Compiler optimization to improve data locality for processor multithreading

Balaram Sinharoy
IBM Corporation, East Fishkill, NY 12533, USA
E-mail: balaram@watson.ibm.com

Over the last decade processor speed has increased dramatically, whereas the speed of the memory subsystem improved at a modest rate. Due to the increase in the cache miss latency (in terms of the processor cycle), processors stall on cache misses for a significant portion of its execution time. Multithreaded processors has been proposed in the literature to reduce the processor stall time due to cache misses. Although multithreading improves processor utilization, it may also increase cache miss rates, because in a multithreaded processor multiple threads share the same cache, which effectively reduces the cache size available to each individual thread. Increased processor utilization and the increase in the cache miss rate demands higher memory bandwidth. A novel compiler optimization method has been presented in this paper that improves data locality for each of the threads and enhances data sharing among the threads. The method is based on loop transformation theory and optimizes both spatial and temporal data locality. The created threads exhibit high level of intra-thread and inter-thread data locality which effectively reduces both the data cache miss rates and the total execution time of numerically intensive computation running on a multithreaded processor.

1. Introduction

High performance multiprocessor systems are usually built using sophisticated microprocessor(s) at each node of a large processor interconnection network. Most applications require data sharing among the processing nodes in a multiprocessor, although the degree of sharing varies considerably from application to application. The scalability of a multiprocessor system is usually sublinear. As the amount and frequency of data sharing among the processing nodes increase, the scalability of a multiprocessor system reduces. To cope with the sublinear scalability and still provide high performance, it is important to use the most powerful processors at each node of the multiprocessor system and take necessary steps to increase the utilization of these processors. Current multiprocessor systems reflect this trend.

Recent developments in microprocessor technology have improved the processor performance by more than two orders of magnitude over the last decade. However the memory subsystem performance and the performance of the processor interconnection network did not improve nearly as fast. Both of these significantly impact the performance of a multiprocessor system. Not only the cycles per instruction (CPI) of the microprocessors are decreasing rapidly, the processor cycle time is also decreasing and it is decreasing at a rate much faster than the memory cycle time. Recent studies show that the number of processor clock cycles required to access main memory doubles in approximately every 6.2 years [4]. This divergence between the processing power of the modern microprocessors and the memory access latency significantly limits the performance of each node in a multicomputer. Trends in the processor technology and the memory technology indicate that the divergence will continue to widen at least for the next several years. It is not unusual to find that as much as 60% of a task's execution time is spent loading the cache, while the processor is stalled [12].

As the gap between the average instruction processing time by the microprocessor and the average memory access time continues to increase, the reliance on compiler optimization techniques will increase to help alleviate the problem. Compiler optimization can help in at least three different ways.

(1) *Program restructuring*: use program control flow structure (perhaps with added information from execution profiles of earlier runs) to determine code transformation that can increase the spatial and temporal locality of the program to reduce the cache miss rates [13,11,23].
(2) *Software prefetching*: use data flow and control flow information to insert cache load instructions well ahead of the time when the cache line is needed [15].

Scientific Programming 7 (1999) 21–37
ISSN 1058-9244 / $8.00

(3) *Multithreading*: attempt to hide the memory latency by switching, in a few processor cycles, to a new thread of execution whenever an event occurs that stalls the processor for a number of cycles (for example, cache misses or TLB[1] misses) [1].

Among the three compiler optimization methods mentioned above, compiler optimization for multithreading is the least understood, even though it does not suffer from some of the disadvantages that other methods do (see Section 2).

Multithreading (or Multiple Context Processor) has been proposed in the literature to hide the increasingly higher memory access latency [1]. Several multithreading approaches has been proposed in the literature, as follows.

- *Blocked multithreading*: If the running thread encounters a long latency operation, the processor suspends the running thread, saves its state of execution and switches to a ready thread (if available), whose state of execution is available on a separate set of registers.
- *Simultaneous multithreading*: The processor simultaneouly fetches instructions from multiple threads and dispatches the instructions to various execution units in it for simultaneous execution in any given cycle [21].
- *Fine-grain multithreading*: The processor switches thread in every cycle. This typically requires a large number of threads ready to run on the processor.

Multithreading allows several independent threads of execution to be in the running state on a single processor in a given time. Except for simultaneous multithreading, only one of the threads actually runs on the processor at a given cycle. Irrespective of the processor multithreading approach taken, all the threads dispatched to the processor will appear to the operating system to be running simultaneously on the processor. In this sense, a multithreaded processor will appear to the software as a multiprocessor where each processor shares the common resources, such as the TLB and the Caches.

The state of execution of all the threads that are dispatched to run concurrently on the processor are saved. Besides the contents of the memory (or the contents of the address spaces that belongs to the thread), the state of execution of a thread is determined by the contents of various registers in the processor (such as, the general purpose registers, the floating point registers, the processor status registers, the condition code registers, etc.) that are used by the thread. In a multithreaded processor, each thread is assigned its private set of such registers. All the threads running on a processor can share the same cache and TLB.

In a blocked multithreaded processor, when the running thread stalls for one of many reasons (such as, cache miss, TLB miss or pipeline stall due to data or control dependency, etc.), instead of waiting for the thread to become active again, the processor switches to one of the other threads (dispatched to it earlier by the operating system) that is ready to run (if any). Since the state of all the concurrent threads are always saved in their private set of registers, the processor can switch to one of the concurrent threads in only a few cycles.[2] In a simultaneous or fine-grain multithreaded processor, each thread is treated homogeneously, irrespective of whether the thread has encountered a long latency operation or not.

Irrespective of the multithreading approach employed, since the dispatched threads share the same Cache and TLB resources, it is important that the compiler creates the threads in a way, so that there is plenty of inter-thread data locality among the dispatched threads. This can improve the performance for all the three types of multithreaded processors.

Several techniques have been developed over the last few years to determine the independent iterations in the loops in numerically intensive computation and partition the iteration space of such loops so that the parti-

[1] TLB or Translation Lookaside Buffer is a small buffer (typically residing on the microprocessor chip, with 128 or 256 entries) which contains a subset of the page frame table. The information in the TLB is used by the processor to avoid the need to consult the page frame table for mapping most virtual addresses to real addresses. If the mapping of a virtual address that is being referred by an executing instruction is not found in the TLB, a TLB miss interrupt is generated, which causes the processor to walk through the page frame table (the exact method may be implementation dependent) and create a new entry in the TLB (if the mapping does not exist in the page frame table, interrupt is generated for a disk I/O).

[2] The number of cycles to switch threads can be as low as one or two cycles. The exact number of cycles required to make the context switch depends on the design of the microprocessor, especially the implementation of the instruction pipeline and the state of execution of an instruction when a long latency operation can be detected. New machine instructions that can interrogate the cache (and TLB) to see if an impending data reference will cause a thread switch or not and initiate a thread switch based on the result and appropriate insertion of such instructions by the compiler in the executable code (similar to inserting prefetch instructions) can reduce the cost of thread switch to zero in many cases.

tions can run more or less independently. Several loop transformation techniques have also been developed that increase the level of parallelism in such loops [11, 16,5,18]. Interprocessor communication and synchronization in a multicomputer system is expensive and optimizing program transformation, such as data alignment can be performed to reduce the cost of communication [19,6].

Multithreading can significantly reduce the communication and synchronization latency in a SMP, NUMA or a Multicomputer system (also known as, distributed memory architecture), thereby reducing the overall execution time of a scientific application on such systems. Multicomputers in which the processors are multithreaded, will be referred to in this paper as *multithreaded multicomputers* (MTMC). In an MTMC, some of the threads (or loop partitions) are assigned to the same processor. This introduces new constraints on the parallelizing compiler on how it partitions the loop nests to generate the threads, how it keeps the execution of different threads synchronized and how it helps the processor in deciding which thread to schedule next in order to reduce the congestion on the processor interconnection network. This can be better explained with the help of the following example.

Example 1. Suppose we want to execute the loop nest in Fig. 1 on an MTMC with p processors and t threads on each processor. Fig. 2 shows the t threads that are created from the loop nest in Fig. 1 for execution on processor k, where $x = (k-1)N/p$ and $y = N/(p * t)$ (assume $p * t$ divides N, where N is the range of the outermost loop). Let the execution of the innermost two loops be a *step*. Each thread evaluates a subarray of array Z of size $5 \times y$ at each step.

