Memory Bank Disambiguation using Modulo Unrolling for Raw Machines

Rajeev Barua, Walter Lee, Saman Amarasinghe, Anant Agarwal [*]
M.I.T Laboratory for Computer Science
Cambridge, MA 02139, U.S.A
{barua,walt,saman,agarwal}@lcs.mit.edu
http://www.cag.lcs.mit.edu/raw

Abstract

This paper presents modulo unrolling, a code transformation technique for enabling array references to be accessed through the fast static network on a Raw machine. A Raw machine comprises of a mesh of simple, replicated tiles connected by an interconnect which supports fast, static near-neighbor communication. Like all other resources, memory is distributed across the tiles. Management of the memory can be performed by well known techniques which generate the requisite communication code on distributed address-space architectures. On the other hand, the fast, static network provides the compiler with a simple interface to optimize such communication. This paper addresses the problem of taking advantage of such static communication for memory accesses. The requirement for static memory communication is the compile-time knowledge of the exact communication required for each memory reference. This knowledge, in turn, can be obtained if a memory reference refers exclusively to memory residing on a single processing tile. We introduce modulo unrolling as a technique which allows the static communication of a large class of array accesses. We show how this technique achieves the goal of static communication by using a relatively small unroll factor. For a set of dense matrix scientific applications, we are able to access all the array references on the static network, enabling scalable speedups on the RAW machine.

1. Introduction

Architectures of modern microprocessors have attempted to increase performance by aggressively exploiting instruction-level parallelism (ILP). Yet designing a truly scalable architecture to exploit ILP has proved elusive. Increasing parallelism places increasing pressure on required register-file and memory bandwidth. Multi-ported register files and memories are only partial solutions because they do not scale. Current designs have explored complex memory systems with several memory banks connected with global buses, with arbitration performed by complex and non-scalable hardware logic. Run-time cost for this complexity is paid even when exact compile-time prediction of memory locations accessed is possible. Multiprocessors provide truly distributed resources, but they incur very high communication costs, restricting them to exploiting coarse-grained parallelism only.

The Raw machine [13] aims to provide truly distributed resources at communication costs low enough to exploit ILP. It distributes the register files, memories and ALUs across a two dimensional mesh of identical tiles, and it supports single-cycle near-neighbor communication through a programmable, software-controlled static network. Distributed memory provides scalable memory bandwidth, with fast access to memories of remote tiles through the static network. A slower dynamic network provides mechanism for compiler-unanalyzable accesses.

The use of a software-controlled static network to connect the distributed memory on Raw opens up new challenges and opportunities for the compiler. A major new task of the compiler is to predict the locations of memory accesses. If the compiler can predict which tile a memory access refers to, then the access can be made on the fast static network. This process is termed *static promotion*. For fallback, accesses which cannot be statically promoted are made on the slower dynamic network. The larger the class of accesses which can be statically promoted, the better the performance will be.

This paper presents modulo unrolling, a code transformation technique for the static promotion of a large class of array references. This technique is applicable for array accesses having indices which are affine functions of enclosing loop induction variables. These accesses are common in scientific codes, an important class of applications. We also present supporting memory-system optimizations for

[*]This research is funded in part by DARPA contract # DABT63-96-C-0036.

scientific code. The techniques have been implemented in RAWCC, the Raw parallelizing compiler. We present performance results of this compiler. We have also developed strategies to efficiently compile dynamic accesses on Raw, but they are beyond the scope of this paper.

Static promotion for Raw is related to the concept of *memory-bank disambiguation* [6] for distributed bank architectures, such as certain VLIWs. Techniques for memory-bank disambiguation also aim to determine the memory bank accessed by a reference at compile-time. Static promotion refers to the enabling techniques such as intelligent data layout and code transformation which make memory disambiguation possible.

The problem of memory back disambiguation can be made trivial by placing all data on one processor. However this approach sequentializes all the accesses to that processor, creates a network hot-spot, and wastes parallel bandwidth to the distributed memory system. In contrast, our technique allows static promotion to be performed on distributed objects, enabling the concurrent accesses to different memory banks and the full utilization of memory bandwidth.

The rest of the paper is organized as follows. Section 2 gives an overview of the Raw architecture and compiler. Section 3 overviews static promotion on Raw architectures. Section 4 describes modulo unrolling, a static promotion strategy for arrays. Section 5 describes other optimizations for array accesses on Raw. Section 6 presents some experimental results. Section 7 describes related work, while Section 8 concludes.

2. Overview of the Raw System

This section gives an overview of the Raw system, which includes the architecture and the compiler.

