Maps: A Compiler-Managed Memory System 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

Abstract

This paper describes Maps, a compiler managed memory system for Raw architectures. Traditional processors for sequential programs maintain the abstraction of a unified memory by using a single centralized memory system. This implementation leads to the infamous "Von Neumann bottleneck," with machine performance limited by the large memory latency and limited memory bandwidth. A Raw architecture addresses this problem by taking advantage of the rapidly increasing transistor budget to move much of its memory on chip. To remove the bottleneck and complexity associated with centralized memory, Raw distributes the memory with its processing elements. Unified memory semantics are implemented jointly by the hardware and the compiler. The hardware provides a clean compiler interface to its two inter-tile interconnects: a fast, statically schedulable network and a traditional dynamic network. Maps then uses these communication mechanisms to orchestrate the memory accesses for low latency and parallelism while enforcing proper dependence. It optimizes for speed in two ways: by finding accesses that can be scheduled on the static interconnect through **static promotion***, and by minimizing dependence sequentialization for the remaining accesses. Static promotion is performed using* **equivalence class unification** *and* **modulo unrolling***; memory dependences are enforced through explicit synchronization and* **software serial ordering***. We have implemented Maps based on the SUIF infrastructure. This paper demonstrates that the exclusive use of static promotion yields roughly 20-fold speedup on 32 tiles for our regular applications and about 5-fold speedup on 16 or more tiles for our irregular applications. The paper also shows that selective use of dynamic accesses can be a useful complement to the mostly static memory system.*

1 Introduction

Rapidly improving VLSI technology places billion-transistor chips within reach in the next decade. Such transistor capacity expands the space of feasible architectural designs from what is possible today. One trend is to use the increasing resources to build a powerful centralized processor, with a large part of the transistor budget devoted to the tasks of out-of-order issue, dynamic management of instruction level parallelism, and increasingly sophisticated kinds of speculation. Such an approach, however, makes design and verification difficult. It entails quadratic hardware complexity and connectivity, requiring long wires whose performance does not scale with technology. In addition, the approach consumes much area while providing diminishing returns, and it is poorly suited for emerging stream and multimedia applications which demand simple but plentiful amount of computing resources and high-throughput IO.

A Raw microprocessor adopts a different approach [15]. It constructs a powerful machine from simple processing elements, which are replicated and distributed across the chip. Instruction-level parallelism on this machine can be orchestrated through space-time scheduling [7]. By keeping the processing elements simple, a Raw microprocessor can devote a large amount of chip space to memory, thus addressing the memory bottleneck problem by moving much of its memory system on chip [2]. For example, a billion-transistor chip with half its area devoted to memory can contain tens of megabytes of SRAM. Using integrated DRAM allows at least four times that amount. This memory capacity makes it possible for the working sets of many programs to be kept entirely on chip.

A critical design issue is how to organize such a large on-chip memory. A fast, single-banked memory of that size is generally recognized to be infeasible. An on-chip version of multi-banked memory suffers from the hardware complexity of a centralized unit servicing multiple processing elements, and it disrupts the opportunity to exploit on-chip locality between memory and processing elements. Without on-chip locality, an average memory access can traverse half the length of a chip. In a billion-transistor, several-gigahertz processor, such an access will become a multi-cycle operation just from the wire delay alone. For example, extrapolations of current treads suggest that crossing a chip will take eight cycles by 2003 and forty cycles by 2007 [9].

A more natural organization is to distribute the memory banks along with the processing elements. The Raw microprocessor adopts this approach. It consists of simple, replicated tiles arranged in a two-dimensional interconnect; each Raw tile contains both a processing element and a memory bank, as well as a switch which provides direct connectivity between neighbors. Unlike traditional memory banks which serve as subunits of a centralized memory system, each memory bank on the Raw microprocessor functions autonomously and is directly addressable by its local processing element, without going through a layer of arbitration logic. This organization enables memory ports which scale

[*] *Proceedings of the Twenty-Sixth International Symposium on Computer Architecture* **(ISCA-26)**, Atlanta, GA, June, 1999.

with the number processing elements. It supports fast local memory accesses without the need for global caches, which consume on-chip area and introduce a complex coherence problem.

In accordance with its design principle of keeping the hardware simple to allow for plentiful resources and a fast clock, a Raw microprocessor does not contain any specialized hardware to support its distributed memory system. Rather, remote memory accesses are performed through two general inter-tile interconnects: a fast static network for compiler analyzable accesses and a slower, fail-safe dynamic network. Furthermore, the abstraction of a unified memory system is implemented entirely in software.

