Data prefetching for linear algebra operations on high performance workstations

Elena Garcia, Jose R. Herrero and Juan J. Navarro

Abstract

In a previous work it was shown that the performance of linear algebra computations, which access large amounts of data, is dependent on the behavior of the memory hierarchy. This research is aimed to use the multilevel orthogonal blocking approach in conjuntion with other software techniques to further improve the performance of linear algebra computations.

The performance of the dense matrix by matrix multiplication executed on a superscalar high performance workstation is improved using binding and nonbinding prefetching to hide the memory latency together with the well known technique of blocking.

In this report our main goal is to improve the performance of the matrix by matrix multiplication on a high performance workstation. This widely applied operation has already been improved by means of techniques such as blocking and software pipelining. In this paper we show the improvements yielded by applying software prefetch in addition to those techniques aforementioned. Prefetching is a technique used to hide memory latency by avoiding the processor to stall because of data dependencies when a miss occurs.

We first present an algorithm where only blocking and software pipelining are used, followed by two new algorithms in which binding and nonbinding prefetching respectively have been added.

Performance results start with 15 MFlops for *jki* form. Blocking and software pipelining techniques get a performance of 128 MFlops for the best algorithm without prefetching, and finally, it is improved up to 166 MFlops when prefetching is applied in a computer with 200 MFlops of peak performance.

1 Introduction

In order to speed up the execution time of any algorithm its features must be considered and adapted to the underlying architecture where the problem is to be solved.

Matrix linear algebra operations in general and matrix multiplication in particular, have a potentially high degree of data reuse. The memory hierarchy present in current computers exploits data reuse, technically known as temporal locality, by keeping the data to be reused in faster levels of memory. On the other hand, superscalar computers offer the possibility of issuing n instructions in the same cycle, when they belong to certain instruction types, so that they can launched to different functional units. Accordingly, compilers can produce an instruction scheduling which minimizes execution time by arranging instructions in n-tuples that can be issued in one cycle, while respecting data dependencies. Matrix multiplication has a high level of floating-point computation so many efforts have been done to achieve an ideal code with one floating-point operation at every cycle. This is pursued by hiding other instructions, i.e. by issuing them together with floating-point instructions.

Using the memory hierarchy efficiently and taking advantage of superscalar operation can therefore be regarded as the most important directions when exploiting the corresponding features of matrix multiplication algorithms.

Many advances have been made in both directions, specially through the application of techniques such as blocking ([GaPS90] [LaRW91] [NaJL94] [Nava94]) and software pipelining ([Lam88] [RaST92] [AiNi88]). Blocking reduces data cache misses. However, this reduction is not enough to obtain optimal performance. Despite using blocking techniques, the processor is stalled during a considerable amount of time waiting for data to come from slower levels of memory. To avoid those stalled cycles, prefetching is required.

Since our machine does not provide any hardware prefetching in its cache, we are talking only about software prefetching. The most common software prefetching technique uses specific prefetching instructions to bring a line of data from a slower level of memory to a closer and faster one. The prefetch instruction is added to the code and intended to be issued several cycles before the load instruction. The prefetch instruction is expected to miss, and the code designed to keep executing while the service is done. This way the actual load usually hits and stalled cycles are avoided. Our technique, however, will be different because we suppose that our instruction set does not provide such instructions. We exploit prefetching in a similar notwithstanding different way.

Our prefetching strategy will take advantage of a third feature our computer offers, namely the fact that the processor does not stall when a miss at first-level cache occurs as long as the following instructions do not access memory and do not use the data fetched. Accordingly, the idea is to place the load instructions several cycles ahead of its use.

If instructions between fetch and use do not stall the processor, in case of first-level cache miss, part of all cycles of the memory latency will be hidden; depending on the distance between the fetch and the actual use of data. Current compilers do not consider misses for code scheduling. This is reasonable since not all loads should be prefetched, only those likely to produce a miss. It is at this point when knowledge of the algorithm's characteristics will help us to design a selective and successful prefetching of data.

When data prefetched is kept in registers until used and no other load is needed, we say we are making a binding prefetch. This way prefetching increases the total number of registers used, and the total number of registers of a processor limits the amount of latency hidden. We will exploit this possibility up to the limit. A way to overcome this limit consists of performing nonbinding prefetches, that is, by repeating the load instructions before the use (it will be a hit) while the prefetched data is disregarded. Both binding and nonbinding prefetching versions will be explained in detail, algorithms developed and results in a real case presented.

Techniques for implementing non-blocking loads to tolerate miss cache latency are given in [FaJo94]. A general view of software prefetching can be found in [CaKP91]. Studies of compiler algorithms to insert prefetch instructions into code are [MoLG92] and [Mowr94]. Another case of exploiting prefetching for linear algebra algorithms can be found in [AgGZ94].

We give now some technical details about the computer and performance measurements used. Afterwards, section 2 presents a brief description of techniques such blocking and software pipelining and provides insight of our analytical model, while in sections 3 and 4 binding and nonbinding prefetch are developed. Finally, comparisons are presented and conclusions drawn in section 5.

for a floating-point operation. Finally, this processor has 32 registers for floating-point data.

The memory hierarchy consist of two direct mapped caches. Their sizes are $Cache_1$=1K word and $Cache_2$=256K words, being the line size $L = 4$ words for both caches. One floating-point data occupies 1 word (8 bytes). The page size is $P = 1K$ word. It has a fully associative TLB consisting of 32 entries, with an LRU replacement policy. Latencies for load instructions are 3 cycles for a hit at the first-level cache; 11 cycles for the first word when a miss occurs (although the whole line fill-in requires 18 cycles); 36 cycles for a miss in the second-level cache and 33 cycles for a TLB miss.

Performance measurements

Performance is shown in Cycles Per Floating-point operation (CPF) and MFlops, and they are related by $MFlops = 200/CPF$ because the processor is clocked at 200 MHz.

The goal is to reduce CPF to the minimum attainable by the processor, that is CPF=1, one floating-point operation issued at every cycle. That would mean hiding all the integer, memory and branch instructions by issuing them together with floating-point operations.

CPF can be described as a combination of these components:

$CPF = CPF(inner) + CPF(overh) + CPF(mem)$

The number of cycles of the innermost loop body are counted to obtain $CPF(inner)$ while $CPF(overh)$ is approximated counting the cycles of the second innermost loop body. $CPF(mem)$ is calculated adding misses per flop (MPF) of each memory level weighted by their corresponding miss latencies ($CyclesperMiss$).

$$CPF(mem) = CPM(Cache_1) \times MPF(Cache_1) \; + \; CPM(Cache_2) \times MPF(Cache_2)$$

$$+ CPM(TLB) \times MPF(TLB)$$

Those MPF are obtained from a simulator of the memory hierarchy of our computer developed for this purpose. Besides, a mathematical model for the memory hierarchy has been developed for the algorithms in this report. This model, together with the estimation of $CPF(inner)$ and $CPF(overh)$ will be compared with the actual execution of the programs; it will help us to explain the reasons for certain performances.

The set of the matrices used to display the results are $N \times N$ squared matrices with N ranging from 10 to 1500. The leading dimension of all matrices is 2000. For the algorithms presented in this report where the inner loop has been fully unrolled we only show the code that is used for dimensions which are multiple of the number of iterations unrolled, i.e. the former innerloop dimension since it has been fully unrolled. The extra code that performs the few calculations of the remanding areas of matrices has been developed and used for the execution results but not included here to simplify the exposition.

2 Blocking and software pipelining

From the simple algorithm shown in the last section, we will develop an improved version using techniques such as blocking and software pipelining. That algorithm will be later used to compare its performance with the new algorithms introducing prefetching which are presented later in this report.