2.1. Raw Architecture

The Raw architecture [13] is a simple, distributed, software-exposed architecture motivated by the desire to scale and maximize the amount of computational resources. Figure 1 depicts the layout of a Raw microprocessor. The design features a two-dimensional mesh of identical tiles, with each tile having its own instruction stream, register file, memory, ALU, and switch. Each processor on a tile is a simple, RISC pipeline. The switches implement both a static network and a dynamic network. The static network is under the control of the instruction streams on the switches, while the dynamic network routes messages conventionally by reading the headers of messages.

The features mentioned result in a design whose resources can scale easily. They also enable a fast clock, since there is no complex logic and no wire is longer than

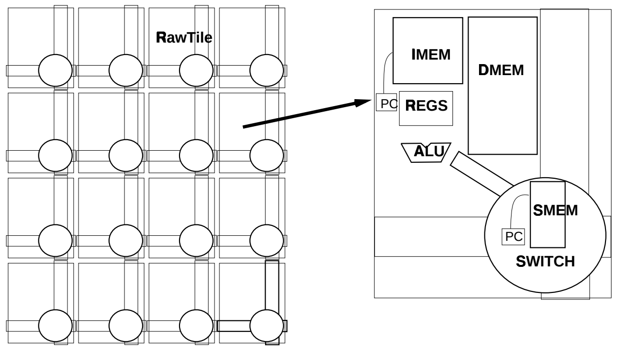

Figure 1. A Raw microprocessor is a mesh of tiles, each with a processor and a switch. The processor contains instruction memory, data memory, registers, and ALU. The switch contains its own instruction memory.

the inter-tile distance. Static network routing and register-level access enable fast communication, with the latency of communication between neighboring tiles being as low as two cycles. Such low latency allows exploitation of fine-grained ILP. This contrasts with multiprocessors which are restricted to coarse-grain parallelism due to much larger network latencies. This architecture is fully exposed to the compiler, which through sophisticated analysis, is able to extract and schedule a very high degree of instruction level parallelism from ordinary sequential programs.

2.2. The Raw compiler

Figure 2 outlines the structure of RAWCC, the Raw compiler implemented using the SUIF compiler infrastructure [15]. RAWCC accepts sequential C or FORTRAN programs and automatically parallelizes them for a Raw machine. The compiler consists of two main phases, the compiler-managed memory system called Maps [4] and the space-time scheduler. Maps uses the information provided by pointer and array analysis to perform static promotion and transformations related to dynamic accesses. This paper focuses on modulo unrolling, one method of static promotion used in Maps. An overview of the complete system can be found in [4]. The space-time scheduler [9] parallelizes each basic block of the program across the processors, obeying dependence and serialization requirements specified by Maps.

3. Static Promotion

The static network has no packetization, dynamic routing, or demultiplexing overhead and can communication values between neighboring tiles in as low as two cycles. It can be invoked when the source and the destination of a

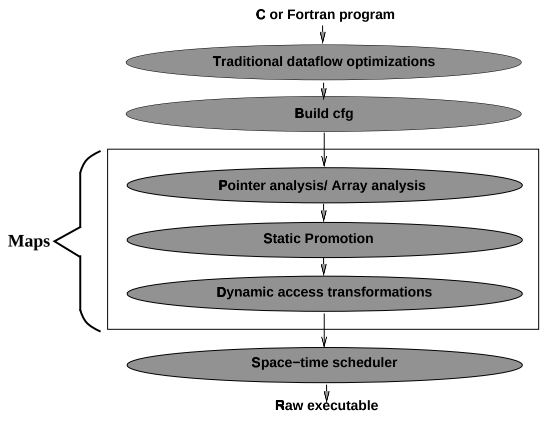
C or Fortran program
Traditional dataflow optimizations
Build cfg
Pointer analysis/ Array analysis
Static Promotion
Dynamic access transformations
Maps
Space−time scheduler
Raw executable

Figure 2. Structure of Raw compiler

message are known at compile-time. To employ the static network to communicate a memory reference value, the compiler must be able to determine that the reference refers exclusively to memory on a single, statically-known tile. This property of the memory reference is called the *static residence property*, and references satisfying this property are *static references*. References which do not satisfy this property are *dynamic references*, and they are communicated using the dynamic network.

The process of producing and identifying static references is called *static promotion*. Static promotion can be achieved through a combination of intelligent mapping of data to tiles and code transformation. This section overviews the static promotion strategy for scalars and array accesses on Raw.

Scalar Static Promotion Scalar variables are assigned to home tile locations. Direct accesses to these variables satisfy the static residence property trivially, and they can be statically promoted. Indirect accesses through pointers cannot be promoted unless every value accessed by a pointer resides on the same processor. Currently, RAWCC does not promote these indirect accesses. Pointer analysis and proper mapping of scalars to home tile locations can enable such promotion, but they are beyond the scope of this paper.

