious approaches.

The remainder of this paper is organized as follows: In section 2, based on the optimal matrix parenthesization problem, we abstract a general DP formulation that is simplified and keep the same computational characteristics with the original formulation. In section 3, we discuss our optimizations of the nonserial polyadic DP formulation. In section 4, we present our experimental results and , finally, in Section 5, we draw conclusions.

2 The Optimal Matrix Parenthesization Problem

Consider the problem of multiplying n matrices, $\{A_1, A_2, ..., A_n\}$, where each A_i is a matrix with r_{i-1} rows and r_i columns. The order in which the matrices are multiplied has a significant impact on the total number of operations required to evaluate the product. The objective of the parenthesization problem is to determine a parenthesization that minimizes the number of operations. Enumerating all possible parenthesizations is not feasible since there are exponentially many of them.

Let $m[i][j]$ be the optimal cost of multiplying the matrices $\{A_i, ..., A_j\}$. This chain of matrics can be expressed as a product of two smaller chains, $\{A_i, ..., A_k\}$ and $\{A_{k+1}, ..., A_j\}$. The cost of multiplying these two matrices is $f(i, j, k) = r_{i-1}r_kr_j$. Hence, the cost of the parenthesization $\{A_i, ..., A_k\}\{A_{k+1}, ..., A_j\}$ is given by $m[i][k] + m[k+1][j] + r_{i-1}r_kr_j$. This gives rise to the following recurrence relation for the parenthesization problem:

$$m[i,j] = \begin{cases} min_{i \le k < j}\{m[i,j], m[i,k]+ & \\ m[k+1,j] + f(i,j,k)\} & 0 \le i < j < n \\ 0 & j = i, 0 \le i < n \end{cases} \tag{1}$$

Equation 1 can be solved if we use a bottom-up approach for constructing the table m that stores the values $m[i][j]$. The algorithm fills table m in an order corresponding to solving the parentesization problem on matrix chains of increasing length and $m[0][n-1]$ is the minimum cost. Visualize this by thinking of filling the table shown as in Figure 1. The value of $m[i][j]$ (red point) depends on the previous computed value in the same row and column. In general, $f(i, j, k)$ is a $O(1)$ computation. So for simplicity, we rewrite DP formulation (1) as:

$$m[i,j] = \begin{cases} min_{i \le k < j}\{m[i][j], m[i,k] + m[k+1,j]\} & \\ \qquad\qquad\qquad\qquad 0 \le i < j < n & \\ 0 & \\ \qquad\qquad\qquad\qquad j = i, 0 \le i < n & \end{cases} \tag{2}$$

$f(i, j, k)$ is independent of the entry $m[i][j]$ in DP table. The simplified DP dynamic programming formulation (2) doesn't change the data dependency in filling the DP table and can be easily implemented using iterative nested three loops:

```
**dp_standard(matrices m, int n)**
for ($j = 1; j \le n; j++$)
    for ($i = j; i \ge 1; i--$) {
        $t = m[i][j]$
        for ($k = i; k < j; k++$)
            t=min2(t, m[i][k]+m[k+1][j])
        $m[i][j] = t$
    }
```

Although equation (2) has similar nested three loops with Floyd algorithm and matrix multiply and the same time complexity $\Theta(n^3)$, it is a quit different algorithm.

- The matrix multiply can use any of the six possible permutations for the order of the three loops without affecting the computed results; In Floyd algorithm, only the order of the innermost two loops may be changed, the outermost loop must remain outermost; In **dp_standard**, only the order of the outermost two loops may be changed, the innermost loop must remain outermost;

- Any entry $m[i][j]$ in Floyd algorithm only depend the const number of previous values; In **dp_standard**, the number of previous entries for computing $m[i][j]$ varies with i and j.

In next section, we proposed a recursive algorithm for implementing the simple nested three loops to improve its cache performance.

