Matrix Multiplication: A Case Study of Enhanced Data Cache Utilization

Nadav Eiron
Computer Science Department, Technion
and
Michael Rodeh and Iris Steinwarts
IBM Haifa Research Lab

Modern machines present two challenges to algorithm engineers and compiler writers: They have superscalar, super-pipelined structure, and they have elaborate memory subsystems specifically designed to reduce latency and increase bandwidth. Matrix multiplication is a classical benchmark for experimenting with techniques used to exploit machine architecture and to overcome the limitations of contemporary memory subsystems.

This research aims at advancing the state of the art of algorithm engineering by balancing instruction level parallelism, two levels of data tiling, copying to provably avoid any cache conflicts, and prefetching in parallel to computational operations, in order to fully exploit the memory bandwidth. Measurements on IBM's RS/6000 43P workstation show that the resultant matrix multiplication algorithm outperforms IBM's ESSL by 6.8-31.8%, is less sensitive to the size of the input data, and scales better.

In this paper we introduce a cache aware algorithm for matrix multiplication. We also suggest generic guidelines that may be applied to compute intensive algorithm to efficiently utilize the data cache. We believe that some of our concepts may be embodied in compilers.

Categories and Subject Descriptors: Don't [**know**]: what to put

General Terms: Here too

Additional Key Words and Phrases: Cache, Prefetching, Blocking, Matrix Multiplication, BLAS

A preliminary version of this paper appeared in [Eiron et al. 1998].
Name: Nadav Eiron
Affiliation: Computer Science Department, Technion — Israel Institute of Technology
Address: Computer Science Department, Technion, Haifa, 32000, Israel. `nadav@cs.technion.ac.il`
Name: Michael Rodeh
Affiliation: IBM Haifa Research Lab
Address: IBM Haifa Research Lab, Haifa, 31905, Israel. `rodeh@il.ibm.com`
Name: Iris Steinwarts
Affiliation: IBM Haifa Research Lab
Address: IBM Haifa Research Lab, Haifa, 31905, Israel. `iriss@il.ibm.com`

1. INTRODUCTION

As the gap between CPU and memory performance continues to grow, so does the importance of effective utilization of the memory hierarchy. This is especially evident in compute intensive algorithms that use large data sets, such as most numeric problems. The problem of dense matrix multiplication is a classical benchmark for demonstrating the effectiveness of techniques that aim at improving memory utilization. A straightforward matrix multiplication algorithm performs $O(N^3)$ scalar operations on $O(N^2)$ data items. Moreover, this ratio can be preserved when performing the multiplication operation as a sequence of operations on sub-matrices. This feature of the problem, which is shared by other numeric problems, allows efficient utilization of the memory subsystem.

The efficient implementation of compute-intensive algorithms that use large data sets present a unique engineering challenge. To allow the implementation to exploit the full potential of the program's inherent instruction level parallelism, the adverse effects of the processor-memory performance gap should be minimized. A well engineered compute-intensive algorithm should:

—Manage with small caches;

—Avoid cache conflicts;

—Hide memory latencies associated with "cold-start" cache misses.

A broad set of techniques has been suggested to adapt numeric algorithms to the peculiarities of contemporary memory subsystems. These techniques include software pipelining [Lam 1988], blocking (tiling) [Callahan et al. 1991a] and data copying [Callahan et al. 1991a; Temam et al. 1993]. Each of these techniques was designed as a solution to one of the first two engineering challenges presented above. The relatively new method of selective software prefetching [Callahan et al. 1991b; Mowry 1994; Mowry et al. 1992] aims at the third challenge. Software prefetching attempts to hide memory latencies by initiating a prefetch instruction sufficiently early, before the data item is used. However, the implementation should be carefully designed to avoid cache pollution by the prefetched data (see [Lee et al. 1994]). If the prefetch instruction is non-blocking, the memory access will be executed in parallel with the computations carried out by the CPU.

Basic linear algebra operations on matrices and vectors serve as building blocks in many algorithms and software packages. The importance of efficient implementation of such operations has led to the development of the BLAS (Basic Linear Algebra Subroutines) library standard. The standard is divided into three levels: BLAS-1 deals with vector/vector operations, BLAS-2 includes matrix/vector operations, and BLAS-3 includes matrix/matrix operations.

Previous attempts to improve the performance of BLAS-2 routines are reported in [Agarwal et al. 1994]. Similar work on BLAS-3 routines is presented in [Navarro et al. 1992]. Both these efforts employ a variety of techniques in order to overcome the memory hierarchy limitations. However, these efforts fail to meet all three challenges simultaneously.

We propose a new cache-aware $O(N^3)$ matrix multiplication algorithm which builds upon known techniques to meet all of the three engineering challenges. Our algorithm is based on two observations: (i) The ratio between the number of scalar operations and the data size remains high, even when the problem is divided into a sequence of sub-matrix multiplications. (ii) The system bus is a valuable hardware resource that should be taken into account in the algorithm design.

Our algorithm uses a blocking scheme that divides the matrices into relatively small non-square tiles, and treats the matrix multiplication operation as a series of tile multiplication phases. The data required for each phase is designed to completely fit in the cache. In addition, our scheme maintains a high ratio of scalar operations to the number of data items for each phase.

To maintain conflict-free mapping of the data regardless of the associativity level of the cache, the algorithm restructures the matrices into an array of interleaved tiles. The copying operation can be carried out during the multiplication process, using only a small copying buffer.

To cope with memory latency, all data required during phase i must be prefetched into the data cache during phase $i-1$. This is done simultaneously with the actual computation. The two activities must be well balanced. In particular, the smaller the latency and the higher the memory bandwidth — the smaller the portion of the cache needed. The order and timing of the prefetch instructions is designed to make sure that relevant data is not flushed from the cache. When doing so, the architectural features of the cache must be taken into account.

We prove that our cache-aware $O(N^3)$ matrix multiplication algorithm does not suffer memory latency when running on an architecture that fits the assumptions of our machine model. The performance of our algorithm is not influenced by the size or layout of the input matrices. Assuming that the data set fits in the main memory of the machine, our algorithm maintains its behavior regardless of the data set size. In addition, unlike traditional blocking based algorithms, our algorithm shows little sensitivity to small changes in the input size.

We implemented our algorithm on an IBM RS/6000 PowerPC 604 based workstation. Our implementation allows instruction level parallelism by using tiling at the register level, combined with loop unrolling and software pipelining. The scheduling of machine instructions builds on the fact that in our algorithm, memory access operations are always serviced by the cache. Our implementation outperforms IBM's BLAS-3 matrix multiplication function by roughly 21.5%, on the average, for double precision data. For some values of N, our implementation runs 31.8% faster.

We go further to generalize the techniques used in our cache aware matrix multiplication algorithm to a set of guidelines that may be applied to a wide variety of compute intensive numeric algorithms. Again, we prove that an algorithm that complies with our guidelines will suffer no memory latencies when running on a platform that fits our abstract machine model.

The following section provides general background on the architecture of memory hierarchies in modern computers. In Section 3 we describe the assumptions that we make on the machine architecture. In Section 4 we outline our techniques and their application for matrix multiplication, while in Section 5 we present our implementation for the IBM RS/6000 platform. Section 6 presents the generic guidelines developed as a generalization of our matrix multiplication algorithm.

2. MEMORY HIERARCHY ARCHITECTURE

Abstraction is a fundamental concept in software design: By presenting to the developer an abstract model of computation, the developer can concentrate on the conceptual aspect of the problem to be solved rather than spending significant effort on the peculiarities of a specific machine. The concept of memory hierarchy is aimed at imitating a flat, uniformly accessible, large memory, by taking advantage of *locality of reference* that most programs exhibit. Since fast memory is expensive, a memory hierarchy is organized in levels – each smaller, faster and more expensive than the level below it. The goal is to provide a memory system which costs almost as little as the lowest and cheapest level of the hierarchy and is as fast as the highest level of the hierarchy. The data stored in the different levels of the memory hierarchy usually contain one another.

2.1 Cache

Cache is the name generally given to the top level of the memory hierarchy encountered once the address leaves the CPU. While in the 80s, microprocessors were often designed without caches, today they often come with two or three levels of caches. The level 1 (L1) cache is the smallest, fastest and most expensive memory. It is usually co-located with the processor to minimize its access time.

A *cache block* is the minimum unit of information that can be present in the cache (hit in the cache) or not (miss in the cache). The cache thus holds a certain amount of *cache lines*, each the size of a block, that may hold the data (the line represents the frame, while the block is the data inside the frame). Consider a cache that contains m lines. A *K-way set associative* cache is organized into m/K *sets* where each set contains K lines. The cache may be viewed as made up of K *ways*, each containing one line from every set. The level of the cache associativity determines the mapping of a lower-level memory block into the cache. A block is first mapped into a cache set, and then it can be placed in any line within that set. A *direct mapped cache* is a cache whose sets contain only one line. A *fully associative cache* is a cache that contains only one set.

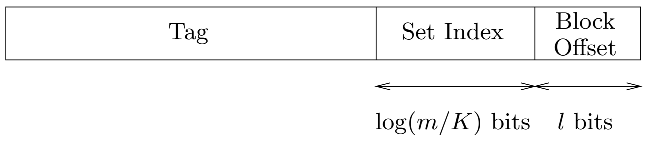


Fig. 1. Mapping into a set associative cache

Figure 1 depicts the mapping of an address into an m-line, K-way set associative cache, with line size 2^l. The index part of the address is used to select one of the m/K cache sets. The offset part is the address of the desired data within the block. The reminder of the address, i.e. the tag, is used to mark the cache line and, by this, to identify the specific block within the line from all other $K - 1$ blocks that reside in the same set.