Array Static Promotion The criteria for choice of a good data layout for arrays are as follows. A good data layout scheme should be amenable to easy static promotion across a wide range of possible accesses to that data. In addition, it must place data corresponding to accesses having high temporal locality onto different tiles, so that the accesses can occur in parallel. Finally, it should attempt to maximize data locality, in that accesses are allocated close to where they are most often required by the space-time scheduler.

Based on these criteria, arrays are uniformly laid out in a *low-order interleaved* manner. In low-order interleaving, consecutive elements of the data structure are interleaved in a round-robin manner across successive tiles in the Raw machine. The low order bits of the address specify the tile location of the memory. This layout is desirable since spatially close array accesses, such as A[i] and A[i+1], are also often temporally close. Low-order interleaving places these on different tiles, thus allowing ILP parallelism between their accesses. For certain programs, one can employ more tailored layouts, but that would destroy the uniformity which makes memory disambiguation and static promotion possible without extensive inter-procedural analysis.

Given this distribution of array data, static promotion of many array accesses can be achieved through *modulo unrolling*. The next section presents this technique in detail.

4. Modulo Unrolling

This section describes modulo unrolling, a technique for the static promotion of a common classes of array accesses. As a motivating example, consider the code in Figure 3(a). Using low-order interleaving for a Raw machine with four tiles, the data layout of A is as shown in Figure 3(b), *i.e.*, any element A[i] is stored on the tile given by the array offset modulo the number of tiles(N), which equals i modulo 4. In the loop, successive A[i] accesses refer to memory on tile 0, 1, 2, 3, 0, 1, 2, 3 ... . The edges out of the tiles in Figure 3(b) point to the program accesses which refer to that tile. As we can see, the A[i] access in Figure 3(a) refers to memory on all four tiles. Hence the access cannot be executed on the static network, because static network execution requires every access made by a given instruction to refer to memory on the same tile.

There is however a way to transform the code to enable static promotion. Figure 3(c) shows the result of unrolling the code in Figure 3(a) by a factor of four. Now, each access always refer to memory on the same tile. Specifically, A[i] always refers to tile 0, A[i+1] to tile 1, A[i+2] to tile 2, and A[i+3] to tile 3.

The above example leads us to the following intuition: when using low-order interleaving to lay out arrays, it may be possible to unroll loops by some factor to statically promote the array accesses in those loops. Note that full unrolling of loops would statically promote all array accesses in the loops. However, full unrolling can be prohibitively expensive in terms of code size, and it is not even possible for loops with unknown loop bounds. The challenge is to devise a method using partial unrolling, whose resulting code size is independent of the data size, and which allows the static promotion of references in loops with unknown loop bounds.

This section presents modulo unrolling, a technique which enables the static promotion of all array accesses whose index expressions are affine functions of enclosing

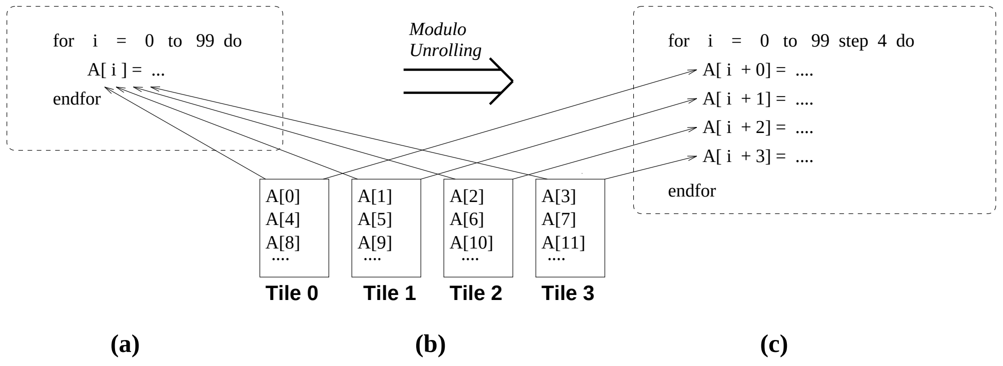

Figure 3. Example of array static promotion. (a) shows the original code. (b) shows the distribution of array A a Raw machine with four tiles. (c) shows the code after unrolling. After unrolling, each access refers to memory on only one tile.

loop induction variables. [1] Affine array accesses along with scalar variables form the bulk of the accesses in dense-matrix scientific codes.

Modulo unrolling is presented in three parts. Section 4.1 derives expressions for the minimum unroll factors required. After unrolling by those factors, the targeted array references will refer to memory on a single tile, but the identity of that tile may not be determinable at compile time. Section 4.3 outlines additional transformations which are required in some cases to make this tile compile-time determinable. Section 4.4 describes how the tile numbers and local offset expressions are actually determined.

4.1. Calculating unroll factors

The following theorem formally states the conditions for modulo unrolling, and the minimum unroll factors required.

