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://cag-www.lcs.mit.edu/raw

July 8, 1998

Abstract

The Raw approach of replicated processor tiles interconnected with a fast static mesh network provides a simple, scalable design that maximizes the resources available in next generation processor technology. In Raw architectures, scalable access to memory is provided by distributing the memory across processor 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 this novel problem of statically determining the exact communication required for each memory reference. We introduce a code transformation called *modulo unrolling* following which memory-bank disambiguation of certain classes of accesses becomes possible. Modulo unrolling transformations ensure that every instance of these memory references will access the memory bank of a single processor tile known at compile-time. Such memory references can then be accessed on the static network. We show that this can be achieved by using a relatively small unroll factor. For dense matrix scientific applications we were able to access all the array references on the static network, thus achieving good speedup on the RAW processor.

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

Proceedings of the Fifth Int'l Conference of High-Performance Computing (HIPC), December, 1998. Also MIT-LCS-TR-759.

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 [9] aims to provide truly distributed resources at communication costs low enough to exploit ILP. It distributes the register files, memories and ALUs into identical tiles, and it provides a fast static compiler-routed network organized as a two-dimensional mesh for inter-tile communication. Each tile executes its own instruction stream, and static network routing is under direct control of the instruction stream. Distributed memory provides scalable memory bandwidth, with fast access to memories of remote tiles through the static network. Short wires allow a very high clock rate. A slower dynamic network provides mechanism for compiler un-analyzable accesses.

The use of a compiler-routed 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 goes to, then the access can be made on the fast static network. This process is termed *static promotion*. The larger the class of accesses which can be statically promoted, the better the performance will be. Other accesses must be made on the dynamic network.

This paper focuses on compiler techniques for Raw by which certain classes of accesses can be statically promoted. One important class of applications are scientific codes, containing array accesses having indices which are affine functions of enclosing loop induction variables. This paper shows how such codes can be statically promoted using a technique we term *modulo unrolling*. We also present supporting memory-system optimizations for scientific code. The techniques have been automated on RAWCC , the Raw parallelizing compiler. We present performance results of this compiler. We have developed strategies to efficiently compile dynamic accesses on Raw, but these are beyond the scope of this paper.

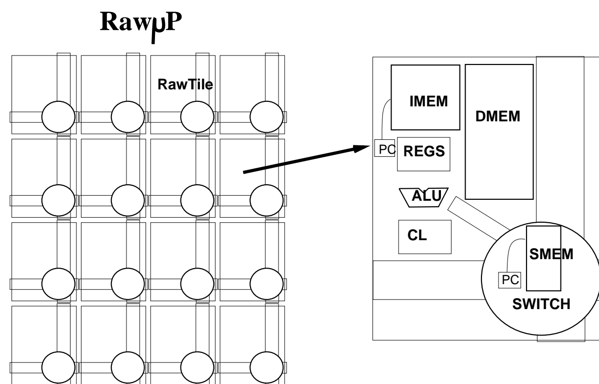

Figure 1: A Raw microprocessor.

Static promotion for Raw is related to the concept of *memory-bank disambiguation* [5] for distributed bank architectures, such as certain VLIWs. Memory-bank disambiguation is an analysis concept which aims to predict at compile-time the memory bank accessed by a reference. Static promotion of a reference is possible exactly when it can be memory bank disambiguated. However, note that while memory-bank disambiguation is an analysis concept, static promotion is a set of program transformations after which such analysis can be successful. Modulo unrolling is one transformation technique for static promotion.

While the problem of memory bank disambiguation can be made trivial by placing all data on one processor, this approach wastes the bandwidth of the distributed memory system. Static promotion for performance includes specification of distributed data layouts of different objects. This is followed by transformations to ensure compile-time bank disambiguation in such a distributed memory environment. Thus static promotion aims to achieve predictability while utilizing the full memory bandwidth of the machine.

The rest of the paper is organized as follows. Section 2 briefly describes the Raw architecture. Section 3 presents an overview of static promotion on Raw architectures. Section 4 describes overviews the RAWCC compiler system. Section 5 describes the static promotion strategy for arrays in more detail. Section 6 describes some optimizations to increase ILP in the context of scientific codes. Section 7 presents some experimental results. Section 8 describes related work, while Section 9 concludes.