This paper presents Maps, Raw's compiler managed memory system which maintains a unified memory abstraction for sequential programs correctly and efficiently. Maps manages correctness by enforcing necessary memory dependence through explicit synchronization on the static network and a new technique called *software serial ordering*. It manages efficiency by minimizing memory dependence and by considering the tradeoff between locality, memory parallelism, and the preference for static accesses over dynamic accesses. These goals are realized through applications of traditional pointer and array analysis. *Static promotion*, the process of creating static accesses, is performed using two new techniques. *Equivalence class unification* creates static accesses by using pointer analysis to guide the placement of data, while *modulo unrolling* creates static accesses out of regular array accesses through unrolling.

From a more general perspective, the static promotion techniques in this paper describe how to distribute data in sequential programs across multiple memory banks and how to disambiguate memory accesses to specific banks. Successful distribution enables independent parallel accesses to memory, while successful disambiguation leads to two advantages. It allows memory accesses to be orchestrated by the compiler through the fast static network, and it enables the compiler to perform locality optimizations based on the known location of that access. These techniques are applicable to any microprocessor having multiple memory banks with non-uniform access times and compiler-exposed communication mechanisms.

We have implemented a SUIF-based compiler [16] that implements Maps by incorporating static promotion and software serial ordering. We have evaluated it on several dense matrix applications, stream applications, and irregular scientific applications. Analysis of current results leads to two basic conclusions. First, most of our benchmarks derive a significant performance improvement from the higher bandwidth and finer disambiguation provided by Maps. Second, though a purely static memory system usually provides good performance, a better memory system is a mostly static one in which dynamic accesses play a complementary but essential role.

The rest of the paper is organized as follows. Section 2 provides the architectural background, compiler overview and execution model for Maps. Section 3 describes the traditional analysis techniques leveraged by Maps. Section 4 describes techniques for static promotion. Section 5 describes support for dynamic accesses. Section 6 presents results, Section 7 discusses related work, and Section 8 concludes.

2 Background

This section provides some background for Maps. It describes the Raw architecture and its memory mechanisms. It also gives an overview of the Raw compiler, focusing on its execution model and the issues faced by its Maps subcomponent.

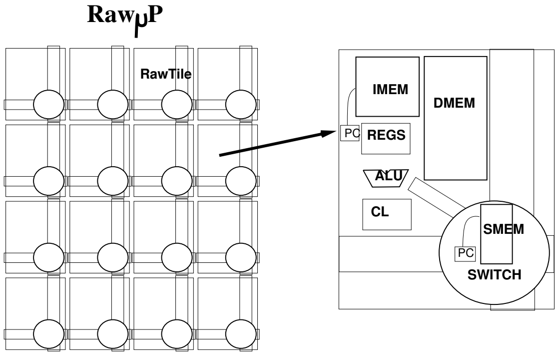

Figure 1: A Raw microprocessor is a mesh of tiles, each with a processing element, some memory and a switch. The processing element contains registers, ALU, and configurable logic (CL). It interfaces with its local instruction and data memory as well as the switch. The switch contains its own instruction memory.

Raw architecture The Raw architecture [15] is designed to address the issue of building a scalable architecture in the face of increasing transistor budgets and wire delays which do not scale with technology. Figure 1 depicts the layout of a Raw machine. A Raw machine consists of simple, replicated tiles arranged in a two dimensional mesh. Each tile has its own processing element, a portion of the chip's total memory, and a switch. The processor is a simple RISC pipeline, and the switch is integrated directly into this processor pipeline to support fast register-level communication between neighboring tiles; a word of data travels across one tile in one clock cycle. Scalability on this machine is achieved through the following design guidelines: limiting the length of the wires to the length of one tile; stripping the machine of complex hardware components; and organizing all resources in a distributed, decentralized manner.

Communication on the Raw machine is handled by two distinct networks, a fast, compiler-scheduled static network and a traditional dynamic network. The interfaces to both of these networks are fully exposed to the software. Each switch on the static network is programmable, allowing statically inferable communication patterns to be encoded in the instruction streams of the switches. This approach eliminates the overhead of composing and routing a directional header, which in turn allows a single word of data to be communicated efficiently. Furthermore, it allows the communication to be integrated into the scheduling of instructions at compile time. Accesses to communication ports have blocking semantics that provide near-neighbor flow control; a processor or switch stalls if it is executing an instruction that attempts to access an empty input port or a full output port. This specification ensures correctness in the presence of timing variations introduced by dynamic events such as interrupts and I/O, and it obviates the lock-step synchronization of program counters required by many statically scheduled machines. The dynamic switch is a traditional wormhole router that makes routing decisions based on the header of each message while guaranteeing in-order delivery of messages. It serves as a fall-back mechanism for non-statically inferable communication. A processor handles dynamic messages via either polling or interrupts.