Theorem 1 *Consider an affine access to a d-dimensional array in a k-dimensional loop nest. Let v_j be the induction variable for the j^{th} loop dimension. Assume the arrays are low-order interleaved on the row-major address. If all loop dimensions j ($\forall j \in [1..k]$) are unrolled by a factor U_j given by the formula below, or any multiple, then each corresponding access in the unrolled code will always access memory on the same tile across iterations.*

We define U_j in terms of D_j:

$$D_j = N \ / \ gcd \ (N, \ \sum_{i=1}^{d} c_{i,j} \prod_{l=i+1}^{d} MAX_l)$$

$$U_j = lcm(D_j, s_j) \,/\, s_j$$

where:

$c_{i,j}$ is coefficient of v_j in the i^{th} array dimension ($\forall i \in [1..d], \forall j \in [1..k]$)
$c_{i,k+1}$ is constant factor in the i^{th} array dimension ($\forall i \in [1..d]$)
N is number of tiles on Raw machine
MAX_i is size of the i^{th} array dimension $\forall i \in [1..d]$
s_j is step size of loop dimension j.

The above formula was derived by choosing to low-order interleave the array on its row-major address. Note that this choice is completely independent of the choice of major ordering in the remaining code generation. The row-major choice in the above is used to partition the array and computation in one particular manner: once that partitioning is done, any ordering can be used on the resultant new local arrays.

Due to lack of space, we present this and most other results in this section without proof. The proof proceeds along the following lines. First, for the given array access, a symbolic expression for its row-major address is written down in terms of the loop induction variables. Next, we derive the address for the same access in the next unrolled iteration by substituting in the variable plus the unroll factor. The desired condition of static promotion is that these two addresses refer to the same tile. This will be true if the difference between the two expressions is a multiple of N, namely that the difference is zero in modulo N arithmetic. It can be shown that the smallest unroll factor satisfying this condition is D_j. U_j represents a correction made for non-unit step size.

Note that Theorem 1 provides the unroll factors induced by a single access. The overall unroll factor of a loop is the lcm of the unroll factors induced by the different accesses in the loop.

Modulo unrolling handles arbitrary affine functions with few other restrictions. Within its framework, it handles imperfectly nested loops, non-unit loop step sizes, and hand-

[1] An *affine function* of a set of variables is defined as a linear combination of those variables, plus a constant. As an example, given i,j as enclosing loop variables, A[i+2j+3][2j] is an affine access, but A[i*j + 4] is not.

```
real X[IMAX][JMAX], Y[IMAX][JMAX]

for I=2 to M-1 do
   for J=2 to M-1 do
      X[I][J] = 0.9 * X[I][1]
      Y[I][J] = 0.9 * ((1.0 - X[I][1]) * Y[1][J] + X[I][1] * Y[M][J])
endfor
```

(a)

ACCESS		***UNROLL FACTORS***		
		Abstract Expression	Without padding (JMAX = 29)	With padding (JMAX padded to 32)
$X[I][J]$	Loop I	$D_I = U_I = N/\gcd(N, 1 * JMAX + 0 * 1)$	$8/\gcd(8, 29) = 8$	$8/\gcd(8, 32) = 1$
	Loop J	$D_J = U_J = N/\gcd(N, 0 * JMAX + 1 * 1)$	$8/\gcd(8, 1) = 8$	$8/\gcd(8, 1) = 8$
$X[I][1]$	Loop I	$D_I = U_I = N/\gcd(N, 1 * JMAX + 0 * 1)$	$8/\gcd(8, 29) = 8$	$8/\gcd(8, 32) = 1$
	Loop J	$D_J = U_J = N/\gcd(N, 0 * JMAX + 0 * 1)$	$8/\gcd(8, 0) = 1$	$8/\gcd(8, 0) = 1$
$Overall$	Loop I		8	1
	Loop J		8	8

(b)

Figure 4. Modulo unrolling for code fragment from Tomcatv. (a) shows the code fragment. (b) shows the unroll factors computed for two of the accesses for $N = 8$ and X and Y array sizes being 29×29. The unroll factors are shown with and without the padding optimization. The last row shows the overall unroll factor computed to be the LCM of all 7 accesses.

linearized multidimensional arrays. It also handles unknown loop bounds, but code with unknown lower bounds may require additional transformations as explained in Section 4.3.

Bounds on unroll factors Unrolling incurs the cost increased code size. It can be shown that the unroll factor U_j derived in Theorem 1 is provably at most N, the number of tiles. In the worst case, since all the k loop dimensions may be unrolled N ways, the overall code growth is at most a factor of N^k. For k >=2 this can be large. In practice, however, the overall code growth is often limited to N irrespective of k.