Since matrices in Fortran are stored columnwise, the best loop ordering in the simple algorithm in figure 1 will access data in continuous memory locations. On the other hand, the

```
    DO 10 j3=1, N, J2
      DO 10 i3=1, N, I2
        do j1=1,J2
         do i1=1,I2
          W2C(i1,j1)=C(i3+i1-1,j3+j1-1)
         enddo
        enddo
        DO 20 k2=1, N, K1
          do j1 = 1,J2
           do k1 = 1,K1
            W2B(k1,j1)=B(k2+k1-1,j3+j1-1)
           enddo
          enddo
          DO 20 i2=i3, i3+I2-1, I1
            do k1 = k2, k2+K1-1
             do i1 = i2, i2+I1-1
              W1A( i1-i2+1,k1-k2+1) = A(i1,k1)
             enddo
            enddo
30          DO 20 j1=j3, j3+J2-1
40            DO 20 k1=k2, k2+K1-1
50              DO 20 i0=i2, i2+I1-1
60                W2C(i0-i2+1,j1-j3+1) = W2C(i0-i2+1,j1-j3+1) +
                            W1A(i0-i2+1,k1-k2+1) * B(k1-k2+1,j1-j3+1)
20        CONTINUE
          do  j1=1,J2
           do  i1=1,I2
            C(i3+i1-1,j3+j1-1)=W2C(i1,j1)
           enddo
          enddo
10  CONTINUE
```

Figure 4: Code with blocking and software pipelining applied, $ji(ki)(jki)$ form

```
      DO 30 j1 = j3,j3+J2-1
        T1  = W2C(i2-i3+1   ,j1-j3+1)
        T2  = W2C(i2-i3+1+1 ,j1-j3+1)
        T3  = W2C(i2-i3+1+2 ,j1-j3+1)
        T4  = W2C(i2-i3+1+3 ,j1-j3+1)
        T5  = W2C(i2-i3+1+4 ,j1-j3+1)
        T6  = W2C(i2-i3+1+5 ,j1-j3+1)
        T7  = W2C(i2-i3+1+6 ,j1-j3+1)
        T8  = W2C(i2-i3+1+7 ,j1-j3+1)
        T9  = W2C(i2-i3+1+8 ,j1-j3+1)
        T10 = W2C(i2-i3+1+9 ,j1-j3+1)
        T11 = W2C(i2-i3+1+10,j1-j3+1)
        T12 = W2C(i2-i3+1+11,j1-j3+1)
        T13 = W2C(i2-i3+1+12,j1-j3+1)
        T14 = W2C(i2-i3+1+13,j1-j3+1)
        T15 = W2C(i2-i3+1+14,j1-j3+1)
        T16 = W2C(i2-i3+1+15,j1-j3+1)
        B0 = W2B(1,j1-j3+1)
        A1 = W1A(1,1)
        A2 = W1A(2,1)
        A3 = W1A(3,1)
        DO 40 k1 = k2,k2+K1-1
         T1  = T1  + A1 * B0
         T2  = T2  + A2 * B0
         T3  = T3  + A3 * B0
         T4  = T4  + W1A(4 ,k1-k2+1) * B0
         T5  = T5  + W1A(5 ,k1-k2+1) * B0
         T6  = T6  + W1A(6 ,k1-k2+1) * B0
         T7  = T7  + W1A(7 ,k1-k2+1) * B0
         T8  = T8  + W1A(8 ,k1-k2+1) * B0
         T9  = T9  + W1A(9 ,k1-k2+1) * B0
         T10 = T10 + W1A(10,k1-k2+1) * B0
         T11 = T11 + W1A(11,k1-k2+1) * B0
         T12 = T12 + W1A(12,k1-k2+1) * B0
         T13 = T13 + W1A(13,k1-k2+1) * B0
         T14 = T14 + W1A(14,k1-k2+1) * B0
         T15 = T15 + W1A(15,k1-k2+1) * B0
         T16 = T16 + W1A(16,k1-k2+1) * B0
         B0 = W2B(k1-k2+2,j1-j3+1)
         A1 = W1A(1,k1-k2+2)
         A2 = W1A(2,k1-k2+2)
         A3 = W1A(3,k1-k2+2)
40      CONTINUE
        W2C(i2-i3+1   ,j1-j3+1) = T1
        W2C(i2-i3+1+1 ,j1-j3+1) = T2
        W2C(i2-i3+1+2 ,j1-j3+1) = T3
        W2C(i2-i3+1+3 ,j1-j3+1) = T4
        W2C(i2-i3+1+4 ,j1-j3+1) = T5
        W2C(i2-i3+1+5 ,j1-j3+1) = T6
        W2C(i2-i3+1+6 ,j1-j3+1) = T7
        W2C(i2-i3+1+7 ,j1-j3+1) = T8
        W2C(i2-i3+1+8 ,j1-j3+1) = T9
        W2C(i2-i3+1+9 ,j1-j3+1) = T10
        W2C(i2-i3+1+10,j1-j3+1) = T11
        W2C(i2-i3+1+11,j1-j3+1) = T12
        W2C(i2-i3+1+12,j1-j3+1) = T13
        W2C(i2-i3+1+13,j1-j3+1) = T14
        W2C(i2-i3+1+14,j1-j3+1) = T15
        W2C(i2-i3+1+15,j1-j3+1) = T16
30    CONTINUE
```

Figure 5: Innermost unrolled code for $ji(ki)(jki)$ form

level cache, is around 128 MFlops (that is 1.56 CPF). The difference between this actual performance and the maximum attainable pointed by $CPF(inner)$ is basically due to memory misses. The model we have developed to measure cache and TLB misses helps us to show the reason why such difference arises.

Next we introduce a brief description of the pattern used to display all models in this report.

Model of algorithm with blocking and software pipelining techniques

The model of $CPF(mem)$ gives an equation for each memory level separately, ($Cache_1$, $Cache_2$ and TLB). They include a first term for the compulsory misses of accesses to Working Areas, a second term reflecting the interference misses, and a third one for the extra misses produced by making the precopies. The TLB is completely associative, so no interference misses have been considered. The global model includes the $CPF(inner)$ and the $CPF(overh)$ estimated from the code in assembly level. To develop the formula for misses in our models we have used some similar techniques to those introduced in [Lan] for only one level of blocking.

We give now three terms as examples of how the model below has been developed:

- *Compulsory misses* for first level working area A (term $\frac{1}{2LJ_2}$ of $MPF(Cache_1)$): All the cache lines must be accessed through inner loop body j1 of figure 5, so the number of misses are $\frac{K_1 I_1}{L}$. Compulsory misses suppose that these lines are kept in first-level cache until the last iteration of inner loop j1, so the number of floating-point operations executed are $2 \times I_1 \times J_2 \times K_1$. Dividing the total number of misses by the number of floating-point operations we obtain the component of $MPF(Cache_1)$ corresponding to compulsory misses originated by the working area of A. Acting in a similar way with B and C, adding the results and simplifying we get the first term of $MPF(Cache_1)$. The third term, corresponding to compulsory misses produced by the copies, is reckoned in a similar way.

- *Interference misses* of first level working area A due to B (term $\frac{K_1}{2LCache_1}$ of $MPF(Cache_2)$): The part of A kept in the working area in the first level can be flushed because of accesses to B and C. The misses of A produced by accesses to B are calculated for each iteration of loop j1. K_1 words of B are accessed and the probability of one of them colliding with a word of A is $\frac{K_1}{Cache_1}$. This probability of miss is multiplied by the number of lines accessed of A are divided by the number of operations executed in one iteration of loop j. That is:
$\frac{\frac{K_1}{Cache_1}\frac{K_1 I_1}{L}}{2K_1 I_1}$. Simplifying we get the first term of the second term of $MPF(Cache_1)$.

- $MPF(Cache_2)$ is calculated in a similar way to that of the first-level cache considering the proper spanning of the misses.

- TLB misses of first-level working area A are just the number of pages of the area divided by the number of operations that it covers. Misses of copies are different since the whole area to be copied is not continuous in memory, and each column of A,B,C belongs to at least one different page.