Memory mechanisms From these communication mechanisms, the Raw architecture provides three ways of accessing memory: local access, remote static access, and dynamic access, in increasing order of cost. A memory reference can be a local access or a remote static access if it satisfies the *static residence property* — that is, (a) every dynamic instance of the reference must refer to memory on the same tile, and (b) the tile has to be known at compile time. The access is local if the Raw compiler places the subsequent use of the data on the same tile as its memory location; otherwise, it is a remote static access. A remote static access works as follows. The processor on the tile with the data performs the load, and it places the load value onto the output port of its static switch. Then, the pre-compiled instruction streams of the static network route the load value through the network to the processor needing the data. Finally, the destination processor accesses its static input port to get the value.

If a memory reference fails to satisfy the static residence property, it is implemented as a dynamic access. A load access, for example, turns into a split-phase transaction requiring two dynamic messages: a load-request message followed by a load-reply message. Figure 2 shows the components of a dynamic load. The requesting tile extracts the resident tile and the local address from the "global" address of the dynamic load. It sends a load-request message containing the local address to the resident tile. When a resident tile receives such a message, it is interrupted, performs the load of the requested address, and sends a load-reply with the requested data. The tile needing the data eventually receives and processes the load-reply through an interrupt, which stores the received value in a predetermined register and sets a flag. When the resident tile needs the value, it checks the flag and fetches the value when the flag is set. Note that the request for a load needs not be on the same tile as the use of the load.

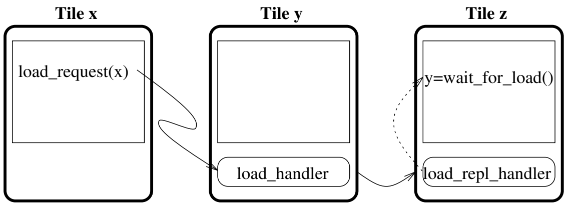

Figure 2: Anatomy of a dynamic load. A dynamic load is implemented with a request and a reply dynamic message. Note that the request for a load needs not be on the same tile as the use of the load.

Table 1 lists the end-to-end costs of memory operations as a function of the tile distance. The costs include both the processing costs and the network latencies. Figure 3 breaks down these costs for a tile distance of two. The measurements show that a dynamic memory operation is significantly more expensive than a corresponding static memory operation. Part of the overhead comes from the protocol overhead of using a general network, but much of the overhead is fundamental to the nature of a dynamic access. For example, a dynamic load requires sending a load request to the proper memory tile, while a static load can optimize away such a request because the memory tile is known at compile time. The need for flow control and message atomicity to avoid deadlocks further contributes to the cost of dynamic messages. Finally the inherent unpredictability in the arrival order and timing of messages requires expensive reception mechanisms such as polling or interrupts. In the static network, its blocking semantics combine with the compile-time ordering and scheduling of static messages to obviate the need for expensive reception mechanisms.

Distance	0	1	2	3	4
Dynamic store	17	20	21	22	23
Static store	1	4	5	6	7
Dynamic load	28	34	36	38	40
Static load	3	6	7	8	9

Table 1: Cost of memory operations.

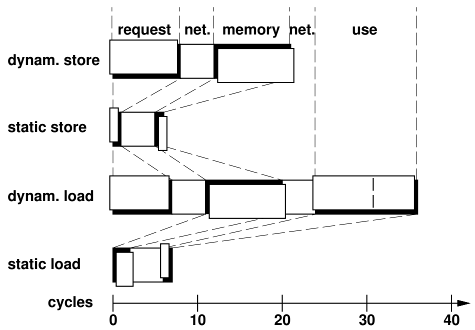

Figure 3: Breakdown of the cost of memory operations between tiles two units apart. Highlighted portions represent processor occupancy, while unlifted portions portion represents network latency.

Compilation overview and execution model Figure 4 outlines the structure of Rawcc, the Raw compiler built on top of SUIF [16]. Rawcc accepts sequential C or FORTRAN programs and automatically parallelizes them for a Raw machine. The compiler consists of two main phases, Maps and the space-time scheduler.