Cache misses are usually classified into three categories, called the *three C's*:

Cold start misses. Also called *compulsory misses.* A miss that is caused by the first access to a block, which, naturally, is not yet resident in the cache.

Capacity misses. If the cache is not large enough to hold all of the blocks needed during the execution of a program, capacity misses would occur due to blocks that are discarded prematurely.

Conflict misses. Also called *collision* or *interference misses.* A type of miss that, by definition, does not occur in a fully-associative cache. A conflict miss occurs on a new access to a block that was resident in the cache but was previously flushed out of the cache even though other sets of the cache contained stale data. This kind of miss occurs when too many blocks are mapped to the same cache set.

When a cache miss occurs, the cache controller must select a block (one out of the K blocks in the set) to be replaced by the new accessed data. The most common *replacement policies* are LRU (Least Recently Used) and Random. LRU records the accesses being made to the blocks in the set. The block replaced is the one that has been unused for the longest time.

When modifying a resident cache line, the *write-policy* determines how data updates are reflected in the lower-level of the memory hierarchy. There are two common write policies:

Write through. The information is written both to the block in the cache and to the block in the lower-level memory. The advantages of the write through policy is that lower levels of the hierarchy are always consistent with the cache, and that read misses never result in writes to the lower-level.

Write back. Also called *copy back.* Updated information is kept in the cache. The modified cache block is written to the main memory only when it is replaced. The main advantage of the write-back policy is that multiple writes within a block require only one write to the lower-level memory, thus requiring a smaller number of memory read/write transactions and a smaller memory bandwidth.

Since data is accessed also on write, there are also two common options on a write miss:

Write allocate. Also called *fetch on write.* The block is loaded on a write miss, followed by the write hit actions, as dictated by the write policy.

No-write allocate. The block is only modified in the lower level and not loaded into the cache.

For a more complete explanation of cache design, refer to [Hennessy and Patterson 1996, Chapter 5].

2.2 Main Memory

In the memory hierarchy, main memory is the next level down the hierarchy, below the cache levels. There are two common measures to the main memory performance, namely memory *latency* and memory *bandwidth.* Memory latency is the time duration which starts when a read operation is issued and ends when the desired data arrives. Memory bandwidth is the maximum amount of data that may be read from the memory in one unit of time.

2.3 Paged Virtual Memory

Before the introduction of virtual memory in the 60s, programmers had to make sure that their programs fit into main memory. Loading and unloading of software modules was done under user program control. Virtual memory subsystems were invented to relieve programmers of this burden. Virtual memories automatically manage two levels of the memory hierarchy: The main memory and the secondary storage [Hennessy and Patterson 1996].

Several general memory hierarchy terms mentioned in Subsection 2.1 also apply to virtual memory. The term *page* is analogue to a block, the minimum unit of information that is managed by the virtual memory subsystem; the notation of a *frame* is analogue to that of a cache line; *page fault* is the term that substitutes cache miss. However, in spite of the common terminology, virtual memory parameters are radically different than cache parameters. While a typical value for one L1 cache block ranges from 16 to 128 bytes, the typical value of a virtual memory page size ranges from 1KB to 64KB. While the cache hit time is 1–2 cycles, the memory latency is 40–100 clock cycles. Also, while cache miss penalty is 8–100 clock cycles, a page fault costs 700,000–6,000,000 clock cycles [Hennessy and Patterson 1996, Section 5.7].

When using cache terminology to describe the mapping of virtual to physical memory, we can refer to the main memory as fully associative, since it may be viewed as a single set of frames. A page can be placed anywhere within the set, i.e. in any frame of the main memory.

With virtual memory, the CPU produces virtual addresses. The length of a virtual address determines the size of the virtual memory. Virtual addresses are translated by a combination of hardware and software, to physical addresses, which are used to access main memory. This process is called *address translation*.

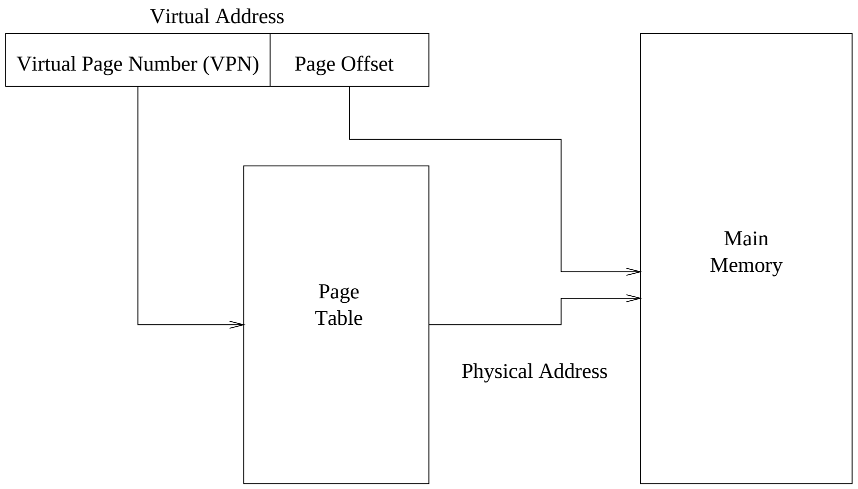


Fig. 2. Translation of a virtual address into a physical address

Figure 2 demonstrates translation from a virtual address into a physical address. The Virtual Page Number (VPN) is used as an index into a data structure called the Page Table. The Page Table Entry (PTE) contains the physical address of the block's frame (or an indication that the page is not resident in the main memory). Physical addresses are obtained by concatenating the physical address of the frame in which the page reside, with the page offset. The cost of the translation process is not negligible: Page tables are usually quite large and do not fit into main memory. As a result, they usually have a recursive structure and are paged in and out. Therefore, the translation process requires at least two memory accesses, one memory access to retrieve the page frame address, and an additional access to retrieve the data.

As opposed to the cache replacement policy which is hardware controlled, the virtual memory replacement policy is primarily controlled by the operating system. In addition, a page fault preempts the current running process and causes a *context switch*. As will be further explained below, context switches either cause a complete cache flush (in case the cache is virtually indexed), or implicitly cause cache contamination by other processes.

2.3.1 *Virtually Versus Physically Indexed Caches.* Virtual addresses, produced by the processor, need to be mapped to physical addresses in order to retrieve information from main memory. A cache that uses virtual addresses for its mapping is called a *virtually-indexed* cache, as opposed to a *physically-indexed* cache that uses physical addresses. The advantage of virtually-indexed caches is that the cache access time is shortened since, on a cache hit, which is the common case, the virtual-to-physical translation process is skipped (since the memory reference is serviced by the cache). However, a virtually-indexed cache must be flushed on every context-switch.

To shorten the translation process, regardless of the cache implementations, a special translation cache, called a Translation Look-aside Buffer (TLB) is used. A TLB entry is like a cache entry where the data portion holds a PTE, and the tag holds a VPN.

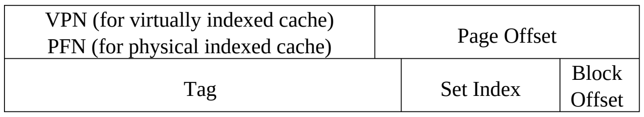

VPN (for virtually indexed cache)
PFN (for physical indexed cache)
Page Offset
Tag
Set Index
Block Offset

Fig. 3. Virtual vs. physical address mapping, when page size $\geq$ way size

Figure 3 depicts the use of an address to select a cache line, where the page size is greater, or equal, to the way size. As can be clearly seen from the figure, the sequence of bits that are used to select the cache set (the index) plus the sequence of bits that are use to select the specific data within the block (the block offset), are a suffix of the page offset. The remainder of the address is only used as the cache tag. If the address is virtual, then the remainder is the VPN + the prefix of the page offset, otherwise the address is physical, and then the reminder is the PFN + the prefix of the page offset. Note that with a page size that is greater or equal to the cache way size, mapping of a virtual address and physical address is the same.

3. THE MACHINE MODEL

When optimizing an algorithm, it is important to properly choose a simple, but sufficiently accurate machine model. The objective is then to define an abstract model such that an algorithm optimized for it will perform well in practice.

In our case, we deliberately decide to ignore the effects of the virtual memory subsystem. Specifically, we ignore the paging mechanism, the use of virtual versus physical addresses for cache indexing, and the use of a TLB to shorten the address translation process. As a consequence, the negative effects of page faults and TLB misses are not taken into account. Furthermore, we assume a virtually-indexed data cache. This assumption is required to allow the algorithm to use virtual addresses when it restructures data in a manner that will assure conflict-free mapping into the cache.

While our machine model may seem very simplistic as compared to contemporary architectures, it is accurate enough for our purposes. Even though the target machine we have for our implementation uses demand-paged virtual memory and more than one level of physically indexed caches, our implementation, which is optimized for the abstract machine, presents better performance than the current state of the art implementations for the architecture in question.

Other assumptions that we make regarding the target machine are:[1]

—The memory subsystem includes at least one level of data cache. Our optimization techniques target only the first level (L1) data cache. We assume that slower caches do not degrade the performance of the L1 cache. Specifically, we require that they do not affect the L1 replacement policy[2].

—The processor supports a non-blocking cache fetch instruction. This may be a specialized prefetch instruction, or a simple non-blocking load instruction [Farkas and Jouppi 1994].

