HAL
open science

Data Prefetching and Targeted Loop Optimizations

Abella Jaume, J.M. Codina, Carlos Ciuraneta, Min Dai, Christine Eisenbeis, Antonio Gonzalez, Josep Llosa, Peter Knijnenburg, Michael O'Boyle, Sid Touati, et al.

To cite this version:

Abella Jaume, J.M. Codina, Carlos Ciuraneta, Min Dai, Christine Eisenbeis, et al.. Data Prefetching and Targeted Loop Optimizations. [Research Report] M2.D2, 1999, pp.47. hal-00647624

HAL Id: hal-00647624

https://inria.hal.science/hal-00647624v1

Submitted on 2 Dec 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Deliverable M2.D2
Data Prefetching and Targetted Transformations

Jaume Abella, Josep M. Codina, Carlos Ciuraneta,
Min Dai, Christine Eisenbeis, Antonio Gonzalez, Josep Llosa, P.M.W. Knijnenburg,
M.F.P. O'Boyle, Sid Ahmed Ali Toutai and Xavier Vera,
INRIA, Leiden University, UPC, University of Edinburgh

November 26, 1999

Contents

1 Overview **1**

2 SOFTWARE PREFETCHING **2**

2.1 INTRODUCTION 2

2.1.1 What is software prefetching? 3

2.2 THE SUPPORTED ARCHITECTURE 4

2.2.1 The MIPS R10000 processor 4

2.3 Structure of our tool 5

2.3.1 Locality analysis 6

2.3.2 Prefetch insertion 6

2.4 Status 7

2.4.1 Results & examples 7

2.5 Future work 8

3 Software Pipelining and Prefetching **10**

3.1 INTRODUCTION 10

3.2 TOPS 11

3.2.1 STATEMENT TRANSFORMATION 13

3.2.2 SOFTWARE PIPELINING - THE DESP ALGORITHM 13

3.2.3 REGISTER ALLOCATION - LOOP UNROLLING 13

3.2.4 LOOP RESTRUCTURING 14

3.3 CODE OPTIMIZATIONS 14

3.3.1 OPTIMIZATION ALGORITHM 15
3.3.2 DATA PREFETCHING 16
3.4 PRELIMINARY RESULTS 16
3.4.1 Without prefetching 16
3.4.2 With prefetching 18
3.5 CONCLUSION AND FUTURE WORK 18

4 Targetted Global Optimisation 22
4.1 Introduction 22
4.2 Example 25
4.3 Notation 26
4.3.1 Iteration Spaces 26
4.3.2 Index Spaces 26
4.3.3 Array Accesses 26
4.4 Rank Modifying Transformations 27
4.4.1 Data Transformations 27
4.4.2 Loop Transformations 28
4.4.3 Form of Transformation 29
4.4.4 Example 30
4.4.5 Combining Loop and Data Transformations 31
4.5 Handling Data Transformations in the Presence of Linearised Arrays . . . 31
4.5.1 Data Transformations on Linearised Arrays 32
4.5.2 Reducing Access Overhead for Linearised Arrays 34
4.6 Handling Data Transformations in the Presence of Reshaped Arrays . . . 35
4.6.1 Data Transformations for Reshaped Arrays 35
4.6.2 Reducing Access Overhead for Reshaped Arrays 36
4.7 Experiments 37
4.7.1 Algorithm 38

4.7.2 Results 38
4.8 Related Work 38
4.9 Conclusion 39

List of Figures

3.1 Optimization steps 12

3.2 OPTIMIZATION ALGORITHM USING DATA DEPENDENCE TEST 17

3.3 Performances from Perfect Club benchmarks 17

3.4 Variation of performances with load latencies - Livermore benchmark - UltraSparc 19

3.5 Variation of performances with load latencies - Livermore benchmark - Dec alpha 19

4.1 Transformations 24

4.2 Example loop 28

4.3 Loop and Data Transformations 28

4.4 Linearised Aliasing 32

4.5 N = 64... 1024 34

4.6 Propagation Algorithm 37

Chapter 1

Overview

This purpose of this deliverable is to describe ongoing work in the area of prefetching and targetted loop transformations. Chapters 2 and 3 describe ongoing work in source-level software prefetching while chapter 4 describes the integration of loop and data transformations for spatial and temporal locality.

Chapter 2 describes an annotation based approach to software prefetching. By using static locality analysis, we can determine those memory references likely to incurr cache misses. We can therefore avoid inserting unneccessary prefetch instructions for data likely yo be already in the cache. Initial results suggest that this is a promising approach.

Chapter 3 decribes a different approach to prefetching, namely source to source software pipelining. The main benefit of software -piplening is that we percolate up load instructions taking into consideration their anticipated latency. This novel work, applies this previously considered low-level approach at the sourse level where preliminary results are very encouraging.

Finally, in chapter 4 we consider targetted transformations. This work unifies the treatment of loop and data transformations by embedding them in a new rank-modifying framework. Within this framework any restructuring transformation has a inverse which can be used to globally optimise a program. This framework allows the generation of a new family of transformations and preliminary results are once again highly promising.

Chapter 2

SOFTWARE PREFETCHING

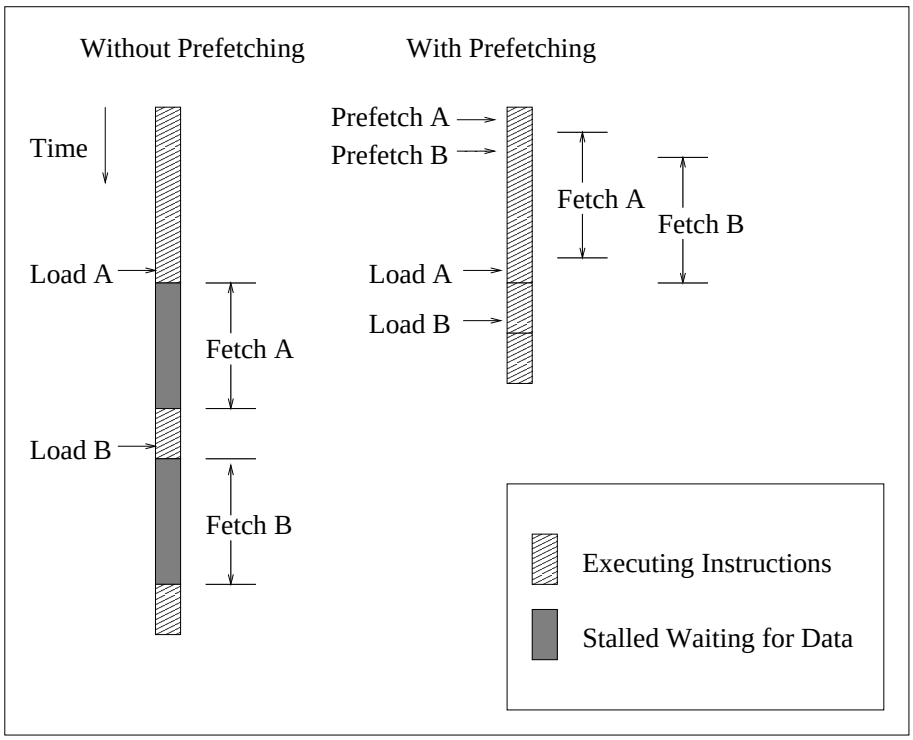
2.1 INTRODUCTION

Memory latency is a major issue for many modern microprocessor based systems. Although microprocessors speeds have been increasing dramatically, the speed of memory have not kept the pace. At the last years the speed of commercial microprocessors have been doubling every three years while the speed of commodity DRAM has improved by little more than 50highest priority in improving the capacity. The result is that, from the perspective of the processor, memory is getting slower at a dramatic rate. Most of computer systems rely on their cache hierarchy to reduce the effective memory access time but while the effectiveness of caches is quite good for general-purpose codes is not as good for scientific and engineering applications. Different techniques to reduce or tolerate latency are caching, Locality Optimizations, Buffering and Pipelining, Software-Prefetching and Multithreading. Reducing latency is preferable over tolerating latency since it actually reduces the demand for main memory bandwidth, wich can be crucial. Reducing latency is preferable over tolerating latency since it actually reduces de demand of memory bandwidth, which can be crucial. Caches provide the foundation for all the latency-hiding techniques, and locality optimization are also attractive since they don't require additional hardware support. After reducing latency, we must to tolerate any remaining latency. The first way to do that is buffering and pipelining accesses, which is an effective means to reduce write latency and requires only a lockup-free cache. To address read latency as well, the choices are either prefetching or multithreading. Software-controlled prefetching appears to be more desirable solution since it requires significantly less hardware support than hardware-controlled prefetching or multithreading and it only requires a single thread of execution, while multithreading needs more than only one. Hardware prefetching is limited to the fact that it is limited to prefetching constat-stride accesses and they may entail a significant hardware cost.

2.1.1 What is software prefetching?

The concept of software prefetching is basically to add new memory instructions in the original code. With these new instructions we can split apart the request of the data and the use of the data, while finding enough parallelism to keep the processor busy in between. To hide the latency within a single thread, the request of data has to be so far in advance of the use of the data in the execution stream. This requires the ability to predict what data is needed ahead of time. Software prefetch requires explicit prefetch instructions to move data into the cache. The format of these instructions resembles a normal load instruction, but without a register specifier. Prefetch instruction also differ from normal load instructions in that they are non-blocking and they not take memory exceptions. The non-blocking aspect allows them to be overlapped with computation, and the fact that they don't take exceptions is useful because it permits more speculative prefetching strategies (e.g., dereferencing pointers before is certain that they point to legal addresses, or the fact that is easiest to prefetch array references in the same loop than split it for not prefetch not legal array points, they could be also illegal references). The challenges of software-controlled software prefetching include the fact that some sophistication is needed to insert the prefetches into the code, and also that the new prefetch instructions involve some amount of execution overhead.

The following example illustrates how prefetching can hide memory latency.



On the left we can see the case without prefetch where the processor stalls when it attempts

to load two locations that are not in cache. On the right the prefetch can be issued far enough in advance so the memory accesses don't stall the processor because the memory accesses will have completed before the loads are executed. Our approximation is software prefetching because we only need compiler and hardware support. Hardware support is guaranteed on last generation processors. At the next section we describe first the supported architecture. Then in next sections we describe our tool and the future work.

2.2 THE SUPPORTED ARCHITECTURE

We have been developing our tool on a platform that allows us to use the Software Prefetching. For this we needed a processor that has prefetch instructions that not take memory exceptions. Also we need a compiler that supports a prefetch instruction.

2.2.1 The MIPS R10000 processor

In this section we will talk about the hardware we have used. It includes the processor and the memory hierarchy, although there are levels of this hierarchy inside the microprocessor. We have been working with the MIPS R10000 microprocessor, which has implemented the prefetch instruction. The R10k is a last generation microprocessor. It has branch prediction, a non blocking cache, superscalar (it can issues at last 4 instruction at the same time) and it has out of order execution. This last means that the processor can execute instructions independent of its situation in the stream of execution while this instruction wouldn't depend on others that haven't been executed. The out of order also permits to hide some latency due to instructions behind a memory instruction that misses can be executed if they don't have any dependence. To support the out of order execution the R10k has a window of 32 instructions that can be scheduled dynamically for best performance. This microprocessor has built-in the first level of cache. It also implements the control unit for the next level. These two levels are 2 way set associative. The first level has 32 bytes per line while for the second level it can be chosen between 64 or 128, it depends on the implemented system. The replacement algorithm is LRU for both caches. In our case the second level cache is 4 Megabytes and has 128 bytes per line. The memory latency between the first and the second level of cache is 3 cycles. This latency is enough low to be hidden by the out of order execution. The latency for the main memory is around 40 cycles but it can be larger due to the factor that we are working on a SGI Silicon Origin 2000 that is a distributed shared memory multiprocessor computer.