The goal of Maps is to provide efficient use of hardware memory mechanisms while ensuring correct execution. This goal hinges on three issues, identification of static accesses, support for memory parallelism, and efficient enforcement of memory dependences. The primary goal of Maps is to identify static accesses. As shown in Table 1, static accesses are several times faster than dynamic accesses, and they enable locality optimization by the space-time scheduler to co-locate data with the computation which access it. In addition, Maps attempts to provide memory parallelism by distributing data across tiles. Not only does it distribute different objects to different tiles, it also divides up aggregate objects such as arrays and structs and distributes them across the tiles. This distribution is important as it enables parallel accesses to different parts of the aggregate objects.

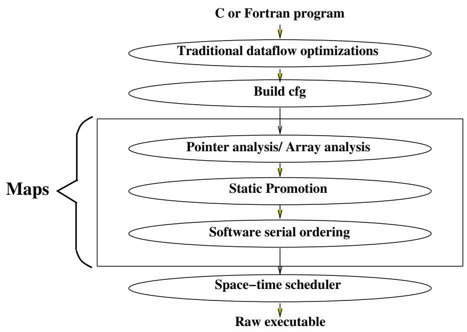

Figure 4: Structure of the Raw compiler

For correctness, Maps must ensure that the memory accesses occurring on different tiles obey the dependences implied by the original serial program. Three types of memory dependences need to be considered: those between static accesses, those between dynamic accesses, and those between a static and a dynamic access. Dependences between static accesses are easily enforced. References mapped to different memory banks are necessarily non-conflicting, so the compiler only needs to avoid reordering potentially dependent memory accesses on each tile. The real difficulty comes from dependences involving dynamic accesses, because accesses made by different tiles may potentially be aliased and require serialization. Maps uses a combination of explicit synchronization and a new technique called software serial ordering to enforce these dependences.

The space-time scheduler follows the analysis and code transformations in Maps. It parallelizes the computation in each forward control flow region across the processors. During this process, it uses the data distribution and disambiguation information provided by Maps, and it respects any dependence and serialization requirements of Maps. Parallelization is achieved by statically distributing the instructions across the tiles and orchestrating any necessary communication at the register level over the static network. The decision of how to map instructions is made while considering the tradeoffs between locality, parallelism, and communication cost. During execution, the instruction streams on different tiles cooperate to exploit parallelism in a forward control flow region one region at a time. Individual instruction streams proceed in a loosely synchronous manner, communicating only when there are register dependences and at the end of the forward control flow regions. For more details on the space-time scheduler, please refer to [7].

3 Analysis techniques

Maps employs several traditional analysis techniques to enhance the effectiveness of its mechanisms. The techniques include pointer analysis and array analysis. This section briefly presents the information they provide.

Pointer analysis is leveraged for three purposes: minimization of dependence edges, equivalence class unification, and software serial ordering. Maps uses SPAN, a state-of-the-art pointer analysis package [12]. Pointer analysis determines the group of abstract data objects each memory reference can refer to. An abstract object is either a static program object, or it is a group of dynamic objects created by the same memory allocation call in the static program. An entire array is considered a single object, but each field in a struct is considered a separate object. Pointer analysis identifies each abstract object by a unique *location set number*. It then annotates each memory reference with a *location set list*, a list of location set numbers corresponding to the objects that the memory reference can refer to. Figure 5(a) shows pointer analysis applied to a sample program. Object creation sites are annotated with their assigned location set numbers. A struct is marked with multiple location set numbers, one for each of its field. For simplicity, location set numbers are assigned only to non-pointer objects; in reality all objects are assigned such numbers. Each memory reference is marked with the location set numbers of the objects it can reference.

Maps defines the concept of *alias equivalence classes* from the program's location set lists. Alias equivalence classes form the finest partition of the location set numbers such that each memory access refers to location set numbers in only one class. Maps derives the classes as follows. First, it constructs a bipartite graph. A node is constructed for each abstract object and each memory reference. Edges are constructed from each memory reference to the abstract objects corresponding to the reference's location set list. Then, Maps finds the connected components of this graph. The location set numbers in each connected component form a single alias equivalence class. Note that references in the same alias class can potentially alias to the same object, while references in different classes can never refer to the same object. Figure 5(b) shows the bipartite graph and the equivalence classes in our sample program.

Maps uses a combination of pointer analysis and array analysis to identify any potential dependences between memory references. The location set lists provided by pointer analysis give precise object-level dependence information: only memory references with common elements in their location set lists can refer to the same data object and be potentially memory dependent. For arrays, however, pointer analysis does not distinguish between references to different elements in an array, so that reference pairs such as $A[1]$ and $A[2]$ are analyzed to be dependent. For these references, Maps uses traditional array dependence analysis to obtain finer grained dependence information [10].