3 A Recursive Implementation

Before presenting the recursive implementation, we need a coordinates transformation for equation (2). Assume (i, j) is in the original coordinates and n is problem size. Setting $i' = i, j' = j + 1$, in the new coordinates, the new problem size is $n' = n + 1$. We can rewrite (2) as:

$$m[i',j'] = \begin{cases} min_{i'+1\le k'<j'}\{m[i'][j'], m[i',k']+m[k',j']\} \\ \qquad\qquad\qquad\qquad 0\le i'<j'<n' \\ 0 \\ \qquad\qquad\qquad\qquad j'=i', 0\le i'<n' \end{cases} \tag{3}$$

In fact, this transformation adds a new diagonal to the original DP table. However, the entries in this new diagonal are unused and don't contribute to the computations (See the gray points at the diagonal in Figure 1). Accordingly, the minimum cost is the value of $m[0][n]$ or $m[0][n'-1]$

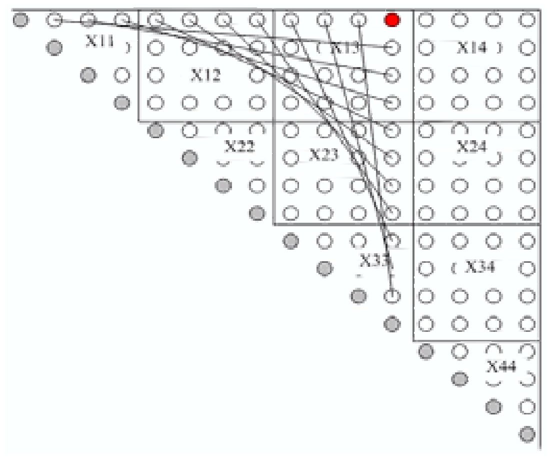

Figure 1. The blocked DP table. The size is $n' \times n'$, $X_{11}, X_{22}, X_{33}, X_{44}$ are three triangular matrices, which size are $\frac{n'}{4} \times \frac{n'}{4}$. $X_{12}, X_{13}, X_{14}, X_{23}X_{24}, X_{34}$ are six rectangular matrices, which size are $\frac{n'}{4} \times \frac{n'}{4}$

Assume that n' is a power of two. If n' is not a power of two, some additional unused entries are added to the original DP table and the new size is a power of two. When implementing the algorithm, several branch instructions can avoid the unused computations. As experiment shown, the overhead of branch can be ignored. So in this section, only that $n' = 2^k$ is a power of two is considered. The DP matrices is partitioned into ten sub-matrices, which consist of three triangular matrices and six rectangular matrices (See Figure 1). X is the original DP matrices with 2^k size, then it is partitioned as follows:

$$\mathbf{X} = \begin{pmatrix} X_{11} & X_{12} & X_{13} & X_{14} \\ & X_{22} & X_{23} & X_{24} \\ & & X_{33} & X_{34} \\ & & & X_{44} \end{pmatrix}$$

According to equation 3, the sub-matrices along diagonal $X_{11}, X_{22}, X_{33}, X_{44}$ are self-contained, that is, any entries only depend on other entries in the same sub-matrices. Furthermore, X_{12} only depends on X_{11} and X_{22}, X_{34} only depends on X_{33} and X_{44}. If combining X_{11}, X_{12}, X_{22} and X_{33}, X_{34}, X_{44} into two larger sub-matrices, respectively, we get two independent DP matrices with 2^{k-1} size and can be divided recursively. Thus, recursive function G is defined:

$$\begin{pmatrix} \text{A} & \text{C} \\ & \text{B} \end{pmatrix} = G \begin{pmatrix} \text{A} & \text{C} \\ & \text{B} \end{pmatrix}$$

where A and B is triangular matrices, C is a rectangular matrices. All entries in three matrices are unknown. Thus, the two DP sub-matrices are computed recursively using G

$$\begin{pmatrix} X_{11} & X_{12} \\ & X_{22} \end{pmatrix} = G \begin{pmatrix} X_{11} & X_{12} \\ & X_{22} \end{pmatrix}$$

$$\begin{pmatrix} X_{33} & X_{34} \\ & X_{44} \end{pmatrix} = G \begin{pmatrix} X_{33} & X_{34} \\ & X_{44} \end{pmatrix}$$

After $X_{11}, X_{12}, X_{22}, X_{33}, X_{34}, X_{44}$ are computed, we can solve the remainder four rectangular sub-matrices. Because of the data dependencies, X_{23} should be computed firstly and can only depends on X_{22} and X_{33}. At this time, X_{22} and X_{33} have been computed, so X_{23} can be computed immediately. We define another recursive function F:

$$\begin{pmatrix} \text{A} & \text{C} \\ & \text{B} \end{pmatrix} = G \begin{pmatrix} \text{A} & \text{C} \\ & \text{B} \end{pmatrix}$$

where A and B is triangular matrices, C is a rectangular matrices. A and B have been computed and C is an unknown matrices only depend on A and B. X_{23} is computed using recursive F:

$$\begin{pmatrix} X_{22} & X_{23} \\ & X_{33} \end{pmatrix} = F \begin{pmatrix} X_{22} & X_{23} \\ & X_{33} \end{pmatrix}$$