* ISA for prefetch On the MIPS IV ISA there are two instruction for prefetch whose mnemonic are 'pref' and 'prefx' respectively. Their format is the following:

PREF hint, offset(base) PREFX hint, offset(base)

The only difference is the addressing mode. The address in the instruction is calculated adding the 16-bit signed number (offset) that is inside the instruction with the value contained in the register (base). For the PREFX instruction is the same but the offset is in a register, so the address is calculated adding the two registers. The hint field supplies information about the data is expected to be used. The hint field indicates which sort of prefetch should be done, depending of the hint it could be done as a NOP. Values of this hint field are: 0 (load), 1(store), 4 (load_streamed), 5 (store_streamed), 6 (load_retained) and 7 (store_retained). For the 0 and 1 values (load and store) prefetch is done like normal memory instructions (except for no work with registers). For 4 and 5 the data is expected to be not reused extensively. For the 6 and 7 the data is expected to be retained in the cache and reused extensively. 1.2.2 The SGI f77 MIPSpro Compiler

We work with the f77 MIPSpro Compiler. This compiler helps us to add compiling directives that tell where to put the prefetch instruction at the low level code. The compiler also can add prefetch automatically. With these directives, software prefetching is also possible in high-level codes and the user may add prefetching without any knowledge about assembler. Once the user has decided where to do prefetching he must add the compiler directives. A directive is composed of the reference to prefetch and a stride. The stride is the distance in loop iterations between different executions of the prefetch instruction. Example of a code with this directive:

```
DO i = 1, 10000, 1
         b(4*i) = 3
C*$* PREFETCH_REF=b(4*(i+33)) ,stride=4
         a(4*i) = 2+b(4*i)
     ENDDO
```

In this example we can a prefetch directive in wich we take profit of the spatial locality of the reference b(4*i). The reference must be prefetched with enough time in advance,in this case 33 iterations of the loop. With the stride 4 we indicate to the compiler that the prefetch must be executed only every 4 iterations of the original loop so that we only prefetch once every 128 bytes (the item size is 8 bytes). When we prefetch an item all the cache line the item belongs is loaded in the cache so only one prefetch for each line is performed.

2.3 Structure of our tool

Our goal is to implement a tool that inserts prefetching instructions for a numeric codes in a high level language as Fortran. To understand better the behaviour of the tool we must differentiate two types of references. The first type is the static reference. An static reference means the reference that is written in the code. The second type is the dynamic

reference. A dynamic reference is an execution of an static one. This nomenclature is also taken for prefetch instructions. The basic algorithm is divided in two major phases. The first phase consists in a locality analysis that is necessary for knowing which references produces misses. The second phase is the prefetch insertion, we need to transform dynamic prefetches into static ones. This means that if we have a reference that produces one miss per four dynamic executions we will have to isolate the dynamic reference (e.g. by unrolling the loop and so the static reference will become into four), but is not needed to do this since the compiler directives allow us to do this automatically without any loop transformation.

2.3.1 Locality analysis

To be sure that prefetches are not unnecessary we need to know which dynamic references misses in the cache. To know the references a locality analysis is needed. With this analysis we select the references to be prefetched. To do this analysis we use our tool of static locality analysis (SPLAT). The first step of the locality analysis is the volume phase. In this phase we consider the volume of data that the loop accesses. With this information we have a good approximation of the data that will be loaded in the cache between two dynamic references of the same reference and if this data may replace the line that the reference accesses. A more precise analysis would require an interference analysis. With this analysis we can know if a previous miss of a reference replaces a cache line because set of conflicts. For the moment we are not able to apply an interference anaylis. This analysis needs to obtain the base addresses of the data structures but for the moment we are not able to obtain this information. Also the interference analysis that makes our locality analysis is focused to direct mapped caches and the target processor has two-way associative caches. However we have noticed that the main number of the interferences are produced in a short interval of time where the prefetch can't hide all the latency and in many cases it adds overhead to the execution. Therefore, the interference analysis is not so important for many codes.

2.3.2 Prefetch insertion

After the locality analysis is done we must to insert the prefetches instructions that will hide the memory latency. At this moment we only add prefetching instructions to those references that have no locality or only have self-spatial locality. We use the Ictineo compiler to get a low-level code and consider this information (the low-level instructions) to put the prefetch far enough in advance. We also consider the information about the stride of the loop and the factor that multiplies the loop index variable to be sure that a cache line is prefetched only once. We don't need to unroll the loop so this is made automatically by the compiler. Although not always the compiler unrolls it enough times to insert the minimum number of prefetch instructions. It can result on unnecessary prefetch instructions. We use this phase to put prefetches far in advance in the loop. For dynamic references we

must to isolate them in static references. The reason for this is not to add of extra code like conditional sentences in order to perform the prefetch only in some iterations. To do that we can use techniques as unrolling or peeling, but is not necessary because within the compiler directives of prefeching we can specify the stride that we want to put the prefetch instructions. Then the compiler is who applies these techniques.

2.4 Status

At this moment our tool only performs prefetch for the innermost references of the loop-nests. We don't consider prefetching before a loop or interloops. We only consider a volume analysis for the locality. We focus on the prefetch to the second level cache, where it seems to have better results because the memory latency between the first and the second level is hidden by the out of order execution. We are comparing our results with the automatic prefetching of the MIPSpro Compiler. The code generated, with prefetch, by the compiler has better performance than the normal version of the program (without prefetching) so our goal is to improve the results obtained by the compiler. We have noticed that in some cases the same code that would be worse in performance than the same code but with different prefetch insertion can be more stables (better performance), but not in every execution. This is due to the factor that in a multiprocess system cache lines can be replaced for other lines from another process. In these cases unnecessary prefetches may be good for performance so they minimize the effect of conflicts produced from other processes because these prefetches refill the cache before data is required.

2.4.1 Results & examples

The following results are obtained using the hardware counters that provides the R10k processor and our last version of the tool. To obtain results that are more significant the example codes have been iterated many times, this is the reason why the number of cycles is so large.

Example 1

```
INTEGER*8 A(1048576),B(1048576)
DO I=1,1048576-7,4
   B(I+4)=3
   A(I)=B(I)+2
ENDDO
```

For the above code we obtained the following results:

Example 1

Time (in cycles)	No prefetch	Automatic prefetch	Our tool
Exclusive	2003041281	1756707667	1753084128
Median (multiprocess)	2336381137	1892494646	2152596719

Example 2

```
REAL*8 A(1048576),B(1048576),C(1048576)
DO I=1,1048576,4
  A(I)=A(1048576-I)
ENDDO
```

Time (in cycles)	No prefetch	Automatic prefetch	Our tool
Exclusive	1408484347	1189681364	1083980160
Median (multiprocess)	1482250224	1298606240	1238017425

In the first example we can notice that the results for our tool executing the code in exclusive, i.e. like it was executed in a monoprocessor machine, are better than the automatic prefetch made by the compiler. In the case of multiprocessor the code generated for the machine tolerates better the latency due to the extra prefetch that the compiler adds. At the second example the generated code from our tool is better in both cases. The reason for this is the best location of the prefetch instructions. We allocate two prefetch instructions while the compiler only allocates one. But there are some codes where we add more overhead. For instance the next example.

Example 3

```
REAL*8 A(393216),B(393216)
DO I=1,393216,8
   A(I)=3
   B(I)=4
ENDDO
```

Example 3

Time (in cycles)	No prefetch	Automatic prefetch	Our tool
Exclusive	473617316	447621946	463165879
Median (multiprocess)	575886003	491786244	548976407

2.5 Future work

In the next year, we will work to improve the performance of our tool in several aspects:

- point to do is to solve the problem with the interference analysis.

- Work with different types of associativity. By this way our tool would be more generic.
- The compiler not always isolates the prefetch instructions as good as possible. It is due to its scheduling phase. The tool would have to do it automatically and force the unroll either with compiler directives or unrolling it before.
- Work on different types of prefetch. For the moment, we only add innermost prefetch instructions but there are some references whose behaviour of locality not depends only of the innermost loop.
- We also seek to have a generic tool not depending on the architecture.
- Another interesting aspect we plan to research is to perform some sort of prefetch for architectures without prefetch instructions.

Chapter 3

Software Pipelining and Prefetching

Abstract

Loop software pipelining is now a well known technique for exploiting Instruction Level Parallelism present in most today's high performance processors. Compilers usually perform software pipelining in a low level code where instructions are or are close to actual assembly instructions because a precise timing information is needed. We present here the TOPS source to source software pipelining framework that aims to act as a preprocessing step in the compiling process very much like the usual automatic parallelizers. It is based on an approximation of the architecture model but the progarmmer can specify through directives presumed load latency for some memory access. Hence, data prefetching can be controlled in softwareand combined with software pipelining. We present here the TOPS tool for source to source software pipelining as well as preliminary experimental results.

3.1 INTRODUCTION

It is widely recognized that loops provide the largest source of optimization in common numerical programs, and loop execution dominates almost total execution time of an application program. A lot of researches have resulted into development of various techniques for exploiting parallelism and data locality within nested loops.

The coarse grain parallelism present in multiprocessors can be exploited by a restructuration of nested loops in the source code, whereas fine grain parallelism (also called Instruction Level Parallelism [RF93] superscalar or VLIW) in today's high performance monoprocessors is rather exploited at the assembly code level by loop software pipelining [AJLA95] .

Memory management is the other important feature of high performance architectures.

It is also handled differently in multi- and mono-processors. In MIMD architectures, the problem is to partition the data set among the local memories of the different processors; this is usually done in the source code. In the mono-processors, the problem is to exhibit good temporal or spatial data locality in the code - this can be performed in the source code - or to efficiently use the registers available - this is usually performed in the assembly code.

What is proposed in this paper is a first step towards a common framework for studying the combination of these different optimizations. We have implemented source to source software pipelining based on DESP [WEJS94] in the Sage++ [BBGS94] compiling environnement. Software pipelining is a loop transformation that increases the parallelism within the loop body by combining instructions from different iterations into the same loop body. Because today's microprocessors have dynamic features that make them able to exploit local parallelism - we believe that this approach makes sense. We have also added the possibility for the programmer to specify presumed latencies of some memory references by means of directives. This allows first to keep data prefetching under control and second, software pipelining reschedules intructions in order to take into account this data prefetching. In the MHAOTEU framework, load latencies can be obtained by data locality analysis that point out memory references that miss in the cache. We first describe the TOPS tool for source to source software pipelining and then give preliminary experimental results with and without specifying load latencies.

3.2 TOPS

There are many ways for improving the execution time of an application program. One approach, termed *hardware processing*, and another, termed *software processing* or *parallel processing*. Software pipelining is a software processing technique which restructures loops by combining instructions from different iterations into the same loop body. For exemple, let us consider an architecture consisting of four functional units: one Integer Unit, one Load/Store Unit, and two Floating Point Units. A simple inner loop and its corresponding simulation assembly code are shown in Figure 1(a) and 1(b). Without any transformation, each loop iteration requires 9 cycles to execute on a given ILP processor. If the loop is software pipelined (Figure 1(c)), each loop iteration in the *kernel* requires 4 cycles in average. The instructions 2 and 0 can be executed in same cycle, as well as instructions 8-6-1, 4-3, 7-5.