Next the complete model is displayed:

$$CPF(inner) = 1.03$$

$$CPF(overh) = \frac{25}{I_1 K_1}$$

$$MPF(Cache_1) = \frac{1}{2L}\left(\frac{1}{J_2}+\frac{1}{I_1}+\frac{1}{K_1}\right) + \frac{1}{2L}\left(\frac{K_1+I_1}{Cache_1}\right) + \frac{1}{2L}\left(\frac{1}{J_2}+\frac{1}{I_2}+\frac{1}{N}\right)$$

$$MPF(Cache_2) = \frac{1}{2L}\left(\frac{1}{J_2}+\frac{1}{I_2}+\frac{1}{N}\right) + \frac{1}{2L}\left(\frac{K_1+I_2+J_2}{Cache_2}\right) + \frac{1}{2L}\left(\frac{1}{J_2}+\frac{1}{I_2}+\frac{1}{N}\right)$$

$$MPF(TLB) = \frac{1}{2P}\left(\frac{1}{J_2}+\frac{1}{I_1}+\frac{I_2}{I_1K_1}\right) + \frac{1}{2}\left(\frac{1}{I_1J_2}+\frac{1}{I_2K_1}+\frac{1}{NI_2}\right)$$

The weightiest terms appear in the $Cache_1$ equation. This model points out that block size should be as large as possible, limited by the growing interferences, specially in the first level of cache. The block size chosen is $I_2 = 320, J_2 = 320, I_1 = 16, K_1 = 48$. With these parameters the working area of A is $I_1 \times K_1 = 768$ words, occupying 75% of $Cache_1$; working areas of B and C are $K_1 \times J_2 = 15360$ words and $I_2 \times J_2 = 102400$ words which together extend throughout 45% of $Cache_2$.

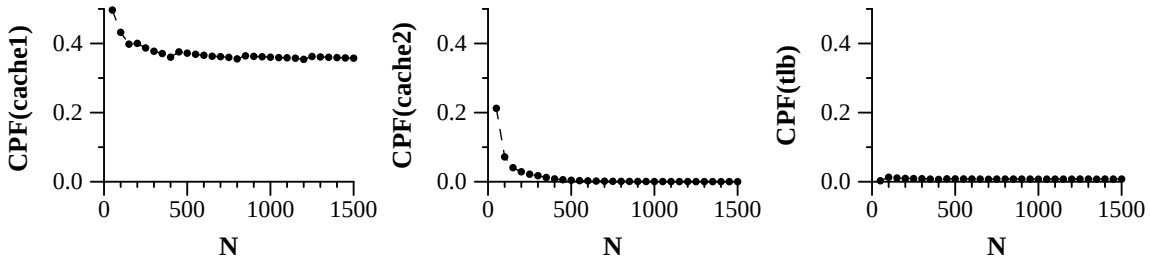

Figure 6: Previous techniques: Memory performance simulated by levels

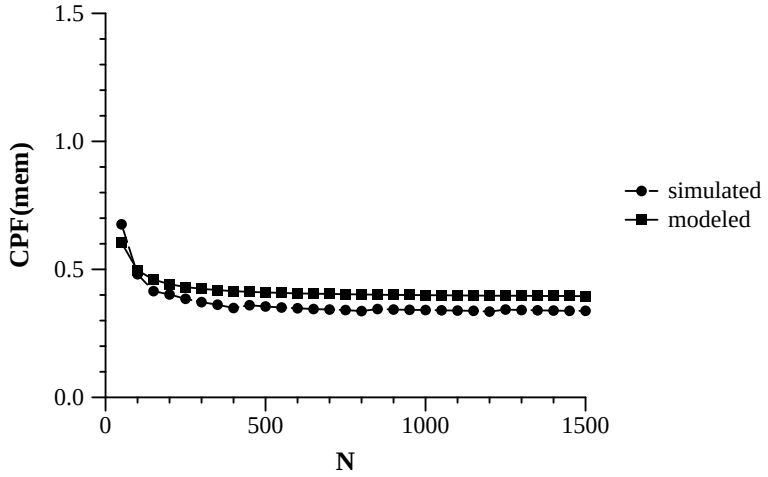

Figure 7: Previous techniques: Memory performance modeled and simulated

Multilevel blocking provides a way to lower the number of misses on the two levels which provides a reduction in $CPF(mem)$. This $CPF(mem)$ has been simulated and the results for each level in the memory hierarchy are shown in figure 6. Their combined effect together with the model estimations are given in figure 7. Note that our model adapts tightly to the simulations.

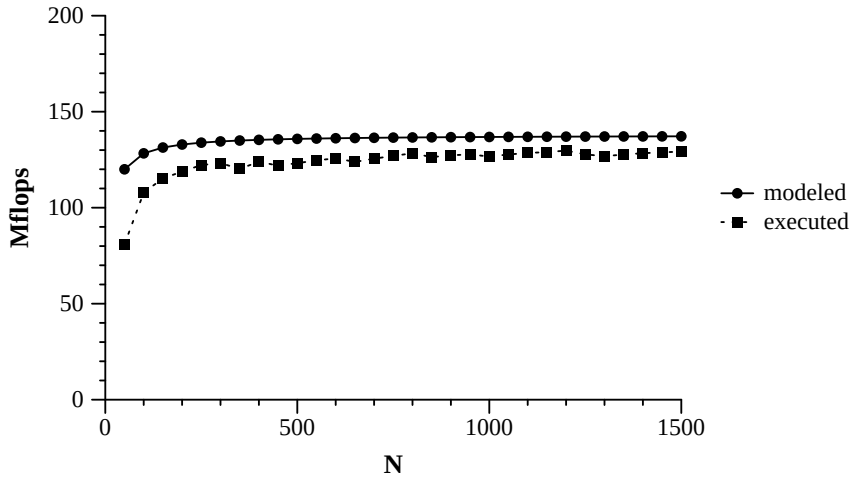

Figure 8: Previous techniques: comparing executions with model results

Finally, figure 8 shows the execution results compared with the theoretical modeling we have developed. The small difference if the latter is most likely due to the estimation of $CPF(overh)$.

The difference between the performance results obtained for the actual execution (1.56 CPF) and the theoretical optimum given by the $CPF(inner)$ (1.03 CPF) is 0.53 CPF for large matrices. Both model and simulations indicate that misses at first-level cache in the inner loop produce the 49 % of that difference. In second position of importance, for the whole algorithm, we have misses at second-level cache with 9.2%, $CPF(overh)$ with 6.5 % and misses at first-level cache in loop j1 (second inner loop) with 5.5 %.

From these results we infer we should be mainly concerned about misses in the first level of cache, and particularly the misses produced in the innermost loop body. The model shows that misses at first-level cache in the inner loop body are due to accesses to the working area B at second level of cache (0.125 CPF) and the interferences they cause to the copy of A at first level of cache (0.124 CPF).

3 Binding prefetch

As it has been noted at the end of the previous section, misses at the first level of cache produce half of the stalled cycles which slow down the whole process. The processor we use, as most high performance processors, avoids stalling whenever possible in order to reduce the impact of first-level cache misses on performance. It can manage a pending memory access and, at the same time, continue executing other instructions which do not access memory nor use the data whose fetch is still in progress.

Suppose that a load instruction I_l is issued at cycle 0. The most relevant features of our processor are the following:

- 1. In case that I_l hits in the first-level cache, the first instruction I_u that uses the data prefetched will not be issued before cycle 3. If the code has other instructions between I_l ana I_u the processor will not stall during cycles 1 and 2.

- 2. In case that I_l produces a miss in first level and a hit in second level, I_u will not be issued until cycle 11. Again if the code provides other instructions between I_l and I_u the processor will not stall the corresponding cycles.

- 3. However, no memory accesses can be done while data is coming from second-level cache to the processor, that is from cycle 3 to 10, because they would stall the processor and not resume until cycle 11. A remarkable characteristic of this processor is that of allowing loads to follow a previous miss when issued within the immediately following two cycles. This at most two load instructions are not required to cause a hit since a *miss under miss* policy is implemented. In case they produce a hit, they proceed normally. When a miss arises, it is queued to be processed after the actual miss finishes.

- 4. Data which originally missed in the first-level cache but hits in the second level cache is available from cycle 11 thanks to a bypassing mechanism. However, the line is not yet in the cache and a load of data from that line will not be a hit if it is done shortly afterwards. If a second load instruction to the *same* line is done, whenever it is issued (cycles 1,3,11,12,...), no instruction using that data will be issued before cycle 18.

The idea used for improving performance is, on one hand, to keep some floating-point operations to issue with load instructions to hide them (as it has been done in the last section); and on the second one, to issue a set of floating-point operations immediately after an expected miss in order to hide its latency. The latter group will not be issued with load instructions because of feature 3.

From these features we conclude that at least 8 extra floating-point operations are needed for each miss in order to execute the code without stalling the processor. We call these 8 cycles the hidden-latency.

In general the number of floating-point operations must exceed the number of loads plus the number of misses times the latency needed to be hidden. This relation between flops (floating-point operations) and accesses is given in next equation:

$$flops \geq loads + misses \times hiddenlatency \tag{1}$$

Once the innermost loop body features are fixed, this equation gives a maximum bound on the memory latency that can be hidden by any scheduling of that innermost loop.