Now, we define two tensor operations $\otimes$ and $\oplus$. Let matrices $A = (a_{ij})_{s\times s}$,$B = (b_{ij})_{s\times s}$,$C = (c_{ij})_{s\times s}$. $\otimes$ is defined:

$$C = A \otimes B, \quad where \quad c_{ij} = min_{k=1}^{n}\{c_{i,j}, a_{i,k} + b_{k,j}\}$$

$\oplus$ is defined:

$$C = A \oplus B, \quad where \quad c_{ij} = min\{a_{i,j}, b_{i,j}\}$$

After X_{23} has been computed, both X_{13} and X_{24} can be computed. Assume that we first compute X_{13}. Because X_{12} and X_{23} are known rectangular sub-matrices, we can compute the partial results of sub-matrices X_{23} using equation (4)

$$X_{13} = X_{13} \oplus (X_{12} \otimes X_{23}) \tag{4}$$

then, we complete the computation of X_{13} by recursive function F:

$$\begin{pmatrix} X_{11} & X_{13} \\ & X_{33} \end{pmatrix} = F\begin{pmatrix} X_{11} & X_{13} \\ & X_{33} \end{pmatrix}$$

Through the same method, X_{24}, X_{14} also can be computed:

$$X_{24} = X_{24} \oplus (X_{23} \otimes X_{34}) \tag{5}$$

$$\begin{pmatrix} X_{22} & X_{24} \\ & X_{44} \end{pmatrix} = F\begin{pmatrix} X_{22} & X_{24} \\ & X_{44} \end{pmatrix}$$

$$X_{14} = X_{14} \oplus (X_{12} \otimes X_{24}) \tag{6}$$

$$X_{14} = X_{14} \oplus (X_{13} \otimes X_{34}) \tag{7}$$

$$\begin{pmatrix} X_{11} & X_{14} \\ & X_{44} \end{pmatrix} = F\begin{pmatrix} X_{11} & X_{14} \\ & X_{44} \end{pmatrix}$$

Dividing the DP matrices recursively into smaller sub-matrices, when the size of sub-matrices is small enough to be contained in cache totally, the recursive can stop. For this small sub-matrices, **dp_standard** is called to finish computing in recursive function G, where all entries in the sub-matrices are unknown. For the small sub-matrices in recursive function F, only one of the three sub-matrices need to be computed. So **dp_return** is defined to be called in this case.

```
dp_return(matrices m, int n)
for ($j = n/2; j < n; j++$)
    for ($i = n/2 - 1; i \geq 0; i--$) {
        $t = m[i][j]$
        for ($k = i + 1; k < j; k++$)
            t=min2(t, m[i][k]+m[k][j])
        $m[i][j] = t$
    }
```

Equations (4)(5)(6)(7) are four elementary operations in the recursive implementation and have an uniform formal: $C = C \oplus (A \otimes B)$, which is implemented as:

```
dp_base(matrices A, matrices B, matrices C, int n)
for ($i = 0; i < n; i++$)
    for ($j = 0; j < n; j++$) {
        $t = C[i][j]$
        for ($k = 0; k < n; k++$)
            t=min2(t, A[i][k]+B[k][j])
        $C[i][j] = t$
    }
```

It is obvious that **dp_base** is analogous to dense matrix multiply, thus most of optimization in dense matrix multiply also can be applied to **dp_base**.

Table 1. Machine configuration for the various platform used for experiments. The cache are of the form Capacity/Block size(C/B).

Parameter	Opteron	Xeon	PowerPC
clock rate	1.6Ghz	2.4GHz	400Mhz
L1 data cahce	64KB/32B	8KB/64B	32KB/128B
L2 cache	1MB/64B	512KB/64B	1MB/128B
RAM	3.5GB	1GB	1GB
L1 data TLB entries	32	64	128
L1 TLB associativity	direct	direct	2
L2 TLB entries	512	512	512
L2 TLB associativity	8	8	16
VM page size	4KB	4KB	64KB
Compiler	pgcc	icc	xlc
Option	-O3	-O3	-O3
Operating system	SuSE	Redhat	AIX

4 Experimental Results

In this section we compare the performance of the standard and recursive implementation with experimental results on a number of machines. We experimented on five commonly used modern computing platforms:Opteron, Xeon(P4) and PowerPC 604e under different compiler environments. Table 1 lists the configurations for the machines. We are primarily interested in execution times of the algorithms and use L1 cache misses, L2 cache misses, and TLB misses to explain the trends in execution time.