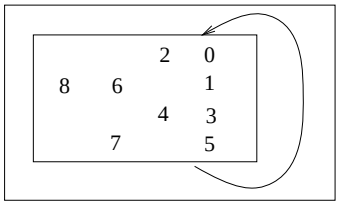
The objective of software pipelining is to increase the ILP inside each inner loop. It "removes" dependences between instructions within each iteration and replaces them by dependence across iterations. This removal of dependences makes more instructions independent from each other and thereby increases the number of instructions that can be executed concurrently.

```
DO  10  I = 3, N
   B[i] = (A[I-1] +A[I-2]
          + A[I+1] +A[I+2])/D
10  CONTINUE
```

(a) Original Loop

Inst.	DO I = 3, N	cycle
0	T1[I] = A[I]	0
1	T2[I] = A[I+1]	1
2	T3[I] = T1[I] + T2[I]	2
3	T4[I] = A[I+3]	3
4	T5[I] = T3[I] + T4[I]	4
5	T6[I] = A[I+4]	5
6	T7[I] = T5[I] + T6[I]	6
7	T8[I] = T7[I] / D	7
		8
8	B[I] = T8[I]	9
	ENDDO	

(b) Simulation Assemble code



(c) Schedule of pipelined loop

IF (I > 6) prelude instructions	
DO I = 1, N, 2	
B[I] = r82	1
r71 = r52 + r61	2
r31 = r12 + r22	3
r11 = A[I]	4
r51 = r31 + r42	5
r21 = A[I+1]	6
r81 = r71 / D	7
r41 = A[I+3]	8
r61 = A[I+4]	9
B[I+1] = r81	10
r72 = r51 + r62	11
r32 = r11 + r21	12
r12 = A[I+1]	13
r52 = r32 + r41	14
r22 = A[I+2]	15
r82 = r72 / D	16
r42 = A[I+4]	17
r62 = A[I+5]	18
ENDDO	
postlude instructions remainder iterations	
ELSE sequential loop	

8 loads, 2 stores / 2 iterations

(d) Pipelined Loop

IF (I > 6) prelude instructions	
DO I = 1, N, 2	
B[I] = r82	1
r71 = r52 + r42	2
r31 = r21 + r22	3
r51 = r31 + r42	4
r21 = r41	5
r81 = r71 / D	6
r41 = r62	7
B[I+1] = r81	8
r72 = r51 + r62	9
r32 = r22 + r21	10
r52 = r32 + r41	11
r22 = r42	12
r82 = r72 / D	13
r42 = A[I+4]	14
r62 = A[I+5]	15
ENDDO	
postlude instructions remainder iterations	
ELSE sequential loop	

2 loads, 2 stores / 2 iterations

(e) After Optimizing

Figure 3.1: Optimization steps

The first problem is to simulate the low level 3-address instructions in the source code. This is done by splitting instructions into virtual 3-address instructions. For avoiding any preliminary assumptions on where the temporary variables are to be stored, we use temporary arrays. It should be noted that this expansion of scalars into arrays is done only internaly and does not remain in the final produced program.Then the usual dependency graph is computed by Sage++. The loop is software pipelined by DESP, followed by a phase of register allocation with possible unrolling. Then the elimination of redudant load/store of subscripted variables [DE97] is performed.

3.2.1 STATEMENT TRANSFORMATION

In our approach, all statements of high level language are first transformed into virtual 3-address instruction. We use temporary array variable for replacing every expression of original statement, including memory accesses, arithmetic operations, ... with the form of $T_i(I) = e$, where e is part of the right hand side expression of original statement. Figure 1(b) shows the transformed code. The transformed instructions provide all informations of data dependence. Sage++ can compute the data dependence for FORTRAN program by using Omega test [PW92a]. It should be noted that index and address computations are not taken into account, hence it should be assumed that induction variables have been identified in a preliminary step. Since we are mostly addressing memory problems, we could also separate only memory accesses (loading of data into registers) and keep computation expressions as there are.

3.2.2 SOFTWARE PIPELINING - THE DESP ALGORITHM

Software pipelining is usually performed by Modulo Scheduling. Modulo Scheduling is an iterative algorithm that generates a schedule for one iteration of the loop body, such that this schedule can be issued every II clock cycles [8]. In TOPS, we use the DESP algorithm. DESP (DEcomposed Software Pipelining) [WEJS94] is a direct method for loop software pipelining. The schedule is represented with a matrix of operations. In this matrix, the row-number of an operation denotes its place in the final schedule whereas the column-number denotes which iteration of that operation is placed in the loop body. DESP takes the dependency graph and the resource constraints and returns the row-numbers and column-numbers of each operation.

3.2.3 REGISTER ALLOCATION - LOOP UNROLLING

Software pipelining increases the lifetime of variables that may span more than one iteration. This means that more than one instance of the same loop variable is alive at some point of the loop, so that more than one register is needed for carrying all these instances.

For expliciting this register allocation in the software, there are two ways. The first way is to use for some variable a set of registers that is shifted at each iteration. The other way is to unroll the loop for describing explicitely in which registers the different instances of some variable are hold. We have chosen to implement the latter technique in order to make the code simple and also because loop unrolling increases in general the opportunities of optimization by the compiler or the processor. Actually this strategy happened to have interferences with the compiler that also unrolls loops.

DESP offers an option to allocate variables on the minimal number of registers. Since this allocation in the source program consists in allocating variables into scalars, we decided to use a very simple variant of the MVE (Modulo Variable Expansion) method. It consists in unrolling the loop only a number of times u equal to the largest number of iterations that a variable lifetime spans. Then u different scalars ("registers") are assigned to each variable.

3.2.4 LOOP RESTRUCTURING

After DESP, the code has to be rewritten according to the row-numbers and column-numbers computed. The code scheme is the following. The first iterations are initialized in the prelude P_r until a steady state - the new loop K - is achieved. Then the last iterations must be completed at the end of the loop in a postlude P_o. This scheme works only for a number of iterations greater than the number of iterations involved in the prelude. If the number of iterations is less, then the sequential loop is executed (S_l). Also, due to unrolling and for avoiding conditional branchs in the final loop, there remains a little number of iterations R_l to perform. Hence the scheme for loop transformation can be presented with $[(IF)P_rKP_oR_l(ELSE)S_l]$. Figure 1(d) shows a transformed loop unrolled 2 times.

3.3 CODE OPTIMIZATIONS

After the phase of loop software pipelining, there remain opportunities for optimization by avoiding redundant memory acccess. Ideally, this optimization should be taken in consideration simultaneously to the software pipelining process, since keeping array variables in registers increases register pressure. For a preliminary evaluation, we have designed a variant of the algorithm described in [9, 10, 11]. Future work will consider combination of software pipelining with removal of redundant memory accesses.

3.3.1 OPTIMIZATION ALGORITHM

Our optimization algorithm is based on the data dependence test [WT92, PW92a, PW92b]. We use the data dependence graph (DDG) to illustrate the relationships between various operations. The data dependence graph can be represented by G(N,E), where N is the set of all nodes, and E is the set of all dependences edges. Some definitions are shown in Figure 2. The algorithm consists of two parts: data dependence and variable reference analysis.

Data Dependence Analysis: There are two steps in eliminating redundancy: loads and stores elimination. First, we consider $True$ and $Input$ dependences in order to avoid partial redundant loads. A load is redundant, if, on every control flow path ending at this node, the loaded value is already available in a register. To detect redundant load, we use the data dependence informations. That is, if there exist $True$ and $Input$ dependences within the loop, some loads can be eliminated. For example in the loop of Figure 1(d), because of the input dependence (S^6, S^{13}), we can remove the instruction 13 because the value loaded by this instruction is already available in register $r61$. However, in order to avoid erasing prematurely this value, we insert a move operation (shifting of registers) for saving the value in $r42$. It may happen that this move instruction is unnecessary. This is discussed in the next step of the algorithm.

Then, $Output$ dependences are considered to eliminate redundant stores. A store S^{in} in a dependence S^{in} and S^{out} is redundant and can be removed if there is no read access to this variable between S^{in} and S^{out}. Thus, the problem of detecting and eliminating redundant stores can be solved by testing $Output$ data dependence. The result of optimization is shown in Figure 1(e). We can see that after this optimization, there are only two memory loads operations instead of eight previously.

Variable Reference Analysis: In our approach, all memory loads and arithmetic operations are allocated into virtual registers. If the dependence distance is bigger than one, some register move operations are necessary for saving different instances of that variable. For managing these register shifts precisely, we first analyse each register variable. Our *Variable Reference Analysis* (VRA) is based on the extension of the methods of [DGS93, BG95]. The number of shifts depends on the dependence distance. A shift operation may be redundant if, on the path from the first definition of that register to the node that uses it, the register that carries a copy of the variable is never used by other instructions. For our example loop, a input dependence is found between instruction 15 and 4 (S^{15}, S^4) with distance 1. In general, a shift should be inserted for replacing load operation (instruction 4). From instruction S^{15} to S^4, the definision of instruction 4 (r11) is never used by other instructions, therefore the shift instruction can be eliminated and $A[I+2]$ in instruction 15 can be directly loaded into $r11$. The details of the algorithm are given in Figure 2. The result for our example is shown in Figure 1 (e).

3.3.2 DATA PREFETCHING

The architecture model used currently in TOPS specifies latencies for each type of instruction. It is fixed for the whole software pipelining process. However, TOPS offers also to the programmer the possibility to specify longer memory latencies. This is done by means of directives (under comments in FORTRAN), where each memory access of the next statement is given a special latency.

For instance, in the loop below, the first directive specifies that the next loop has to be software pipelined. The second line gives the latencies assigned to the load operations of the next statement (5 for `A(I)`, 5 for `B(I)` and 1 for `C(I-1)`, that is supposed to be already present in the cache since it is computed in the previous iteration).

```
CPipeline_Loop
CLatencies (1, 5) (2, 5) (3, 1)
          DO 20 I = 2, N
          C(I) = A(I)*B(I) + C(I-1)
20        CONTINUE
```

Typically one expects that a long latency is given to data that are known to miss. For taking into account spatial locality, one can also unroll the loop a number of times equal to the line size and prefetch only the first line element when known. This avoids useless prefetchings that tend to increase register pressure. This is what is are currently implementing.

We are also considering connecting our TOPS tool to MHAOTEU. This can be done by connecting TOPS to OCTAVE that has almost exactly the same library functions as Sage++. Once it is connected, we will experiment on MHAOTEU benchmarks by using the analysis tools of MHAOTEU for deciding which variables to prefetch. In the next section, we give results of experiments with and without prefetching.

3.4 PRELIMINARY RESULTS

3.4.1 Without prefetching

In this section, we report our experimental results obtained by using only Software Pipelining described in section 3.2 and our array reference optimization method described in section 3.3.1. The source Fortran codes are from Perfect Club Benchmark suites.

For examining the efficiency of our method, we first profile the program for finding frequently executed program regions and we optimize them. Each benchmark with the highest percentage of execution time functions and with nested loops (CSS/terr, LGS/choos, APS/trid, detdxf, SDS/matmul, chosol, LWS/interf and TIS/olda) are chosen to measure

Definition 1 :

Let G = (N, E) be a data dependence graph of a loop,
if $\exists$ e = (S^{in}, S^{out}) $\in$ E, we say there is a dependence between S^{in} and S^{out}, and S^{out} follows S^{in}.