The possibility of considering all the loads as potential misses and try to hide them all must be considered. However, all the rules aforementioned must be met, which prevents us from doing so. Furthermore, the potential spacial locality would not be exploited, since we only expect one miss per line.

The conclusion is that prefetching must be selective to be successful. The code should be designed to estimate which load instructions are likely to miss. In our particular algorithm we can *predict* exactly when misses will appear in the code. To do so, we take into consideration that loads in the innermost loop body of the algorithm developed in section 2 run through continuous memory locations. Then, if the dimension of the fully unrolled loop I_1 is chosen to be multiple of the line size, the missing accesses will arise for the same load instructions through the iterations of the innermost loop. A way to predict which load instructions will miss, is to align the memory area accessed to a line boundary. This is possible since all accesses in the inner loop body are loads to working areas. They can be easily aligned in

```
30          DO 20 j1=j3, j3+J2-1, J0
40            DO 20 k1=k2, k2+K1-1, K0
60               W2C[i2:i2+I1,j1:j1+J0] = W2C[i2:i2+I1,j1:j1+J0] *
                   W1A[i2:i2+I1,k1:k1+K0] * W2B[k1:k1+K0,j1:j1+J0]
20       CONTINUE
```

Figure 9: Code for binding prefetch

execution time just leaving some extra words left at the back of each area and shifting the base pointer to the right place before the precopy is done. With this method the prediction is very accurate and we can assure which loads will not produce a miss and prefetch the reminding instructions that can possibly miss.

So far we have studied how the processor behaves in case of miss and the way to reduce and fix the number of misses in one iteration of the innermost loop to $1+\frac{I_1}{L}$ (where L is the line size). Next, we study the relation between memory accesses and floating-point operations and the way to find the best dimension I_1 to hide all the latency cycles of first-level cache.

We find now the latency-hidden attainable by the algorithm in last section. Our previous inner loop body in figure 5 had $2 \times I_1$ flops, $1+I_1$ loads and $1+ \frac{I_1}{L}$ misses. Applying these parameters to equation 1 for a line size L=4, it turns out that there is no value for I_1 which allows hiding more then 3 cycles. Moreover, I_1 must be multiple of L and we get that I_1=4 get a latency-hidden of 2 cycles, I_1=8 of 3 cycles, I_1=12 of 3.33 cycles and I_1=16 of 3.5 cycles. The consequence is that this inner loop is not suitable for our purpose since more flops per load are needed.

Unrolling in only one direction does not satisfy equation 1 for a latency-hidden of 8. We consider now unrolling in all three directions. Inner loops j1 and k1 are strip-mined in streams of J_0 and K_0 respectively. Loop interchange is done and the new two innermost loop are fully unrolled. The changes done over the code in figure 4 are shown in figure 9. The unrolling of the three inner loops is summarized in line 60 using the colon notation of matrices.

The final inner loop will have $2 \times I_1 \times J_0 \times K_0$ flops. Once again loads and stores of C can be taken out of the final inner loop, which therefore has $I_1 \times K_0 + K_0 \times J_0$ loads left to working areas A and B.

We need to count the number of misses of the new loop body and then use equation 1. The number of misses depends on the way the precopies of blocks of A and B have been done, specifically whether they have been stored by rows or by columns. We have decided to precopy matrix A by rows and B by columns, so continuous memory locations run in both cases following direction k. There are two reasons for this. First, in this way only parameter K_0 should be multiple of L. Second, the number of registers needed in the innermost loop for accumulating values of C is $I_1 \times J_0$, so both dimensions I_1 and J_0 should be small and not compelled to be multiples of L. Once these decisions are taken, the number of misses in the inner loop can be estimated again, resulting in $\frac{K_0}{L} \times (I_1 + J_0)$.

Applying equation 1 we get that hidden-latency $= L \times \left(\frac{2I_1J_0}{I_1+J_0} - 1\right)$. From this equation we infer that I_1 and J_0 will give a hidden-latency greater than 8 only when $2 \times I_1 \times J_0 > 3 \times (I_1 + J_0)$. This result implies that unrolling with larger values of I_1 and J_0 is preferable. Unfortunately, there is a limit in the number of registers available which prevents us from

to 240 to fit in the second-level cache. Using these parameters executions are around 138 MFlops (that is 1.44 CPF).

Model for binding prefetch

Here is the model developed for our algorithm with binding prefetch. It is very similar to the model given in section 2 since outer loop have not change. The most important difference is one term missed in $MPF(Cache_1)$ equation. This evaluates the compulsory misses of Working Area B hidden by binding prefetching. This term is given at the bottom in $MPF(Cache_1 hidden)$ equation.

$$CPF(inner) = 1.01$$

$$CPF(overh) = \frac{84}{I_1 K_1 J_1}$$

$$MPF(Cache_1) = \frac{1}{2L}\left(\frac{1}{J_2} + \frac{1}{K_1}\right) + \frac{1}{2L}\left(\frac{K_1 + I_1}{Cache_1}\right) + \frac{1}{2L}\left(\frac{1}{J_2} + \frac{1}{I_2} + \frac{1}{N}\right)$$

$$MPF(Cache_2) = \frac{1}{2L}\left(\frac{1}{J_2} + \frac{1}{I_2} + \frac{1}{N}\right) + \frac{1}{2L}\left(\frac{K_1 + I_2 + J_2}{Cache_2}\right) + \frac{1}{2L}\left(\frac{1}{J_2} + \frac{1}{I_2} + \frac{1}{N}\right)$$

$$MPF(TLB) = \frac{1}{2P}\left(\frac{1}{J_2} + \frac{1}{I_1} + \frac{I_2}{I_1 K_1}\right) + \frac{1}{2}\left(\frac{1}{I_1 J_2} + \frac{1}{I_2 K_1} + \frac{1}{N I_2}\right)$$

$$MPF(Cache_1 hidden) = \frac{1}{2L I_1}$$

Figure 11 shows the results of the simulations, the model and the model plus the hidden term. Considering that simulations only count misses and there is no cycle counter and no hiding effect, the model plus term hidden ($MPF(Cache_1 hidden)$) should be equal to the simulations.

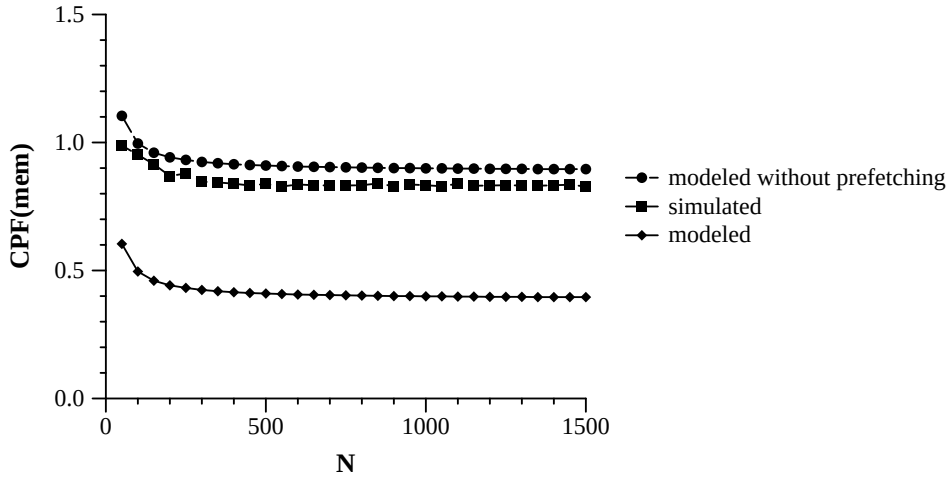

Figure 11: Binding prefetch: Memory performance modeled and simulated

Comparing those results with the model in figure 11, the new algorithm seems to improve a lot. Compared to the previous model in section 2 both $CPF(inner)$ and $CPF(overh)$ have