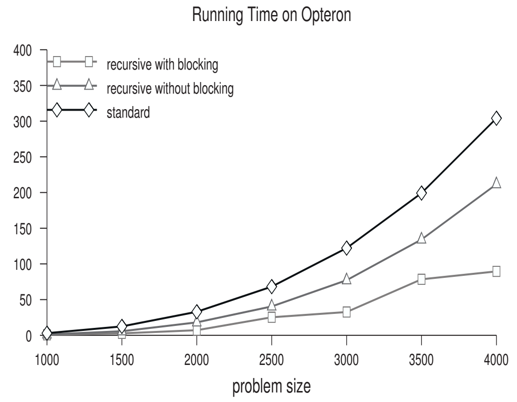

Figure 2. Running times comparison on Opteron

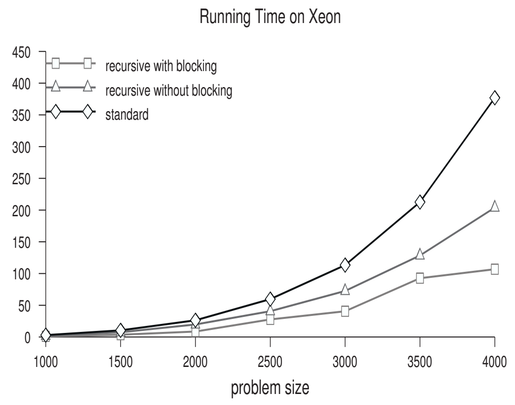

Figure 3. Running times comparison on Xeon

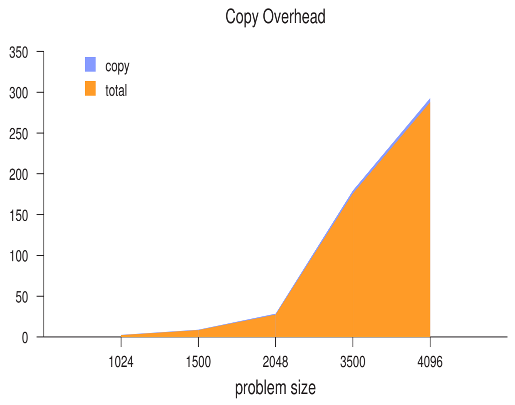

Figure 5. The copy overhead

4.1 Running Time

First, we evaluate the pure recursive algorithm without blocking, that is, the recursive return until the size of matrices is 1. Figure 2,3,4 shows the execution time plots comparing the standard DP implementation with recursive algorithm on three different platforms, respectively. The plots clearly demonstrate the recursive algorithm outperform the standard algorithm. The recursive algorithm achieve average 50% faster than the standard algorithm. The speedups increase on three platform with the problem size. The greater speedup in the case of 4000 problem size is about 2 times.