2 Raw Architecture

The Raw architecture [9] is motivated by the desire to maximize the performance per silicon area of a machine. The design is kept simple to maximize the amount of processing resources that can fit on a chip and to enable a fast clock. Rather than using complex parallelism-discovering hardware, it relies on the

compiler to discover and schedule the parallelism.

Figure 1 depicts the layout of a Raw microprocessor. The design features a two-dimensional mesh of identical tiles, with register files, memories and ALUs distributed across tiles. A fast static compiler-routed network is provided for accesses whose tile numbers can be determined at compile-time. Each tile has separate instruction streams for the processor and the switch. Routing on the static network is under explicit control of the switch instruction stream. In addition, a dynamic network is provided as a fall-back for compile-time unanalyzable accesses. Each tile also has a limited amount of configurable logic (CL) implemented as FPGA logic, though this feature is not explored in this paper. The Raw prototype we have developed uses a MIPS R2000 processor on each tile, and the switch uses a stripped-down R2000 core.

The features mentioned result in a design whose resources can scales easily. They also enable a very fast clock speed, since there is no complex logic and no wire is longer than the inter-tile distance. Static network routing and register-level access enable extremely fast messages, with a effective two-cycle latency between neighboring tiles. Such low communication latency allows exploitation of fine-grained instruction-level parallelism, much finer grained than those that can be exploited on multiprocessors. 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.

3 Static Promotion Overview

This section overviews the static promotion strategy for scalars and array accesses on Raw. Static promotion is meaningful only within the context of a particular *data layout*, since it involves disambiguating these references to a single node at compile-time. We present the data layout and static promotion schemes for both scalars and arrays.

Scalar Static Promotion Scalar variables are allocated to maximize data locality by mapping them close to the processors where the space-time scheduler most often accesses that variable. Basic blocks accessing a variable fetch it, compute based upon its value, and finally store it back to its home location.

Once layout of scalars is determined as above, statically promoting scalars becomes a trivial matter. The first read and last write to every scalar in each basic block are simply statically promoted to the scalar's home location. Other accesses are renamed to transient temporary variables, which are created, communicated, and destroyed within basic blocks as decided by the space-time scheduler.

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.

Arrays, structures, and heap-allocated objects are all laid out in a uniform *low-order interleaved* manner. This implies that consecutive elements of the data structure are interleaved in a round-robin manner across successive tiles in the Raw machine. The name is derived from the fact that in this scheme, the low order bits of the address specify its home location. 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. Note that blocking, a coarse-grain layout scheme often employed in multiprocessors, would be unsuitable for Raw because it provides coarse-grain parallelism, not ILP. For certain programs, one can employ more tailored layouts, but that would destroy the uniformity which makes memory disambiguation and static promotion so easy even on an inter-procedural level.

Having specified the data layout, we can now consider static promotion of array accesses. The basic requirement for compiling a memory access using the static network is that *each invocation of that access must evaluate to the same processor every time it is executed.* This is normally not true even for the most trivial accesses. For example, for the single access A[i], different values of i can place the element on different processors, thus disallowing static promotion without transforming the program first. In section 5 we explain the program transformations and code-generation strategies which enable static promotion for a large classes of array accesses.

4 The RAWCC compiler system overview

Figure 2 outlines the structure of RAWCC , the compiler implemented for Raw using SUIF [10]. The memory and control-flow transformations form the front-end of the compiler. These passes transform input programs into a form that the space-time scheduler, described in [6], can schedule. The front-end includes the focus of this paper, namely the mechanisms for static promotion and the tile identification of promoted accesses. In addition, the front-end performs dependence analysis and optimizations, dynamic access optimizations and code generation, and other memory and control optimizations.