—The L1 data cache write policy is copy-back.

—The L1 data cache replacement policy is Least Recently Used (LRU).

—The CPU follows a load/store (register-register) architecture.

We use the following parameters in the description of our algorithm: The L1 data cache holds C bytes arranged in lines of size L. The cache is K-way set associative. We denote by M the number of machine cycles required to fetch a complete cache line from memory (contemporary machines have $20 \leq M \leq 100$).

4. THE ALGORITHM

Our algorithm is designed to carry out matrix multiplication of the form

$$\mathcal{C} = \mathcal{A} \cdot \mathcal{B}$$

[1]The last three assumptions were chosen to reflect common modern RISC architectures. Conceivably, different assumptions regarding these issues could have been used instead in the design.

[2]This assumption is reasonable, since in most architectures, slower caches are both larger and have a higher degree of associativity than the L1 cache. For example, see [Digital Equipment Corp. 1998].

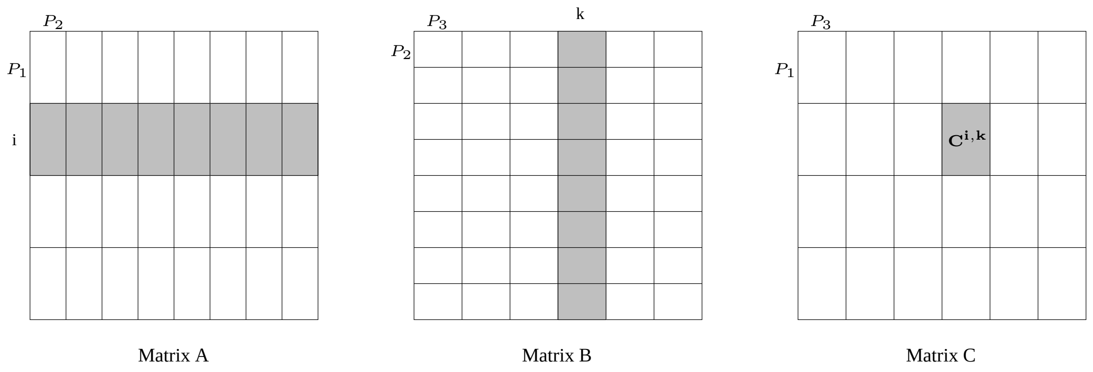


Fig. 4. Tile multiplication

where $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ are real matrices of sizes $N_1 \times N_2$, $N_2 \times N_3$ and $N_1 \times N_3$, respectively. We denote by I the matrices' element size, in bytes. We make no assumption regarding the layout, or relative location, of the input matrices in memory. In the following subsections we outline the algorithm, describing the use of each of the optimization techniques and the way in which the algorithm combines them. We end up with a matrix multiplication algorithm, that by construction, does not suffer memory latency when running on an architecture that fits the assumptions of our machine model.

4.1 Tolerating Capacity and Cold Start Misses

In this subsection we assume that the cache is fully associative. We postpone the discussion on eliminating cache conflicts and pollution to the following subsections.

The algorithm partitions the input matrices into tiles (see Figure 4). The matrix $\mathcal{C}$ is divided into tiles of size $P_1 \times P_3$. Each one of these tiles is generated by a sum of products of a horizontal stripe of $\mathcal{A}$-tiles and a vertical stripe of $\mathcal{B}$-tiles. The matrix $\mathcal{A}$ is thus divided into tiles of size $P_1 \times P_2$ while $\mathcal{B}$ is divided[3] into tiles of size $P_2 \times P_3$. Denoting by $\mathcal{M}^{i,j}$ the (i,j)th tile of the matrix $\mathcal{M}$, we have:

$$\mathcal{C}^{i,k} = \sum_{j=0}^{N_2/P_2-1} \mathcal{A}^{i,j} \cdot \mathcal{B}^{j,k}$$

Since we have $N_1N_3/(P_1P_3)$ $\mathcal{C}$-tiles to compute and since the computation for each tile requires N_2/P_2 tile multiplications, we have a total of $(N_1N_2N_3)/(P_1P_2P_3)$ tile multiplication phases. In each phase we multiply an $\mathcal{A}$-tile of size $P_1 \times P_2$, by a $\mathcal{B}$-tile of size $P_2 \times P_3$, updating a $\mathcal{C}$-tile of size $P_1 \times P_3$, using $P_1P_2P_3$ scalar multiplications and $P_1P_2P_3$ scalar additions. The number of data items accessed in each phase is $P_1P_2 + P_2P_3 + P_1P_3$.

In order to achieve optimal performance, all data used in a specific phase must be present in the data cache at the beginning of the phase. If this condition is

[3]Naturally, this implies that P_1, P_2 and P_3 must all divide the respective dimension, N_i. Padding of the matrices may be used to meet this requirement.

```
for (i = 0 ; i < N1/P1 ; i++)
    for (j = 0 ; j < N3/P3 ; j++)
        for (k = 0 ; k < N2/P2 ; k++) {
            C^{i,j} = C^{i,j} + A^{i,k} · B^{k,j};
        }
```

Fig. 5. The algorithm's outer loops.

indeed met, the system bus remains unused by the tile multiplication code and can instead be used to bring the data required for the *next* phase into the cache. Let W denote the number of machine cycles it takes to multiply an $\mathcal{A}$-tile by a $\mathcal{B}$-tile and store the result in $\mathcal{C}$. Then:

$$W = \Theta(P_1 P_2 P_3).$$

The total amount of data (in bytes) required for each phase is:

$$I \cdot (P_1 P_2 + P_2 P_3 + P_1 P_3).$$

Assuming that each prefetch instruction fills a single line of the cache, the number of prefetch instructions we must issue is

$$\frac{I}{L} \cdot (P_1 P_2 + P_2 P_3 + P_1 P_3).$$

If

$$M \frac{I}{L} \cdot (P_1 P_2 + P_2 P_3 + 2P_1 P_3) \leq W \tag{1}$$

then a single phase runs long enough to allow us to prefetch all the data required by the next phase on time. Note that in Equation (1) the size of the $\mathcal{C}$-tile is multiplied by 2. This is because $\mathcal{C}$-tiles are modified and therefore must be written back to memory, occupying the system bus.

To allow for the latency-free operation of the algorithm, the cache must be large enough to concurrently hold all the data required for a certain phase (i.e., one tile of each matrix: $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$), as well as the data that will be used in the next phase. All in all, the cache must be able to hold two tile triplets, so the total cache size must be at least

$$2I \cdot (P_1 P_2 + P_2 P_3 + P_1 P_3) \leq C. \tag{2}$$

Note that Equation (2) places an upper bound on P_1, P_2 and P_3, while the timing considerations of Equation (1) place a lower bound on these values.

For machines with a large CPU-memory performance gap, it might happen that Euqations (1) and (2) cannot hold simultaneously for any value of P_1, P_2 and P_3. However, in Equation (1) we calculated the number of prefetch instructions required in a single phase assuming that all three tiles should be replaced. The number of prefetch instructions required may be reduced by ordering the phases so that one of the tiles is reused. The code in Figure 5 replaces $\mathcal{C}$-tiles only once in every N_2/P_2 phases. The reuse of $\mathcal{C}$-tiles is beneficial, since these tiles are the only ones that are modified and therefore must be written back to memory. Using the scheme presented in Figure 5, the number of prefetch instructions needed in a single phase is reduced to:

$$\frac{I}{L} \cdot (P_1P_2 + P_2P_3 + \frac{P_1P_2P_3}{N_2}).$$

This means, that the condition of Equation (1) may be relaxed to:

$$M\frac{I}{L} \cdot (P_1P_2 + P_2P_3 + 2\frac{P_1P_2P_3}{N_2}) \leq W. \tag{3}$$

In case this does not yield feasible values of P_1, P_2 and P_3, the implementation's performance may degrade.

Note that, unlike the general case of Equation (1), the relaxed equation spaces the prefetch instructions by a duration of time that depends on N_2. As a consequence, the implementer cannot assume a constant number of operations between two prefetch instructions, making the exact timing of the prefetch instructions harder to program.

Assuming a memory bus that does not allow for pipelined memory access or outstanding requests, all prefetch instructions must be spaced at intervals of at least M units of time apart, to prevent stalling the CPU. With a more advanced bus that allows multiple outstanding memory requests, this requirement can be relaxed, as long as the request queue never overflows. For buses that allow pipelined memory access, a more detailed calculation of the effective value of M should be carried out.

4.2 Avoiding Cache Conflicts

Our algorithm is based on the assumption that any two triplets of tiles (one from each matrix) that are used in two consecutive tile multiplication phases can simultaneously reside in the cache. So far, we have assumed full associativity of the cache. Most practical caches have a limited degree of associativity. Restructuring the data by copying it to properly designed locations can be used to avoid cache conflicts. Temam, Granston and Jalby discusses in [Temam et al. 1993] the utilization of copying to avoid cache conflicts. In the case of matrix multiplication, copying is potentially beneficial, as it takes only $O(N^2)$ time while the computation takes $O(N^3)$ time, assuming that every data element is copied only once.

When using a K-way set associative cache ($K > 1$, K is even[4]), the following condition allows any two triplets of tiles to be mapped into the cache simultaneously:

Condition 1. *Associate with each of the three matrices a multi-set of cache set indices of size sufficient to hold one tile of that matrix. Use copying to map each tile of a specific matrix to the multi-set of cache sets that is associated with that matrix. The mapping is conflict-free if the multi-set union of the multi-sets for the three matrices does not contain any index more than $K/2$ times.*

This condition is sufficient but not necessary, since it is equivalent to having *any* two triplets fit in the cache, and not just the $(N_1N_2N_3)/(P_1P_2P_3)$ triplet pairs used in matrix multiplication. Note that a scheme that complies with Condition 1 may be easily designed to copy each data element only once.