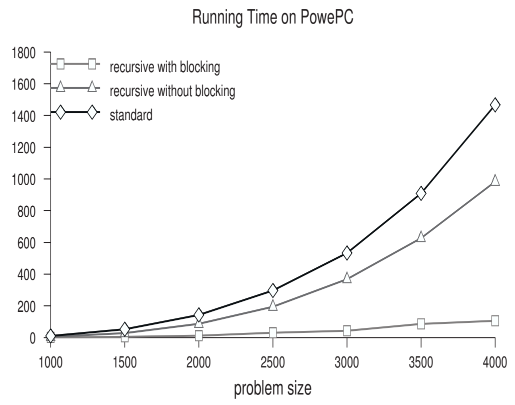

Figure 4. Running times comparison on PowerPC 604

The performance of blocking is largely sensitive to the block size and cache size, in our current work, we only select an appropriate by experimental statistical method instead of analytical model. We measured the running time with various block sizes on three platforms and choose the block size combinations (16, 8), (16 32), (32,32) for Opteron, Xeon and PowerPC, respectively. Figure 2 shows the execution time plots on the Opteron. The blocked recursive algorithm runs about 2-5 times faster than the standard algorithm. Figure 3 shows the execution time plots on the Xeon. The blocked recursive algorithm runs about 1-3 times faster than the standard algorithm.Figure 4 shows the execution time plots on the PowerPC. The blocked recursive algorithm runs about 10 times faster than the standard algorithm. The greater speedups on the PowerPC is because then latency of cache misses is more than the other two machines. For example, the L1 cache miss latency on the Xeon is 13.19ns, however, PowerPC need 80.05ns for L1 cache miss latency. In the implementation of blocked recursive algorithm, the sub-matrices are copied and constructed a new matrices. However, the copy overhead is much smaller than the total running time (See Figure 5).

4.2 Memory Behavior

The running time of the recursive algorithm on the different machines above varies a function of problem size. This variation arises from the way the code exercises different components of the memory system. We now only explore these effects using Oprofile, which is performance monitor tool on Opteron.

4.2.1 Memory References

Figure 6 compares the memory references of the standard DP implementation and the recursive implementation with and without blocking. The recursive without blocking doesn't reduce the number of memory references. When computing equations (4)(5)(6)(7), The recursive with blocking copies sub-matrices to new matrices, then computes the blocked new matrices. Although resulting in copy overhead, this method reduces the number of memory references greatly.

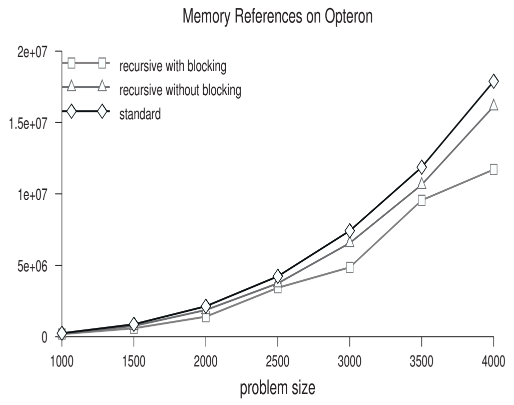

Figure 6. Memory References comparison on Opteron

4.2.2 L1 Data Cache

Figure 7 shows that L1 data cache misses for three implementations on the Opteron. The recursive algorithm without blocking reduces the number of L1 data cache misses 40% than the standard implementation. In the recursive implementation, the number of L1 data cache misses decreases 1 times than the standard implementation. We note that the recursive without blocking is more sensitive to the problem size than the recursive with blocking. In the experiment, the problem size varies with 500. The number of L1 data cache misses in recursive without blocking increases with the increasing problem size by 500. However, when the problem size is increased by 500, the number of L1 data cache misses in recursive with blocking almost doesn't increase. Only when the problem size is increased by 1000, we notice the notable increase in cache misses.

4.2.3 L2 Data Cache

Figure 8 shows the L2 cache misses for three implementations on the Opteron. The recursive algorithm without blocking reduces the number of L2 data cache misses 1-2 times than the standard implementation. But the L2 cache misses have a notable increase when the problem size is larger than 3000. The recursive algorithm without blocking reduces the number of L2 data cache misses about 5 times than the standard implementation. Although the number of L2 cache misses is much smaller than that of L1 data cache misses, the miss latency of L2 cache misses is 2-3 times than that of L1 cache misses. So the decreasing of L2 cache misses is also important.