Several of the tasks shown in the front-end are outlined below. First, the *modulo unrolling transforma-*

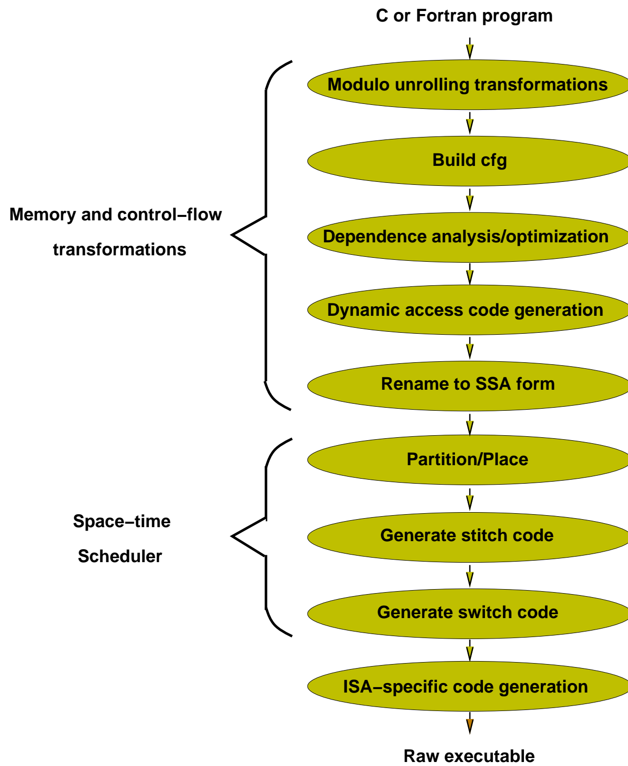

Figure 2: Outline of Raw compiler passes

tions pass performs unrolling and other transformations described in section 5 for the static promotion of array accesses. Unrolling also serves to increase the size of basic blocks forming the bodies of innermost loops, thus increasing the amount of available instruction-level parallelism. Next, a *control-flow graph* of the program is constructed in the usual way. Following that, *dependence analysis/optimization* operate on each basic block and inserts dependence edges between pairs of accesses that cannot be disambiguated. Section 6 describes how this optimization is performed. Next *dynamic access code generation* generates code for accesses which cannot be statically promoted onto the fast static network. Several strategies are employed to optimize this phase, but they are beyond the scope of this paper. Finally *renaming* performs all legal conversion of scalar and memory accesses to SSA forms. The objective is to eliminate anti and output dependences.

The space-time scheduling phase of the compiler takes the program generated by the front-end phase as input, and it parallelizes each basic block in the program across the processors. Within each basic block, the parallelization problem can be visualized as the mapping of a Directed Acyclic Graph (DAG) onto the two-dimensional topology of the Raw machine. The DAG nodes represent instructions, and DAG edges represent dependence and synchronization relations. Nodes are mapped onto tiles, and edges are mapped

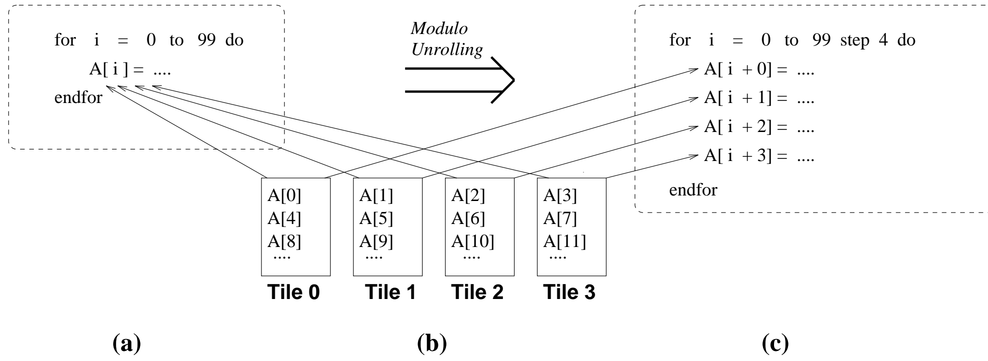

Figure 3: Example of array static promotion. (a) shows the original code. (b) shows the memories of a 4 processor Raw machine, showing the distribution of array A. (c) shows the code after unrolling. After unrolling, each access refers to locations from only one processor.

onto messages between tiles on the Raw static network. The mapping is done in a manner that maximizes parallelism, data locality, and load balance while guaranteeing deadlock-free behavior. Details are described in [6].