4.2. Padding Optimization

For most of the simple index functions which occur in practice, a simple optimization enables us to restrict the overall code growth. The optimization arises out of the following observation. It can be shown that if the last dimension size MAX_d is a multiple of the number of tiles N, and the affine function representing the last array dimension index refers to at most one loop variable, then at most one of the loops in the enclosing loop nest needs to be unrolled for that access. This would imply that the overall code growth would be limited to N, irrespective of k. This leads to the *padding optimization*: pad the last array dimension size to be a multiple of N for all arrays. This is profitable whenever there are accesses to the array whose last dimension refer to at most one loop variable. In addition, padding the last dimension greatly simplifies the code generation, as shown in Section 4.4.

As an example of how modulo unrolling is used to automatically compute the unroll factors, consider Figure 4. Figure 4(a) shows a code fragment from Tomcatv, one of the Spec92 benchmarks. Figure 4(b) shows the unroll factors computed using modulo unrolling for the $X[I][J]$ and $X[I][1]$ accesses on different rows, for both I and J loops. The last row shows the overall unroll factors. The second columm shows the expressions for unroll factors using the formula in Theorem 1. Assuming $N = 8$ and array sizes for the X and Y arrays being 29×29, the third and fourth columns show the unroll factors with and without the padding optimization. In the fourth column, the last dimension size $JMAX = 29$ is assumed to be padded to 32, the next multiple of N. As we can see, the unroll factors for the $X[I][J]$ access for the I and J loops are 1 and 8 after padding. They would have been larger (8 and 8) without padding. If the unroll factors induced by all the accesses are similarly computed (not shown), and the LCM per dimension taken, this will yield the overall unroll factors in the last row.

In some cases the padding optimization fails to bound the overall code growth to N. These include cases which are not simple index functions, as well as cases where the loop nest contains multiple simple index functions which sometimes may induce unrolls on different loop dimensions. For these cases, if the code growth is deemed excessive, the array accesses can simply be executed on the dynamic network, thus avoiding any code growth. The dynamically executed accesses however do not interfere with the promotion of other accesses.

4.3. Additional transformations

After the code is unrolled by the factors dictated by Theorem 1, all affine array accesses will always refer to the same tile. In addition in most cases after unrolling, the tile numbers of the accesses are compile-time constants. However, for a loop with an unknown lower bound and a non-unit step size, the repeating pattern of tile numbers may depend on the lower bound. As an example, consider the code in Figure 5. When the lower bound lb is 0, the tiles referenced by successive accesses is 0, 2, 0, 2, ..., but if lb is 2 the pattern changes to 1, 3, 1, 3, ... As a result, static promotion is not possible.

```
for i=lb to 99 step 2 do
  A[i] = ...
endfor
```

Figure 5. Example loop with unknown lower bound and non-unit step size

To allow static promotion for a loop with unknown loop bounds and non-unit step size, a switch statement is needed in the output code. The switch is made on the value of lb mod N and has D_j / U_j cases, each executing the original loop unrolled by a factor U_j but with different patterns of tile numbers.

4.4. Affine code generation

Once loops are unrolled and any required additional transformations performed, each affine access will refer to the same tile. This section outlines how the constant tile numbers and the expressions for local offsets within the tiles are actually computed.

Code generation effectively distributes a single array of S elements in the original program across the tiles, so that each tile has an array of size $\lceil S/N \rceil$. Using low-order interleaving, the tile number of an access is its global offset modulo N, and the local offset is the global offset divided by N. When the last dimension is padded, as is done because of the padding optimization above, the tile number is simply the last dimension modulo N. In addition, the local offset is obtained by replacing the last dimension index with the index divided by N.

Strip mining While this last observation may be used to generate code directly, we automate this process by strip mining the last dimension by N and strength reducing the divide operations. This process is very similar to that used in [2] for a different purpose. Strip mining replaces the last dimension by itself divided by N, and it adds a new dimension at the end with index being the original last dimension index `mod` N. The division expressions are strength reduced in all cases, and the `mod` expressions representing tile numbers are reduced to constants using compiler knowledge of the modulo values of loop variables combined with modulo arithmetic [2].

Startup and cleanup code Note that unrolling may generate cleanup code after the unrolled loop if the number of iterations is not a multiple of the unroll factor. In addition, we generate startup code when the lower bound is unknown so that we can start the main unrolled loop at the next higher multiple of N, thus making the tile numbers known inside the main loop.

```
idiv4 = 0;
for i=0 to 99 step 4 do
  A[idiv4][0] = ...
  A[idiv4][1] = ...
  A[idiv4][2] = ...
  A[idiv4][3] = ...
  idiv4++
endfor
```

Figure 6. Statically promoted code for example in Figure 3

Figure 6 shows the final result of array static promotion on the original code in Figure 3 for a four-tile Raw machine. The code is first unrolled by $N = 4$ and the last array dimension is strip mined by $N = 4$. The division expression is strength reduced to the variable 'idiv4'. The new last dimension in Figure 6 represents the tile numbers, which have been reduced to the constants 0, 1, 2 and 3. The tile access pattern in the transformed loop is 0, 1, 2, 3, 0, 1, 2, 3 ... as in the original code, except that now each access always refer to the same tile. This transformed code is finally mapped by the space-time scheduler to the Raw executable.