$E_F \subseteq$ E is the set of flow dependence

$E_I \subseteq$ E is the set of input dependence

$E_O \subseteq$ E is the set of output dependence

Definition 2 :

Let R_{def} (R_{use}) be a set of definition (use) references

R^{in}_{def} is node S^{in} definition reference

R^{in}_{use} is node S^{in} use reference

R^{out}_{def} is node S^{out} definition reference

R^{out}_{use} is node S^{out} use reference

Algorithm :

```
FOR (every nodes S^in and S^out ∈ N)
  IF ( ∃ e = ( S^in , S^out ) ∈ E_F (or ∈ E_I)) Then
     compute the distance (d) of S^in and S^out
     IF ( d == 0 ) THEN
          del( S^out )
          FOR (all r = R^out_def ∈ R_use )
               r ← R^in_use (or r ← R^in_def )
     ENDIF
     IF ( d == 1 ) THEN
        IF ( ∃ r = R^out_def ∈ R_use from S^in to S^out )
             S^out is replaced by a shift operation
        ELSE
           del( S^out )
           FOR (all r = R^out_def ∈ R_use )
               r ← R^in_use (or r ← R^in_def )
     ENDIF
     IF ( d > 1 ) THEN
        S^out is replaced by a set of shift operations
     ENDIF
  ENDIF
  IF ( ∃ e = ( S^in , S^out ) ∈ E_O ) Then
     del( S^out )
  ENDIF
ENDFOR
```

Figure 3.2: OPTIMIZATION ALGORITHM USING DATA DEPENDENCE TEST

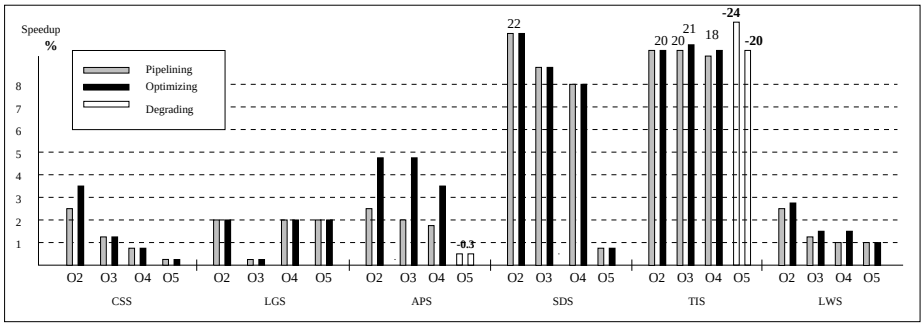


Figure 3.3: Performances from Perfect Club benchmarks

the performance benefit of our efficient code transformation. The experimental loops are single basic block loops that do not contain function call and branch. The target processor assumed for these experiments are four issue rates processors. All speedups are reported over the original progam with different optimization levels. The pipelined code is generated by DESP algorithm and optimized by our techniques described in section.

Figure 3.3 shows the performance and speedup results of each benchmark. For comparing the effectiveness of our approach, we compile and execute each program in every optimization levels of FORTRAN compiler on DEC-Alpha processor. The software pipelined loops in all programs on each optimization level required less registers than target machine has. The performances can be improved till 20%. However, for function *olda*, after pipelining loops, there was a larger amount of spill code generated on optimization level O5, the performance is degraded. Also, we can see, at optimization level O2 or O3, the maximum performance improvement are obtained.

Software pipeline and optimization techniques improve the performance of CSS, LGS and APS by 0 to 3 percent. In these programs, the treated functions have only small percentage of total execution time. However, for SDS and TIS program with 81 and 99 percent of execution time, we can get over 20 percent of improved performance. Even so, our methods can provide a significant performance improvement.

3.4.2 With prefetching

For testing prefetching we analyzed Livermore loop 23 on UltraSparc and DEC-alpha. We try different optimization levels because our source to source software pipeliner may interfere with transformations done by the compiler. Results are given on figures 3.4 and 3.5. On these figures execution time for the five optimizations options and different values for load latencies (5, 10, 20, 50, 100) and compared to the original code. We can observe that in both cases small latencies give speedups and the best performance is obtained for latency 5, that gives a speedup of 2. As expected, long latencies induce very high register pressure and cause performance degradation. As a matter of fact, analysis of assembly code shows a lot of spurious register spillings. It is also interesting to note that optimization O5 gives surprisingly bad performance on Dec-alpha. This is because O5 optimization performs already software pipelining, that results in another unrolling of the loop and code size increase.

3.5 CONCLUSION AND FUTURE WORK

This paper presented the motivations and the implementation of a framework for source to source software pipelining and software data prefetching. For preliminary experiments, we have based the optimization on a simple virtual instruction-level parallel processor and

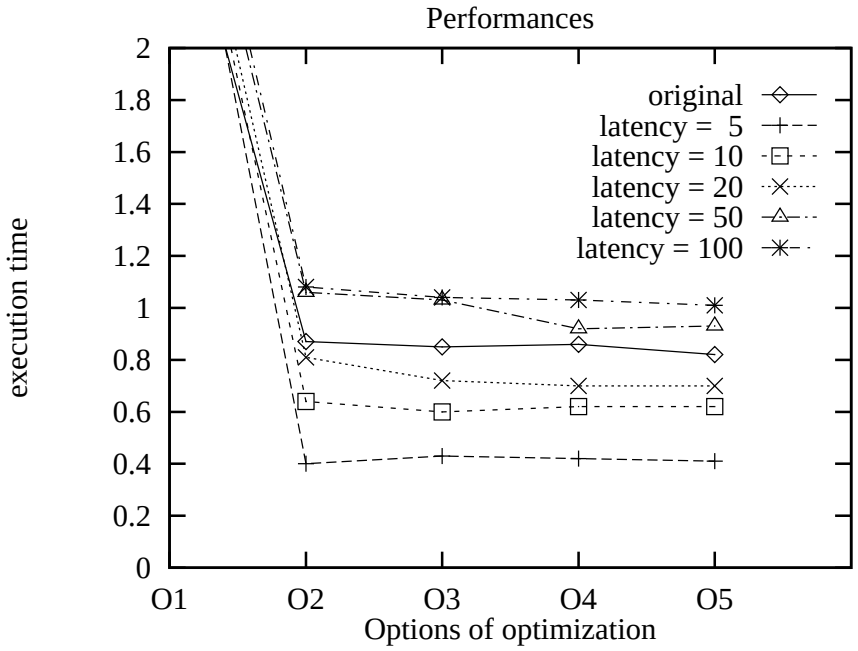


Figure 3.4: Variation of performances with load latencies - Livermore benchmark - UltraSparc

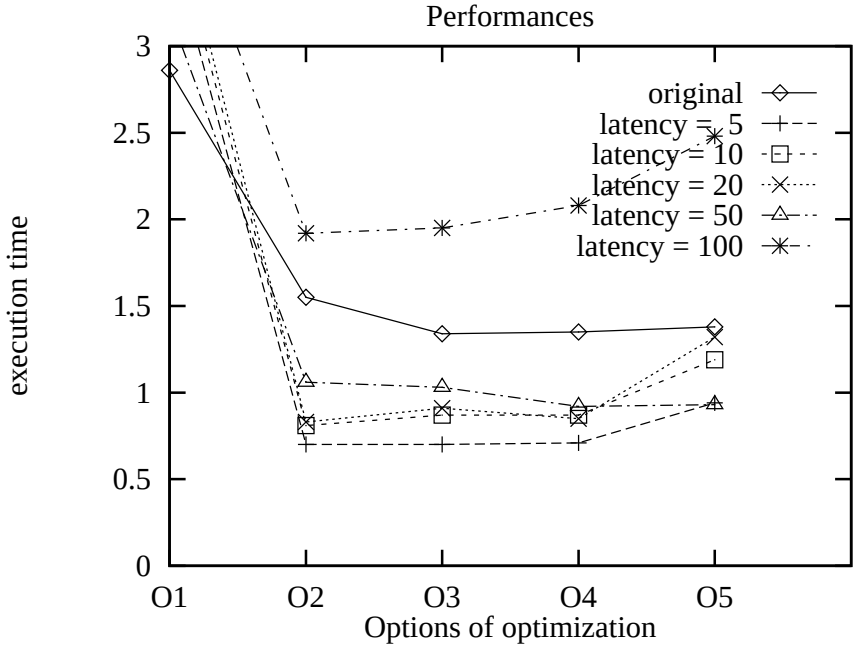


Figure 3.5: Variation of performances with load latencies - Livermore benchmark - Dec alpha

a very greedy register allocation schme and show that we are able in some cases to attain and sometime increase the performance of the code generated by the compiler with the "software-pipelining" option O5. This is very encouraging for the future.

For improving software pipelining itself, there are many directions we are considering, for instance mixing of load/store elimination and loop scheduling, incorporating a performant register allocation algorithm, studying precisely the relationship between the number of "virtual" registers explicited in the source code and the actual number of registers in the assembly code.

As for data prefetching, our TOPS tool is very convenient for performing scheduling of loads in advance. Experiments have highlighted sensitivity of software pipelining to specified load latencies. We expect that this optimization gives good results when only missing references (identified by MHAOTEU analysis) are fetched in advance, and different latencies are tried for finding the best optimization scheme.

Bibliography

[AJLA95] V.H. Allan, R.B. Jones, R.M. Lee, and S.J. Allan. Software Pipelining. *ACM Computing Surveys*, 27(3):367–432, September 1995.

[BBGS94] F. Bodin, P. Beckman, D. Gannon, and J.G.S. Srinivas. Sage++: A class library for building Fortran and C++ restructuring tools. *Proc. of the Second Object-Oriented Numerics Conference*, avril 1994.

[BG95] R. Bodik and R. Gupta. Optimal placement of load-store operations for array access in loops. Technical Report DCS 95-03, University of Pittsburgh, 1995.

[DE97] Min Dai and Christine Eisenbeis. Source to source software pipelining. *Parallel and Distributed Computing and Networks (PDCN'97)*, August 1997.

[DGS93] E. Duesterwald, R. Gupta, and M.L. Soffa. A practical data flow framework for array reference analysis and its application in optimizations. *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 68–77, June 1993.

[PW92a] W. Pugh and D. Wonnacott. Eliminating false data dependences using the omega test. *Proc. of the SIGPLAN 1992 (PLDI)*, pages 140–151, 1992.

[PW92b] W. Pugh and D. Wonnacott. Static analysis of upper and lower bounds on dependences and parallelism. Technical Report CS-TR-3250, University of Maryland, 1992.

[RF93] B. R. Rau and J. A. Fisher. Instruction-level parallel processing : History, overview, and perpective. *The Jornal of Supercomputing*, 7:9–50, 1993.

[WEJS94] J. Wang, C. Eisenbeis, M. Joudan, and B. Su. DEcomposed Software Pipeline : a new persective and a new approach. *International Journal on Parallel processing*, 22(3):357–379, 1994.

[WT92] M. J. Wolfe and C.W. Tseng. The power test for data dependence. *IEEE Transaction On Parallel*, 3(5), September 1992.