[4]This assumption on K is used for simplicity. If it does not hold, use $\lfloor K/2 \rfloor$ instead of $K/2$ in the discussion that follows.

When using a 2-way set associative cache, Condition 1 may be met by mapping the tiles so that all tiles of a single matrix are mapped to the same sets in the cache, and each cache set is mapped only once. Such a mapping may be formed by interleaving the matrices' tiles, so that the offset between two tiles of the same matrix is a multiple of the way size of the cache. Since the cache is assumed to be 2-way, we can have two tiles from each of the matrices resident in the cache simultaneously.

For a direct mapped cache, Condition 1 cannot be satisfied. To allow conflict-free mapping for direct mapped caches, we must use the fact that not every possible pair of three tiles from the matrices $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ is used. Each matrix will have two possible sets of cache set indices for its tiles, with half of the tiles using one set and the other half using the second set. The tile mapping is chosen so that whenever a tile is replaced by a tile from the same matrix, the two tiles use different sets, and therefore, do not conflict.

To design such a mapping, we divide the cache lines into two equal-sized subsets: a "black" subset and a "white" subset. Each of these subsets is designed to hold one tile from each matrix. Each of the matrices will have half of its tiles mapped to black cache sets and the other half mapped to white cache sets. For the sake of this explanation, we assume that each dimension of the matrices is divided into an even number of tiles (i.e., N_1/P_1, N_2/P_2 and N_3/P_3 are all even). The matrix $\mathcal{A}$ is copied so that all even-numbered *vertical stripes* of tiles are mapped to the black part of the cache and all odd-numbered vertical stripes of tiles are mapped to the white part of the cache. The matrices $\mathcal{B}$ and $\mathcal{C}$ are copied so that all even-numbered *horizontal stripes* of tiles are mapped to the black part of the cache and all odd-numbered horizontal stripes of tiles are mapped to the white part of the cache. It can now be easily verified, that when multiplying tiles in the order of Figure 5, whenever a tile is replaced by a tile from the same matrix, the tile replacing it is colored differently, and therefore the two tiles do not conflict. Note that this method can also be implemented while copying each data element only once.

While it is possible to do all the copying operations required before engaging in the multiplication process itself, it is also possible to do the copying *on the fly*, assuming sufficient memory bandwidth is available. Consider reordering of the outer loops to multiply a vertical stripe of $\mathcal{A}$-tiles by a horizontal stripe of $\mathcal{B}$-tiles while updating all of the matrix $\mathcal{C}$. This order requires a stripe from $\mathcal{A}$ and a stripe for $\mathcal{B}$, totaling $(N_1 + N_3)P_2$ elements, to be copied while $N_1N_3P_2$ arithmetic operations are carried out, leaving enough time to do the copying on the fly. Using copying on the fly requires an initialization stage where the first stripe of $\mathcal{A}$ and the first stripe of $\mathcal{B}$ are copied, taking $O((N_1 + N_3)P_2)$ memory delays. After this initialization, the copying operation will be carried out in parallel to the calculations carried out by the CPU. Note that copying takes only $O(N_1N_2 + N_2N_3)$ time. In our sample implementations, we have chosen to copy it off-line, namely, before the first multiplication operation takes place.

4.3 Avoiding Cache Pollution

As observed in [Lee et al. 1994; Navarro et al. 1992] care must be taken, when performing prefetch operations, in order to assure that essential data is not flushed out of the cache. Similarly, we have to assure that fresh prefetched data is not flushed out before it is used for the first time. Therefore, we have to examine what cache lines are chosen for flushing by the replacement policy when a new cache line is brought into the cache. Since the replacement policy is applied to each set separately, we limit our discussion to a single set.

Consider a K-way set associative cache. Recall that $K/2$ lines of each set are being used by the active triplet of tiles, and the other $K/2$ lines are used to fetch the next triplet before the current phase is over. Let $\{p_i^j\}_{i=0}^{K/2-1}$ be the sorted time instances in which prefetch instructions have been executed during the jth phase, $\{f_i^j\}_{i=0}^{K/2-1}$ be the sorted time instances in which the *first* access to each line holding data for the jth phase has been executed, and $\{l_i^j\}_{i=0}^{K/2-1}$ be the sorted time instances in which the *last* access to each such line during the jth phase took place.

Let us now describe the conditions on the access pattern that will allow pollution-free prefetching. The first condition governs the prefetching code:

Condition 2. *Every block of the tiles used in the jth phase is accessed (prefetched) only once during phase $j-1$.*

Assuming that the above condition holds, the following condition necessary and sufficient for pollution-free prefetching:

Condition 3. *For every $0 \le i \le K/2-1$ and every phase j, denote by r the smallest index for which $f_r^j > p_i^j$ and by s the smallest index for which it holds that $p_s^{j-1} > l_i^{j-1}$. If no such r exists, the prefetch instruction p_i^j is pollution-free. Otherwise, denote by F the cache lines accessed by $\{f_k^j\}_{k=0}^{r-1}$ and by P the cache lines accessed by $\{p_k^{j-1}\}_{k=0}^{s-1}$. The prefetch instruction p_i^j is pollution-free iff $P \subseteq F$.*

Theorem 1. *When both Condition 2 and Condition 3 hold, each prefetch will replace stale data in the cache.*

Proof. Examine the state of the cache set when the prefetch at time p_i^j is executed. If r, as defined in Condition 3, does not exist, then all first accesses to data in the jth phase occurred before p_i^j. Therefore, all cache lines holding data to be used in phase j, have time stamps after the beginning of the jth phase. On the other hand, stale data that remained from the previous phase, has time stamps that are older than the beginning of the current phase, and will therefore be chosen for flushing by the LRU policy.

If r exists, then only the lines in F have been accessed during the jth phase, up to the point in time of p_i^j. In such a case, the candidate to be flushed from the cache may either be a stale data line or a line holding data for the current phase that was not yet accessed. Note that lines of both types will have time stamps from phase $j-1$. Since in the jth phase, we already have i prefetch instructions that are assumed to be pollution-free, i lines that contained stale data were already removed – the lines that were marked with time stamps $\{l_k^{j-1}\}_{k=0}^{i-1}$. Therefore, if a line that contains stale data will be replaced by the new prefetched line, it will be the line accessed at time l_i^{j-1}. To assure that no line holding data for the current phase is to be flushed from the cache, we must check that all the lines that were prefetched during the $j-1$th phase before time l_i^{j-1} have already had their time stamp updated in the jth phase. The set P defined in Condition 3 contains all

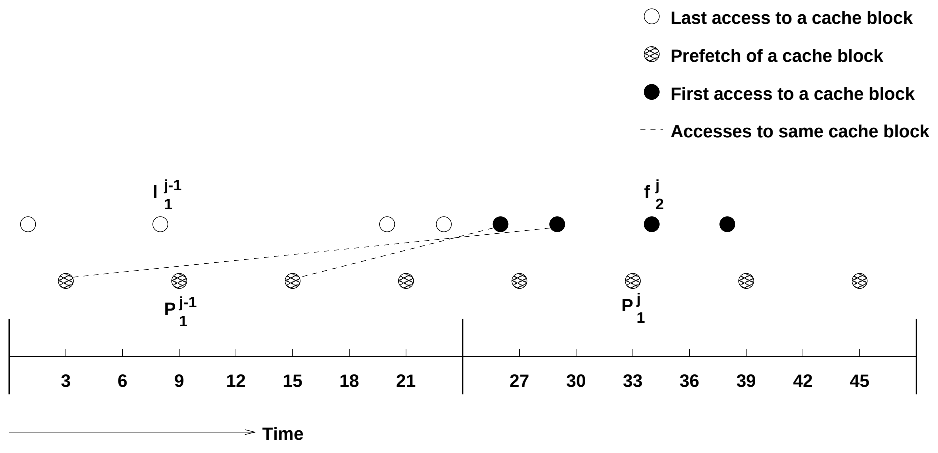

Last access to a cache block
Prefetch of a cache block
First access to a cache block
Accesses to same cache block
l j-1 1
f j 2
P j-1 1
P j 1
3 6 9 12 15 18 21 27 30 33 36 39 42 45
Time

Fig. 6. Determining if a prefetch instruction is pollution-free

these lines. If $P \subseteq F$, these lines have indeed already been accessed in the jth phase, assuring a pollution-free prefetch at time p_i^j. If, on the other hand, there is a line in P that is not in F, then there is a line holding current data in the LRU queue before the oldest line that contains stale data, making the prefetch at time p_i^j a polluting prefetch. $\square$

To illustrate Condition 3, consider Figure 6. In this example, we consider one set of an 8-way set associative cache, which therefore has 8 lines. The algorithm is blocked into phases, and each phase processes 4 lines that reside in the set we consider. Naturally, there are 4 prefetch instructions for lines of this set in each phase.

Figure 6 shows two phasees: the $j-1$th phase (machine cycles: 0 to 23), and the jth phase (machine cycles: 24 to 47). The horizontal axis is time. The prefetch operations are marked by small hashed circles, and are occur every 6 machine cycles. The last access to each line that holds data for phase $j-1$ is marked by the white circles. First accesses to lines in the jth phase are marked by the black circles. Dashed lines join a first access to the prefetch access that brought the relevant block into the cache. Naturaly, there are 4 last accesses and 4 first accesses.