4 Static promotion of memory accesses

Static promotion is the act of making a reference satisfy the static residence property. Without analysis, the Raw compiler is faced with two unsatisfactory choices: map all the data to a single tile, which makes all memory accesses trivially static at a cost of no memory parallelism; or distribute all data, which enables memory parallelism but requires expensive dynamic accesses. This section describes two compiler techniques for static promotion which preserve some memory parallelism. Section 4.1 describes equivalence class unification, a general promotion technique based on the use of pointer analysis to guide the placement of data. Section 4.2 describes modulo unrolling, a code transformation technique applicable to most array references in the loops of

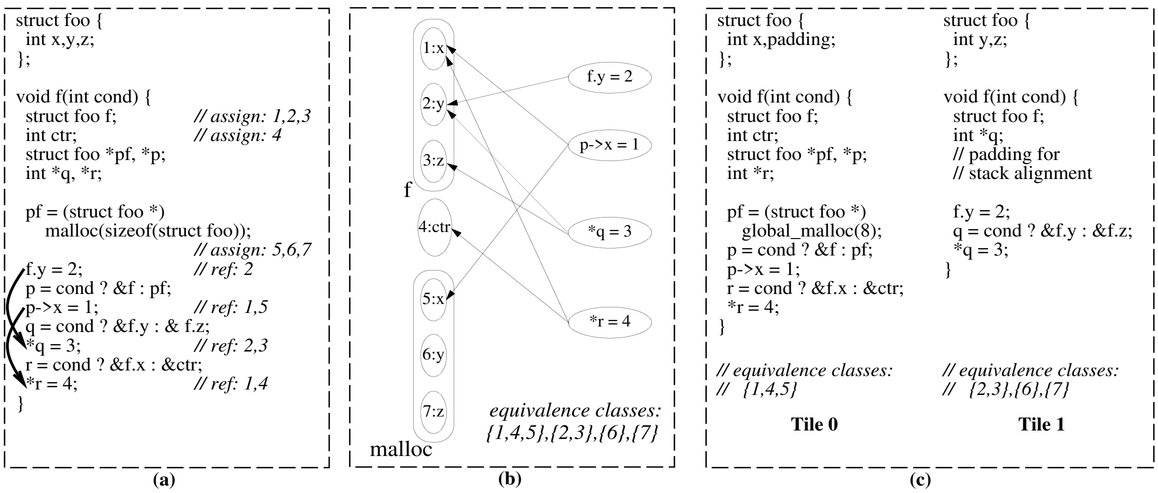

Figure 5: A sample program processed through pointer analysis and ECU. (a) shows the program annotated with the information provided by pointer analysis. The arrows represent memory dependences derived from pointer analysis. (b) shows its bipartite graph and its equivalence classes. (c) shows the program after it is distributed through ECU and space-time scheduling.

scientific applications. Section 4.3 explains the limitations of static promotion and motivates the need for an effective dynamic fall-back mechanism.

4.1 Equivalence class unification

Equivalence class unification (ECU) is a static promotion technique based on pointer analysis. It uses the alias equivalence classes we derive from pointer analysis to help guide the placement of data. ECU promotes all memory references in a single alias equivalence class by placing all objects corresponding to that class on the same tile. By mapping objects for every alias equivalence class in such a manner, all memory references can be statically promoted. By mapping different alias equivalence classes to different tiles, memory parallelism can be attained.

Elements in aggregate objects such as arrays and structs are often accessed close together in the same program. Distribution and static promotion of arrays are addressed in Section 4.2. For structs, SPAN differentiates between accesses to different fields, so that fields of a struct can be in different alias equivalence classes and distributed across the tiles. Figure 5(c) shows how equivalence class unification is applied to our sample program. Note that aggregate objects distributed across more than one tile have the same memory address on each tile. This property allows a single pointer to refer to the entire object, and it enables address computation to be mapped to any arbitrary tile. The alignment requirement is ensured by doing the appropriate padding on the distributed objects, stack, and heap.

4.2 Modulo unrolling

The major limitation of equivalence class unification is that an array is treated as a single object belonging to a single equivalence class. Mapping an entire array to a single memory bank sequentializes accesses to that array and destroys the parallelism found in many loops. Therefore, we use a different strategy to handle the static promotion of array accesses. First, arrays are laid out in memory through *low-order interleaving*. In this scheme, consecutive elements of an array are interleaved in a round-robin manner across the memory banks on the Raw tiles. We then apply modulo unrolling, a code transformation technique which statically promotes array accesses in loops.