Chapter 4

Targetted Global Optimisation

Abstract

This paper is concerned with integrating global data transformations and local loop transformations in order to minimise overhead. By first developing an extended algebraic transformation framework, a new technique to allow the static application of global data transformations to reshaped arrays is presented, eliminating the need for expensive temporary copies and hence eliminating any communication and synchronisation. In addition, by integrating loop and data transformations, poor spatial locality and expensive array subscripts that may have been introduced can be eliminated. A specific optimisation algorithm is derived where initially experiments show it to give a significant improvement in execution time over existing approaches

4.1 Introduction

In order to achieve acceptable performance on current distributed shared memory machines, it is essential to make efficient use of the memory hierarchy and minimise overhead. Typically, a loop based approach [24] is used which is local in nature, as each loop nest is separately examined and optimised. Although each loop nest in isolation may perform well, they may perform poorly when combined due to significant communication between loop nests.

Another approach is to consider data orientated techniques [13], traditionally developed for distributed memory compilation but also used for distributed shared memory [1, 9]. This approach is primarily concerned with mapping arrays to processors and has a global, program wide, effect. It tries to globally trade off costs for an entire program, in contrast to loop based approaches. However, this global approach breaks down when a particular

array has two different layouts, for instance, when it is reshaped at a subroutine boundary. At present, there is no efficient static means to apply data transformations to reshaped arrays. Expensive temporary copies must normally be made at run-time on entry to the procedure and restored on exit [7], introducing additional communication.

Another problem associated with global data transformations is that in determining an optimisation that is globally acceptable, it may have an adverse affect on the performance of a particular statement within a loop nest. For instance, data alignment may transpose the layout of one array, reducing overall memory access, but in certain loop nests cause poor stride access through the local data cache by destroying spatial locality.

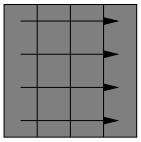
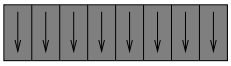
These two problems of data transformations, namely, reshaped arrays and the adverse effect on loop nests, are the subject of this paper. By developing an extended framework for loop and data transformations, we have developed a technique to statically determine the data layout for reshaped arrays, eliminating the need for temporary copies and the associated overhead. To solve the second problem of the (potential) local adverse effects of global data transformations on loop nests, loop transformations can be used to undo these adverse effects.

In order to optimise programs globally, it is essential that a compiler is able to combine loop and data transformations. Recent work [1, 6, 10, 22] has focussed on combining the loop based with data layout approaches [13] in order to trade-off these conflicting requirements. However, these approaches are restricted in that transformations, including strip-mining and linearisation, cannot be directly incorporated within their representation.

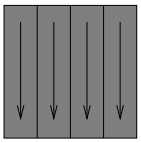
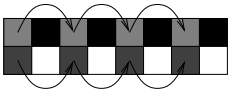
This paper develops a new approach to combining loop and data transformations, introducing rank modifying transformations which allow generalised linearisation and strip-mining of loop and data spaces. Its main practical use is that it allows, for the first time, the static application of data transformations, such as global index reordering and data strip-mining to reshaped arrays. Applying global data transformations to reshaped arrays can, however, produce complex and inefficient code but, by integrating loop and data transformations, we may also systematically eliminate any complex array access function introduced. This dramatically improves spatial locality by restructuring data to have stride-1 access wherever possible.

This paper is organised as follows. In the next section, a motivating example showing the applicability of our integrated approach is presented. Section 3 presents the notation used within this paper and is followed by section 4 which describes the form and properties of our novel rank modifying transformations. Sections 5 and 6 develop techniques for efficient data layout propagation which is followed in section 7 by a small experiment showing the significant improvement of such a scheme. Section 8 briefly reviews related work and is followed by some concluding remarks.

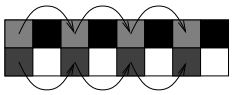
Original Code (1)	Reshaped Array (2)	Original Access (3)	Reshaped Access (4)

`REAL A(0:3,0:3)`
`Do i = 0, 2`
`  Do j = 1,3`
`  A(i,j) = A(i+1,j-1)`
`  + D(j,i)`
`  Enddo`
`Enddo`
 	
`call Reshape(a)`
	`Subroutine Reshape (B)`
	`REAL B(0:1,0:7)`
	`Do j = 0, 7`
	`  Do i = 0,1`
	`  B(i,j) = B(i,j) + 1`
	`  Enddo`
	`Enddo`
		 a	 b

After Data Transformation(5)	**Propagated Transformation (6)**	**New Access (7)**	**New Reshaped Access (8)**

`REAL A(0:3,0:3)`
`Do i = 0,2`
`  Do j = 1,3`
`  A(j,i) = A(j-1,i+1)`
`  + D(j,i)`
`  Enddo`
`Enddo`
 	
`call Reshape(A)`
	`Subroutine Reshape (B)`
	`REAL B(0:1,0:7)`
	`Do j = 0, 7`
	`  Do i = 0,1`
	`  B(mod(4*mod((2*j+i),4)`
	`  +(2*j+i)/4,2),`
	`  (4* mod((2*j+i),4)`
	`  +(2*j+i)/4)/2)  + = 1`
	`  Enddo`
	`Enddo`
		 a	 b

Loop Restructuring (9)	More Loop Restructuring (10)	Access Pattern(11)

`Subroutine Reshape (B)`
`REAL B(0:1,0:7)`
`Do j = 0, 3`
`  Do i = 0,3`
`  B(mod(j,2),2*i+j/2) += 1`
`  Enddo`
`Enddo`
	`Subroutine Reshape (B)`
	`REAL B(0:1,0:7)`
	`Do j1 = 0,1`
	`  Do j2 = 0,1`
	`  Do i = 0,3`
	`  B(j2,2*i+j1) +=1`
	`  Enddo`
	`  Enddo`
	`Enddo`
		 b

Data Restructuring (12)	**Loop and Data Restructuring (13)**	**Access Pattern(14)**

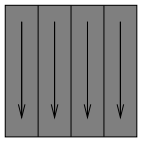
`Subroutine Reshape (B)`
`REAL B(0:3,0:3)`
`Do j = 0, 7`
`  Do i = 0,1`
`  B(j/2,i+2*mod(j,2)) += 1`
`  Enddo`
`Enddo`
	`Subroutine Reshape (B)`
	`REAL B(0:3,0:3)`
	`Do j1 = 0,1`
	`  Do i = 0,1`
	`  Do j2 = 0,3`
	`  B(j2,i+2*j1) += 1`
	`  Enddo`
	`  Enddo`
	`  Enddo`
	`Enddo`
		 b

Figure 4.1: Transformations

4.2 Example

In this section, we consider a simple example to illustrate the two probelems tackled in this paper: application of data transformations to reshaped arrays and the impact of global data transformations on local loop structure. To illustrate one of the applications, consider the program fragment in figure 1, box 1. In Fortran arrays are stored column-wise and therefore the reference to array `A` has a non stride-1 access pattern. This is shown in the diagram in box 3. The program fragment in box 2 contains references to `B` which are reshaped reference to `A`[1]. Here, however, there is perfect stride-1 access to array `B` as shown in box 4. One way to improve the access to array `A`, is to interchange the two loops. Loop permutation of the code in box 1 is not possible, however, due to the data dependence $[1, -1]^T$and thus a compiler may wish to apply a data transformation described in [22] to ensure stride-1 access. The necessary data transformation is a simple permutation matrix which, when applied to array `A`, gives the code in box 5 with the corresponding stride-1 access pattern in box 7. Such an approach has the benefit of always being legal, though its effect must be propagated to all accesses, including reshaped ones. Existing approaches [7] do not propagate information when there is reshaping and prefer either to insert a copy or prevent the data restructuring. Using the techniques described in section 6, we can apply the data transformation to the reshaped array giving the code in box 6 and the access pattern shown in 8. Although correct, it is immediately apparent that such a code structure and access pattern will be expensive, outweighing any benefit to the improved stride access to `A`. Even after application of the strength-reduction techniques for `mod` operations described in [1], calculation of the access function of `B` will be prohibitive. Furthermore, this will not improve the pathological "leapfrog" access to `B`.

The second table in figure 1 shows different attempts to improve the loop structure and data access pattern of the code in box 6. Using a combination of strip-mining and linearisation, the loop can be transformed to that shown in box 9. The access pattern, shown in box 11, is unchanged, but the access function is considerably simplified. By the application of a further rank modifying loop transformation, we have the code in box 10, where all `mod` and `div` operators have been removed. This, however, does not affect the data access pattern shown in box 11. If instead a data transformation is applied to `B`, we have the code shown in box 12. Finally, if both loop and data transformations are combined, we obtain the code in box 13 which has stride-1 access shown in box 14. Thus, by a combination of rank modifying data and loop transformations, we have stride-1 access on all arrays without excessively expensive access functions. In section 7 we develop automatic techniques which select the appropriate transformations and show the impact of such restructuring on actual performance.

[1]Such aliasing can occur either due to equivalencing (as shown), or more usually, due to reshaping across subroutine boundaries.

4.3 Notation

In this section, we briefly describe the notation used to develop the transformation framework. It is based on an algebraic representation of program constructs.

4.3.1 Iteration Spaces

The iterators in a Fortran program, surrounding any statement, can be represented as an $m \times 1$ column vector

$$J = [j_1, j_2, \ldots, j_m]^T$$

where m is the number of enclosing loops or iterators. The loop range or affine bounds of the iterators can be described by a system of inequalities defining the *polyhedron*

$$\mathbf{B}J \leq \mathbf{b} \tag{4.1}$$

where $\mathbf{B}$ is a $(\ell \times m)$ integer matrix and $\mathbf{b}$ an $(\ell \times 1)$ vector, for some ℓ. The integer values taken on by J define the *iteration space* of the iterators.

4.3.2 Index Spaces

The data storage of an array `A` can also be viewed as a polyhedron. We introduce *formal indices* $\mathcal{I}$ for the array to describe the array index domain

$$\mathcal{I} = [i_1, i_2, \ldots, i_N]^T$$

where N is the dimension of array `A`. The formal indices have a certain range which describe the size of the array, or *index space*, as follows:

$$\mathbf{A}\mathcal{I} \leq \mathbf{a} \tag{4.2}$$

where $\mathbf{A}$ is a $(\ell \times N)$ integer matrix and $\mathbf{a}$ an $(\ell \times 1)$ vector, for some ℓ. The integer values taken on by $\mathcal{I}$ define the *index space* of the indices.

4.3.3 Array Accesses

The subscripts in a reference to an array `A` represent a function that maps the values of the iteration space to the index domain. If J is the iteration vector, we assume in this paper that these subscripts can be written in the following form:

$$\mathcal{U}J + u \tag{4.3}$$

where $\mathcal{U}$ is a $N \times m$ matrix and u is a $N \times 1$ vector. All access vectors and loop and array bounds are assumed to start from 0 throughout this paper, for ease of presentation. The transformations can be trivially extended to encompass non-zero offsets.

4.4 Rank Modifying Transformations

In this section, we first describe the form and properties of rank modifying transformations. This is followed by an illustrative example.

4.4.1 Data Transformations

A data transformation is applied to the index space of a particular array and *all* accesses to that array throughout the program and is therefore *global* in nature. Rank-decreasing data transformations, such as array linearisation, are frequently used in FORTRAN programs.