When considering the jth phase, the lines that were prefetched during phase $j-1$ now become the current data set. The lines that were current in the $j-1$th phase, now hold stale data. The prefetch instructions executed during the jth phase are said to be *pollution-free* if the data they replace is stale data. When a prefetch instruction gets executed, the cache line that is flushed out is determined by the LRU policy. The LRU queue holds the sorted time stamps for the last access to each of the 8 lines. Let us consider the state of the LRU queue at the beginning of the jth phase.

The LRU queue at the beginning of the jth phase contains the sorted merger of the two series: l^{j-1} and p^{j-1}. In our example, $p^{j-1} = \{3, 9, 15, 21\}$ and $l^{j-1} = \{1, 8, 20, 23\}$. The first candidate to be replaced is a stale line; it was accessed most recently at time 1. Therefore the first prefetch of the jth phase is definitely a pollution-free prefetch. However, the next candidate to be flushed from the cache is a relevant line whose time-stamp is 3. Luckily, the second first-access in the jth phase (at time 29), to the line that was prefetched at time 3, moves the line far behind the stale lines, to the back of the LRU queue. In our example $f^j = \{26, 29, 34, 38\}$.

Consider the second prefetch instruction of the jth phase, at time $p_1^j = 33$, assuming that all preceding prefetch instructions were pollution-free. To determine if this new prefetch is pollution-free, we need to examine the LRU queue at time 33. At that point in time, one prefetch instruction has already been executed (as $i = 1$). Since the preceding prefetch is assumed to be pollution-free, it flushed the oldest stale line from the cache set. This line is the one that was accessed at time l_0^{j-1}. The oldest stale data line that remains in the cache is therefore the line accessed at time l_1^{j-1}. To make sure that the second prefetch of the jth phase is pollution-free we must check that all prefetches of the $j-1$th phase that occurred before time l_1^{j-1} have been touched in the jth phase before time p_1^j. As seen from the figure, there is one prefetch before time l_1^{j-1} (as $s = 1$) and there are two first accesses before time p_1^j (as $r = 2$). The prefetch instruction accessed the line that makes up the set P, while the two first accesses access the lines that make up the set F. As demonstrated by the dashed lines in the figure, these first accesses indeed contain references to the line in P (that is, $P \subseteq F$). Therefore the prefetch at time p_1^j replaces the stale line that was last accessed at time l_1^{j-1}, and is thus pollution-free.

5. IMPLEMENTATION

To experiment with our approach, we implemented our matrix multiplication algorithm for both single and double precision square matrices. The target platform was a 133MHz PowerPC 604 based IBM RS/6000 43P Model 7248 workstation. The algorithm was implemented in C and was compiled using the IBM XL-C compiler [Stewart 1994]. Where necessary, hand-tuning of the resulting machine code was carried out. To gauge the performance improvements over known techniques, we compared our results to IBM's Engineering Scientific Subroutine Library (ESSL) [IBM Corp. 1997], which is the state of the art implementation of BLAS provided by IBM for this platform.

The version of ESSL we used was specificaly optimized for PowerPC 604 based IBM RS/6000 workstations. According to Gustavson [Gustavson 1998], ESSL is written in Fortran, with some hand-tuned assembly code[5]. ESSL implements most of the known cache related optimizations, including: off-line copying, software prefetching, and tiling at the register, L1 cache, and L2 cache levels. Hence, ESSL may be considered as a formidable banchmark.

5.1 Platform Description

The PowerPC 604 [IBM Microelectronics and Motorola Inc. 1994] is a superscalar, super-pipelined RISC processor. The CPU contains one floating point unit (FPU) and one load-store unit (LSU). It has 32 architectural floating point registers and 32 architectural integer registers. The processor has a 16KB on-chip instruction cache

[5]ESSL also utilizes the `fma` instruction in its matrix multiplication functions. See Section 5.1 below for details on this machine instruction.

and a separate 16KB on-chip data cache ($C = 16384$). Both caches are *physically indexed*, four-way set associative ($K = 4$), and have a line size of 32 bytes ($L = 32$). The write policy for the on-chip data cache is copy-back and the replacement policy is LRU. Access to the cache is done via non-blocking load/store instructions. Note that the PowerPC 604 processor adheres to all of the assumptions of our machine model, except for the use of a physically indexed L1 data cache.

In addition to the L1 cache the machine has an off-chip L2 *directly mapped and physically indexed,* unified cache of size 512KB. The line size of the L2 cache is 32 bytes. The L2 cache controller implements full inclusion of both on-chip L1 caches. Note that this L2 cache design contradicts our assumption that slower caches do not interfere with the replacement policy of the L1 cache.

The AIX operating system uses the PowerPC 604's MMU to implement demand paged virtual memory. The MMU manages two separate TLBs – one for instruction access, and the second for data access. Each TLB contains 128 entries and is two-way set associative. The page size is 4KB, the same as the size of a way of the L1 data cache. This allows us to ignore the page-number part of the virtual address when mapping data to the L1 cache, making the distinction between physically and virtually indexed caches irrelevant.

To complete the picture, let us now examine the features of the PowerPC 604 instructions, which are relevant to our work. The PowerPC architecture supports a floating point multiply-add (`fma`) instruction which performs two floating point operations: a multiplication and an addition. This instruction has four register operands and performs the following calculation: $\mathrm{fp1} \leftarrow \mathrm{fp2}{\cdot}\mathrm{fp3}{+}\mathrm{fp4}$. The pipelined structure of the PowerPC CPU supports issuing one independent floating point instruction in every cycle. The usage of the `fma` instruction is most appropriate for matrix multiplication, since it allows the PowerPC 604 to complete two floating point operations in every cycle. Thus, the IBM RS/6000 43P Model 7248 may achieve, theoreticaly, throughput of up to $2 \cdot 133\text{MHz} = 266$ MFLOPS. By implementing the tile multiplication code using `fma` instructions that use the FPU while loads and stores execute in parallel in the LSU, we assume that the value of W (the amount of work for a single tile multiplication phase) is roughly equal to $P_1P_2P_3$ machine cycles.

Floating point load instructions that hit in the on-chip L1 data cache usually complete in 3 cycles. Since the L1 data cache access is pipelined, one load/store instruction may complete in every cycle.

A load instruction that misses the L1 data cache but hits in the L2 data cache completes in approximately 20 cycles. A load instruction that misses both caches takes roughly 80 cycles (we therefore assume that $M = 80$). While the PowerPC 604 may have up to four outstanding load/store instructions, memory access is not pipelined.

As far as prefetching is concerned, the PowerPC 604 supports a special Data Cache Block Touch (`dcbt`) instruction that fetches the cache line which corresponds to its virtual effective address (VEA) into the cache. This instruction was designed to implement non-faulting software prefetching, as is frequently recommended for use in compile-time algorithms [Hennessy and Patterson 1996, Chapter 5]. Heuristic prefetching, the common approach to use prefetching, may cause faults or exceptions that severely degrade performance. Therefore, the common approach is to avoid prefetching when a fault or exception would result.

While our algorithm could have used the `dcbt` instruction, we decided to use a standard non-blocking register load instruction instead, for the following reasons:

(1) The `dcbt` instruction does not take any action if the VEA causes a TLB miss. Had we used the `dcbt` instruction, we would have lost the confidence that a data item that was prefetched was actually brought into the data cache. Moreover, in our algorithm, most TLB misses occur while prefetching data, and not while actually loading it for immediate use.

(2) The `dcbt` instruction format requires two general purpose registers as operands to calculate the VEA of the data to fetch (`dcbt` rA,rB), unlike the single register and a constant offset used by standard floating point load instructions (for example, `lfs` frD, d(rA)). Therefore, `dcbt` requires a different way of calculating the address to be fetched, thus increasing the overhead of prefetching. In addition, the use of `dcbt` increases register pressure, to which our implementation is especially sensitive, since loops are unrolled as long as registers are available.

(3) The XL-C compiler does not provide a source language feature that triggers the generation of `dcbt` instructions. Had we used it, we would have had to use assembly language to program both the `dcbt` instruction itself and the associated address calculations. On the other hand, C allows the use of `volatile` pointers to force the compiler to generate a load instruction ahead of time (namely, a prefetch), without using the `dcbt` instruction.

5.2 Implementation Details

In this section, we describe our single and double precision implementations. For single precision data we implement in full the matrix multiplication algorithm described in Section 4. For double precision data, the requirements from the memory subsystem, both in terms of bandwidth and of size, are doubled. As a consequence, we were not able to implement our algorithm in full. Instead, we present a partial implementation that hides most, but not all, of the memory latency. However, the two implementations are very similar in nature. In the following sections we describe in detail the single precision implementation. For the double precision implementation, the obstacles that prevented us from implementing the algorithm in full are explained, as is our variant that maximizes performance under these conditions.

As described in Section 4.1, when breaking up matrix multiplication into phases, the algorithm designer can sequence the phases to maximize tile reuse. Since our target platform suffers from high memory latency, we indeed took advantage of this observation. In particular, we chose to reuse every $\mathcal{C}$-tile in every N/P_2 consecutive phases before replacing it.

5.2.1 *Single Precision Implementation.* For the single precision implementation, we chose the following values for the tile-size parameters: $P_1 = P_3 = 32$ and $P_2 = 16$.