Modulo unrolling is a framework for determining the unroll factor needed to statically promote all array references inside a loop. We illustrate this technique through a simple example. Consider the source code in Figure 6(a). Using low-order interleaving, the data layout for array A on a four-tile Raw machine is shown in Figure 6(b). In the loop, successive A[i] accesses refer to memory banks on tile 0, 1, 2, 3, 0, 1, 2, 3, etc. The edges out of any access point to the memory banks the access refers to. As we can see, the A[i] access in Figure 6(a) refers to memories on all four tiles. Hence the access as written cannot be statically promoted.

Intelligent unrolling, however, can enable static promotion. Figure 6(c) shows the result of unrolling the code in Figure 6(a) by a factor of four. Now, each access always refers to elements on the same memory bank. 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. Therefore, all resulting accesses can be statically promoted. It can be shown that this technique is always applicable for loops with array accesses having indices which are affine functions of enclosing loop induction variables. These accesses are often found in dense matrix applications and multimedia applications. For a detailed explanation and the symbolic derivation of the unrolling factor, see [1].

4.3 Uses for dynamic references

A compiler can statically promote all accesses through equivalence-class unification alone, and modulo unrolling helps improve memory parallelism during promotion. There are several reasons, however, why it may be undesirable to

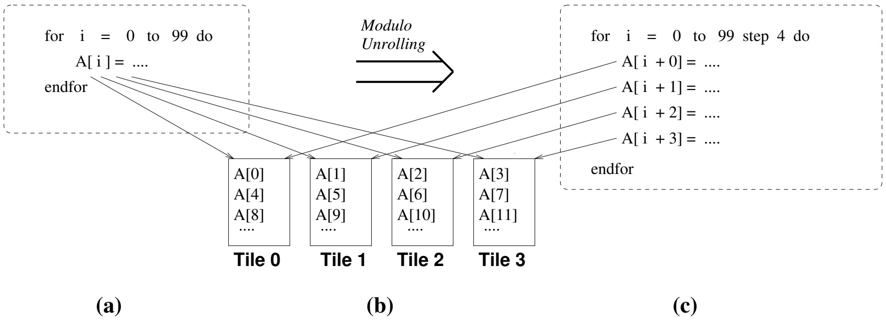

Figure 6: Example of modulo unrolling. (a) shows the original code; (b) shows the distribution of array A on a 4-tile Raw machine; (c) shows the code after unrolling. After unrolling, each access refers to memory on only one tile.

promote all references. First, modulo unrolling sometimes requires unrolling of more than one dimension of multi-dimensional loops. This unrolling can lead to excessive code expansion. To reduce the unrolling requirement, some accesses in these loops can be made dynamic. In addition, static promotion may sometimes be performed at the expense of memory parallelism. For example, indirect array accesses of the form $A[B[i]]$ cannot be promoted unless the array $A[]$ is placed entirely on a single tile. This placement, however, yields no memory parallelism for $A[]$. Instead, Maps can choose to forgo static promotion and distribute the array. Indirect accesses to these arrays would be implemented dynamically, which yields better parallelism at the cost of higher access latency. Moreover, dynamic accesses can improve performance by not destroying static memory parallelism in critical parts of the program. Without it, arrays with mostly affine accesses but a few irregular accesses would have to be mapped to one tile, thus losing all potential memory parallelism to the arrays. Finally, dynamic accesses can increase the resolution of equivalence class unification. A few isolated "bad references" may cause pointer analysis to yield very few equivalence classes. By selectively removing these references from promotion consideration, more equivalence classes can be discovered, enabling better data distribution and improving memory parallelism. The misbehaving references can then be implemented as dynamic accesses.

For these reasons, it is important to have a good fallback mechanism for dynamic references. More importantly, such mechanism must integrate well with the static mechanism. The next section explains how these goals are accommodated.

For a given memory access, the choice of whether to use a static or a dynamic access is not always obvious. Because of the significantly lower overhead of static accesses, the current Maps system makes most accesses static by default, with one exception. Arrays with any affine accesses are always distributed, and two types of accesses to those arrays are implemented as dynamic accesses: non-affine accesses, and affine accesses which require excessive unroll factors for static promotion. Automatic detection of other situations which can benefit from dynamic accesses is still ongoing research. However, Section 6 shows two programs, Unstructured and Moldyn, whose performance can be improved when dynamic accesses are selectively employed.