A $(k \times N)$ linearisation matrix L is a transformation which maps an N dimension index vector $\mathcal{I}$ to a new k dimension space $\mathcal{I}'$.

$$\mathcal{I}' = L\mathcal{I} \tag{4.4}$$

Each array access $\mathcal{U}$ must be globally updated such that

$$\mathcal{U}' = L\mathcal{U} \tag{4.5}$$

Data transformations are therefore *left-hand* transformations when applied to array access functions. The new bounds of the new iteration space must also be determined. They are of the form:

$$\mathbf{A}'\mathcal{I}' \leq \mathbf{a}' \tag{4.6}$$

where

$$\mathbf{A}' = X\mathbf{A}L^{\dagger} \quad \text{and} \quad \mathbf{a}' = X\mathbf{a} \tag{4.7}$$

and

$$X = \begin{bmatrix} L & O \\ O & L \end{bmatrix} \tag{4.8}$$

In equation (4.7), $L^{\dagger}$ is a transformation that is inverse to L on the index space of `A`. That is, for every index point $\mathcal{I}$ of `A`, $L^{\dagger}(L(\mathcal{I})) = \mathcal{I}$. We call such an inverse transformation $L^{\dagger}$ a *local inverse* for L on the index space of `A`.

For rank decreasing transformations, the only additional condition to be added is that $k < N$, i.e., the new index space is smaller than the original. The equations for applying a singular transformation that increases the number of dimensions of an array are precisely

```
Real A(0:3,0:7), B(0:3,0:7)

Do j = 0,3
 Do i = 0,3
  A(i,j) = A(j,i) + B(i,j)
 Enddo
Enddo
```

Figure 4.2: Example loop

	Data	Loop	Combined

$\begin{bmatrix} 1 & 4 \end{bmatrix}$	`Real A(0:31), B(0:3,0:7)`
	
	`Do j = 0,3`
	`  Do i = 0,3`
	`  A(i+4*j) = A(j+4*i) +B(i,j)`
	`  Enddo`
	`Enddo`
		`Real A(0:3,0:7), B(0:3,0:7)`
		
		`Do i = 0,15`
		`  A(mod(i,4),i/4) = A(i/4,mod(i,4))`
		`  + B(mod(i,4),i/4)`
		`Enddo`
			`Real A(0:31), B(0:3,0:7)`
			
			`Do i = 0,15`
			`  A(i) = A(i/4+4*(mod(i,4))`
			`  + B(mod(i,4)+4*(i/4))`
			`Enddo`

$\begin{bmatrix} (\cdot)\%2 \\ (\cdot)/2 \end{bmatrix}$	`Real A(0:1,0:1,0:7), B(0:3,0:7)`
	
	`Do j = 0,3`
	`  Do i = 0,3`
	`  A(mod(i,2),i/2,j) =`
	`  A(mod(j,2),j/2,i) + B(i,j)`
	`  Enddo`
	`Enddo`
		`Real A(0:3,0:7), B(0:3,0:7)`
		
		`Do j = 0,3`
		`  Do i2 = 0,1`
		`  Do i1 = 0,1`
		`  A(i1+2*i2,j) = A(j,i1+2*i2)`
		`  + B(i1+2*i2,j)`
		`  Enddo`
		`  Enddo`
		`Enddo`
			`Real A(0:1,0:1,0:7), B(0:3,0:7)`
			
			`Do j2 = 0,1`
			`  Do j1 = 0,1`
			`  Do i2 = 0,1`
			`  Do i1 = 0,1`
			`  A(i1,i2,j1+2*j2)`
			`  = A(j1,j2,i1+2*i2)`
			`  + B(i1+2*i2,j1+2*j2)`
			
			`  Enddo`
			`  Enddo`
			`  Enddo`
			`Enddo`

Figure 4.3: Loop and Data Transformations

the same form as that for rank-decreasing. In the case of rank increasing matrices used, for instance, in strip-mining $k > N$. For convenience we will denote such matrices by S.

4.4.2 Loop Transformations

Loop transformations are applied to the iterators in a loop nest and to all array accesses within a loop nest and are thus *local* in nature. Rank-decreasing loop transformations, such as loop collapsing, are used to reduce loop overhead while rank-increasing ones, such as tiling, are frequently used to exploit locality.

A $(k \times m)$ linearisation matrix L is a transformation which maps an m dimensional index

vector J to a new k dimensional vector J'.

$$J' = LJ \tag{4.9}$$

Each access $\mathcal{U}$ within the loop nest must be updated such that

$$\mathcal{U}' = \mathcal{U}L^{\dagger} \tag{4.10}$$

Thus, loop transformations are *right-hand* acting transformations when applied to array accesses.

The new bounds of the new iteration space must be determined and are of the form:

$$\mathbf{B}'J' \leq \mathbf{b}' \tag{4.11}$$

where

$$\mathbf{B}' = X\mathbf{B}L^{\dagger} \text{ and } \mathbf{b}' = X\mathbf{b} \tag{4.12}$$

and

$$X = \begin{bmatrix} L & O \\ O & L \end{bmatrix} \tag{4.13}$$

Once again, the only additional condition to be added is that $k < N$ for rank-decreasing transformations and $k > N$ for rank increasing ones, which will be denoted by S.

4.4.3 Form of Transformation

In this paper, we restrict attention to generalised strip-mining and linearisation. In order to describe such transformations in an algebraic framework, it is necessary that the transformation matrices may now include integer division and modulo operations as well as integers. If we need to include the operation "divide by n" as an entry in a matrix, we will write this entry as $(\cdot)/n$. Likewise, we write $(\cdot)\%n$ for the "modulo n" operation.

We need to define how to calculate with these extended matrices. Briefly, if we multiply a matrix with such entries with a vector, we simply substitute the values of the vector elements into the operation. For example,

$$\begin{bmatrix} (\cdot)\%4 & 1 \\ (\cdot)/4 & 2 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 2\%4+3 \\ 2/4+6 \end{bmatrix} = \begin{bmatrix} 5 \\ 6 \end{bmatrix} \tag{4.14}$$

On the other hand, if we multiply such a matrix by another matrix, we multiply the input or result of the operation by the appropriate integer. For example,

$$\begin{bmatrix} 1 & 4 \end{bmatrix} \times \begin{bmatrix} (\cdot)\%4 \\ (\cdot)/4 \end{bmatrix} = [(\cdot)\%4 + 4 * ((\cdot)/4)] \tag{4.15}$$

This transformation maps an integer n to $n\%4 + 4 * (n/4)$. It can easily be checked that $n\%4 + 4 * (n/4) = n$ and hence this transformation can be replaced by the 1×1 identity matrix.

Now, on the index space of `A(0:3,0:7)`,

$$\begin{bmatrix} (\cdot)\%4 \\ (\cdot)/4 \end{bmatrix} \times \begin{bmatrix} 1 & 4 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{4.16}$$

Hence these matrices are inverse to one another over the index space of `A`. In general, in this paper we will use the following rank modifying matrices.

$$L = \begin{bmatrix} 1 & n_1 & n_1 \times n_2 & \dots & n_1 \times \cdots \times n_{m-1} \end{bmatrix} \tag{4.17}$$

and

$$L^\dagger = \begin{bmatrix} (\cdot)\%n_0 \\ ((\cdot)/n_0)\%n_1 \\ \vdots \\ ((\cdot)/(n_0 \times \cdots \times n_{m-1}))\%n_m \end{bmatrix} \tag{4.18}$$

where $n_0, \dots, n_{m-1}$ are constants corresponding to the size of the appropriate index/iterator dimension. It is clear that L and $L^\dagger$ are local inverses.

4.4.4 Example

To illustrate this formulation, consider the program in figure 4.2 and the following transformation which maps the 2 dimensional array `A` to a 1 dimensional linearised form:

$$L = \begin{bmatrix} 1 & 4 \end{bmatrix} \quad \text{and} \quad L^\dagger = \begin{bmatrix} (\cdot)\%4 \\ (\cdot)/4 \end{bmatrix} \tag{4.19}$$

In the previous section we have shown that these matrices are local inverses. The array accesses to `A` are updated thus:

$$\begin{bmatrix} 1 & 4 \end{bmatrix} \times \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} j \\ i \end{bmatrix} = \begin{bmatrix} 4 & 1 \end{bmatrix} \begin{bmatrix} j \\ i \end{bmatrix} \tag{4.20}$$

$$\begin{bmatrix} 1 & 4 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} j \\ i \end{bmatrix} = \begin{bmatrix} 1 & 4 \end{bmatrix} \begin{bmatrix} j \\ i \end{bmatrix} \tag{4.21}$$

i.e., `A(i,j)` $\mapsto$ `A(i+4*j)` and `A(j,i)` $\mapsto$ `A(j+4*i)`.

The new index space $\mathbf{A}'\mathcal{I}' \leq \mathbf{a}'$ is calculated as follows:

$$\mathbf{A}' = \begin{bmatrix} 1 & 4 & 0 & 0 \\ 0 & 0 & 1 & 4 \end{bmatrix} \left[\begin{array}{cc} -1 & 0 \\ 0 & -1 \\ \hline 1 & 0 \\ 0 & 1 \end{array}\right] \begin{bmatrix} (\cdot)\%4 \\ (\cdot)/4 \end{bmatrix} = \left[\begin{array}{c} -1 \\ \hline 1 \end{array}\right] \tag{4.22}$$

$$\mathcal{I}' = \begin{bmatrix} 1 & 4 \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} = \begin{bmatrix} i'_1 \end{bmatrix} \tag{4.23}$$

$$\mathbf{a}' = \begin{bmatrix} 1 & 4 & 0 & 0 \\ 0 & 0 & 1 & 4 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ \hline 3 \\ 7 \end{bmatrix} = \begin{bmatrix} 0 \\ \hline 31 \end{bmatrix} \tag{4.24}$$

giving

$$\begin{bmatrix} -1 \\ \hline 1 \end{bmatrix} \begin{bmatrix} i'_1 \end{bmatrix} \leq \begin{bmatrix} 0 \\ \hline 31 \end{bmatrix} \tag{4.25}$$

i.e., `A(0:31)`. The updated array access and index space are shown in figure 4.3, row 1, column 2. The application of a similar loop transformation is shown in column 3.

4.4.5 Combining Loop and Data Transformations

Although our formulation allows rank modifying loop and data transformations to be applied independently, in practice they are often combined. For instance, the strip-mining data transformation gives rise to access functions containing `div` and `mod`, both of which will be prohibitively expensive. If, however, the surrounding loop were also strip-mined, the accesses would become simplified. For example, consider the code in figure 4.3, row 2, column 2, after the application of a rank-increasing data transformation. If a related *loop transformation* is subsequently applied, we have the program shown in row 2, column 4, where the accesses to `A` are now simplified. Thus, this framework allows a natural method to apply dimension changing transformations, frequently avoiding the use of `div` and `mod` without the need for special optimisations akin to strength reduction as described in [1].

4.5 Handling Data Transformations in the Presence of Linearised Arrays

The previous sections have described the form of rank-modifying transformations. This section illustrates the usefulness of such a framework by combining rank-modifying transformations with more standard non-singular ones. It develops a technique to allow the application of data transformations to linearised arrays without incurring excessive communication and synchronisation overhead.

The first subsection describes a technique to apply data transformations to linearised arrays. This is followed by a section describing how rank modifying loop transformation can remove some of the introduced `div` and `mod` operators.