Due to data dependency constraints among the threads, threads need to synchronize at the end of each step. Let the running thread be t_r and let it be data dependent on thread t_d (i.e., t_r uses data element(s) computed by t_d). When t_r finishes step n, it cannot proceed with step $n+1$ unless t_d has finished step n. Since the threads run asynchronous of each other during a step's execution, either of t_r or t_d can finish step n before the other. If the running thread, t_r, is blocked, it disables itself by setting a bit in the Thread Execution Mask (TEM) on the processor (subsequently t_r will be enabled by t_d when t_d finishes its step n). When t_r is disabled or when there is a thread switching event during the execution of t_r, the processor switches to an enabled thread (thread whose corresponding bit in TEM is zero) with the highest priority, if there is any.

```
for j = 1, 5 * N
    for i = 1, N
        Z[i, j] = Z[i − 1, j − 5] + Z[i − 4, j − 6];
endfor
```

Fig. 1. Loop nest to be mapped on a multithreaded multicomputer.

In this implementation, threads T_1 and T_t are special. These are the only threads that cause interprocessor communication at the end of processing the innermost two loops. T_1 sends a packet to processor p_{k-1} and T_t receives a packet from processor p_{k+1}. Since these are the only threads on the processor that have data dependency on threads on a different processor, they should be given the highest priority. This allows the threads on the communicating processors to run as early as possible. It also reduces the buffer contention at the I/O ports of the communicating nodes. No other synchronization operations are needed for the execution of the loop nest and the loop nest is now optimized to run on the MTMC.

The example in Fig. 2 elucidates various compilation issues in an MTMC. To keep the network traffic low in a multicomputer system, it is still important to map threads to the processors in a manner so that interprocessor communication cost is low. However threads mapped to the same processor share the same cache and TLB, so data and instruction sharing among the local threads increase the utilization of these resources. The optimum number of threads to be run on a processor depends on the data dependence vectors, cache configuration and available parallelism. Optimization steps such as array replication and array privatization [3] has been shown to reduce interprocessor communication cost in a multicomputer for many scientific applications. In an MTMC with n processing nodes and with t threads per node, all the t threads can share the private copies of the arrays, thus the memory requirement on each node does not go up when these optimizations are applied (to run nt threads simultaneously on a multicomputer will require nt copies, instead of just n, as in MTMC).

Since the threads run asynchronously for the most part, it is important to have support for special thread synchronization instructions such as enable and disable threads. These synchronization instructions can be generated in a manner similar to the way send/receive primitives are generated today in a multicomputer. Some of the threads assigned to a processor may communicate only with other threads that are assigned to

```
T1:     for j1 = 1, N
                        for j = (j1 − 1) ∗ 5 + 1, j1 ∗ 5
                                for i = x + 1, x + y
                                        Z[i, j] = Z[i − 1, j − 5] + Z[i − 4, j − 6];
                                endfor
                        endfor
                        send (Z[x + 1 . . . x + y, (j1 − 1) ∗ 5 + 1 . . . j1 ∗ 5]);
                        step[1] = step[1] + 1;
                        if (step[1] > step[2]) disable (T1);
        endfor

Tl      for j1 = 1, N
for 2 ⩽ l < t           for j = (j1 − 1) ∗ 5 + 1, j1 ∗ 5
                                for i = x + 1 + (l − 1) ∗ y, x + l ∗ y
                                        Z[i, j] = Z[i − 1, j − 5] + Z[i − 4, j − 6];
                                endfor
                        endfor
                        step[l] = step[l] + 1;
                        if (step[l] > step[l + 1]) disable (Tl);
                        enable (Tl−1);
        endfor

Tt:     for j1 = 1, N
                        receive (Z[x + t ∗ y + 1 . . . x + (t − 1) ∗ y,
                                        (j1 − 2) ∗ 5 + 1 . . . (j1 − 1) ∗ 5]);
                        for j = (j1 − 1) ∗ 5 + 1, j1 ∗ 5
                                for i = x + 1 + (t − 1) ∗ y, x + t ∗ y
                                        Z[i, j] = Z[i − 1, j − 5] + Z[i − 4, j − 6];
                                endfor
                        endfor
                        step[t] = step[t] + 1;
                        enable (Tt−1);
        endfor
```

Fig. 2. Partitions of the loop nest in Fig. 1 that are mapped onto the processor $k, 1 \leqslant k \leqslant p$, in a p processor system where each processor has t threads, T_1 to T_t.

the same processor. Since they don't generate any interprocessor communication, these threads should be given lower priority compared to the threads that are on the periphery and generate interprocessor communication. By proper identification of the communication intensive threads and giving them higher priority for execution, the compiler can help in reducing the network latency and I/O buffer conflicts in an MTMC.

In the following section, we discuss why memory access latency hiding is important and various ways to hide the latency. Section 3 provides a brief overview of the loop transformation theory. In Section 4 and Section 5, we discuss methods to create threads and perform loop transformation to improve the intra-thread and inter-thread locality of reference. The methods presented improves both the spatial (Section 4.1) as well as the temporal (Section 4.2) locality. In Section 6, we discuss the results of applying the algorithm for a simple data parallel loop nest. In Section 7, we conclude the paper by discussing the benefits of processor multithreading and how compiler can help in creating threads with high data reference locality.

2. Latency hiding

Microprocessor performance is often measured in terms of the average number of processor cycles required to execute an instruction, known as cycles per instruction (CPI). Ignoring disk and the communication network, CPI has two major parts: $\text{CPI}_{\text{processor}}$, which represents the number of cycles per instruction under the assumption that there is no TLB or Cache miss and $\text{CPI}_{\text{storage}}$, which represents the aver-

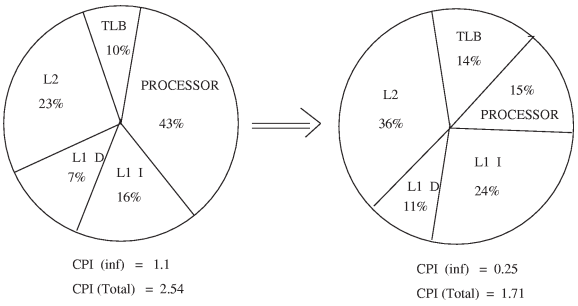

Fig. 3. Changes in the fractions of the total CPI spent performing various memory operations and processor execution as the processing speed increases without any change in the memory subsystem.

age number of cycles spent per instruction due to the processor being stalled for a cache or TLB miss to be resolved. Today's high-end superscalar processors has low $CPI_{processor}$ and it is decreasing dramatically due to many innovations in the microprocessor technology and instruction level parallel processing (ILP). As the processor cycle time decreases due to better pipelined design and innovations in the semiconductor technology, the $CPI_{storage}$ is increasing over the time, even though the absolute performance of the memory subsystem is improving modestly. Fig. 3 illustrates the effect of memory subsystem on the overall system speedup by considering a hypothetical system, in which the processor speed has increased by 4.4 times whereas the speed of the memory subsystem (with two levels of cache, split L1 and L2) remained the same. It shows that the processing speed of the system improved only by 1.33 times.

To improve the overall system speed, $CPI_{processor}$ and $CPI_{storage}$ need to be properly balanced. The performance of the memory subsystem can be improved to some extent by increasing the bus bandwidth, the number of memory ports, the number of memory banks, etc. All of these have technological limitations and cost implications. Moreover, these techniques can only increase the memory bandwidth, but the memory access latency does not decrease significantly. To reduce the effective memory latency two major techniques have been proposed in the literature: *Prefetching* and *Multithreading*, which are discussed next.

2.1. Latency hiding through prefetching

Prefetching is a method in which a cache line is fetched from a lower level to a higher level in the memory hierarchy, before a data reference is actually made by the processor to a byte in the cache line. Prefetching has been studied extensively in the literature [14, 17]. Prefetching can be implemented through hardware (usually using a simple algorithm) or software (by inserting "prefetch" instructions appropriately in the program).