PROPOSITION 1. *For single precision implementation on the IBM RS/6000 43P Model 7248, the choice* $P_1 = P_3 = 32$ *and* $P_2 = 16$ *complies with both Equations*

```
struct Triplet {
        float A_tile[P1][P2];
        float C_half1[P1][P3/2];
        float B_tile[P2][P3];
        float C_half2[P1][P3/2];
};
```

Fig. 7. Conflict free mapping

(2) and (3), assuming the input matrices are at least of size 6×6.

PROOF. Each $\mathcal{A}$-tile and each $\mathcal{B}$-tile has 512 elements (or 2KB in size), while each $\mathcal{C}$-tile has 1024 elements (or 4KB in size). We see that a triplet has 2KB + 2KB + 4KB = 8KB. Since the cache is 16KB in size, Equation (2) is satisfied.

The total amount of work per phase is $W = P_1P_2P_3 = 32 \cdot 16 \cdot 32 = 16384$. Plugging in Equation (3) the values for M, P_1, P_2, P_3, I, and L, we have:

$$M\frac{I}{L} \cdot (P_1P_2 + P_2P_3 + 2\frac{P_1P_2P_3}{N_2}) = 80\frac{4}{32}(512 + 512 + 2\frac{16384}{N}) \leq 16384 = W$$

This inequality holds for all $N \geq 16/3$. Since we assume $N \geq 6$, Equation (3) is satisfied. □

When choosing to reuse $\mathcal{C}$-tiles the frequency of prefetches depends on N. Since N is an input parameter, prefetching data from $\mathcal{C}$ at the correct frequency would have required the use of code that relatively often checks whether a prefetch should be executed. For efficiency reasons, we prefetch $\mathcal{C}$ outside of the tile multiplication code, while prefetching of $\mathcal{A}$- and $\mathcal{B}$-tiles is done inside the inner-most loop. $\mathcal{A}$- and $\mathcal{B}$-tiles are prefetched at a constant rate of one prefetch instruction every 128 cycles. Data from $\mathcal{C}$ is prefetched at a much slower rate. $\mathcal{C}$ prefetches would not cause the code to stall on memory accesses, even though they are not precisely timed, since the processor allows up to 4 outstanding memory requests. This suffices because, even if we did one prefetch of $\mathcal{C}$ for each prefetch from $\mathcal{A}$ and $\mathcal{B}$, we would have had 3 prefetch instructions executed in 256 cycles, which leaves more than 80 cycles per instruction.

To allow conflict-free mapping of two tile triplets into the cache, the matrices are first copied into a page-aligned array of the `Triplet` structure, shown in Figure 7.

PROPOSITION 2. *For single precision implementation on the IBM RS/6000 43P Model 7248, copying the tiles of the three matrices to a page-aligned array of the* `Triplet` *structure satisfies Condition 1.*

PROOF. The array is page-aligned, each structure is exactly the size of two pages, and the page size is the same as the L1 way size. This implies that every $\mathcal{A}$-tile is mapped into sets 0 to 63 of the cache, exactly once, and so is every $\mathcal{B}$-tile. For sets 64 to 127 of the cache, there are two blocks of each $\mathcal{C}$-tile mapped into each set. As the cache is 4-way set associative, Condition 1 is satisfied, allowing the simultaneous mapping of any two triplets of tiles into the cache. □

To implement the tile multiplication code, we used tiling at the register level. We divided $\mathcal{B}$-tiles into sub-tiles of size 4×4 elements each. $\mathcal{A}$- and $\mathcal{C}$-tiles are divided accordingly into vertical stripes. In our inner-most loops we load a single sub-tile of $\mathcal{B}$ into 16 registers and then traverse a 4-element wide vertical stripe of both $\mathcal{A}$ and $\mathcal{C}$. In each iteration, we multiply four elements of $\mathcal{A}$ by a sub-tile of $\mathcal{B}$, while updating four elements of $\mathcal{C}$, totaling 16 scalar multiplications This inner-most loop is unrolled to allow prefetching of $\mathcal{A}$- and $\mathcal{B}$-tiles without using conditional constructs.

Mapping of elements within the tile storage area is designed such that first access made by the tile multiplication code occur in increasing address order. $\mathcal{A}$ and $\mathcal{C}$-tiles are copied in vertical stripes, four elements wide. Each such 4-element sub-row occupies consecutive memory addresses, and these sub-rows are ordered in column-major order. $\mathcal{B}$-tiles are copied such that every sub-tile of size 4×4 occupies a contiguous area of memory. These sub-tiles are again arranged in column-major order within the tile.

Prefetches are carried out according to the tiles' layout in memory. The prefetches of $\mathcal{A}$ and $\mathcal{B}$ are interleaved within the unrolled inner-most loop; one from $\mathcal{A}$ and then one from $\mathcal{B}$. Prefetches for the two halves of $\mathcal{C}$ are interleaved, with each half accessed in increasing address order. Every block is prefetched only once, satisfying Condition 2.

Proposition 3. *For single precision implementation on the IBM RS/6000 43P Model 7248, our implementation complies with Condition 3.*

Proof. Our tile multiplication code loads a complete line of $\mathcal{B}$ into registers, operates on the registers, and never access that same line again within this tile multiplication phase. Thus every line of the $\mathcal{B}$-tile is touched only once during a phase, making the first access within a phase the last one. Since prefetching also touches each line in the $\mathcal{B}$-tile exactly once, the number of prefetches from a $\mathcal{B}$-tile is the same as the number of accesses to the current $\mathcal{B}$-tile. Note that both access patterns are in consecutive, increasing set indices order. The lines of $\mathcal{A}$ are also prefetched in the order of their set indices and at the same frequency as the lines of $\mathcal{B}$. Our code is arranged so that each prefetch of a line of $\mathcal{A}$ and $\mathcal{B}$ follows the single access to the corresponding $\mathcal{B}$-line in the current data set. This implies that the line of $\mathcal{A}$ that resides in the same set as a certain line of $\mathcal{B}$ is prefetched only after that line of $\mathcal{B}$ is used in the current phase. The $\mathcal{A}$-tile is also traversed in the order of memory addresses, but at a rate that is 8 times greater than the rate at which prefetches are done (each element of $\mathcal{A}$-tile is accessed 8 times during a phase when multiplied by elements from each of the 8 sub-tile columns of the $\mathcal{B}$-tile). Hence, when a prefetch occurs (whether it is for data from $\mathcal{A}$ or from $\mathcal{B}$), the first accesses to the current data (both from $\mathcal{A}$ and from $\mathcal{B}$) that reside in the same set have already occurred. In the terminology of Condition 3, this means that $f_i^j < p_0^j$ for all j and i. Hence, we have for all m that $f_m^j \leq p_i^j$, both when prefetching the data from $\mathcal{A}$ and when prefetching the data from $\mathcal{B}$, and therefore both prefetches are pollution-free by the first case in Condition 3.

As far as prefetching $\mathcal{C}$-tiles, our implementation complies with Condition 3, when examining a sequence of tile multiplication phases through which a single $\mathcal{C}$-tile is used. Condition 3 trivially holds (i.e., no value of r exists for any prefetch instruction p_i), since $\mathcal{C}$ prefetching is carried out outside of the tile multiplication code. During the first tile multiplication in a sequence that uses a specific $\mathcal{C}$-tile, the complete $\mathcal{C}$-tile is accessed. Therefore, all first accesses precede any prefetch of data that is mapped to the same cache set, or, in the terminology of Condition 3, $f_1^j < p_0^j$ for all j (there are two blocks of a single $\mathcal{C}$-tiles mapped to each set, and therefore the last first access is numbered 1). $\square$

5.2.2 *Double Precision Implementation.* For double precision matrix multiplication, the bandwidth requirements are higher by a factor of 2 as compared to single precision matrix multiplication. In addition, the space taken up in the cache is also doubled. This aggravates the problem of hiding the memory latency. Even if we forgo prefetching of $\mathcal{C}$-tiles all together, we still have to comply with the following condition, derived from Equation (3):

$$M\frac{I}{L}(P_1P_2 + P_2P_3) \le P_1P_2P_3 = W$$

or:

$$\frac{P_1 + P_3}{P_1P_3} \le \frac{L}{MI}.$$

To make the left hand side as small as possible, assuming we fix the value of P_1P_3, we should take $P_1{=}P_3$. Note that $I \cdot P_1P_3$ is exactly the size of a $\mathcal{C}$-tile. Therefore, making P_1P_3 larger will increase the portion of the cache consumed by the $\mathcal{C}$-tiles. For $P_1 = P_3 = 32$, a single $\mathcal{C}$-tile already occupies 8KB, which is half the total cache size of the PowerPC 604, leaving only at most 2KB for a single tile of $\mathcal{A}$ or $\mathcal{B}$. This implies[6] that P_2 cannot be greater than 8. As shown below, even with this choice of parameters, we have only 64 cycles between two prefetch instructions; this is not enough for completely hiding the memory latency associated with accesses to $\mathcal{A}$ and $\mathcal{B}$. Therefore, the timing constraints do not allow prefetching $\mathcal{C}$-tiles. Moreover there is no room in the cache for a second $\mathcal{C}$-tile.

In view of the above, for the double precision implementation, we choose the following values for the tile size parameters: $P_1 = P_3 = 32$ and $P_2 = 8$. Given these tile parameters, each $\mathcal{A}$-tile and each $\mathcal{B}$-tile is 256 elements (or 2KB) in size, while each $\mathcal{C}$-tile is 1024 elements (or 8KB) in size. Since we only prefetch $\mathcal{A}$- and $\mathcal{B}$-tiles, the cache should be capable of holding two pairs of tiles from $\mathcal{A}$ and $\mathcal{B}$ in addition to the current $\mathcal{C}$-tile. The total amount of space taken up in the cache is therefore exactly 16KB.