5 Static Promotion using Modulo Unrolling

In this section we show how static promotion can be done for certain classes of array accesses. As a motivating example, consider the code in Figure 3(a). Using low-order interleaving for a four processor Raw machine, the data layout of A is as shown in Figure 3(b), *i.e.,* any element A[i] is stored on the processor given by the array offset expression $\mathtt{mod}\ N$=4, or $i \ \mathtt{mod}\ 4$. In the loop, successive A[i] accesses go to processor numbers 0, 1, 2, 3, 0, 1, 2, 3 ... . The edges out of 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 all four tiles. Hence the access as written cannot be executed on the static network, because static network execution requires every access made by a given instruction to evaluate to 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 goes to the same processor. Namely, A[i] always goes to processor 0, A[i+1] to processor 1, A[i+2] to processor 2, and A[i+3] to processor 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 enable static execution on certain classes of array accesses. Note that full unrolling of loops would statically promotes all accesses because it reduces

array offsets to constants whose processor locations are immediately known at compile-time. However, full unrolling can be prohibitively expensive in terms of code size, and is not even possible for unknown loop bounds. The challenge is to devise a method using partial unrolling, whose resulting code size is independent of the data size, and in addition to allows static promotion with unknown loop bounds.

In this section we present *modulo unrolling*, a technique which enables static promotion of all array accesses whose index expressions are affine functions[1] of enclosing loop induction variables. Affine accesses along with scalar variables form the bulk of the accesses in dense-matrix scientific codes.

Modulo unrolling is presented in three parts. Section 5.1 derives expressions for the minimum unroll factors required. After unrolling by those factors, the code satisfies the *static property* which is that all accesses always go to the same processor, even if it may not be possible to determine the processor at compile-time. Section 5.2 outlines additional transformations which are required in some cases to ensure that the accesses are compile-time determinable, a property essential for static promotion. Section 5.3 describes how the processor numbers and local offset expressions are actually determined.

5.1 Calculating unroll factors

The following theorem states that partial unrolling always enables static promotion for all affine functions. It also gives the minimum unroll factors required.

Theorem 1 (Unroll factor theorem) *Consider an affine access to a d-dimensional array inside of a k-dimensional loop nest. 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 the same processor 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:

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

$c_{i,j}$*'s define the affine access* $\forall i \in [1..d], \forall j \in [1..k+1]$ *with values:*

$c_{i,j}$ = *coefficient of* v_j *in the* i^{th} *array dimension (* $\forall i \in [1..d], \forall j \in [1..k]$ *)*

$c_{i,k+1}$ = *constant factor in the* i^{th} *array dimension (* $\forall i \in [1..d]$ *)*

N = *number of tiles on Raw machine*

MAX_i = *size of the* i^{th} *array dimension* $\forall i \in [1..d]$

s_j = *step size of loop dimension* j.

Due to lack of space, we present this and most other results in this section without proof. The proof of this result proceeds along the lines of deriving the address computation offsets for the affine access for a given iteration, as well as an iteration advanced by D_j along loop dimension j. Proving that the offsets are equal in modulo N arithmetic proves the result, since for low-order interleaved arrays, the offset modulo N yields the processor number. U_j represents a correction made for non-unit step size.

The compilation strategy is as follows. Theorem 1 provides the unroll factors induced by a single access. The overall unroll factor of a loop is the Least Common Multiple (LCM) of the unroll factors induced by the different accesses in the loop. First the loops are unrolled by this factor. Next, additional transformations described in Section 5.2 are performed as needed to ensure static promotion. Finally, code generation occurs as shown in Section 5.3.

Modulo unrolling handles arbitrary affine functions with few other restrictions. Imperfectly nested loops are automatically handled, with no penalty or special cases. Non-unit loop step sizes are handled. The method handles unknown loop bounds, as theorem 1 provides the static property without assuming anything about the bounds. However the code may need additional transformations for unknown lower bounds as explained in Section 5.2. The method works unchanged even if multidimensional arrays were hand-linearized by the programmer, as the offset of the array element from the base remains unchanged in either case.