Designing software prefetching algorithms with low rates of inaccurate fetches are difficult for an arbitrary scientific computation, although it is not uncommon for scientific computation to have predictable data reference patterns, especially for data parallel computation. Inaccurate prefetching of cache lines can pollute the cache and put unnecessary burden on the available bandwidth of the memory subsystem and the processor interconnection network. This can potentially increase both the memory access latency and communication latency in a multicomputer due to increased level of congestion and queueing at various stages.

For prefetching algorithms to be effective the application should have predictable data reference patterns that can be detected well ahead of the time the reference is actually made. In addition, prefetching does not necessarily mean a cache hit, because the line may get evicted due to collision in the congruence class or cache invalidation due to cache coherency protocol. Besides increasing the cache interference, prefetching can also reduce the effective memory bandwidth by prefetching lines that are never used. Furthermore, execution of the prefetch instruction can cause unnecessary ILP overhead when the data is actually available in the higher level of the memory.

To be more accurate and effective, for many scientific applications prefetch instruction needs to be executed close to the point of execution where the data is actually referred. How many cycles ahead of the reference should the corresponding prefetch instruction be executed needs to be estimated based on information such as, where in the system the current value of the data is available, latency at different memory levels and the number of cycles the processor will spend executing the intervening instructions, all of which are difficult to estimate correctly in a real system. Besides requiring the optimizing compiler to have low level knowledge of the implementation of a given architecture, this approach may not always leave enough time for the cache line to be brought in early enough so that the processor does not stall.

For many scientific computation with regular loop strides, it is possible to predict accurately and well in advance the data cache lines that will be referred in near future. However, in general the optimum prefetch distance (in terms of the processor cycle) depends on the dynamic behavior of the system [8] and is hard to predict by software or hardware.

2.2. Latency hiding through multithreading

All forms of multithreading achieves higher performance by utilizing cycles that are wasted in a single-threaded processor. The loss in the single-threaded processor can be full cycles, when the processor does not have any instruction ready to execute from a single thread, or partial cycles when the available number of instructions from a single thread is less than the number of functional units in the processor. Among the three multithreaded processors, only simultaneous multithreading can recover loss of partial cycles, whereas blocked and fine-grain multithreaded processors can recover full cycle losses that are caused due to long latency operation. Recovery of partial cycle losses are important for workload which does not have high instruction level parallelism.

However, to keep the processor busy, multithreading may require increased memory bandwidth due to the following two reasons:

(1) Since more than one threads' footprint occupies the cache for a multithreaded processor, the effective size of the cache available to each thread may become smaller, especially if there is no sharing of data or instruction among the threads. In this case, each thread will have more cache misses per instruction, causing extra bandwidth requirement from the memory subsystem.
(2) A non-multithreaded processor has at most one thread that can put memory transaction requests to the memory subsystem, whereas a t-way multithreaded processor can have at most t threads having outstanding memory requests. So multithreading potentially can require higher memory bandwidth.

To exploit the full benefit of multithreading, cache should be designed to allow multiple outstanding misses and the memory subsystem should have high bandwidth (see Section 2). Cache misses in a uniprocessor or a multiprocessor system can be classified into the following three categories[3]:

compulsory misses – the first time a cache line is referred it must be a miss,

capacity misses – cache misses because the cache is not big enough to hold the working set size of the application,

conflict misses – cache miss because of not having high enough associativity.

Multithreading hides (partially or fully) the memory access latency for cache and TLB misses (for blocked multithreading, latency hiding takes place only when there is another thread available and ready to switch to and the cost of thread switch is significantly lower than the average latency to resolve such misses). However since more than one threads' footprints share the cache, the number of conflict and capacity misses can be potentially higher with multithreading, unless the threads are created carefully with data sharing among the threads as a criterion for thread creation.

3. Loop transformation

To reap the full benefit of multithreading, the threads dispatched to a multithreaded processor should have high degree of data sharing. Compiler optimization can help by performing loop transformation that improves cache reuse by the references within a thread and by creating threads with inter-thread locality. Compiler also needs to generate the necessary synchronization primitives as discussed in Section 1.

Given a dependence matrix $D_{n\times k} = [\bar{d}_1\ \bar{d}_2\ \ldots\ \bar{d}_k]$, where $\bar{d}_i$'s are the dependence vectors, any non-singular matrix $T_{n\times n}$ is valid for transformation if and only if

$$T \cdot \bar{d}_i > 0, \quad i \in \{1, 2, \ldots, k\}. \tag{1}$$

Once a non-singular tranformation matrix T is determined that optimizes the required criterion, the process of loop tranformation involves changing the original loop variables to new loop variables in the following two steps:

(1) change the subscript expressions to reflect the new loop variables and
(2) determine bounds for the new loop variables.

Changing loop variables. The loop variables, $\bar{i}$, in the original loop nest is related to the new loop variables, $\bar{i}'$, in the transformed loop nest by the equation

$$\bar{i} = T^{-1} \cdot \bar{i}'.$$

If an m-dimensional variable is referred by the expression $A \cdot \bar{i} + \bar{c}$ in the original program (where A is a two-dimensional matrix of size $m \times n$), then the same array reference is made by the expression $A \cdot T^{-1} \cdot \bar{i}' + \bar{c}$ in the tranformed program.

[3] Another cache miss category, called *coherence misses* (arises when a private cache line on the local processor is invalidated by a remote processor), is only relevant for multiprocessor systems with hardware cache coherency protocol, often used in shared memory systems.

Determining new loop bounds. To determine the bounds of the new loop variables, *Fourier–Motzkin* elimination process can be used [24]. Example 2 explains the method. The time complexity of the Fourier–Motzkin method is exponential [20], however the depths of the loop nests in most applications are small and the method has been found to be useful [22].

Example 2. Let us consider a doubly nested loop, with loop variables i and j and loop bounds $L_1 \leqslant i \leqslant U_1$ and $L_2 \leqslant j \leqslant U_2$. Assume that the original loop nest is transformed using the transformation matrix, T, where

$$T = \begin{bmatrix} 1 & -3 \\ 0 & 1 \end{bmatrix}, \qquad T^{-1} = \begin{bmatrix} 1 & 3 \\ 0 & 1 \end{bmatrix}.$$

Since

$$\begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} k \\ l \end{bmatrix}$$

we have

$$L_1 \leqslant k + 3l \leqslant U_1, \qquad L_2 \leqslant l \leqslant U_2,$$

that is

$$l \geqslant \frac{L_1 - k}{3},\ l \leqslant U_2 \ \Rightarrow\ \frac{L_1 - k}{3} \leqslant U_2,$$
$$l \leqslant \frac{U_1 - k}{3},\ l \geqslant L_2 \ \Rightarrow\ L_2 \leqslant \frac{U_1 - k}{3}.$$

So the loop bounds in the transformed loop nest can be written as

$$L_1 - 3U_2 \leqslant k \leqslant U_1 - 3L_2,$$

$$\max\left(L_2, \frac{L_1 - k}{3}\right) \leqslant l \leqslant \min\left(U_2, \frac{U_1 - k}{3}\right).$$

4. Improving locality of reference

In this section, we discuss methods of determining the transformation matrix T, that increases the locality of reference of a thread and also helps in creating threads with inter-thread data sharing. Our objective is to determine transformation matrix, T, such that once a cache line is fetched, it is reused as many times as possible before it is evicted from the cache due to capacity misses.

There are two different transformations that can increase the locality of reference by increasing the reuse of a fetched cache line:

- Improving spatial locality: transform the innermost loops such that successive iterations use the same data cache line (for the same reference to the same array).
- Improving temporal locality: transform the outermost loops such that the temporal distance between iterations that reuse the same memory locations (by a different reference to the same array) is reduced.

The objectives are different for these two loop transformations and for deeply nested loops, they may work on different sets of loops. The spatial locality transformation works on the innermost loop(s) and attempts to generate cache hit on successive iterations, whereas the temporal locality transformation starts with the outermost loop and attempts to increase the locality among data references made among distant iterations. Some of the early work in this area was done by [23]. In [23], Wolf et al. provides a mathematical formulation of reuse and data locality in terms of four different types of data reuse: self-temporal, self-spatial, group-temporal and group-spatial. They also explains how well-known loop transformation via interchange, reversal, skewing and tiling can improve the data locality. However, they do not provide any general algorithm in the paper for such loop transformation.