The total amount of work per phase is $W = P_1P_2P_3 = 32 \cdot 8 \cdot 32 = 8192$. During this time we prefetch one $\mathcal{A}$-tile and one $\mathcal{B}$-tile, totaling:

$$\frac{I}{L}(P_1P_2 + P_2P_3) = \frac{8}{32}(256 + 256) = 128$$

cache lines. This means that a prefetch instruction should be executed once every $8192/128 = 64$ cycles. The prefetch instructions for $\mathcal{A}$- and $\mathcal{B}$-tiles are interleaved in the tile multiplication code. As we have $M = 80$ for this machine, memory latencies are not completely hidden, and sub-optimal performance will result (see Section 5.3 for a discussion of the performance penalties).

[6]We restrict ourselves to values of P_1, P_2 and P_3 that are powers of 2 to make our pointer arithmetic more efficient.

To provide partial remedy to the performance penalty observed in the double precision case, we have developed a variant that departs from our generic cache-aware matrix multiplication algorithm. We take advantage of the fact that $\mathcal{C}$ is used for output only, and therefore the initial values of $\mathcal{C}$ are all zeros. We therefore use a single memory buffer to hold $\mathcal{C}$-tiles, which is always resident in the cache. Whenever a new $\mathcal{C}$-tile is required, the old one is saved into the original $\mathcal{C}$ matrix and the buffer is cleared. By using a single, cache-resident, buffer to hold $\mathcal{C}$-tiles, we compensate for our inability to prefetch the data from $\mathcal{C}$. However, since the L1 cache uses allocate on write policy, copying back of the $\mathcal{C}$-tiles allocates cache lines to hold the modified data of the $\mathcal{C}$ matrix. This allocation may cause flushing of prefetched data from $\mathcal{A}$- and $\mathcal{B}$-tiles that is intended for use in the next tile multiplication phase. Since the combined size of an $\mathcal{A}$-tile and of a $\mathcal{B}$-tile is only 4KB, compared to 8KB for a single $\mathcal{C}$-tile, the saving of delays for access to $\mathcal{C}$-tiles is advantageous.

5.3 Results

5.3.1 *Double Precision.* Figure 8 (a) shows the performance in MFLOPS of our EDCU (Enhanced Data Cache Utilization) single $\mathcal{C}$-buffer double precision matrix multiplication implementation vs. IBM's BLAS-3 (shown as ESSL). Figure 8 (b) shows the relative performance of these two implementations.

For double precision data we get the following average performance figures for the sizes of inputs we checked: IBM's ESSL implementation achieves 115 MFLOPS, our standard implementation achieves 130 MFLOPS, and our single $\mathcal{C}$-buffer implementation achieves 140.8 MFLOPS, a 21.5% average advantage over ESSL. Another feature of our algorithm that can be clearly seen from Figure 8 is that, performance-wise, our design is less sensitive to changes in the size of the data in comparison to ESSL. The performance instability evident in ESSL allows the single $\mathcal{C}$-buffer implementation to outperform the ESSL implementation by up to 31.8% on some double precision matrices. By running our implementation with the prefetching instructions removed, we see that prefetching alone contributes 12–15% of the total performance gain. The contribution of the other techniques we used (tiling, copying, etc.) is not easy to measure directly, and is beyond the scope of this work.

5.3.2 *Single Precision.* For single precision data, the tile sizes we use are sufficient to allow prefetching of all the required data. However, the memory bandwidth is not taxed as heavily in the single precision implementation, as it is in the double precision implementation, and so the potential for performance improvement via memory latency hiding is smaller. This is clearly demonstrated by the performance achieved by the naive 3-loop $O(N^3)$ matrix multiplication algorithm. While for double precision data the naive 3-loop implementation achieves only 13.6 MFLOPS on average, allowing us to achieve a 935% performance increase over it, its performance is almost doubled to 23.7 MFLOPS on average for single precision numbers. For the single precision implementation, we got the following average performance figures for the sizes of input we checked: IBM's ESSL implementation achieves 154.9 MFLOPS while our EDCU implementation achieves 165.5 MFLOPS, a 7% average increase over ESSL, and up to 13% for some single precision matrices.

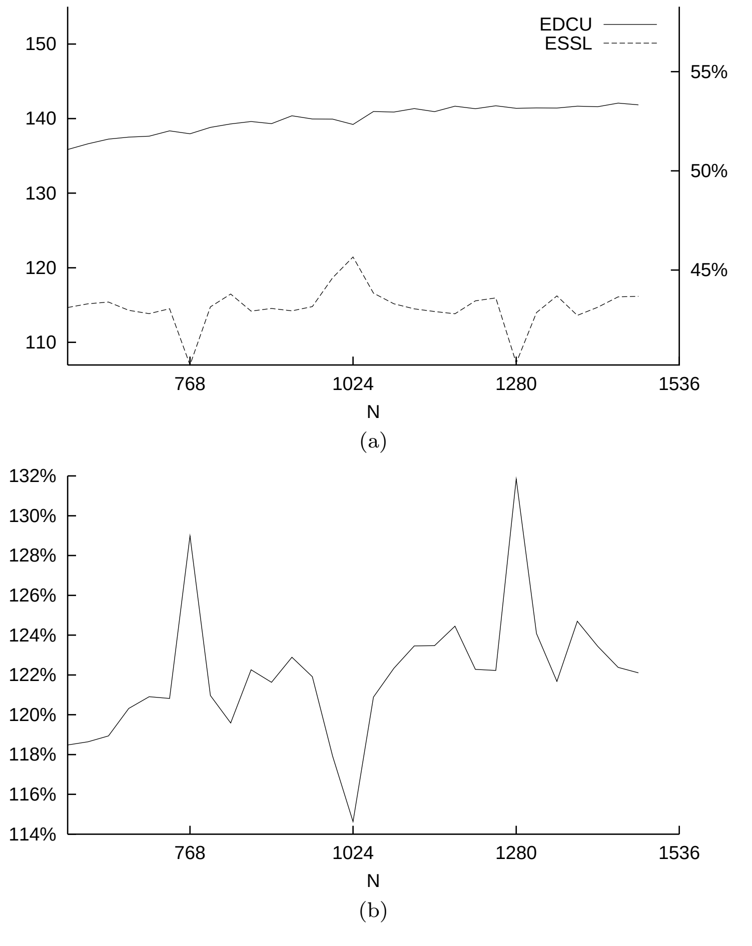
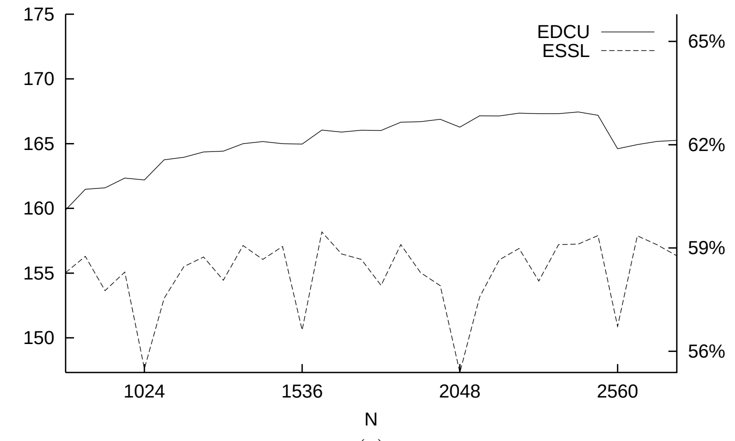
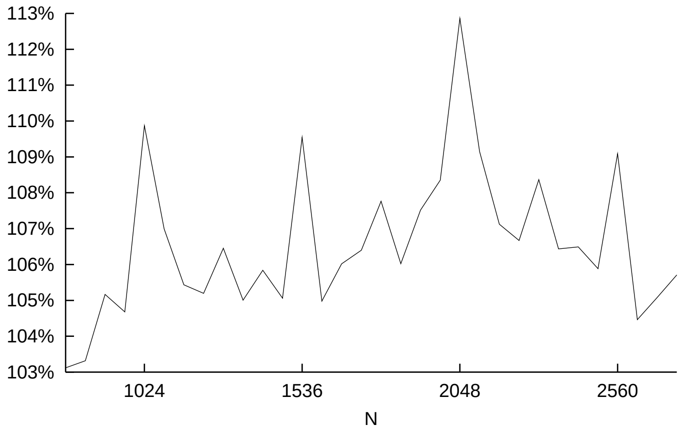


Fig. 8. Performance of double precision matrix multiplication on the RS/6000 43P Model 7248: (a) In MFLOPS and percentage of theoretical peak, (b) Relative to ESSL

5.3.3 *Analysis.* The main reasons for not reaching the full potential of the CPU are related to the specific machine which does not allow a complete implementation of our algorithm. First, the machine has a relatively small L1 cache, when considering its relatively high memory latencies. Our double precision implementation could not prefetch the $\mathcal{C}$-tile, forcing us to use instead, a single $\mathcal{C}$ buffer that is copied back on every N/P_2 tile multiplication phases. The copy back operation pollutes the cache, causing delays in subsequent accesses to $\mathcal{A}$- and $\mathcal{B}$-tiles. In addition, since the tile sizes we use allow only 64 cycles between prefetch instructions, we cannot fully hide memory latency, which is as high as 80 cycles. As a consequence, the tile multiplication code of the double precision implementation is prolonged by roughly 25% (the difference between 80 and 64). Second, the direct mapped L2 cache forces a full inclusion policy on the instruction and data L1 caches. Therefore, some data may be flushed out of the L1 cache because of



(a)



(b)

Fig. 9. Performance of single precision matrix multiplication on the RS/6000 43P Model 7248: (a) In MFLOPS and percentage of theoretical peak, (b) Relative to ESSL

conflicts in the L2 cache, which may even result from instruction access. Third, as noted in Section 4.2, before engaging in the actual multiplication process, we copy our input matrices into an array of interleaved tiles. For the sake of simplicity, we chose to carry these copying operations off-line. These copying operations take time and our measurements indicate that the overhead for the input sizes we used is at least 14% of the peak for double precision data. Clearly, since this overhead is $O(N^2)$ its relative influence on performance diminishes as the size of the data increases. We have also estimated the overhead of the control constructs of the two inner most loops to be roughly 5%, for both the single precision and the double precision implementations. Outer loops contribute even more overhead to the total running time of the implementation.