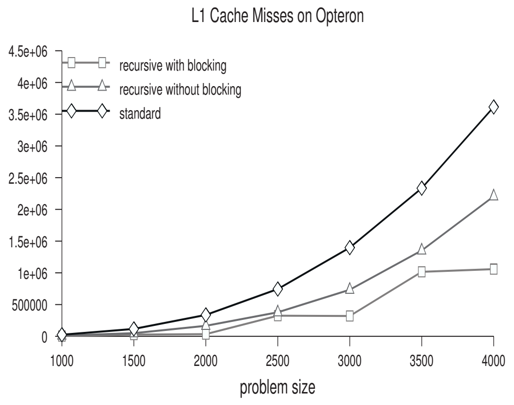

Figure 7. L1 Cache misses comparison on Opteron

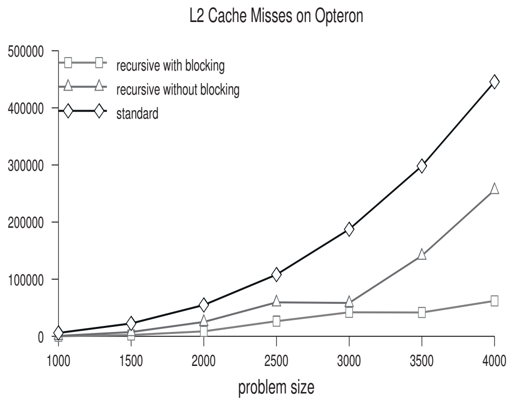

Figure 8. L2 Cache misses comparison on Opteron

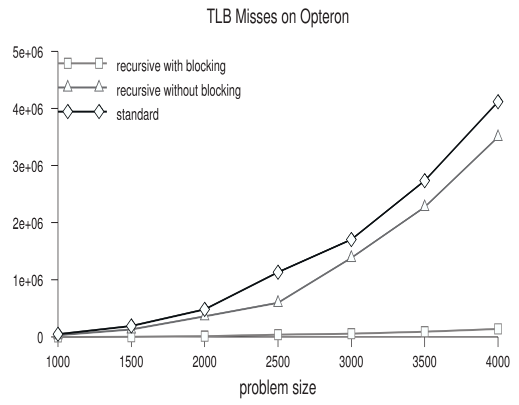

Figure 9. TLB misses comparison on Opteron

4.2.4 TLB

Another important reason for decreasing in running time for recursive algorithm is explained by TLB misses. Figure 9 shows the L2 cache misses for three implementations on the Opteron. In the recursive implementation, TLB thrashing is avoided by blocking the sub-matrices, so the number of TLB misses is reduced greatly. Comparing above running time plots with cache and TLB performance plots, we note that trend in running time almost follows with the trend in TLB misses. Although we don't measure the TLB misses on the PowerPC, the larger 10 times reductions in running time can be explained by the larger page size 64KB. The DP formulation only needs computing a triangular matrices, the data is layed out by column/row index by an additional index array. So in the standard and recursive without blocking implementation cause irregular memory accesses. In this case, larger page size results in more TLB thrashing. The blocking recursive implementation rearrange the data accesses at the cost of sub-matrices copy operations, however, it improves the TLB performance greatly.

5 Conclusions

We have demonstrated decreased running times for nonserial polyadic dynamic programming algorithm by improving locality using combination of algorithmic ideas and architectural capabilities. We have related these performance gains to improved memory system behavior of the new programs. The DP table is triangular matrices and the data is accessed through an indirect index array. The recursive algorithm can not improve the number of memory references because of row-major or column-major data layout. A new data layout is necessary to improve the performance further.

References

[1] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. petitet, R. Vuduc, C. Whalely, and K. Yelick, Self adapting linear algebra algorithms and software, Proceedings of the IEEE, 2005, vol. 93, no. 2, pp. 293-312.

[2] M. Puschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, SPIRAL: Code generation for DSP transforms, Proceedings of the IEEE special issue on "Program Generation, Optimization, and Adaptation," Vol. 93, No. 2, 2005, pp. 232-275

[3] R. Whalely and J. Dongarra, Automatically Tuned Linear Algebra Software, High Performance Computing Applications, Nov. 1998

[4] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, Cache-Oblivious Algorithms, Pro. 40th Ann. Symp. Foundations of Computer Science, pp. 17-18, Oct. 1999.

[5] T. Chilimbi, M. Hill and J. Larus, Cache-Conscious Structure Layout, Pro. ACM SIGPLAN Conf. Programming Language Design and Implementation, 1999.

[6] G. Venkataraman, S. Sahni, and S. Mukhopadhyaya, A Blocked All-Pairs Shortest-Paths Algorithm, Pro. Scandinavia Workshop Algorithms and Theory, 2000.

[7] J. S. Park, M. Penner, V. K. Prasanna, Optimizing Graph Algorithms for Improved Cache Performance. IEEE Trans. on Parallel and Distributed Systems, Vol. 15, No. 9, Setp. 2004.

[8] T.F. Smith and M.S. Waterman, Identification of common molecular subsequences, Journal of Molecular Biology, 1981, Vol. 147(1), pp. 195-197.

[9] R. B.Lyngso, M. Zuker, Fast evaluation of internal loops in RNA secondary structure prediction, Bioinformatics, 1999, Vol 15, 6: 440-445

[10] A Grama, A Gupta, G Karypis, V Kumar, Introduction to Parallel Computing, Addison Wesley, 2003.