4.1. Spatial locality

Spatial locality can be improved by transforming the innermost loops in a way such that the likelyhood of the next iteration to make reference to the same cache line is increased. The exact tranformation depends on the layout of the arrays in the memory. For an array B of size $N \times M$, the reference $B[i, j]$ maps to the memory location

$$B[i, j] \rightarrow \text{offset } + i + (j - 1) * N$$

in a column major layout and to

$$X[i, j] \rightarrow \text{offset } + (i - 1) * M + j$$

in a row major layout. For our analysis we assume row major layout. Similar results can be obtained for column major layout.

The distance between the memory locations that a given reference to an array refers to in two successive iterations is called *spatial reuse distance*. *Spatial reuse fraction* is the conditional probability that a given array reference will refer the same cache line in the $(i+1)$-th iteration assuming that in the i-th iteration the reference is made to any byte within the cache line at random.

For more general references, $X[A.\bar{I}+\bar{c}]$, with unit loop strides the spatial reuse distance can be obtained as

$$\begin{aligned}&\left[A\cdot\bar{I}+\bar{c}\right]-\left[A\cdot(\bar{I}+\bar{e}_n)+\bar{c}\right]\\&=A\cdot\bar{e}_n=\bar{A}_n=\begin{bmatrix}a_{1n}\\ \ldots\\ a_{mn}\end{bmatrix}\end{aligned}$$

where $\bar{A}_n$ is the n-th column of A. If the array X is of size $M_1\times M_2\cdots\times M_n$ and the stride of the innermost loop is Δ_n, then we define the *stride vector* as

$$\text{Stride vector}=\bar{u}=\begin{bmatrix}u_1\\ \ldots\\ u_{n-1}\\ u_n\end{bmatrix}=\begin{bmatrix}M_2M_3\ldots M_n\\ \ldots\\ M_n\\ \Delta_n\end{bmatrix}.$$

The k-th element of the stride vector denotes the number of array elements a reference to X is advanced by in the next iteration if a_{kn} is 1.

For each reference $X[A\cdot\bar{I}+\bar{c}]$, the same cache line is reused in next iteration only if

$$\bar{A}_n^T\cdot\bar{u}<\text{cache line size}.$$

Example 3. In two successive iterations, $[i,j]$ and $[i,j+1]$, the reference $X[i+2j+2,i+j+3]$ in Fig. 4 refers to the memory locations (assuming array X of size $M\times N$, and row major mapping of the array)

$$\begin{aligned}&X[i+2j,2i+3j+2]\\&\rightarrow(i+2j)*N+(2i+3j+2),\\&X\left[i+2(j+1),2i+3(j+1)+2\right]\\&\rightarrow(i+2(j+1))*N+(2i+3(j+1)+2).\end{aligned}$$

So the spatial reuse distance between successive iterations is $2N+1$. In matrix notation, the spatial reuse distance can be expressed as

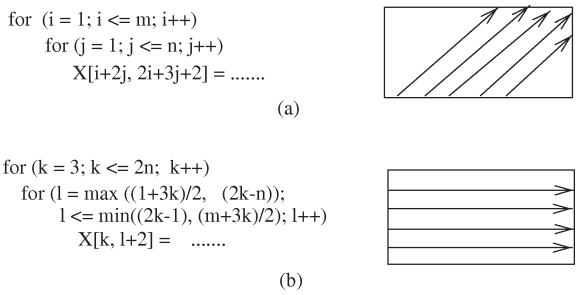

Fig. 4. Spatial locality transformation. The program in (a) is transformed into the program in (b), with the corresponding change in the access pattern of the successive array elements, which reduces cache misses when the array is mapped to the memory in row-major order.

$$\begin{aligned}\bar{e}_2^T\cdot[A]^T\cdot\bar{u}&=\bar{e}_2^T\cdot\begin{bmatrix}1&2\\2&3\end{bmatrix}^T\cdot\begin{bmatrix}N\\1\end{bmatrix}\\&=[\,2\quad 3\,]\cdot\begin{bmatrix}N\\1\end{bmatrix}=2N+3.\end{aligned}$$

If the cache line size is 128 bytes, the spatial reuse fraction is only 20.3% for $N=50$ and 0 for $N\geqslant 63$.

$$T=\begin{bmatrix}1&2\\2&3\end{bmatrix}\quad\text{and}\quad T^{-1}=\begin{bmatrix}-3&2\\2&-1\end{bmatrix}.\quad(2)$$

Applying the transformation in Eq. (2), reference $X[i+2j,2i+3j+2]$ changes to $X[k,l+2]$, as shown in Fig. 4 and the resulting spatial reuse distance is 1. If the cache line size is 128 bytes, the spatial reuse fraction is $127/128\approx 99\%$.

4.1.1. Spatial locality transformation

To determine the spatial reuse transformation, we first determine all the references to the array elements in the loop nest and prioritize them. If profile directed feedback is not available to prioritize the references, heuristics such as [2] can be used, which has been shown to provide branch predictions reasonably well for SPEC benchmarks.

For each reference matrix, A_i, we want to determine a non-zero column vector $\bar{t}'_n$ so that

$$A_i\cdot\bar{t}'_n=[0\;0\;\ldots\;\gamma]^T$$

where γ is such that $\Delta_n|\gamma|<$ cache line size. To increase the likelyhood of spatial reuse, we choose γ as small as possible.

We define the *spatial reuse condition* for reference A_i as

$$[A_i]_{row\;k}\cdot\bar{t}'_n=0\quad\text{for } 1\leqslant k\leqslant n-1.$$

Input: Ordered set of reference matrices $S = \{A_1, A_2, ...\}$.
Priority order $P = \{p_1, p_2...\}$.
Set of dependence vectors $D = \{\bar{d}_1, \bar{d}_2, ...\}$.
Set of stride vectors $\{\bar{u}_1, \bar{u}_2....\}$.
Output: Transformation matrix M for Spatial reuse.
$M =$ Identity matrix;
$d = n$; /* loop depth */
while ($S \neq$ *NULL*) {
Find largest i for which spatial reuse condition holds.
/* determines the last column of $T^{-1} = [t_{n1}...t_{nd}]^T$ */
Delete A_j from S for $j > i$. If ($i == 0$) terminate.
$k =$ *rank*$[A_1\ A_2\ ...\ A_i]^T$
If ($k < d$) express $t_1, ...t_k$ in terms of $t_{k+1}, ..., t_d$
Find smallest $|\gamma_j|$ in the priority order of A_j for $1 \leqslant j \leqslant i$
If ($(u_d)_j * |\gamma_j| >$ *line_size*) delete A_j from S.
Expand last column of T^{-1} to n.
Complement T^{-1} to full rank and find T.
If $T \cdot \bar{d}_i > 0$ for all $\bar{d}_i \in D$ then $M = T \cdot M$
else delete A_i from S and continue with next iteration
Compute spatial reuse fractions, f_j for $1 \leqslant j \leqslant i$
Order A_j in increasing order of $p_j = p_j * f_j$ for $1 \leqslant j \leqslant i$
Order entries in S in the priority order.
$d = d - 1$
}
return (M)

Fig. 5. Spatial transformation algorithm.

The spatial reuse fraction for A_i is

$$\text{Spatial reuse fraction} = \frac{\text{cache line size} - u_n * |\gamma|}{\text{cache line size}}.$$

Fig. 5 shows the algorithm to determine the spatial transformation matrix. The algorithm first tries to determine the best transformation for the innermost loop and then successively goes to the outermost loop until there is no reference matrix available which satisfies the spatial reuse condition, or the spatial reuse fraction is zero for all the reference matrices. At each iteration, the algorithm determines the last column of the transformation matrix, T, which is then expanded to the full rank.