Original	Data Transformation	Loop Transformation

`Real A(0:1,0:3) Real B(0:7)`
`Equivalence(A,B)`
`.`
`.`
`Do j = 0,3`
`  Do i = 0,1`
`  A(i,j)=B(i+2*j)+C(i,j)`
`  Enddo`
`Enddo`
`Do j = 0,7`
`  B(j) = j*3`
`Enddo`
	`Real A(0:1,0:4) Real B(0:9)`
	`Equivalence(A,B)`
	`.`
	`.`
	`Do j = 0,3`
	`  Do i = 0,1`
	`  A(i,i+j)=B(3*i+2*j)+C(i,j)`
	`  Enddo`
	`Enddo`
	`Do j = 0,7`
	`  B(3*mod(j,2) + 2*(j/2)) =`
	`  9*mod(j,2) + 6*(j/2)`
	`Enddo`
		`Real A(0:1,0:4) Real B(0:9)`
		`Equivalence(A,B)`
		`.`
		`.`
		`Do j = 0,3`
		`  Do i = 0,1`
		`  A(i,i+j)=B(3*i+2*j)+C(i,j)`
		`  Enddo`
		`Enddo`
		`Do j = 0,3`
		`  Do i = 0,1`
		`  B(3*i + 2*j) = 9*i+6*j`
		`  Enddo`
		`Enddo`

Figure 4.4: Linearised Aliasing

4.5.1 Data Transformations on Linearised Arrays

Data transformations are left-hand transformations that must be applied to every reference to the particular array throughout the program. Difficulties occur when references to a linearised array access exist. Let $\mathcal{I}_1$ be the index domain of the array to be transformed and $\mathcal{I}_2$ be the the linearised domain:

$$\mathcal{I}_2 = L\mathcal{I}_1 \tag{4.26}$$

where L is the linearising transformation. We therefore have:

$$\mathcal{I}_1 = L^{\dagger}\mathcal{I}_2 \tag{4.27}$$

If we wish to apply a non-singular data transformation $\mathcal{A}$ globally [22], this gives the new index domain $\mathcal{I}_1'$:

$$\mathcal{I}_1' = \mathcal{A}\mathcal{I}_1 \tag{4.28}$$

and analgeous to equation (4.26) we have

$$\mathcal{I}_2' = L'\mathcal{I}_1' \tag{4.29}$$

where L' is the appropriate linearisation and therefore

$$\mathcal{I}_2' = L'\mathcal{I}_1' = L'\mathcal{A}\mathcal{I}_1 = L'\mathcal{A}L^{\dagger}\mathcal{I}_2 \tag{4.30}$$

Thus, when applying $\mathcal{A}$ to the index domain $\mathcal{I}_1$, we must apply $L'\mathcal{A}L^{\dagger}$ to the linearised domain $\mathcal{I}_2$. Now, given two references $\mathcal{U}_1$ and $\mathcal{U}_2$, where $\mathcal{U}_2$ is a linearised reference, then on applying $\mathcal{A}$ we have as usual $\mathcal{U}_1' = \mathcal{A}\mathcal{U}_1$. However, for the linearised access we have

$$\mathcal{U}_2' = L'\mathcal{A}L^{\dagger}\mathcal{U}_2 \tag{4.31}$$

The linearisation transformations L and L' and their inverses are readily derived from the array bounds before and after applying $\mathcal{A}$. For example, in the case of a two dimensional array we have

$$L = \begin{bmatrix} 1 & n \end{bmatrix} \text{ and } L' = \begin{bmatrix} 1 & n' \end{bmatrix} \tag{4.32}$$

To illustrate this, consider the example in figure 4.4, column 1. Let

$$\mathcal{A} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \tag{4.33}$$

i.e., a simple data skew which is applied to the array `A`. The access to array `A` in the first loop is readily found:

$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} j \\ i \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} j \\ i \end{bmatrix} \tag{4.34}$$

i.e., `A(i,i+j)` which is shown in figure 4.4, column 2. To update the equivalenced access to array `B`, we need to determine L' and $L^{\dagger}$. Examining the original array bounds we have:

$$L = \begin{bmatrix} 1 & 2 \end{bmatrix} \text{ and } L^{\dagger} = \begin{bmatrix} (\cdot)\%2 & (\cdot)/2 \end{bmatrix}^T \tag{4.35}$$

Applying $\mathcal{A}$ gives the new array bounds `A(0:1,0:4)` hence

$$L' = \begin{bmatrix} 1 & 2 \end{bmatrix} \tag{4.36}$$

If we apply $L'\mathcal{A}L^{\dagger}$ to the linearised access to `B` in the first loop we get:

$$\begin{bmatrix} 1 & 2 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} (\cdot)\%2 \\ (\cdot)/2 \end{bmatrix} \times [2,1] \times \begin{bmatrix} j \\ i \end{bmatrix} = [2*j, 3*i] \tag{4.37}$$

i.e., `B(3*i + 2*j)` which is shown in figure 4.4 column 2. Repeating the procedure for the second loop we have:

$$\begin{bmatrix} 1 & 2 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} (\cdot)\%2 \\ (\cdot)/2 \end{bmatrix} \times [1] \times [j] = [3*(j\%2) + 2*(j/2)] \tag{4.38}$$

i.e., `A(3*mod(j,2) + 2*(j/2))` Thus, our formulation allows systematic application of data transformations such as data alignment even in the presence of linearised array accesses.

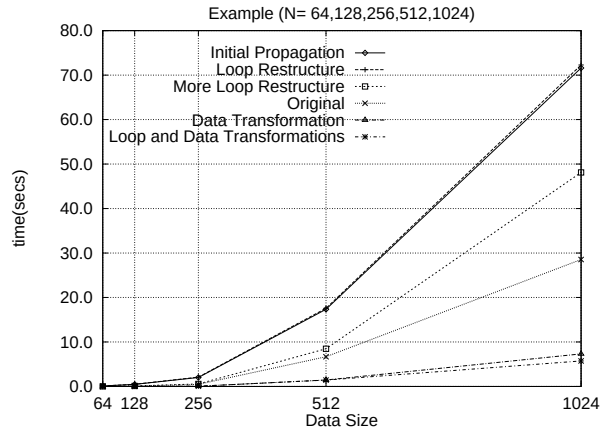


Figure 4.5: N = 64... 1024

4.5.2 Reducing Access Overhead for Linearised Arrays

Although correct, the program in figure 4, column 2 is far from ideal due to the `mod` and `div` operators. We wish to remove these by introducing two new iterator variables corresponding to `mod(j,2)` and `j/2` which is achieved by strip-mining.

The `mod` and `div` operators are introduced by rank-increasing transformation such as S or, in this case, the $L^\dagger$ matrix in equation (4.30). If we can apply a transformation such that this is eliminated, then the corresponding `mod(j,2)` and `j/2` will be eliminated.

Let T be defined as follows:

$$T = \mathcal{U}_2^{-1} L^\dagger \mathcal{U}_2 \tag{4.39}$$

and hence

$$T^{-1} = \mathcal{U}_2^{-1} L \mathcal{U}_2 \tag{4.40}$$

Applying this transformation to the loop and array accesses, using equation (4.31) gives the new access matrix:

$$\mathcal{U}_2' T^{-1} = L' \mathcal{A} L^\dagger \mathcal{U}_2 \mathcal{U}_2^{-1} L \mathcal{U}_2 = L' \mathcal{A} \mathcal{U}_2 \tag{4.41}$$

which is free from any rank-increasing matrices. Applying such a transformation to the second loop gives the code in column 3 of figure 4.4. Thus, by combining loop and data transformations within one framework, we can can readily restructure programs so as to partially undo the effect of previous transformation applications.

4.6 Handling Data Transformations in the Presence of Reshaped Arrays

This section extends the results of the previous section by developing a technique to allow the application of data transformations to reshaped arrays without incurring excessive communication and synchronisation overhead. However, this is at the cost of complex subscripts and poor strides access and we therefore examine loop transformations to reduce introduced overheads due to global data transformations.

4.6.1 Data Transformations for Reshaped Arrays

Application of data transformations for linearised accesses is relatively straightforward in the sense that it is easy to determine both L' and $L^{\dagger}$. Difficulties occur with reshaped arrays in that the shape of the array after application of a data transformation is not fixed, i.e., there are several legal new array layouts. A reshaped array access can be considered to be described by the following equation:

$$\mathcal{I}_2 = SL\mathcal{I}_1 \tag{4.42}$$

The reshaped domain $\mathcal{I}_2$ is considered to be constructed by first linearising $\mathcal{I}_1$ to a flat one-dimensional array which is then strip-mined to the appropriate dimension and size. The value of L is readily available given $\mathcal{I}_1$, so for any $\mathcal{I}_2$, S is easily derived. For instance, if an array is declared of size `A(0:7,0:3)` in the main program but reshaped to size `A(0:1,0:15)` in a subroutine we have:

$$L = [1, 8] \quad \text{and} \quad S = \begin{bmatrix} (\cdot)\%2 \\ (\cdot)/2 \end{bmatrix} \tag{4.43}$$

Note that $S \times L \neq I$ otherwise no reshaping takes place. We therefore have

$$\mathcal{I}_1 = L^{\dagger} S^{\dagger} \mathcal{I}_2 \tag{4.44}$$

If we wish to apply a data transformation $\mathcal{A}$, this gives the new index domain $\mathcal{I}_1'$

$$\mathcal{I}_1' = \mathcal{A}\mathcal{I}_1 \tag{4.45}$$

and analogeous to equation 4.42 we have

$$\mathcal{I}_2' = S'L'\mathcal{I}_1' \tag{4.46}$$

for some S' and L'. Therefore

$$\mathcal{I}_2' = S'L'\mathcal{I}_1' = S'L'\mathcal{A}\mathcal{I}_1 = S'L'\mathcal{A}L^{\dagger}S^{\dagger}\mathcal{I}_2 \tag{4.47}$$

An access $\mathcal{U}_2$ to the reshaped array is transformed to $\mathcal{U}_2'$:

$$\mathcal{U}_2' = S'L'\mathcal{A}L^{\dagger}S^{\dagger}\mathcal{U}_2 \tag{4.48}$$

We have S, L and thus $L^{\dagger}$. Again, L' is readily determined after applying $\mathcal{A}$ to $\mathcal{I}_1$. The difficulty occurs in determining S' as there are no restrictions on its form except for legality. In other words, the new dimensions of the reshaped array are not fixed after applying a data transformation on the original domain. To illustrate this, consider the original program in figure 4.1, box 1 and the reshaped access to array `A`, namely, `B` in box 2. Here

$$L = [1, 4] \text{ and } L^{\dagger} = \begin{bmatrix} (\cdot)\%4 \\ (\cdot)/4 \end{bmatrix} \tag{4.49}$$

$$S = \begin{bmatrix} (\cdot)\%2 \\ (\cdot)/2 \end{bmatrix} \text{ and } S^{\dagger} = [1, 2] \tag{4.50}$$

If we apply the permutation matrix

$$\mathcal{A} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{4.51}$$

then

$$L' = [1, 4] \tag{4.52}$$

As stated before, we have freedom with S'. If we choose $S' = S$, i.e., preserving the original shape of the array, we have the program in figure 4.1, box 6. As is immediately apparent, though correct, this access function to `B` will be extremely costly.