5. Other optimizations for array accesses

This section outlines two additional optimizations for array accesses on Raw, dependence elimination and array permutation transformation.

Dependence elimination Dependence edges are introduced between accesses which the compiler can either determine to be the same or is unable to prove to be different. Unnecessary dependence edges restrict ILP, since they imply access sequentialization and thus restrict scheduling

Benchmark	Source	Lang.	Lines of code	Primary Array size	Seq. RT (cycles)	Description
fpppp-kernel	Spec92	Fortran	735	-	8.98K	Electron Interval Derivatives
btrix	Nasa7:Spec92	Fortran	236	15×15×15×5	287M	Vectorized Block Tri-Diagonal Solver
cholesky	Nasa7:Spec92	Fortran	126	3×32×32	34.3M	Cholesky Decomposition/Substitution
vpenta	Nasa7:Spec92	Fortran	157	32×32	21.0M	Inverts 3 Pentadiagonals Simultaneously
tomcatv	Spec92	Fortran	254	32×32	78.4M	Mesh Generation with Thompson's Solver
mxm	Nasa7:Spec92	Fortran	64	32×64, 64×8	2.01M	Matrix Multiplication
life	Rawbench	C	118	32×32	2.44M	Conway's Game of Life
jacobi	Rawbench	C	59	32×32	2.38M	Jacobi Relaxation

Table 1. Benchmark characteristics. Column *Seq. RT* shows the run-time for the uniprocessor code generated by the Machsuif MIPS compiler.

freedom. For scientific codes containing affine array accesses, three simple rules suffice to disambiguate most accesses which can be disambiguated. First, accesses determined to refer to different tiles by the method in Section 4 are always memory independent. Second, even among accesses referring to the same processor, accesses belonging to the same uniformly generated set differing by a non-zero constant must also be memory independent. [2] Finally, accesses to different un-aliased arrays are always different.

Array permutation transformation Sometimes the static promotion technique described in Section 4 may demand that the outer loop in a loop nest be unrolled and leave the inner loop as is. This is ineffective in terms of exposing ILP within basic blocks, because the basic blocks are now all very small. Array permutation transformation is a solution which replaces instances of the array inducing the outer loop unrolls by a permuted but otherwise identical array, such that accesses to the array now induce unrolls on inner loop. When all loops in a program request the same permutation, we change the orientation of the original array to match the permutation. When different loops request conflicting permutations, it might be profitable to copy from one permutation array to another in between loops.

Array permutation is currently performed by hand in places where it is profitable. It can be automated by discovering requested permutations and using a cost model to determine when copying is profitable.

6. Experimental Results

This section presents some performance results of the Raw compiler which implements modulo unrolling. Experiments are performed on the Raw simulator, which simulates the Raw prototype described in Section 2.1. A description of parameters of the Raw prototype, including instruction and communication latencies can be found in [9].

[2]Two affine array accesses are in the same uniformly generated set if they access the same array, and their index expressions differ by at most a constant. For example, A[i] and A[i+2] are in the same uniformly generated set, but A[i] and A[i+j] are not [1].

Benchmark	N=1	N=2	N=4	N=8	N=16	N=32
fpppp-kernel	0.48	0.68	1.36	3.01	6.02	9.42
btrix	0.83	1.48	2.61	4.40	8.58	9.64
cholesky	0.88	1.68	3.38	5.48	10.30	14.81
vpenta	0.70	1.76	3.31	6.38	10.59	19.20
tomcatv	0.92	1.64	2.76	5.52	9.91	19.31
mxm	0.94	1.97	3.60	6.64	12.20	23.19
life	0.94	1.71	3.00	6.64	12.66	23.86
jacobi	0.89	1.70	3.39	6.89	13.95	38.35

Table 2. Benchmark Speedup. Speedup compares the run-time of the RAWCC-compiled code versus the run-time of the code generated by the Machsuif MIPS compiler.

The benchmarks we select include programs from the Raw benchmark suite [3], program kernels from the nasa7 benchmark of Spec92, tomcatv of Spec92, and the kernel basic block which accounts for 50% of the run-time in fpppp of Spec92. Since the Raw prototype currently does not support double-precision floating point, all floating point operations in the original benchmarks are converted to single precision. Table 1 gives some basic characteristics of the benchmarks.