The values of the smallest $|\gamma_j|$ can be determined using the euclidean algorithm to find GCD. For example, let $n = 5$ and after applying the spatial reuse conditions we can express t_1 and t_2 in terms of t_3, t_4 and t_5. Assume that there are two references and that

$$\gamma_1 = 5t_3 + 10t_4 + 15t_5 \tag{3}$$

and

$$\gamma_2 = 9t_3 + 3t_4. \tag{4}$$

Since γ_1 and γ_2 must be a multiple of 5 and 3 respectively, for the smallest values of γ_1 and γ_2 we must have

$$t_3 + 2t_4 + 3t_5 = 1 \tag{5}$$

and

$$3t_3 + t_4 = 1. \tag{6}$$

From Eqs. (5) and (6), we can write

$$-5t_3 + 3t_5 = -1. \tag{7}$$

Since the GCD of the coefficients at the left hand side divides the constant in the right hand side, there is a solution to Eq. (7), which can be obtained from the euclidean algorithm. The final solution for $[t_3\ t_4\ t_5]$ that minimizes γ_1 and γ_2 is $[2 - 5\ 4]$.

4.2. Temporal locality

So far we have discussed about transformation that will increase the reuse of the same cache line in the

```
for (i = 6; i <= m; i++)
    for (j = 2; j <= n; j++)
       B[i,j] = B[i-2, j-3]+B[i-3, j-5];
```

Fig. 6. Simple program with three temporal reuses.

next iterations. To increase the temporal locality, we look for a tranformation that decreases the distance between the iterations that make reference to the same memory location. Thus this transformation has the effect of reducing the footprint of a loop nest in the data cache.

Let us consider the program in Fig. 6. Variable $A[i,j]$ is referenced in three different iterations $[i,j]$, $[i+2, j+3]$ and $[i+3, j+5]$ by the three references $A[i,j]$, $A[i-2, j-3]$ and $A[i-3, j-5]$ respectively.

The distance between the two iterations in which reference to the same memory location is made is called *reuse vector*. For the example in Fig. 6, there are three reuse vectors which forms the reuse matrix

$$R = \begin{bmatrix} 2 & 3 & 1 \\ 3 & 5 & 2 \end{bmatrix}.$$

For a more general array reference, let us consider two references to array X, $X[A \cdot \bar{i}_1 + \bar{c}_1]$ and $X[A \cdot \bar{i}_2 + \bar{c}_2]$, where $A = [\bar{\alpha}_1\ \bar{\alpha}_2\ ...]^T$. Iterations $\bar{i}_1$ and $\bar{i}_2$ refer to the same memory location when

$$A \cdot \bar{i}_1 + \bar{c}_1 = A \cdot \bar{i}_2 + \bar{c}_2.$$

The *temporal reuse distance* vectors between the two iterations, $\bar{r} = \bar{i}_1 - \bar{i}_2$, can be found by solving

$$A \cdot [\bar{i}_1 - \bar{i}_2] = A \cdot \bar{r} = \bar{c}_2 - \bar{c}_1 \tag{8}$$

that is,

$$\bar{r} = \text{null space}(A) + \bar{s} = \text{span}\{\bar{a}_1, \bar{a}_2, \ldots \bar{a}_k\} + \bar{s} \tag{9}$$

where $k = n - \textit{rank}(A)$, $\{\bar{a}_1, \bar{a}_2, ...\bar{a}_k\}$ is the set of basis vectors of the null space of A and $\bar{s} = [s_1\ s_2...]^T$, a special solution of Eq. (8).

To increase the temporal reuse, the transformation matrix should be chosen in a way so that the higher dimensional elements of the vector $T \cdot \bar{r}$ are zero or very small. To reduce the number of reuse vectors, we choose the maximum of all the reuse distances in Eq. (9)

$$\bar{r} = [r_1\ r_2 \ldots r_n]^T$$

where

$$r_l = \begin{cases} s_l & \text{if } a_{jl} = 0 \text{ for all } j,\ 1 \leqslant j \leqslant k, \\ \text{range of } [\bar{\alpha}_l.\bar{i} + \bar{c}_1 - \bar{c}_2] & \\ \quad \text{for } i \in \text{Iteration Space}, & \text{otherwise.} \end{cases}$$

For more general array references, let us consider the references $X[A \cdot \bar{i} + \bar{c}_1]$ and $X[B \cdot \bar{i} + \bar{c}_2]$, where $A = [\bar{\alpha}_1\ \bar{\alpha}_2\ ...]^T$ and $B = [\bar{\beta}_1\ \bar{\beta}_2\ ...]^T$. Let $\{\bar{a}_1, \bar{a}_2, ...\bar{a}_k\}$ and $\{\bar{b}_1, \bar{b}_2, ...\bar{b}_k\}$ be the sets of basis vectors of the null space of A and B, respectively. Same memory location will be referred to in iterations $\bar{i}_1$ and $\bar{i}_2$ by these two references if and only if

$$A \cdot \bar{i}_1 + \bar{c}_1 = B \cdot \bar{i}_2 + \bar{c}_2 = \bar{x} \text{ (say)}.$$

We can write

$$\bar{i}_1 = \textit{span}\{\bar{a}_1, \bar{a}_2, \ldots \bar{a}_k\} + \bar{s}_1(\bar{x} - \bar{c}_1),$$

$$\bar{i}_2 = \textit{span}\{\bar{b}_1, \bar{b}_2, \ldots \bar{b}_k\} + \bar{s}_2(\bar{x} - \bar{c}_2).$$

So we have

$$\bar{r} = \textit{span}\{\bar{a}_1, \bar{a}_2, \ldots \bar{a}_k, \bar{b}_1, \bar{b}_2, \ldots \bar{b}_k\} + \bar{s}_2(\bar{x} - \bar{c}_2) - \bar{s}_1(\bar{x} - \bar{c}_1).$$

Again to keep the number of reuse vectors small and still capture the maximum reuse distance, we choose

$$\bar{r} = [r_1\ r_2 \ldots r_n]^T$$

where

$$r_k = \begin{cases} c_{1k} - c_{2k}, & \text{when } \bar{a}_i = \bar{b}_i, \\ \text{range of } [\bar{\alpha}_k.\bar{i}_1 - \bar{\beta}_k.\bar{i}_2 + \bar{c}_1 - \bar{c}_2] & \\ \quad \text{for } i_1, i_2 \in \text{Iteration Space}, & \text{otherwise.} \end{cases}$$

4.2.1. Temporal reuse transformation

To improve the temporal data locality for a loop nest, we construct the transformation matrix T, by adding rows $\bar{t}_i$ successively for $1 \leqslant i \leqslant n$ to it. The new rows should be determined following the three criteria:

(1) $\bar{t}_i$ is linearly independent of rows already in T.

Input: Reuse matrix $R_{n\times m} = [\bar{r}_1\ \bar{r}_2 \ldots]$ in priority order $P = \{p_1, p_2 \ldots\}$.
Set of dependence vectors $D = \{\bar{d}_1, \bar{d}_2, \ldots\}$.
GCD of the elements of each vector $\bar{r}_i$: $\bar{g} = [g_1\ g_2\ \ldots]$.
Output: Transformation matrix T for Temporal reuse.

$q = 1$
while ($R \neq$ *empty*$||q \neq n$) {
if ($r_{qj} == 0$) for all $1 \leqslant j \leqslant n$ continue with next iteration.
Construct m linear equations
$$\bar{t}^T \cdot \bar{r}_j = x_j g_j \quad \text{for } 1 \leqslant j \leqslant m$$
If ($q \neq 0$) do singular value decomposition of
$$T = W \Sigma V^T$$

Replace t_j by $y_q v_{qj} + y_{q+1} v_{(q+1)j} + \cdots y_n v_{nj}$ to obtain
$[f_1(\bar{y})\ \ldots\ f_n(\bar{y})] \cdot \bar{r}_j = x_j g_j \quad \text{for } 1 \leqslant j \leqslant m$
Solve for $\bar{y}$ in terms of $\bar{x}$. Find minimum $|x_i|$'s in lexicographical order so that
$\bar{t}^T \cdot \bar{d}_i = [f_1(\bar{y}) \ldots f_n(\bar{y})] \cdot \bar{d}_i = [h_1(\bar{x}) \ldots h_n(\bar{x})] \cdot \bar{d}_i > \bar{0}$ for all $\bar{d}_i \in D$

If no such $\bar{x}$ or $\max_j(|x_j g_j|)$ is too large, tile or generate threads (Section 5.2).
Delete $\bar{d}_i$'s for which $\bar{t}^T \cdot \bar{d}_i > 0$.
Adjust m.
$q = q + 1$
}
If ($q \neq n$) complement T to full rank.