4.6.2 Reducing Access Overhead for Reshaped Arrays

In this subsection we examine two methods whereby loop transformations may be used to reduce any introduced overhead due to singular matrices. Just as in the case of linearised arrays, we can remove some of the overhead due to `mod`s etc. by applying a loop transformation to remove some of those rank-increasing transformations which introduce `mod`s. Let T be the loop transformation to remove these operators

$$T = \mathcal{U}_2^{-1}L^{\dagger}S^{\dagger}\mathcal{U}_2 \tag{4.53}$$

1. Given the original and reshaped array indices $\mathcal{I}_1$ and $\mathcal{I}_2$, determine the transformations L, S, $L^\dagger$ and $S^\dagger$.
2. Given the data layout transformation $\mathcal{A}$, update all reshaped array accesses such that $\mathcal{U}' = \mathcal{A}L^\dagger S^\dagger \mathcal{U}$
3. Update the reshaped array indices such that $\mathcal{I}_2' = \mathcal{A}L^\dagger S^\dagger \mathcal{I}_2$.
4. Let $T = \mathcal{A}L^\dagger$.
5. For each loop nest containing reference to the array, apply the loop transformation T, if it is legal to do so.

Figure 4.6: Propagation Algorithm

and hence

$$T^{-1} = \mathcal{U}_2^{-1} S L \mathcal{U}_2 \tag{4.54}$$

Applying such a transformation would have the following affect on the access $\mathcal{U}_2'$ from equation (4.48)

$$\mathcal{U}_2' T^{-1} = S' L' \mathcal{A} L^\dagger S^\dagger \mathcal{U}_2 T^{-1} = S' L' \mathcal{A} \mathcal{U}_2 \tag{4.55}$$

If this transformation is applied to the code in box 6 in figure 1, we produce the program shown in box 9. Although this removes some of the expensive functions, there still remain `mod`s due to S'. A possibly more straightforward approach would be to select the reshaping matrix S' to be $S' = L'^\dagger$. This has the effect that arrays are remapped into the same shape as the original array when applying a propagated data transformation $\mathcal{A}$. Thus equation (4.47) simplifies to:

$$\mathcal{I}_2' = S' L' \mathcal{I}_1 = S' L' \mathcal{A} \mathcal{I}_1 = \mathcal{A} L^\dagger S^\dagger \mathcal{I}_2 \tag{4.56}$$

If we applied this data transformation to the code in box 6 of figure 1 we arrive at the code in box 12. Finally, a simple method to improve performance is to examine the array access matrix and then strip-mine those iterators which are arguments of `divs` or `mod`, and reorder the iterators to give the code in box 13 with the stride-1 access shown in box 14. In other words, apply a loop transformation $T = \mathcal{A}L^\dagger$ and update the loop bounds and array accesses accordingly.

4.7 Experiments

In this section we develop a simple algorithm to eliminate the overhead associated with reshaped arrays and apply the technique to three SPEC benchmarks demonstrating the usefulness of our approach.

4.7.1 Algorithm

In this subsection, we develop an algorithm for propagating data transformations to reshaped arrays. Given the techniques developed in sections 5 and 6, it is relatively straightforward to incorporate then into a compiler phase order. Consider the algorithm in figure 4.6. It is applied after the global data partitioning/alignment has been chosen and after barrier synchronisation placement has been determined, but before loop optimisations and code generation. Once our compiler has determined the global data layout, it must consider reshaped arrays (step 1). The application of the reshaped data transformation (step 2) is followed by the update of array declarations (steps 3) before a loop transformation is constructed (step 4) to remove any remaining `mod`s etc (step 5). As step 5 may reorder the loop nest, its legality must be checked before application.

4.7.2 Results

To show the use of the analysis developed in this section, we ran each of the program versions shown in figure 1 on the SGi Challenge for varying data sizes. Figure 4.5 shows their relative performance compared to the original program. The basic data propagation scheme is more than twice as poor as the original. For this reason, it is not suprising that compiler writers disable data transformations in the presence of reshaped arrays. Although subsequent loop transformations do improve performance, they still do not match the performance of the original. When both loop and data transformations are applied, we finally have an improvement over the original. For $N = 1024$ this improvement is by over a factor of 4.

4.8 Related Work

There is a large body of work concerned with improving program performance using program transformations. In [24], unimodular loop transformations are used to improve locality, which is extended to the non-singular case in [18]. These papers restrict themselves to loop transformations and do not consider cases where there are conflicting loop orders. In such instances, [20] proposes a simple heuristic, limited to just loop permutations, which is extended to the unimodular case in [3]. Although usefull, these approaches are limited in that they do not consider array layout transformations.

Other researchers have considered data transformations, primarily with respect to data alignment and partitioning. In [2, 8, 13, 17] approaches based on graph theory, integer programming and linear algebra are explored. Most of this work, however, considers alignment to be part of a mapping process rather than a program transformation and thus, parallels with loop transformation are absent. In [21], alignment as a program level trans-

formation is first presented while [16] uses a similar representation for uni-processor spatial locality. In [6], data linearisation transformations are considered as a means to change array layout, while data permutation and strip-mining transformations are considered in [1]. In [22], we developed a new framework describing non-singular data transformations equivalent in standing to loop transformations and described how they may be used to improve program performance. Again, these approaches are limited as they restrict themselves to just data transformations.

There has been recent work considering the combination of loop and data transformations in improving program performance. Using a hyperplane formulation of data transformations and non-singular loop transformations, an algorithm, which considers a restricted set of loop and data transformations, is proposed as a means of improving locality [6]. In [10, 11], a similar formulation is used, but considers a wider class of transformations. These approaches, however, have an asymmetric treatment of loop and data transformations. Furthermore, transformations such as iteration space [12] and data space tiling have not been integrated into the above work. In [1], data tiling is used to improve spatial locality, but the representation used does not allow easy integration with other data space transformations or general loop transformations. In this paper, we overcome this problem by developing an extended transformation framework based on rank modifying transformations.

In [7], the general case of array aliasing, particularly across array boundaries, is considered and preliminary techniques to recover the structure of linearised arrays are developed. They also develop ad hoc techniques to recover loop structure after data restructuring but cannot, at present, handle the application of data transformations, such as data partitioning, across aliased arrays. The techniques developed in this paper, however, allow the static application of data transformations, including array partitioning across reshaped arrays, providing the necessary results for [7].

4.9 Conclusion

In this paper, we have presented a new algebraic framework that allows the integration of loop and data transformations. This enables existing transformations to be described in a unifying manner and provides also the basis for new program optimisations. In particular, we develop techniques which allow the application of optimising data transformations to reshaped arrays without incurring excessively expensive code. Future work will consider integrating this with complete inter-procedural analysis.

There are, however, many other uses of such a framework such as auto-parallelisation and locality optimisation. The central problem of loop or data only approaches is that of balancing conflicting requirements throughout the program. More specifically, if one part of a program, be it a loop or an array access, requires a particular transformation but another

part requires a completely distinct transformation, how do we determine transformations that trade-off such requirements to give a globally acceptable result? Future work will investigate how conflicting requirements on a loop transformation may be resolved by using a data transformation and vice-versa. Future work will also further investigate the mathematical properties of the transformation representation used in this paper and, in particular, develop formal validity tests and investigate further optimisation algorithms.

Bibliography

[1] J.M. Anderson, S.P. Amarasinghe and M.S. Lam, **Data and Computation Transformations for Multiprocessors**, Proc. PPoPP, 1995.

[2] D. Bau, I. Kodukula, V. Kotlyar, K. Pingali and P. Stodghill, **Solving Alignment using Elementary Linear Algebra**, Proc. Seventh Annual workshop on Languages and Compilers for Parellel Computing, Ithaca, New York, 1994.

[3] A.J.C. Bik and P.M.W. Knijnenburg, **Reshaping Access Patterns for Improving Data Locality**, Proc. Sixth Workshop on Compilers for Parallel Computers, 1996.

[4] U. Banerjee, **Loop Transformations for Restructuring Compilers**, Kluwer Academic Publishers, 1993.

[5] F. Bodin and M.F.P. O'Boyle, **A Compiler Strategy for SVM** Third Workshop on Languages, Compilers and Runtime Systems, New York, Kluwer Press, May 1995.

[6] M. Cierniak and W. Li, **Unifying Data and Control Transformations for Distributed Shared-Memory Machines**, Proc. PLDI, 1995.

[7] M. Cierniak and W. Li, **Validity of Interprocedural Data Remapping**, Tech Rep 642, University of Rochester, 1996.

[8] M. Gupta, **Automatic Data Partitioning on Distributed- Memory Multicomputers**, PhD thesis, University of Illinois Urbana-Champaign, 1992.

[9] T.E.Jeremiassen and S.J. Eggers, **Reducing False Sharing on Shared Memory Multiprocessors through Compile-Time Data Transformations**, Proc. PPoPP, 1995.

[10] M. Kandemir, J. Ramanujam and A. Choudhary, **A Compiler Algorithm for Optimizing Locality in Loop Nests**, Proc. ICS, 1997.

[11] M. Kandemir, A. Choudhary, J. Ramanujam and P.Banerjee, **A Matrix Approach to the Global Locality Optimization Problem**, Proc. of PACT'98, IEEE Press, Paris, October 1998.

[12] I. Kodukla, N. Ahmed and K.Pingali, **Data-centric multi-level blocking**, in ACM SIGPLAN PLDI 1997.

[13] K. Kennedy and U. Kremer. **Automatic Data Layout for High Performance Fortran**, Proc. Supercomputing, 1995.

[14] D. Kulkarni and M. Stumm. **Loop and Data Transformations: A Tutorial** University of Toronto, Tech Rep CSRI-337, June 1993.

[15] P.M.W. Knijnenburg, E. Ayguadé and J. Torres. **Multi-transformations of Nested Loops for Parallelizing Compilers**, Tech. Rep. 96–14, Leiden University, 1996.

[16] S.-T. Leung and J. Zahorjan, **Optimizing Data Locality by Array Restructuring**, University of Washington, Department of Computer Science and Engineering, Tech Rep 95-09-01, September, 1995.

[17] J. Li and M. Chen, **Index Domain Alignment: Minimising Cost of Cross-Re ferencing between Distributed Arrays**, IEEE Proceedings of the Third Symposium on the Frontiers of Massively Parallel Computation, October 1990.

[18] W. Li and K. Pingali. **A Singular Loop Transformation Framework Based on Nonsingular Matrices**, Proc. of the Fifth Workshop on Languages and Compilers for Parallelism, August 1992.

[19] W. Li and K. Pingali. **A Singular Loop Transformation Framework Based on Nonsingular Matrices**, Int'l J. of Parallel Programming, **22**(2), pp. 183–205, 1994.

[20] K. McKinley, S Carr. and C-W Tseng, **Improving Data Locality with Loop Transformations** ACM TOPLAS 1996.

[21] M.F.P. O'Boyle and G.A. Hedayat, **Data Alignment: Transformation to Reduce Communication on Distribu ted Memory Architectures**, Proc. Scalable High Performance Computing Conference, *IEEE Press*, Virginia, April 1992.

[22] M.F.P. O'Boyle and P.M.W. Knijnenburg, **Non-Singular Data Transformations: Definition, Validity and Applications**, *accepted for publication in* the International Journal of Parallel Programming.

[23] M.F.P. O'Boyle and P.M.W. Knijnenburg, **Integrating Loop and Data Transformations for Global Optimisation**, Technical Report Leiden University, Dept. Computer Science, 98-08, July 1998.

[24] M.E. Wolf and M. Lam. **A Loop Transformation Theory and An Algorithm to Maximise Parallelism**, IEEE Transactions on Parallel and Distributed Systems 2(4), October 1991.