Last, but not least, one must bear in mind that the measurements we made were carried out under a general-purpose time sharing operating system. Context switches, interrupt processing, page faults and other system activities contribute to the total running time of the implementation, even when measuring just CPU time, by causing pollution of the caches and the TLBs.

6. GENERIC GUIDELINES

In this section we present a generalization of the matrix multiplication algorithm described in Section 4. We generalize the algorithm to a set of conditions and guidelines that may be applied to a much wider class of compute intensive algorithms that use a large data set. Implementations that follow these guidelines allow processing of the input data set in a manner that does not suffer from memory latency, when running on architectures that comply with our abstract machine model.

6.1 Tolerating Capacity and Cold Start Misses

For the sake of this discussion we assume that the cache is fully associative. Again, we defer the discussion of other cache designs to the following subsection.

Consider an algorithm which runs in time $f(D)$ to process a data set $\mathcal{D}$ of size D bytes. We impose the following requirements:

—The algorithm may be broken into phases, such that the total running time of the algorithm remains $f(D)$.

—Each phase of the algorithm accesses a subset δ_i of the data.

—Each of the subsets δ_i occupies l cache lines.

—Each phase modifies m lines out of the l lines it uses.

—Each phase executes in W cycles, assuming all memory accesses hit in the L1 data cache. Notice that W is a function of the size of δ_i.

If

$$M \cdot (l + m) \leq W \tag{4}$$

then a single phase runs long enough to allow us to prefetch all the data required by the next phase on time. Note that Equation (4) includes the term m that represents the system-bus transactions that will be executed in order to write the modified lines back into the lower levels of the memory hierarchy.

Assuming that the time complexity W of a single phase is super-linear in the size of its inputs, Equation (4) places a lower bound on the data size used in each phase, since, to satisfy the condition of Equation (4), we must have:

$$M \leq \frac{W}{l + m}$$

Since $l + m$ is clearly at most linear in l, we get a lower bound on W/l. Since W is super-linear in the size of the data, this equation holds for a sufficiently large value of l, and thus it imposes a lower bound on l.

To allow for latency-free computation during every phase, the cache must be large enough to simultaneously hold the data required in every phase as well as the data that will be used in the next phase. Therefore, we have the following requirement:

$$2 \cdot l \leq \frac{C}{L} \tag{5}$$

This requirement places an upper bound on the size of the data set used by each phase.

Assuming that both (4) and (5) are met by the algorithm and the machine characteristics, we may resort to a blocked implementation. Prefetch instructions may be spread within each phase so that, if prefetching is pollution-free, all the data required for a phase is present in the L1 cache at the beginning of the phase. This allows us to completely eliminate capacity misses and hide all cold-start misses, provided that the cache is fully-associative.

In this discussion, we made the assumption that the data sets used by any two phases are disjoint. If this assumption may be relaxed then some of the phases will reuse data of previous phases, thereby reducing the load on the system-bus. This, of course, leads to a relaxation of Equation (4), analogous to Equation (3), that supports efficient implementations even when memory bandwidth is too low to satisfy Equation (4).

6.2 Avoiding Cache Conflicts

We now consider cache designs that are not fully associative. When using a K-way set associative cache where K is even (otherwise use $\lfloor K/2 \rfloor$ instead of $K/2$), the following condition allows any two data sets to be mapped into the cache simultaneously:

Condition 4. *The data should be restructured so that each data set δ_i occupies at most $K/2$ lines in each cache set and at most l cache lines total, such that no more than m of them are modified in any phase.*

This condition is sufficient but not necessary, since it is equivalent to having *any* two data sets fit in the cache, and not just two data sets used in consecutive phases.

Condition 4 does not apply to direct mapped caches. In general, for direct mapped caches, the data should be restructured such that for every phase i, the cache lines occupied by δ_i and δ_{i+1} are disjoint. While this can be achieved by copying each δ_i to a pre-designated location, such copying involves considerable overhead. However, in many practical cases (see Subsection 4.2), by exploiting the internal structure of the phases' data sets and the code of the algorithm, mapping that allows conflict-free operation with direct-mapped caches can still be obtained with only moderate copying.

6.3 Avoiding Cache Pollution

Careful examination of the conditions on access patterns described in Section 4.3 shows that these conditions were not specific to the case of matrix multiplication. Theorem 1 and its proof may be applied, unchanged, even in the generalized setting.

7. CONCLUSION

To achieve good performance, numeric algorithms should balance computation with data movement. We have presented a new cache-aware $O(N^3)$ matrix multiplication algorithm. We have proved this algorithm to suffer no memory latency when running on an architecture that fits the assumptions of the machine model introduced. Furthermore, this algorithm uses only the smallest part of the cache that can still balance the memory bandwidth. Using a larger cache than the minimum required will have no further impact on performance.

Our experiments show that, even for platforms that are not ideally suited for the suggested techniques, the implementation of the matrix multiplication algorithm we present is superior.

While the technique presented in this work was demonstrated for $O(N^3)$ matrix multiplication, we also presented generic guidelines and conditions for cache-aware design of compute intensive algorithms, that promise latency-free operation. It is yet to be established in practice whether these guidelines may be effectively implemented on real-life architectures.

Insight regarding effective exploitation of contemporary memory hierarchies has been gained. We believe that some of our concepts may be embedded in compilers.

ACKNOWLEDGMENTS

Many members of the IBM Haifa Research Lab helped us throughout this research. We are specifically indebted to David Bernstein for his helpful advice, that contributed greatly to this work.

REFERENCES

Agarwal, R. C., Gustavson, F. G., and Zubair, M. 1994. Improving performance of linear algebra algorithms for dense matrices, using algorithmic prefetch. *IBM J. of Research and Development 38*, 3, 265–275.

Callahan, D., Kennedy, K., and Porterfield, A. 1991a. The cache performance and optimizations of blocked algorithms. In *Proceedings of ASPLOS'91* (1991), pp. 63–74.

Callahan, D., Kennedy, K., and Porterfield, A. 1991b. Software prefetching. In *Proceedings of ASPLOS'91* (1991), pp. 40–52.

Digital Equipment Corp. 1998. *21164 Alpha Microprocessor Data Sheet*. Digital Equipment Corp.

Eiron, N., Rodeh, M., and Steinwarts, I. 1998. Matrix multiplication: A case study of algorithm engineering. In *Proceedings of WAE'98* (August 1998), pp. 40–52. Max-Plank-Institut für Informatik.

Farkas, K. and Jouppi, N. 1994. Complexity/performance tradeoffs with non-blocking loads. In *Proceedings of ISCA'94* (1994), pp. 211–222.

Gustavson, F. G. 1998. Personal Communication.

Hennessy, J. L. and Patterson, D. A. 1996. *Computer Architecture: A Quantitive Approach* (Second ed.). Morgan Kaufmann Publishers Inc.

IBM Corp. 1997. *IBM Engineering and Scientific Subroutine Library for AIX, Version 3 – Guide and Reference*. IBM Corp.

IBM Microelectronics and Motorola Inc. 1994. *PowerPC 604 RISC Microprocessor User's Manual*. IBM Microelectronics and Motorola Inc.

Lam, M. S. 1988. Software pipelining: An effective technique for vliw machines. In *Proceedings of SIGPLAN'88* (1988), pp. 318–328.

Lee, J. H., Lee, M. Y., Choi, S. U., and Park, M. S. 1994. Reducing cache conflicts in data cache prefetching. *Computer Architecture News 22*, 4, 71–77.

Mowry, T. C. 1994. *Tolerating Latency Through Software-Controlled Data Prefetching*. Ph. D. thesis, Stanford University.

Mowry, T. C., Lam, M. S., and Gupta, A. 1992. Design and evaluation of a compiler algorithm for data prefetching. In *Proceedings of ASPLOS'92* (1992), pp. 62–73.

Navarro, J. J., García-Diego, E., and Herrero, J. R. 1992. Data prefetching and multilevel blocking for linear algebra operations. In *Proceedings of ICS'96* (1992), pp. 109–116.

Stewart, K. E. 1994. Using the xl compiler options to improve application performance. In *PowerPC and POWER2, Technical Aspects of the New IBM RISC System/6000*. IBM Corp.

Temam, O., Granston, E. D., and Jalby, W. 1993. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Proceedings of SUPERCOMPUTING'93* (1993), pp. 410–419.