Speedup We compare results of the Raw compiler with the results of a MIPS compiler provided by Machsuif [12] targeted for an R2000. Table 2 shows the speedups attained by the benchmarks for Raw machines of various sizes. Note that these speedups do not measure the advantage Raw is attaining over modern architectures due to a faster clock, nor do they measure the disadvantages of single-issue versus multiple-issue. The results show that the Raw compiler is able to exploit ILP profitably across the Raw tiles for all the benchmarks. The average speedup on 32 tiles is 19.7.

The speedup numbers demonstrate the effectiveness of the static promotion. The modulo unrolling strategy is able to statically promote all array accesses in these applica-

Benchmark	N=1	N=2	N=4	N=8	N=16	N=32
fpppp-kernel	0.48	0.68	1.36	3.01	6.02	9.42
btrix	0.83	1.92	1.74	2.80	3.79	4.84
cholsky	0.88	0.92	1.31	1.82	1.78	1.75
vpenta	0.70	1.00	1.29	1.48	1.45	1.33
tomcatv	0.92	1.43	2.14	2.69	2.81	2.81
mxm	0.94	1.49	2.08	2.70	2.64	2.48
life	0.94	1.22	2.62	3.91	4.16	4.26
jacobi	0.89	1.06	1.47	2.08	2.30	2.24

Table 3. Benchmark Speedup under trivial static promotion, where all arrays are mapped to a single tile.

tions. For some of the applications we achieve close to linear speedup. Note that the speedups are attained from sequential code using automatic parallelization, and not for code tailored to any high-performance architecture.

Most of the speedup attained can be attributed to the exploitation of ILP, but unrolling plays a beneficiary role as well. In RAWCC, unrolling speeds up a program by exposing scalar optimizations across loop iterations. This latter effect is most evident in jacobi and life, where consecutive iterations share loads to same array elements which can be eliminated through common subexpression elimination. The large number of such shared references in jacobi explains why it achieves super-linear speedup. For most other applications, this shared effect was less significant.

Table 3 shows the speedup of the applications using the trivial static promotion technique of mapping all arrays to a single tile. The resulting speedups no longer scale to 32 tiles. Mapping all the memory to one tile limits the memory bandwidth, creates a communication hotspot, and prevents locality of access while exploiting ILP. In contrast, intelligent static promotion using modulo unrolling do not suffer from any of these defects.

Table 4 shows the code growth factor for all the benchmarks using modulo unrolling and using full unrolling on an eight-tile machine. The code growth factor is the ratio of the code size after the given transformation to the original code size. The code sizes are collected in terms of intermediate instructions of the SUIF intermediate format. For fppp-kernel, the code does not grow for because it has no array accesses and no loops. For six out of eight benchmarks, the code growth factor for modulo unrolling (with padding) is either close to or less than the number of tiles ($N = 8$). As observed in Section 4.1, the overall unroll factors for most commonly occurring affine accesses, namely those with simple index expressions, are bounded by N. The two exceptions, mxm and btrix, have larger code growth because of the exception noted in Section 4.2. They contain multiple simple index functions in the same loop which induce unrolls on different loop dimensions. Nevertheless, the code growth in all cases is much less than for

Benchmark	**Code growth factor**	
	Modulo unrolling	Full unrolling
fppp-kernel	1.0	1.0
btrix	15.3	5868.9
cholesky	8.6	1894.3
vpenta	7.2	367.3
tomcatv	9.8	331.0
mxm	28.1	1389.3
life	10.2	633.6
jacobi	8.1	512.6

Table 4. Code growth factor of benchmarks using modulo unrolling and using full unrolling on 8 processors. For most, the code is about 8 times larger with modulo unrolling. Code growth will increase further with data size for full unrolling, but not for modulo unrolling.

full unrolling, a naive alternative approach to static promotion of accesses to distributed memory. Even when possible, the full unrolling factors increase with data size, while the modulo unrolling factors do not.

7. Related Work

Memory bank disambiguation was introduced by Ellis in the Bulldog Compiler [6] targeting a point-to-point VLIW machine. For such VLIWs, he shows that successful disambiguation means that an access can be executed through a fast “front door” to a memory bank, while an unsuccessful access must be sent over a slower “back door.” However, most VLIWs today use global buses for communication, not a point-to-point network. VLIW machines of various degrees of scalability have been proposed, ranging from completely centralized machines to machines with distributed functional units, register files, and memory [10]. The lack of point-to-point VLIWs seems to explain the dearth of work on memory bank disambiguation for compiling for VLIWs.

The modulo unrolling scheme we propose is a descendant of a simple technique presented by Ellis [6]. He observes that unrolling can sometimes help disambiguate accesses, but he does not attempt to formalize the observation or propose an algorithm. Instead, his technique is restricted to certain array accesses which must be user identified, and he relies on user annotations to provide the unroll factors needed for disambiguation. In contrast, we present a fully automated and formalized technique for dense matrix codes. This involves a theory to predict the unroll factors required for affine function accesses along with a method to automatically generate code in which the processor number for each array access is known.