5 Support for dynamic accesses

Maps provides mechanisms for correctness and efficiency of dynamic accesses. For correctness, Maps enforces memory dependences involving dynamic accesses through static synchronization and *software serial ordering*. Maps improves performance by reducing the amount of dependences that need to be enforced through epochs and memory update operations.

5.1 Enforcing dynamic dependences

Maps handles dependences involving dynamic accesses with two separate mechanisms, one for the type of dependences between a static access and a dynamic access, and one for the type of dependences between two dynamic accesses. A static-dynamic dependence can be enforced through explicit synchronization between the static reference and either the initiation or the completion of the dynamic reference. When a dynamic store is followed by a dependent static load, this synchronization requires an extra dynamic store acknowledgment message at the completion of the store. Because the source and destination tiles of the synchronization message are known at compile-time, the message can be routed on the static network.

Enforcing dependences between dynamic references is a little more difficult. To illustrate this difficulty, consider the dependence which orders a dynamic store before a potentially conflicting dynamic load. Because of the dependence, it would not be correct to issue their requests in parallel from different tiles. Furthermore, it would not suffice to synchronize the issues of the requests on different tiles. This is because there are no timing guarantees on the dynamic network: even if the memory operations are issued in correct order, they may still be delivered in incorrect order. One obvious solution is complete serialization as shown in Figure 7(a), where the later memory reference cannot initiate until the earlier reference is known to complete. This solution, however, is expensive because it serializes the slow round-trip latencies of the dynamic requests, and it requires store completions to be acknowledged with a dynamic message.

We propose *software serial ordering* to efficiently ensure such dependences. Figure 7(b) illustrates this technique. Software serial ordering leverages the in-order de-

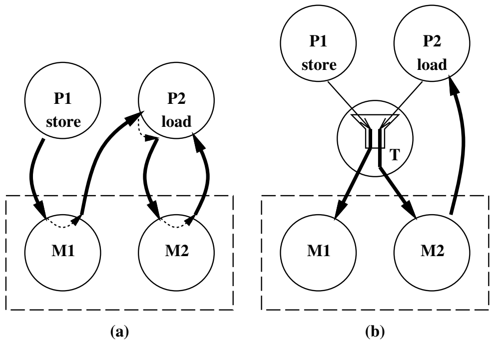

Figure 7: Two methods for enforcing dependences between dynamic accesses. P1 and P2 are processing nodes initiating two potentially conflicting dynamic requests; both diagrams illustrate an instance when the two requests don't conflict. M1 and M2 are the destinations of the memory requests. The light arrows are static messages, the dark arrows are dynamic messages, and the dashed arrows indicate serialization. The dependence to be enforced is that the store on P1 must precede the load on P2. In (a), dependence is enforced through complete serialization. In (b), dependence is enforced through software serial ordering. T is the turnstile node. The only serialization point is the launches of the dynamic memory requests at T. Note that Raw tiles are not specialized; any tile can serve in any or all of the following roles, as processing node, memory node, or turnstile node.

livery of messages on the dynamic network between any source-destination pair of tiles. It works as follows. Each equivalence class is assigned a *turnstile* node. The role of the turnstile is to serialize the request portions of the memory references in the corresponding equivalence class. Once memory references go through the turnstile in the right order, correct behavior is ensured from three facts. First, requests destined for different tiles must necessarily refer to different memory locations, so there is no memory dependence which needs to be enforced. Second, requests destined for the same tile are delivered in order by the dynamic network, as required by the network's in-order delivery guarantee. Finally, a memory tile handles requests in the order they are delivered.

Note that in order to guarantee correct ordering of processing of memory requests, serialization is inevitable. Our system keeps this serialization low, and it allows the exploitation of parallelism available in address computations, latency of memory requests and replys, and processing time of memory requests to different tiles. For efficiency, software serial ordering employs the static network to handle synchronization and data transfer whenever possible. Furthermore, different equivalence classes can employ different turnstiles and issue requests in parallel. Interestingly, though the system enforces dependences correctly while allowing potentially dependent dynamic accesses to be processed in parallel, it does not employ a single explicit check of runtime addresses.

5.2 Dynamic optimizations

Epochs Without optimizations, all dynamic memory requests in a single alias equivalence class have to go through a turnstile for the entire duration of the program. If the compiler schedules a dynamic memory request on a tile other than its turnstile, it would have to separately guarantee that the memory request does not get reordered with any past or future potentially dependent references on its way to the memory tile. In the general case, this requirement is prohibitively expensive and provides no benefit.