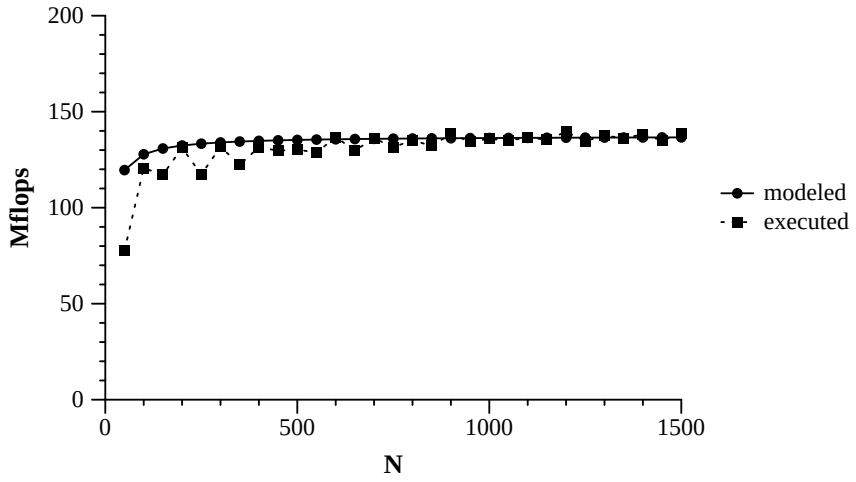

Figure 12: Binding prefetch: comparing executions with model results

improved and there is one term of $CPF(Cache_1)$ hidden. However, from the model we get that the number of interference misses of A at first-level cache grows from 0.0078 MFP to 0.015 and although between 70% to 80% of the misses of A are hidden, the global improvement is not so good. Compared to figure 7, the total number of stalled cycles is almost the same because of traffic in memory.

Finally, figure 12 shows the execution results and compares them with the model. The conclusion of this section is that applying binding prefetching improves performance. However, since not all misses in the first level of cache have been hidden, we consider that in our case binding prefetching has mot been completely successful.

4 Nonbinding prefetching

Despite using the blocking approach and binding prefetch developed in previous sections, memory latency for first level cache misses could not be completely hidden. A prefetch could not be initiated enough cycles prior to using data due to the limited number of registers available. A way to overcome this drawback is by means of Nonbinding Prefetching. Both binding and nonbinding prefetch are techniques which aim to move data from slower levels of memory to faster levels sufficiently in advance of the actual use of data. However, while the former brings a word to a register for immediate use, the latter aims to bring it to the cache memory for quick access in a near future. When Nonbinding Prefetching is used, extra load instructions are placed sufficiently in advance to ensure that a later access to the same cache line is a hit in the first-level cache. Ordinary load instructions are used for issuing the additional loads since our processor's instruction set does not include any specific prefetch instructions. Then, since normal load instructions bring data into registers, data prefetched into a register is disregarded and the register is immediately available for other operations. As a result, the prefetch instruction does not need to remain close to the data use because of the limited number of registers. Now, prefetching does not add data dependencies of 10 cycles for every miss. This way, the code is flexible enough to be ordered optimally.

We will keep some features of the algorithm developed so far, such as blocking at register level and aligning working areas to control misses. Once again, precopies of A and B are done

by rows and columns respectively, and K_0 is given the value of the line size (L=4), so I_1+J_0 lines are accessed in the loop body.

Our algorithm in previous section had I_1+J_0 misses, so this is the number of memory instructions added. Despite these instructions will increase the code, they should not add execution time since the inner loop schedule should issue them with other I_1+J_0 floating-point operations. This way the minimum CPF attainable is still 1 floating-point instruction per cycle. Equation 1 must include this new term:

$$flops \geq loads + prefetches + misses \times latencyhidden \tag{2}$$

This equation gives the maximum number of latency cycles per miss that any schedule could hide with those parameters. The minimum number of registers needed does not change from previous section; although the more flexible code now will not need many more. Trying several schedules we found one with I_1=3 and J_0=4 which covers all the misses of first-level cache (latency-hidden = 8.7, using 25 registers).

One design decision to be taken is where to place prefetching loads. In general, if a prefetch load is issued too early, two risks are introduced: the line prefetched can be flushed out from the cache before its use, and second, the prefetched line can flush out a line which would otherwise cause a hit. In our particular problem, data is accessed in continuous locations to working areas reducing the interferences. We decided to place the prefetch instructions one iteration before the use. This way, prefetch instructions are issued early enough and the code is simpler. On the other hand, simulations show that it is not too early to make it inefficient. Prefetching instructions for data used in the first iteration are done outside the inner loop, before it is started. The last iteration does not need to be treated apart as those useless prefetches are done to some extra space reserved for this purpose in the working areas.

The scheduling is so flexible now that it is possible to follow a pattern that satisfies all restrictions. This way, our inner loop body executes the K_0 unrolled iterations in order and loads and floating-point operations are placed in a natural way.

Before we introduce our instruction schedule we show that since misses in the first-level cache will not stall the processor, outer loops and working areas should be modified. We can guarantee no misses in first level of cache. Therefore, a working area in that level becomes useless. Blocking and working areas are aimed at reducing misses at *second*-level cache. The final DCD is given in figure 13 and the corresponding Fortran code in figure 14. A bigger working area of A is reused through loop j, while the area of B required to proceed with the computations is kept as small as possible. Loads and stores of C can, once more, be taken out of loop k1. C is not accessed in the loop body and reuse in second-level cache does not make it worth copying.

The inner loop schedule is given in appendix 2.

As in sections 2 and 3 we present the developed analytical model for this algorithm.

Model of nonbinding prefetch

Terms in this model have been developed as in previous sections, but the resulting equations are not similar because outer loops are different. At the bottom we include the $MPF(Cache_1)$ equation without prefetching to compare the improvement of the non-binding prefetching with the same algorithm without it.

This model is different from the others. It has only two working areas at second cache level. In $MPF(Cache_1)$ no interference misses term appear since they have been hidden as well as part of the compulsory misses. First term displays compulsory misses of A and B made out of the innermost loop body because of software pipelining and misses of C. Second term displays misses done because of the precopies.

$$CPF(inner) = 1.01$$

$$CPF(overh) = \frac{84}{I_1 K_2 j_1}$$

$$MPF(Cache_1) = \frac{1}{2}\left(\frac{1}{j_1 K_2} + \frac{1}{I_1 K_2} + \frac{6}{I_1 j_1 K_2}\right) + \frac{1}{2L}\left(\frac{1}{N} + \frac{1}{I_2}\right)$$

$$MPF(Cache_2) = \frac{1}{2L}\left(\frac{1}{N} + \frac{1}{I_2} + \frac{1}{K_2}\right) + \frac{1}{2L}\left(\frac{K_2 + I_2 + j_1}{Cache_2}\right) + \frac{1}{2L}\left(\frac{1}{N} + \frac{1}{I_2}\right)$$

$$MPF(TLB) = \frac{1}{2P}\left(\frac{1}{I_2} + \frac{1}{j_1}\right) + \frac{1}{2}\left(\frac{1}{I_2 N} + \frac{1}{K_2 I_2}\right)$$

$$MPF(Cache_1 nonpref) = \frac{1}{2}\left(\frac{1}{L j_1} + \frac{1}{L I_1} + \frac{6}{I_1 j_1 K_2}\right) + \frac{1}{2L}\left(\frac{1}{N} + \frac{1}{I_2}\right)$$

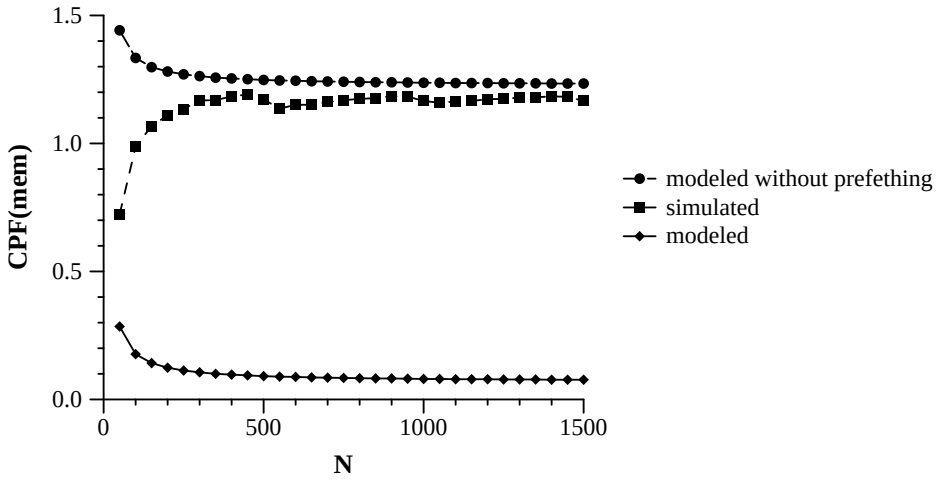

Figure 15: Nonbinding prefetch: Memory performance modeled and simulated

Figure 15, as in figure 11, compares misses simulated and the model with and without cycles hidden (with the two different equations for first-level cache). As in the binding prefetching section, the global number of misses has actually increased. This time, however, we succeed in hiding all misses and, consequently, performance is remarkably better.