5.1.1 Bounds on unroll factors

Unrolling incurs a price in terms of 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. However, theorem 2 shows that for almost all affine functions which appear in practice, the overall code growth is limited to N, irrespective of k.

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

Figure 4: Example loop with unknown lower bound and non-unit step

Theorem 2 (Simple index theorem) *Provided the last dimension size MAX_d is a multiple of N, and the affine function representing the last array dimension index refers to at most a single loop variable, then at most one of the loops in the enclosing loop nest needs to be unrolled. If MAX_d is a not a multiple of N, it can be padded up to the next higher multiple, and the result still applies.*

This theorem implies that simple index expressions, which are restricted affine functions of the form $c_1 * i + c_2$, where i is a loop variable and c_1, c_2 are any integer constants, satisfy this case. Hence their overall code growth induced is at most N, independent of k. Since most real scientific programs have accesses of this form, including almost all of the accesses in the 8 real benchmarks we examined in the results section, the code growth is almost always bounded by N. For the rare access which yields a higher code growth penalty for static promotion, we can simply execute on the dynamic network, and avoid the penalty. The dynamically executed accesses, however, does not interfere with the promotion of other accesses.

Note that a code growth factor of N is no worse than the overall code growth seen for multiprocessors in most cases. Each of the N nodes of a multiprocessor typically executes very similar code, usually a transformed copy of the original code, and about equal in size. Hence the overall code growth across processors is about a factor N for multiprocessors as well.

We always pad the last dimension in order for theorem 2 to be true. In addition, padding the last dimension greatly simplifies the code generation, as shown in section 5.3.

5.2 Additional transformations for static promotion

After the code is unrolled by the factors dictated by theorem 1, all affine array accesses will always goto the same processor. In addition in most cases after unrolling, the processor numbers of the accesses are compile-time constants. However, in the rare case of an unknown loop lower bound combined with a non-unit step size on that loop, the processor number pattern may depend on the value of the lower bound. Intuitively, the reason is that a changed lower bound may permanently alter the processor number pattern. As an example,

consider the code in Figure 4. In case the lower bound lb is 0, the processor pattern from successive accesses is 0, 2, 0, 2, ... while if lb is 2 the pattern changes to 1, 3, 1, 3, ...

Hence a switch statement is needed in the output code when unknown loop bounds and non-unit step size occur together. It can be shown that the switch variable has the value lb mod N and has D_j / U_j cases each containing the original loop unrolled by a factor U_j.

5.3 Affine code generation

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

Code generation effectively distributes a single array of S elements in the original program across the processors, so that each processor has an array of size ceiling(S / N). Using low-order interleaving, the processor 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, it can be shown that the processor number is simply the last dimension modulo N, and the local offset is obtained by replacing the last dimension index by itself divided by N.

Strip mining While this last observation may be used to generate code directly, we automate this process by noting that strip mining the last dimension is performed in [2] for a different purpose. We use the software developed for [2] to strip mine the last dimension by N and strength reduce the divide operations. 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 processor numbers are reduced to constants using compiler knowledge of the modulo values of loop variables combined with modulo arithmetic.

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 processor numbers known inside the main loop.

Figure 5 shows the final result of array static promotion on the original code in Figure 3 for a four processor Raw machine. The code is first unrolled by the demanded factor, in this case four, and then the

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

Figure 5: Statically promoted code for example in figure 3

last array dimension is strip mined by N, namely four. The division expression is strength reduced to the variable 'idiv4'. The new last dimension in figure 5 respresents the processor numbers, which have been reduced to the constants 0, 1, 2 and 3. Note that the processor pattern in the transformed code is hence 0, 1, 2, 3, 0, 1, 2, 3 ... as was demanded in the original code. This transformed code is finally mapped by the space-time scheduler to the Raw executable.

6 Memory system optimizations

This section outlines two additional optimizations for the memory system on Raw, with emphasis on scientific code. They are dependence elimination and array permutation transformation.

Dependence elimination for scientific codes 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 they restrict scheduling freedom. For scientific codes containing affine function array accesses, three simple rules suffice to disambiguate (prove to be different) most accesses which can be disambiguated. First, accesses statically promoted to different processors by the method in Section 5 are always different. Second, even among accesses going to the same processor, accesses belonging to the same uniformly generated set[2] [1] differing by a non-zero constant must also be different. Finally, accesses to different un-aliased arrays are always different.