Fig. 7. Temporal locality transformation algorithm.

(2) $\bar{t}_i \cdot \bar{d}_i > 0$ for all $\bar{d}_i \in D$.
(3) Minimize

$$\left| \bar{t}_i \cdot \begin{bmatrix} r_{i1} * u_{i1} \\ r_{i2} * u_{i2} \\ .... \end{bmatrix} \right|. \tag{10}$$

Condition (1) guarantees that the transformation is non-singular. Condition (2) guarantees that the transformation is a legal transformation and condition (3) finds the best transformation for temporal locality. However it is not possible to evaluate condition (3), because we don't know the unit trip vector $\bar{u}$ that results after the loop nest have been transformed, until the transformation matrix T is fully known. The best we can do is try to minimize $|\bar{t}_i \cdot \bar{r}|$. So condition (3) is modified to

(3′.) Minimize $|\bar{t} \cdot \bar{r}|$.

The temporal locality transformation algorithm is shown in Fig. 7. At the q-th iteration the algorithm attempts to add the q-th row, $\bar{t}^T$, to the partially built tranformation matrix $T_{(q-1)\times n}$, such that the new row is linearly independent with the existing rows of T and the new temporal reuse distance along the q-th dimension, $\bar{t}^T \cdot \bar{r}_j$, is minimized. Since g_j is the GCD of the elements of $\bar{r}_j$, we have

$$\bar{t}^T \cdot \bar{r}_j = x_j g_j \quad \text{for } 1 \leqslant j \leqslant m. \tag{11}$$

To guarantee that $\bar{t}^T$ is linearly independent with the existing rows of T, the matrix T can be transformed into the row echeleon form to determine the unit row vectors that span the set of vectors linearly independent[4] of the row vectors of T. If the rank of T is $q-1$, and the unit row vectors that span the space that is

[4] Alternatively, singular value decomposition theorem [7] can be used which decomposes matrix $T_{m\times n}$ as $W \Sigma V^T$, where $W_{m\times m}$ and $V_{n\times n}$ are orthogonal matrices and $\Sigma_{m\times n}$ is a diagonal matrix. However, this method may not always work, because V may have elements which are real numbers but not rational numbers. If the rank of T is $q-1$, it can be shown that the last $(n-q+1)$ column vectors of V, $\{\bar{v}_q, \bar{v}_{q+1}, \ldots, \bar{v}_n\}$ spans the null space of T. Since an element, $y_q \bar{v}_q + y_{q+1} \bar{v}_{(q+1)} + \cdots + y_n \bar{v}_n$, in the null space of the column vectors of T also belongs to the space orthogonally complement to the row vectors of T (hence linearly independent of the rows of T), $\bar{t}$ can be written as

$$\bar{t} = y_q \bar{v}_q + y_{q+1} \bar{v}_{(q+1)} + \cdots + y_n \bar{v}_n.$$

orthogonally complement to T are $\{\bar{v}_q, \bar{v}_{q+1}, \ldots, \bar{v}_n\}$, then $\bar{t}$ can be written as

$$\bar{t} = y_q \bar{v}_q + y_{q+1} \bar{v}_{(q+1)} + \cdots + y_n \bar{v}_n. \quad (12)$$

Using Eq. (12) to replace $\bar{t}$ in Eq. (11), we get

$$\left[f_1(\bar{y}) \ldots f_n(\bar{y})\right] \cdot \bar{r}_j = x_j g_j \quad \text{for } 1 \leqslant j \leqslant m. \quad (13)$$

Eqs. (13) is a system of linear equations with $(n - q + 1)$ unknown variables y_j's and m equations. If it has one or more solutions, we can express y_j, $q \leqslant j \leqslant n$ in terms of x_j, $1 \leqslant j \leqslant m$. Using these values of y_j's in Eq. (12), we can search for the minimum values (in the lexicographical order) of $|\bar{x}|$ within the valid transformation space

$$\begin{aligned} \bar{t}^T \cdot D &= \left[f_1(\bar{y}) \ldots f_n(\bar{y})\right] \cdot D \\ &= \left[h_1(\bar{x}) \ldots h_n(\bar{x})\right] \cdot D > \bar{0}. \end{aligned} \quad (14)$$

This can be accomplished by successively solving the set of linear inequalities

$$\left[h_1(\bar{x}) \ldots h_n(\bar{x})\right] \cdot D \geqslant e_i$$

for unit vectors $\bar{e}_i$, $i = \{n, n-1, \ldots, 1\}$, by Fourier–Motzkin elimination process and determining the minimum $x_j, 1 \leqslant j \leqslant m$, in the priority order for each set.

5. Thread creation

The loop nest generated after applying the methods described in the previous sections on the original loop nest of a scientific computation reduces the nuber of cache misses during its execution. As described in Section 2.2, to further reduce the processor stall time due to the cache misses as well as other long latency operations, we need to generate multiple threads from the modified loop nests, so that all the threads can run concurrently on a multithreaded processor. The following two questions are relevant in generating multiple threads to run on the same processor which are discussed subsequently.

- If the architectural characteristics and program behavior (such as, average cache miss latencies, average cache miss rates for the given cache configuration) are known at compile time, what should be the appropriate number of threads that should be generated.
- What factors need to be considered in order to create the threads by partitioning the loop nests.

5.1. Determining appropriate number of threads

The appropriate number of threads needed for optimal performance of a loop nest on a multithreaded processor depends on factors such as the rate at which thread switch events (for example, cache misses, TLB misses in a cache coherent shared memory system, send/receive or get/put events in a distributed memory machine) are generated and the average latency for such events to resolve, which in turn depends on the design of the memory subsystem, the multiprocessor architecture. If estimates for these parameters are known at compile time, the analysis presented in this section can be used to determine the appropriate number of threads to run on the processor.

The multithreaded processor can be seen as a queueing system with two servers in sequence. The first server generates the event on which thread switch takes place and the second server resolves the event. We will describe the queueing system for the case when thread switching takes place on L1 cache misses. The method can be easily generalized to other thread switching events.

For thread switching on L1 cache misses, the first server should consist of the processor, the L1 cache and the thread switching network and the second server should consist of the memory subsystem below the L1 cache. Only one thread can be in the first server (being executed by the processor or in the process of being thread switched), whereas multiple threads may be in the second server (for example, multiple outstanding cache misses being serviced by the memory subsystem). At the end of being server at the first server (that is, when the running thread gets a L1 cache miss and have spent C number of cycles to switch thread), the thread must go to the second server to have its cache miss serviced. The first server can be seen to consist of two substations: the processor and the switching network. Processor utilization is the utilization of the first substation in the first server.

Let there be p threads in the system at all time which circulates around the two queueing facilities. Let the system be in state P_i, when there are i threads in the first server (that is, the total number of threads ready to run on the processor plus the thread being run) and $p - i$ threads in the second server. We assume that both the servers has an exponential service time. If t is the average number of cycles between misses and C is the number of cycles to switch thread, then the average rate at which the processor services a thread can be written as

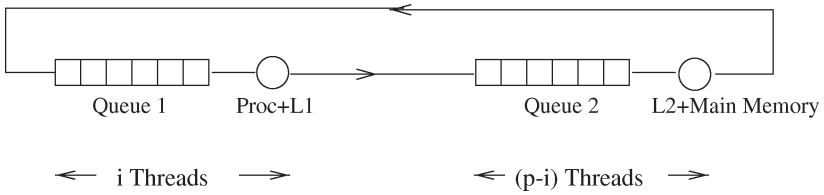

Fig. 8. Queueing model to determine processor utilization when thread switch takes place on L1 cache misses. In state i, there are i threads in the first server and $(p - i)$ threads in the second server.