This algorithm achieves 166 MFlops which is the 83 % of peak performance. This means an important improvement in the performance obtained in this widely used linear algebra operation.

Figure 16 compares the execution with our model. The block size used results from setting I2=K2=480. Working areas cover the 88% of the second level cache.

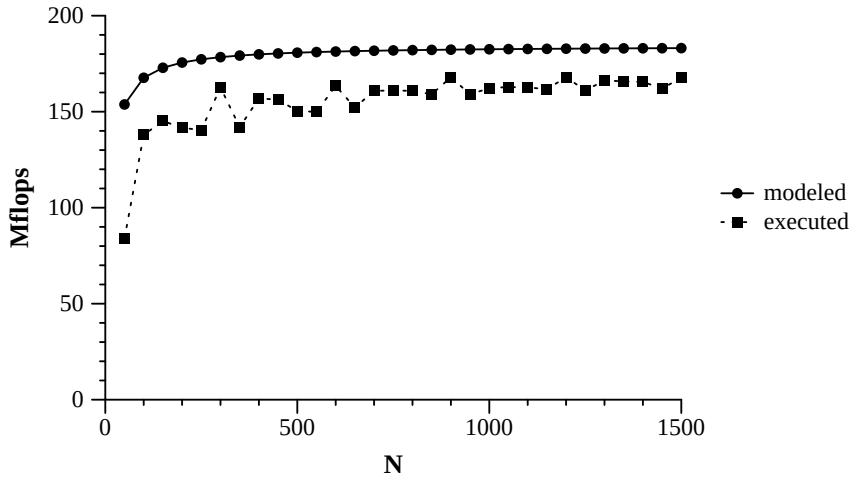

Figure 16: Nonbinding prefetch: comparing executions with model results

Simulated results of $CPF(inner)$, $CPF(overh)$ and $CPF(mem)$ add up $1.14\,CPF$. That would mean around 173 MFlops while our algorithm runs at $1.22\,CPF$ and 166 MFlops. The reason why those 173 MFlops are not achieved is that the unrolling is still too tight to the number of misses in each iteration. The value of first cache miss latency of 8 has been found experimentally in simpler algorithms. It is likely that, when first-level cache is so continuously accessed, some more cycles are needed to bring the lines from second-level cache. Larger unrolling would be the solution, but our number of registers does not allow us to do so.

The behavior of the computer in such circumstances should be better studied, considering the number of registers available and other resources. A better application of the use of registers to the way first-level cache works would completely solve the problem, and therefore, produce even better results.

5 Conclusions

We started from an initial matrix multiplication algorithm in which the important techniques of blocking and software pipelining were applied. Taking advantage of the fact that the processor does not stall due to a missing access, we applied software prefetch to improve performance. Unrolling the innermost loops as much as possible and aligning the working areas accessed in the new innermost loop we could design a schedule to hide latencies of misses at first-level cache. Because of limits of the target architecture, binding prefetch, where data prefetched is kept in registers until it is used, could not hide all misses achieving only a partial improvement. Finally, using nonbinding prefetch and changing outer loops we could hide all stalled cycles of first-level cache misses.

Figure 17 compares the three matrix by matrix multiplication algorithms presented in this report. It also includes the performance of a simple jki algorithm.

Performance has been enhanced with blocking at the register level ($CPF(inner)$ improvement) and the number of stalled cycles due to misses in the first level cache reduced ($CPF(mem)$ improvement). Both techniques have been applied to the inner loop, so when the matrix dimension is not a multiple of the unrolling factor, the remaining part of the

matrices are executed with a less efficient algorithm. This can be noticed in the curve for binding prefetch and specially for nonbinding prefetch.

Results for dimension 1000 are around 5 MFlops for the simple algorithm. Blocking and software pipelining enhances performance up to 128 MFlops. Binding prefetching tries to hide cycles stalled because of misses at first-level cache but it does not succeed completely because of the limited number of registers, being only able to improve the result up to 138 MFlops. Finally nonbinding prefetching obtains 166 MFlops of performance. Consider that this result is better than the very routine given by the manufacturer, who owns the complete information about the computer behavior.

Taking account that the inner loop bodies given in this report follow a regular pattern, it is proposed as a future work the possibility of adapt this method to be executed by a compiler when continuous memory access could be guaranteed.

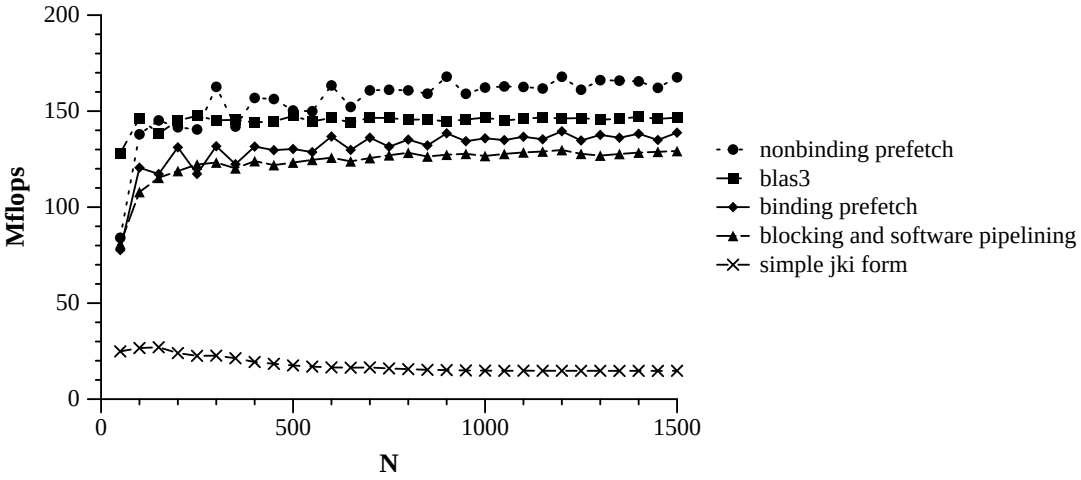

Figure 17: Comparing execution results

6 APPENDIX 1: Inner loop schedule for binding prefetch

We include here the schedule for the inner loop body in our algorithm with binding prefetch. Instead of the assembly code, we show an explanation diagram where the real assignments to registers are hidden to avoid confusion. The indexes are related to the register block, and the matrix M represents the temporal registers that keep the product results.

Each line corresponds to an execution cycle and when a miss load appears, it is pointed out in the right side of the line. Computation evolves in direction k, that is, execution proceeds computing one column of A with the corresponding row of B. We have tried to keep this ordering in k direction in both load instructions (left column) and floating-point operations (right column), although floating-point instruction have been mixed not to much to avoid the data dependencies.