[2]Two affine array accesses are said to be 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.

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.

Array permutation transformation Sometimes the static promotion technique described in Section 5 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. One solution termed *array permutation transformation* is to replace instances of the array inducing the outer loop unrolls by accesses to another array, which has its dimensions permuted in a manner that now induces unrolls on inner loops. 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.

This optimization is currently performed by hand. It may be automated by discovering requested permutations and using a cost model to determine when copying is profitable.

7 Experimental Results

This section presents some performance results of the Raw compiler. Experiments are performed on the Raw simulator, which simulates the Raw prototype described in Section 2. A description of parameters of the Raw prototype, including instruction and communication latencies can be found in [6].

The benchmarks we select include programs from the Raw benchmark suite [4], 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 Mach-

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
cholsky	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.

suif [8] 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. 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 approach. The modulo unrolling strategy is able to statically promote a 100% of the array accesses in all these applications. For some of the applications we achieve close to the best-case linear speedup. Note that the speedups are attained from sequential code using automatic parallelization, and not for code tailored to any high-performance architecture.

The code sizes for 7 of the 8 applications are within a factor of N of the original, as predicted by Section 5.1.1 for simple index expressions. The only exception is mxm, for which the loop body is unrolled N^2 times. The reason for this exception is that the loop body of mxm has references from two different uniformly generated sets, each inducing an unroll factor of N on a different loop. Of the over 30 inner loops in the 8 applications, only one loop in mxm had this property. Nevertheless, performance remains high. Note that this factor N^2 code growth can be avoided in all cases by fetching some of the accesses on the dynamic network. Further, note that in all cases the code size using modulo unrolling is still independent of the *data size*, unlike using full unrolling.

The size of the datasets in these benchmarks is intentionally made to be small to feature the low communication overhead of Raw. Traditional multiprocessors, with their high overheads, would be unable to attain speedup for such datasets [3].

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 optimized 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.

8 Related Work

The Raw architecture and compiler is influenced by previous work in several areas. Due to space limitations, we do not present related work on the architectural aspects of Raw. For a detailed comparison to other architectures, see [9].

Section 1 points out that static promotion is related to memory bank disambiguation, and it lists their differences. Memory bank disambiguation was a term used by Ellis in the Bulldog Compiler [5] for a point-to-point VLIW model. 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 [7]. 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 [5]. He observes that unrolling can sometimes help disambiguate accesses, but he does not attempt to formalize the observation into a theory or 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 VLIW machines such as the Multiflow Trace [7]. These machines use global buses rather than a point-to-point network. Relative memory disambiguation [7] aims to discover if two memory access are necessarily different, though not necessarily known. 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.

9 Conclusions

The static network on the Raw machine presents a unique opportunity to optimize memory-fetching communication on a distributed memory machine. We present modulo unrolling as a powerful optimization technique which takes advantage of this opportunity. This technique has two additional 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.

Modulo unrolling is applicable to array memory accesses whose indices are affine functions of loop indices, which makes it useful for most memory accesses in scientific, dense matrix applications. We show that this technique can be achieved by using a relatively small unroll factor, usually no more than the amount which is needed to expose enough parallelism to the available processors. We have implemented this technique in the Raw compiler, and we have demonstrated the effectiveness of the technique on a suite of dense matrix applications. In all cases, we are able to extract instruction level parallelism which is scalable to 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] S. Amarasinghe, J. Anderson, C. Wilson, S. Liao, B. Murphy, R. French, and M. Lam. Multiprocessors from a Software Perspective. *IEEE Micro*, pages 52–61, June 1996.

[4] 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.

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

[6] 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.

[7] G. Lowney et al. The Multiflow Trace Scheduling Compiler. In *Journal of Supercomputing*, pages 51–142, January 1993.

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

[9] 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.

[10] R. Wilson et al. SUIF: A Parallelizing and Optimizing Research Compiler. *SIGPLAN Notices*, 29(12):31–37, December 1994.