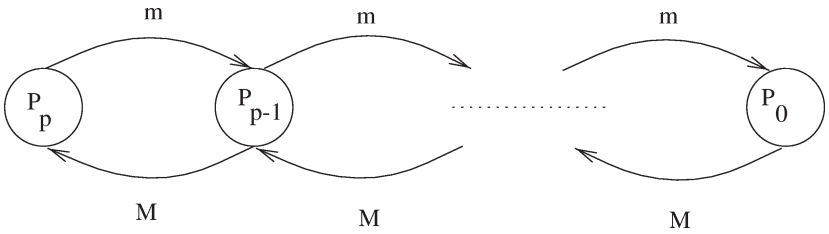

Fig. 9. State-transition-rate diagram for the queuing network shown in Fig. 10.

$$m = 1/(t + C).$$

If the average L1 miss latency is T, then the average rate at which L2 misses are serviced can be written as[5] $M = 1/T$.

Fig. 11 shows the state-transition-rate diagram for the system [9]. From Fig. 11, we can easily write down the equilibrium equations as,

$$P_p m = P_{p-1} M, \tag{15}$$

$$P_p m + P_{p-2} M = P_{p-1} M + P_{p-1} m, \tag{16}$$

... and for state i,

$$P_{p-i+1} m + P_{p-i-1} M = P_{p-i}(M + m). \tag{17}$$

From (15), we have

$$P_{p-1} = P_p \frac{m}{M}. \tag{18}$$

From (18) and (16), we have

$$P_{p-2} = P_p \left(\frac{m}{M}\right)^2. \tag{19}$$

[5]Both the average number of cycles between the thread switching events, t, and the time to resolve such events, T may depend on the number of threads, p, on the processor and other system parameters such as network congestion, etc. For simplicity of analysis, we have ignored such dependencies.

Similarly, using (19), (18) and (17) recursively, we obtain the probablity for state $p - i$ as

$$P_{p-i} = P_p \left(\frac{m}{M}\right)^i. \tag{20}$$

Since the sum of the probablities is 1, we have

$$1 = \sum_{i=0}^{p} P_i = P_p \left[\sum_{i=0}^{p} \left(\frac{m}{M}\right)^i\right].$$

That is,

$$P_p = \frac{1}{\sum_{i=0}^{p} \eta^i}$$

where $\eta = m/M$. Utilization of the first server is given by

$$U' = 1 - P_0 = 1 - P_p \eta^p = 1 - \frac{\eta^p}{\sum_{i=0}^{p} \eta^i}.$$

However, we are only interested in the utilization of the first substation in the first server (that is, we want to exclude the thread switch time from the utilization). This can be obtained by multiplying U' with $t/(t + C)$ to obtain

$$U(p) = \left[1 - \frac{\eta^p}{\sum_{i=0}^{p} \eta^i}\right] * \frac{t}{t + C} \tag{21}$$

where C is the thread switch time (in number of processor cycles) and

$$\eta = \frac{T}{t + C}.$$

Eq. (21) can be used to determine the appropriate number of threads that should run on a processor. For example, if cache misses occur every 40 cycles (on an average) and it takes 20 cycles on an average to resolve a cache miss, then $\eta = 0.5$ and the processor utilizations are 67%, 85%, 93%, 97% and 98.5% respectively with 1,2,3,4 and 5 threads on the processor. So the system performance can improve by almost 50% when the number of concurrent threads on the processor increases from 1 to 5.

5.2. Creating the threads

Loop transformation to increase the temporal locality depends on the solution of Eqs. (13) and (14). If there is no solution to these equations while processing the q-th loop in the loop nest or if the value of $\max_j(|x_j g_j|)$ is too large compared to the cache size of the system, then temporal locality can not be exploited (that is, the entire cache is flushed out before reference to the same memory location is repeated) for the q-th loop.

However performance loss due to cache misses for the lack of temporal locality on the q-th loop can be reduced by choosing one of the following alternatives.

(1) If the underlying architecture is a distributed memory machine and the loop nest being processed has not being distributed over the processors, then we can tile the loop and distribute the tiles over the processors. Since inter-processor communication is expensive in such systems, we should try to keep the distance between communications steps long by trying to maximize $|\bar{t}_i \cdot \bar{r}|$'s (in priority order).

(2) If alternative (1) is not applicable and alternative (2) has not been used before for the loop nest being processed, then the q-th loop should be partitioned to generate multiple threads. The threads can be generated by strip mining the loop with step size equal to the appropriate number of threads. The threads thus created will have significant data sharing. This alternative is second in priority among the four alternatives so that the synchronization among the threads are less frequent.

(3) If alternative (1) or alternative (2) are not applicable, then to increase the temporal locality, the q-th loop should be tiled with tile size between the minimum reuse distance and the maximum reuse distance on the q-th dimension.

(4) Since the reuse vectors are prioritized according to their likelihood to be used in a data reference, the algorithm in Fig. 7 can still proceed by discarding the last reuse vector and recompute the q-th step of the algorithm. Thus we loose the temporal reuse for the low priority reference, but still may be able to perform temporal locality transformation which involves higher priority references.

Alternative (2), when used, generates the new threads. If during temporal locality transformation, alternative (2) is not used, then the following guideline can be used to select one of the outermost loops for partitioning and generating the threads.

If the data dependence matrix of the loop nest has a row all of whose elements are zero, then the corresponding loop should be partitioned to create the threads, because the loop carries no dependence and so no synchronization code needs to be generated. If all the loop carries dependences, then synchronization primitives, such as *enable thread* and *disable thread* needs to be generated for each of the generated threads to have correct data-flow among the threads (see Fig. 2). To reduce the number of such primitives, the loop to be partitioned should be selected so that the maximum amount of computation can be performed between synchronization need. For the example in Fig. 2, if threads are created from the loop on j, synchronization is required after every (5*5N)/(p*t) iterations, where 5 is the minimum of the distance vectors along the j-axis, $5N$ is the loop range for j, p and t are the number of processors in the multicomputer and the number of threads per processor. Similarly, if threads are created from the loop on i, synchronization is required after every (1*N)/(p*t) iterations. So loop on j is chosen in Fig. 2 to generate the threads. The selection of the loop for generating the threads can be generalized for more general type of array references.

6. Experimental results

The data locality optimization is applied to a few simple loop nests which can be transformed by hand. Although these loop nests are simple, they show little

Table 1

Miss Rates (per 100 references) and execution times for two different memory latency and switch times, when the cache size is 128 KBytes and associativity is 4

Line	Non-Th Orig			Non-Th Tran			Thr Orig			Thr Trans		
Size (B)	Miss Rate	Exec (A)	Exec (B)	Miss Rate	Exec (A)	Exec (B)	Miss Rate	Exec (A)	Exec (B)	Miss Rate	Exec (A)	Exec (B)
32	25.00	148.5	47.25	8.15	57.51	24.50	25.01	28.35	20.59	15.56	22.74	17.91
64	12.50	81.0	30.37	4.08	35.53	19.01	12.51	20.93	17.05	7.41	17.90	15.60
128	6.26	47.30	21.95	2.04	24.52	16.25	6.27	17.22	15.28	3.24	15.42	14.42
256	3.12	30.34	17.71	1.02	19.00	14.88	3.14	15.36	14.39	1.21	14.21	13.84
512	1.56	21.92	15.60	0.51	16.25	14.19	1.58	14.44	13.95	0.52	13.81	13.65

Table 2

Miss Rates (per 100 references) and execution times for two different memory latency and switch times, when the cache associativity is 4, cache line size is 128 Bytes

Cache	Non-Th Orig			Non-Th Tran			Thr Orig			Thr Trans		
Size (KB)	Miss Rate	Exec (A)	Exec (B)	Miss Rate	Exec (A)	Exec (B)	Miss Rate	Exec (A)	Exec (B)	Miss Rate	Exec (A)	Exec (B)
32	6.25	47.24	21.94	2.04	24.52	16.25	6.27	17.22	15.28	3.24	15.42	14.42
64	6.25	47.24	21.94	2.04	24.52	16.25	6.27	17.22	15.28	3.24	15.42	14.42
128	6.25	47.24	21.94	2.04	24.52	16.25	6.27	17.22	15.28	3.24	15.42	14.42
256	6.25	47.24	21.94	2.04	24.52	16.25	4.35	16.08	14.73	3.24	15.42	14.42

cache reuse for typical cache sizes and demand some of the highest memory bandwidth (that is, bytes per floating point operations) that is needed for any scientific computation. Due to such high demands on the memory subsystem, these loop nests typically spend significant portion of its execution time just waiting for the data to arrive (even though prefetching can reduce the latency, it does not reduce the memory bandwidth required, instead prefetching often increases the memory bandwidth required). In this section, we present in detail the results from one such simple loop nest (the results are similar for others).