The alignment of working areas has been done to produce misses of B in row 2 and misses of A in row 4.

```
33:
        ldt C(1,1)
        ldt C(2,1)
        ldt C(3,1)
        ldt C(4,1)
        ldt C(1,2)
        ldt C(2,2)
        ldt C(3,2)
        ldt C(4,2)
        ldt C(1,3)
        ldt C(2,3)
        ldt C(3,3)
        ldt C(4,3)
        ldt B(1,1)
        ldt B(1,2)
        ldt B(1,3)
        ldt A(1,1)
        ldt A(2,1)
        ldt A(3,1)
        ldt A(4,1)
34:
                              mult A(1,1) B(1,1) M(1,1,1,1)
                              mult A(2,1) B(1,1) M(2,1,1,1)
                              mult A(1,1) B(1,2) M(1,1,1,2)
                              mult A(2,1) B(1,2) M(2,1,1,2)
        ldt B(2,1)            mult A(3,1) B(1,1) M(3,1,1,1)
        ldt B(2,2)            mult A(4,1) B(1,1) M(4,1,1,1)
        ldt A(1,2)            mult A(3,1) B(1,2) M(3,1,1,2)
        ldt A(2,2)            mult A(4,1) B(1,2) M(4,1,1,2)
        ldt A(3,2)            addt M(1,1,1,1) C(1,1)
        ldt A(4,2)            addt M(2,1,1,1) C(2,1)          miss
                              addt M(3,1,1,1) C(3,1)
                              addt M(4,1,1,1) C(4,1)
                              addt M(1,1,1,2) C(1,2)
                              addt M(2,1,1,2) C(2,2)
                              mult A(1,1) B(1,3) M(1,1,1,3)
                              mult A(2,1) B(1,3) M(2,1,1,3)
```

	mult A(3,1) B(1,3) M(3,1,1,3)		
	mult A(4,1) B(1,3) M(4,1,1,3)		
ldt B(2,3)	addt M(3,1,1,2) C(3,2)		
ldt B(3,1)	addt M(4,1,1,2) C(4,2)	miss	
	mult A(1,2) B(2,1) M(2,1,2,1)		
	mult A(2,2) B(2,1) M(2,2,2,1)		
	mult A(3,2) B(2,1) M(2,3,2,1)		
	mult A(4,2) B(2,1) M(2,4,2,1)		
	addt M(1,1,1,3) C(1,3)		
	addt M(2,1,1,3) C(2,3)		
	addt M(3,1,1,3) C(3,3)		
	addt M(4,1,1,3) C(4,3)		
	addt M(1,2,2,1) C(1,1)		
	addt M(2,2,2,1) C(2,1)		
ldt A(1,3)	addt M(3,2,2,1) C(3,1)		
ldt A(2,3)	addt M(4,2,2,1) C(4,1)		
ldt A(3,3)	mult A(1,2) B(2,2) M(1,2,2,2)		
ldt A(4,3)	mult A(2,2) B(2,2) M(2,2,2,2)		miss
	mult A(3,2) B(2,2) M(3,2,2,2)		
	mult A(4,2) B(2,2) M(4,2,2,2)		
	mult A(1,2) B(2,3)		
	mult A(2,2) B(2,3)		
	mult A(3,2) B(2,3)		
	mult A(4,2) B(2,3)		
ldt B(4,1)	addt M(1,2,2,2) C(1,2)		
ldt B(3,2)	addt M(2,2,2,2) C(2,2)	miss	
	addt M(3,2,2,2) C(3,2)		
	addt M(4,2,2,2) C(4,2)		
	mult A(1,3) B(3,1) M(1,3,3,1)		
	mult A(2,3) B(3,1) M(2,3,3,1)		
	mult A(3,3) B(3,1) M(3,3,3,1)		
	mult A(4,3) B(3,1) M(4,3,3,1)		
	addt M(1,2,2,3) C(1,3)		
	addt M(2,2,2,3) C(2,3)		
	addt M(3,2,2,3) C(3,3)		
	addt M(4,2,2,3) C(3,4)		
ldt A(1,4)	addt M(1,3,3,1) C(1,1)		
ldt A(2,4)	addt M(2,3,3,1) C(2,1)		
ldt A(3,4)	addt M(3,3,3,1) C(3,1)		
ldt A(4,4)	addt M(4,3,3,1) C(4,1)	miss	
ldt B(4,2)	mult A(1,3) B(3,2) M(1,3,3,2)		
	mult A(2,3) B(3,2) M(2,3,3,2)		
	mult A(3,3) B(3,2) M(3,3,3,2)		
	mult A(4,3) B(3,2) M(4,3,3,2)		
	mult A(1,4) B(4,1) M(1,4,4,1)		
	mult A(2,4) B(4,1) M(2,4,4,1)		
	addt M(1,3,3,2) C(1,2)		
	addt M(2,3,3,2) C(2,2)		
	addt M(3,3,3,2) C(3,2)		
ldt B(3,3)	addt M(4,3,3,2) C(4,2)	miss	
	mult A(3,4) B(4,1) M(3,4,4,1)		
	mult A(4,4) B(4,1) M(4,4,4,1)		
	mult A(1,4) B(4,2) M(1,4,4,2)		

```
                        mult A(2,4) B(4,2) M(2,4,4,2)
                        mult A(3,4) B(4,2) M(3,4,4,2)
                        mult A(4,4) B(4,2) M(4,4,4,2)
                        addt M(1,4,4,1) C(1,1)
                        addt M(2,4,4,1) C(2,1)
                        addt M(3,4,4,1) C(3,1)
                        addt M(4,4,4,1) C(4,1)
        ldt B(4,4)      addt M(1,4,4,2) C(1,2)
                        addt M(2,4,4,2) C(1,2)
                        mult A(1,3) B(3,3) M(1,3,3,3)
                        mult A(2,3) B(3,3) M(2,3,3,3)
                        addt M(3,4,4,2) C(3,2)
                        addt M(4,4,4,2) C(4,2)
                        mult A(3,3) B(3,3) M(3,3,3,3)
                        mult A(4,3) B(3,3) M(4,3,3,3)
                        addt M(1,3,3,3) C(1,3)
                        addt M(2,3,3,3) C(2,3)
        ldt B(1,1)      mult A(1,4) B(4,3) M(1,4,4,3)
        ldt B(1,2)      mult A(2,4) B(4,3) M(2,4,4,3)
        ldt B(1,3)      mult A(3,4) B(4,3) M(3,4,4,3)
        ldt A(1,1)      mult A(4,4) B(4,3) M(4,4,4,3)
        ldt A(2,1)      addt M(3,3,3,3) C(3,3)
        ldt A(3,1)      addt M(4,3,3,3) C(4,3)
        ldt A(4,1)      addt M(1,4,4,3) C(1,3)          miss
                        addt M(2,4,4,3) C(2,3)
                        addt M(3,4,4,3) C(3,3)
                        addt M(4,4,4,3) C(4,3)
        bne 34
35:
        stt C(1,1)
        stt C(2,1)
        stt C(3,1)
        stt C(4,1)
        stt C(1,2)
        stt C(2,2)
        stt C(3,2)
        stt C(4,2)
        stt C(1,3)
        stt C(2,3)
        stt C(3,3)
        stt C(4,3)
        bne 33
```

7 APPENDIX 2: Inner loop schedule for nonbinding prefetch

This section presents the inner loop schedule for nonbinding prefetch.

The schedule is organized as in the previous section. Prefetching instructions are pointed out with a mark on the right side. Computation evolves again in direction k but, this time the pattern is even more obvious than the binding prefetching schedule.

```
33:
        ldt C(1,1)
        ldt C(2,1)
        ldt C(3,1)
        ldt C(1,2)
        ldt C(2,2)
        ldt C(3,2)
        ldt C(1,3)
        ldt C(2,3)
        ldt C(3,3)
        ldt C(1,4)
        ldt C(2,4)
        ldt C(3,4)
        ldt A(1,1)          prefetch
        ldt A(2,1)          prefetch
        ldt A(3,1)          prefetch
        ldt B(1,1)          prefetch
        ldt B(1,2)          prefetch
        ldt B(1,3)          prefetch
        ldt B(1,4)          prefetch
34:
        ldt A(1,1)              mult A(1,1) B(1,1) M(1,1,1,1)       prefetch
        ldt A(1,2)              mult A(2,1) B(1,1) M(2,1,1,1)
        ldt A(2,2)              mult A(3,1) B(1,1) M(3,1,1,1)
                                mult A(1,1) B(1,2) M(1,1,1,2)
                                mult A(2,1) B(1,2) M(2,1,1,2)
                                mult A(3,1) B(1,2) M(3,1,1,2)
                                addt M(1,1,1,1) C(1,1)
                                addt M(2,1,1,1) C(2,1)
                                addt M(3,1,1,1) C(3,1)
                                addt M(1,1,1,2) C(1,2)
                                addt M(2,1,1,2) C(2,2)
        ldt A(3,2)              addt M(3,1,1,2) C(3,2)
        ldt A(2,1)              mult A(1,1) B(1,3) M(1,1,1,3)       prefetch
        ldt B(2,1)              mult A(2,1) B(1,3) M(2,1,1,3)
        ldt B(2,2)              mult A(3,1) B(1,3) M(3,1,1,3)
                                mult A(1,1) B(1,4) M(1,1,1,4)
                                mult A(2,1) B(1,4) M(2,1,1,4)
                                mult A(3,1) B(1,4) M(3,1,1,4)
                                addt M(1,1,1,3) C(1,3)
                                addt M(2,1,1,3) C(2,3)
                                addt M(3,1,1,3) C(3,3)
                                addt M(1,1,1,4) C(1,4)
                                addt M(2,1,1,4) C(2,4)
        ldt B(2,3)              addt M(3,1,1,4) C(3,4)
        ldt B(2,4)              mult A(1,2) B(2,1) M(1,2,2,1)
```