A different type of memory disambiguation is relevant on the more typical bus-based VLIW machines such as the Multiflow Trace [10]. Relative memory disambiguation [10] aims to discover if two memory access can refer to the same memory location. Successful disambiguation implies that accesses can be executed in parallel. Hence, relative memory disambiguation is more closely linked to dependence and pointer analysis techniques than to static promotion.

Loop unrolling has been applied for other purposes by researchers. Some techniques for software pipeline [8] uses symbolic loop unrolling internally to decide the software pipeline schedule. Unrolling has been studied as a method to increase ILP by Weiss [14] and Davidson [5]. Loop unrolling is typically combined with register renaming [11, 7] to increase ILP further by removing anti and output dependences. While RAWCC uses unrolling primarily for static promotion, it nevertheless obtains the benefit of the increased ILP as well. We have implemented renaming in our compiler using conventional techniques to fully exploit the benefits of unrolling.

8. Conclusions

This paper presents modulo unrolling, a technique for performing memory bank disambiguation of array references at compile-time. On a Raw machine, where the communication channels between the memory banks and the processors are exposed to the compiler, the technique has allowed the Raw compiler to manage the memory resources statically and efficiently for dense matrix application.

This paper shows that modulo unrolling performs static promotion by unrolling loops by a small factor, usually no more than the amount which is needed to expose enough parallelism to the available processors. In addition to static promotion, modulo unrolling has two other benefits. First, it enables full exploitation of the bandwidth of the machine. Second, it increases parts of programs which we can statically analyze, which in turn enables the compiler to orchestrate significant amount of ILP. For a set of scientific dense matrix applications, the technique has enabled the Raw compiler to extract instruction level parallelism scalable with the number of processors.

References

[1] A. Agarwal, D. Kranz, and V. Natarajan. Automatic Partitioning of Parallel Loops for Cache-Coherent Multiprocessors. In *22nd International Conference on Parallel Processing*, St. Charles, IL, August 1993. IEEE. Also in IEEE Transactions on Parallel and Distributed Systems, vol 6, pp 943-961, September 1995.

[2] S. Amarasinghe. Parallelizing Compiler Techniques Based on Linear Inequalities. In *Ph.D Thesis, Stanford University. Also appears as Techical Report CSL-TR-97-714*, Jan 1997.

[3] J. Babb, M. Frank, V. Lee, E. Waingold, R. Barua, M. Taylor, J. Kim, S. Devabhaktuni, and A. Agarwal. The raw benchmark suite: Computation structures for general purpose computing. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, Apr. 1997.

[4] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal. Maps: A Compiler-Managed Memory System for Raw Machines. Technical report, M.I.T. LCS-TM-583, July 1998.

[5] J. Davidson and S. Jinturkar. Improving Instruction-level Parallelism by Loop Unrolling and Dynamic Memory Disambiguation. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, Ann Arbor, MI, November 1995. IEEE Computer Society.

[6] J. R. Ellis. Bulldog: A Compiler for VLIW Architectures. In *Ph.D Thesis, Yale University*, 1985.

[7] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimizations. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, Jan 1981.

[8] M. S. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proc. Int'l Conf. on Programming Language Design and Implementation (PLDI)*, pages 318–328, June 1988.

[9] W. Lee, R. Barua, D. Srikrishna, J. Babb, V. Sarkar, S. Amarasinghe, and A. Agarwal. Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, October 1998.

[10] P. Lowney, S. Freudenberger, T. Karzes, W. Lichtenstein, R. Nix, J. O'Donnell, and J. Ruttenberg. The Multiflow Trace Scheduling Compiler. In *Journal of Supercomputing*, pages 51–142, Jan. 1993.

[11] S. A. Mahlke, W. Y. Chen, J. Gyllenhaal, and W. Hwu. Compiler Code Transformations for Superscalar-Based High-Performance Systems. *Proceedings of Supercomputing*, 1992.

[12] M. D. Smith. Extending suif for machine-dependent optimizations. In *Proceedings of the First SUIF Compiler Workshop*, pages 14–25, Stanford, CA, Jan. 1996.

[13] E. Waingold, M. Taylor, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, S. Devabhaktuni, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring It All to Software: The RAW Machine. *IEEE Computer*, September 1997. Also as MIT-LCS-TR-709.

[14] S. Weiss and J. E. Smith. A Study of Scalar Compilation Techniques for Pipelined Supercomputers. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA, October 1987.

[15] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S. Liao, C.-W. Tseng, M. Hall, M. Lam, and J. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *ACM SIGPLAN Notices*, 29(12), Dec. 1996.