We have implemented a multithreaded cache simulator which traps all the data references made by a program at runtime and can simulate multithreading on cache misses. To perform the experiment, four separate sets of cache misses and total execution times are collected: for the original program, for the transformed program, original program with multiple threads and transformed program with multiple threads. As shown in the tables below, the execution time is obtained for two different system parameters: (A) average cache miss latency $=$ 80 cycles and thread switch time $=$ 8 cycles and (B) average cache miss latency $=$ 20 cycles and the thread switch time $=$ 4 cycles. The various system parameters used are, cache associativity $=$ 4, and 4 threads. In the first table, line size is kept fixed at 128 lines and the cache size is varied from 32 KB to 256 KB. In the second table the cache size is kept fixed at 128 KB and the line size is varied from 32 Bytes to 512 Bytes. The following tables show that going from original loop nest to the transformed loop nest (both uni-threaded), the cache miss rates decreases almost three folds. This can reduce the total execution time and the stress on the memory hierarchy (which is very important to improve MP efficiency in a symmetric multiprocessor system) quite significantly. When the original and the transformed loop nests are multithreaded, the transformation improves the misses about two-folds. However, in this case most of the cache misses are hidden due to multithreading so the total execution time reduces drastically (the actual values primarily depend on the average L1 miss latency and the thread switch time, both of which varies significantly from machine to machine).

6.1. Loop kernel

Original loop kernel

```
for (i = L1; i <= U1; i++)
  for (j = L2; j <= U2; j++)
    A[i,j] = A[i-3, j-1]+A[i-6, j-2];
```

The reuse matrix R, transformation matrix T and the transformed reuse matrix R' are

```
for (k = L1-3*U2+t; k <= U1-3*L2; k+T)
  for ((l = max(L2,(L1-k)/3); l <= min((U1-k)/3,U2); l++)
     A[k+3*l, l] = A[k+3*l-3, l-1] + A[k+3*l-6, l-2];
```

Program Code 1.

$$T = \begin{bmatrix} 1 & -3 \\ 0 & 1 \end{bmatrix}, \qquad R = \begin{bmatrix} 3 & 6 & 3 \\ 1 & 2 & 1 \end{bmatrix},$$

$$R' = T \cdot R = \begin{bmatrix} 1 & -3 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 3 & 6 & 3 \\ 1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}.$$

Thread t, $0 \leqslant t \leqslant T-1$, in the transformed program with T threads Transformed program with multithreading ($0 \leqslant t \leqslant T-1$), see Program Code 1.

7. Conclusion

Lost processor cycles due to cache misses increase a program's execution time significantly. Multithreaded processors has been proposed in the literature to reduce this performance loss. Multithreading increases the processor utilization, but it may also increase the cache miss rate, because multiple threads simultaneously share the cache which effectively reduces the cache size available to each individual thread. Increased processor utilization and the increase in the cache miss rate in a multithreaded processor system demands higher memory bandwidth. In this paper, we have discussed a novel compiler optimization method based on loop transformation theory to reduce the data cache misses for a uniprocessor by increasing the temporal and spatial locality of reference. The algorithms presented work for general array references. The method helps in creating threads in a multithreaded processor with increased intra and inter-thread locality. The method can also be extended for partitioning and mapping loop nests on a multicomputer so that the number of iterations between communication steps is large. This reduces the total communication and synchronization cost on a multicomputer.

References

[1] A. Agarwal, Performance tradeoffs in multithreaded processors, *IEEE Transactions on Parallel and Distributed Systems* **3**(5) (1992), 525–539.

[2] T. Ball and J.R. Larus, Branch prediction for free, *SIGPLAN Notices* **28**(6) (1993), 300–313.

[3] W. Blume et al., Automatic detection of parallelism: A grand challenge for high-performance computing, *IEEE Parallel and Distributed Technology: Systems and Applications* **2**(3) (1994), 37–47.

[4] K. Boland and A. Dolles, Predicting and precluding problems with memory latency, *IEEE Micro* **14**(4) (1994), 59–67.

[5] P. Boulet, A. Darte, T. Risset and Y. Robert, (Pen)-ultimate tiling, in: *Proc. of Scalable High Performance Computing Conference*, Knoxville, TN, May 1994, pp. 568–576.

[6] S. Chatterjee, J. R. Gilbert, F. Long, R. Schreiber and S. Teng, Generating local addresses and communication sets for data-parallel programs, in: *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, May 19–22, 1993, pp. 149–158.

[7] G.H. Golub and C.F. Van Loan, *Matrix Computations*, The Johns Hopkins University Press, 1989.

[8] S.F. Hummel, I. Banicescu, C.-T. Wang and J. Wein, Load balancing and data locality via fractiling: An experimental study, in: *Languages, Compilers and Run-time Systems for Scalable Computers*, B. Szymanski and B. Sinharoy (eds.), Kluwer Academic Publishers, 1995.

[9] L. Kleinrock, *Queueing Systems*, Wiley, 1976.

[10] W. Kaplow and B.K. Szymanski, Impact of memory hierarchy on program partitioning and scheduling, in: *Proc. 28th Hawaii Int. Conf. on System Sciences*, Maui, Hawaii, Vol. II, Jan. 1995, pp. 93–102.

[11] W. Li and K. Pingali, A singular loop transformation framework based on non-singular matrices, in: *Proceedings of the Fifth Workshop on Programming Languages and Compilers for Parallel Computing*, August 1992.

[12] E. Markatos, Scheduling for locality in shared-memory multiprocessors, Ph.D. Thesis, University of Rochester, Rochester, New York, 1993.

[13] S. McFarling, Program optimizations for instruction caches, in: *Proc. of 3rd Int. Conf. on Arch. Support for Prog. Lang. and Op. Sys.*, April 1989, pp. 183–191.

[14] T.C. Mowry, S. Lam and A. Gupta, Design and evaluation of a compiler algorithm for prefetching, in: *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, Boston, MA, USA, 12–15 Oct. 1992.

[15] T.C. Mowry, Tolerating latency through software-controlled data prefetching, Ph.D. Thesis, Stanford University, March 1994.

[16] J. Ramanujam and P. Sadayappan, Tiling multidimensional iteration spaces for multicomputers, *J. Parallel Distrib. Comput.* **16** (1992), 108–120.

[17] R.H. Saavedra, W. Mao and K. Hwang, Performance and optimization of data prefetching strategies in scalable multiprocessors, *J. Parallel Distrib. Comput.* **22**(3) (1994), 427–448.

[18] B. Sinharoy and B.K. Szymanski, Finding optimum wavefront of computation, *Parallel Algorithms and Applications* **2** (1994), 5–26.

[19] B. Sinharoy and B.K. Szymanski, Data and task alignment in distributed memory architectures, *J. Parallel Distrib. Comput.* **21** (1994), 61–74.

[20] J. Subhlok and K. Kennedy, Integer programming for array subscript analysis, *IEEE Transactions on Parallel and Distributed Systems* **6**(6) (1995), 662–668.

[21] D.M. Tullsen, S.J. Eggers and H.M. Levy, Simultaneous multithreading: maximizing on-chip parallelism, in: *Proceedings of the 22rd Annual International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995, pp. 392–403.

[22] R.P. Wilson et al., SUIF: an infrastructure for research on parallelizing and optimizing, *SIGPLAN Not.* **29**(12) (1994), 31–37.

[23] M.E. Wolf and M.S. Lam, A data locality optimizing algorithm, in: *ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, June 26–28, 1991, pp. 30–44.

[24] J. Xue, An algorithm to automate non unimodular transformations of loop nests, in: *Proceedings of the Fifth IEEE Symposium on Parallel and Distributed Processing*, December 1993, pp. 512–519.