ldt A(1,3)	mult A(2,2) B(2,1) M(2,2,2,1)	
ldt A(3,1)	mult A(3,2) B(2,1) M(3,2,2,1)	prefetch
ldt A(2,3)	mult A(1,2) B(2,2) M(1,2,2,2)	
ldt A(3,3)	mult A(2,2) B(2,2) M(2,2,2,2)	
	mult A(3,2) B(2,2) M(3,2,2,2)	
	addt M(1,2,2,1) C(1,1)	
	addt M(2,2,2,1) C(2,1)	
	addt M(3,2,2,1) C(3,1)	
	addt M(1,2,2,2) C(1,2)	
	addt M(2,2,2,2) C(2,2)	
	addt M(3,2,2,2) C(3,2)	
	mult A(1,2) B(2,3) M(1,2,2,3)	
ldt B(3,1)	mult A(2,2) B(2,3) M(2,2,2,3)	
ldt B(3,2)	mult A(3,2) B(2,3) M(3,2,2,3)	
ldt B(1,1)	mult A(1,2) B(2,4) M(1,2,2,4)	prefetch
ldt B(3,3)	mult A(2,2) B(2,4) M(2,2,2,4)	
ldt B(3,4)	mult A(3,2) B(2,4) M(3,2,2,4)	
	addt M(1,2,2,3) C(1,3)	
	addt M(2,2,2,3) C(2,3)	
	addt M(3,2,2,3) C(3,3)	
	addt M(1,2,2,4) C(1,4)	
	addt M(2,2,2,4) C(2,4)	
	addt M(3,2,2,4) C(3,4)	
	mult A(1,3) B(3,1) M(1,3,3,1)	
	mult A(2,3) B(3,1) M(2,3,3,1)	
ldt A(1,4)	mult A(3,3) B(3,1) M(3,3,3,1)	
ldt B(1,2)	mult A(1,3) B(3,2) M(1,3,3,2)	prefetch
ldt A(2,4)	mult A(2,3) B(3,2) M(2,3,3,2)	
ldt B(3,4)	mult A(3,3) B(3,2) M(3,3,3,2)	
	addt M(1,3,3,1) C(1,1)	
	addt M(2,3,3,1) C(2,1)	
	addt M(3,3,3,1) C(3,1)	
	addt M(1,3,3,2) C(1,2)	
	addt M(2,3,3,2) C(2,2)	
	addt M(3,3,3,2) C(3,2)	
	mult A(1,3) B(3,3) M(1,3,3,3)	
	mult A(2,3) B(3,3) M(2,3,3,3)	
	mult A(3,3) B(3,3) M(3,3,3,3)	
	mult A(1,3) B(3,4) M(1,3,3,4)	
ldt B(1,3)	mult A(2,3) B(3,4) M(2,3,3,4)	prefetch
ldt B(4,1)	mult A(3,3) B(3,4) M(3,3,3,4)	
ldt B(4,2)	addt M(1,3,3,3) C(1,3)	
	addt M(2,3,3,3) C(2,3)	
	addt M(3,3,3,3) C(3,3)	
	addt M(1,3,3,4) C(1,4)	
	addt M(2,3,3,4) C(2,4)	
	addt M(3,3,3,4) C(3,4)	
	mult A(1,4) B(4,1) M(1,4,4,1)	
	mult A(2,4) B(4,1) M(2,4,4,1)	
	mult A(3,4) B(4,1) M(3,4,4,1)	
ldt B(1,4)	mult A(1,4) B(4,2) M(1,4,4,2)	prefetch
ldt B(4,3)	mult A(2,4) B(4,2) M(2,4,4,2)	
ldt B(4,4)	mult A(3,4) B(4,2) M(3,4,4,2)	

```
                                addt M(1,4,4,1) C(1,1)
                                addt M(2,4,4,1) C(2,1)
                                addt M(3,4,4,1) C(3,1)
                                addt M(1,4,4,2) C(1,2)
                                addt M(2,4,4,2) C(2,2)
                                addt M(3,4,4,2) C(3,2)
                                mult A(1,4) B(4,3) M(1,4,4,3)
                                mult A(2,4) B(4,3) M(2,4,4,3)
                                mult A(3,4) B(4,3) M(3,4,4,3)
        ldt A(1,1)              mult A(1,4) B(4,4) M(1,4,4,4)
        ldt A(2,1)              mult A(2,4) B(4,4) M(2,4,4,4)
        ldt A(3,1)              mult A(3,4) B(4,4) M(3,4,4,4)
        ldt B(1,1)              addt M(1,4,4,3) C(1,3)
                                addt M(2,4,4,3) C(2,3)
        ldt B(1,2)              addt M(3,4,4,3) C(3,3)
        ldt B(1,3)              addt M(1,4,4,4) C(1,4)
        ldt B(1,4)              addt M(2,4,4,4) C(2,4)
                                addt M(3,4,4,4) C(3,4)
        bne 34
35:
        stt C(1,1)
        stt C(2,1)
        stt C(3,1)
        stt C(1,2)
        stt C(2,2)
        stt C(3,2)
        stt C(1,3)
        stt C(2,3)
        stt C(3,3)
        stt C(1,4)
        stt C(2,4)
        stt C(3,4)
        bne 33
```

References

[AgGZ94] R. C. Agarwal, F. G. Gustavson and M. Zubair, Improving performance of linear algebra algorithms for dense matrices, using algorithmic prefetch, IBM Journal of Research and Development, Vol. 38, NO. 3, May 1994.

[AiNi88] Aiken, A. and Nicolau, A. Optimal loop parallelization, Proc. of the SIGPLAN Conference on Programming Language Design and Implementation, SIGPLAN Notices, 23, 7, June 1988, pp. 308 – 317.

[CaKP91] David Callahan, Ken Kennedy and Allan Porterfield, Software Prefetching, ASPLOS'91, pp. 40–52.

[FaJo94] Keith I. Farkas and Norman P. Jouppi, Complexity/Performance Tradeoffs with Non-Blocking Loads, Proc of the Int. Symp. on Computer Architecture, 1994, pp. 211–222.

[GaPS90] K. A. Gallivan, R. J. Plemmons and A.H. Sameh, Parallel Algorithms for Dense Linear Algebra Computations, in Parallel Algorithms for Matrix Computations by K. A. Gallivan et al. SIAM, 1990, pp. 1–82.

[Lam88] Monica Lam, Software Pipelining: An Effective Technique for VLIW Machines, Proc. of the SIGPLAN'88, pp 318–328.

[LaRW91] M. S. Lam, E. E. Rothberg and M. E. Wolf, The Cache Performance and Optimizations of Blocked Algorithms, ASPLOS 1991, pp. 67-74.

[Mowr94] Todd C. Mowry, Tolerating Latency Through Software-Controlled Data Prefetching, PhD Thesis, Stanford University, May 1994.

[MoLG92] Todd C. Mowry, Monica S. Lam and Anoop Gupta, Design and Evaluation of a Compiler Algorithm for Prefetching, ASPLOS'92, pp. 62–73.

[NaJL94] J.J. Navarro, A. Juan and T. Lang, MOB Forms: A Class of Multilevel Block Algorithms for Dense Linear Algebra Operations, to appear in Proceedings of the ACM International Conference on Supercomputing, 1994.

[Nava94] J.J Navarro et al., Deliverable HwA4: Memory Organization and Management for Linear Algebra Computations. Deliverable of the APPARC Basic Research Action. 1994.

[RaST92] B. Ramakrishna Rau, Michael S. Schlansker, P. P. Tirumalai, Code Generation Schema for Modulo Scheduled Loops, pp 158–169.