Sometimes, however, Maps can determine that all the dynamic memory accesses to an alias equivalence class in a region of the program are independent from each other. We can such a region an *epoch*. A trivial example of an epoch is a region whose dynamic accesses to an equivalence class are all loads. Other epoch detection mechanisms include pointer analysis, array dependence analysis, and relative memory disambiguation.

In epochs, it would be desirable to disable the turnstile and allow the accesses to proceed independently and without serialization. Maps supports epochs by placing memory barriers before and after the region. The barriers are implemented by explicitly checking for the completion of all accesses through load-replys or store-acknowledgments. Though barriers are expensive operations, their costs can easily be amortized away if an epoch includes one or more time-intensive loops.

Updates Updates are memory handlers which implement simple read/modify/write operations on memory elements. They take advantage of the generality of Raw's active-message dynamic network. The compiler migrates simple read/modify/write memory operations from the main program to the memory handlers. The modify operation is required to be both associative and commutative. Common examples include increment/decrement, add, multiply, and max/min.

Updates improve performance of dynamic accesses in three ways. First, a program can dispatch an update just like a store and then proceed without waiting for its completion. Second, an update collapses two expensive and serial dynamic memory operations, a load and a store, into one. Finally, the associativity and commutativity of the updates effectively removes dependences between different updates. This elimination can help increase the utility of epochs by finding regions with independent updates to an alias equivalence class.

6 Results

We have implemented Maps on Rawcc, the Raw compiler based on the SUIF compiler infrastructure [16]. This section presents evaluation of Maps. Evaluation is performed on a cycle-accurate simulator of the Raw microprocessor. The simulator uses a MIPS R2000 as the processing element on each tile. It faithfully models both the static and dynamic networks, including any contention effects. Application speedup is derived from comparison with the performance of code generated by the Machsuif Mips compiler [14] executed on the R2000 processing element of a single Raw tile. To expose instruction level parallelism across basic blocks, Rawcc employs loop unrolling and control localization [7]. Inner loops are usually unrolled as many times as there are number of tiles.

Benchmark	Type	Source	Lines of code	Seq. RT (cycles)	Primary Array size (words)	Description
Cholesky	Dense Mat.	Nasa7:Spec92	126	34.3M	16×16×32	Cholesky Decomposition/Substitution
Swim	Dense Mat.	Spec95	486	96.2M	513×33	Shallow Water Model
Tomcatv	Dense Mat.	Spec92	254	78.4M	32×32	Mesh Generation with Thompson's Solver
Vpenta	Dense Mat.	Nasa7:Spec92	157	21.0M	32×32	Inverts 3 Pentadiagonals Simultaneously
Ocean	Dense Mat.	Splash/Jade	1174	309.7M	256×256	Ocean Movement Simulation
Adpcm	Multimedia	Mediabench	295	2.8M	10240	Speech compression
SHA	Multimedia	Perl Oasis	608	1.0M	512×16	Secure Hash Algorithm
MPEG-kernel	Multimedia	UC Berkeley	86	14.6K	32×32	MPEG-1 Video Software Encoder kernel
Moldyn	Irreg. Sci.	CHAOS	805	63M	256×3, 32000×2	Molecular Dynamics
Unstructured	Irreg. Sci.	CHAOS	850	150M	17377×3	Computational Fluid Dynamics

Table 2: Benchmark characteristics. Column *Seq. RT* shows the run-time for the uniprocessor code generated by the Machsuif MIPS compiler.

Table 2 gives the characteristics of the benchmarks used for the evaluation. Benchmarks include five dense matrix applications, three multimedia applications, and two scientific applications with irregular memory access patterns. They are all sequential programs. Some benchmarks are full applications; others are key kernels from full applications. Cholesky and Vpenta are extracted from Nasa7 of Spec92. MPEG-kernel is the portion of MPEG which takes up 70% of the total run-time. Because the Raw simulator currently does not support double-precision floating point, all floating point operations are converted to single precision.

Benchmark		N=1	N=2	N=4	N=8	N=16	N=32
Cholesky		0.88	1.75	3.33	6.24	10.22	17.15
Swim		0.88	1.43	2.70	4.47	8.97	17.81
Tomcatv		0.92	1.64	2.76	5.52	9.91	19.31
Vpenta		0.78	1.90	3.36	7.06	12.17	20.12
Ocean		0.88	1.16	1.97	3.05	4.09	4.51
Adpcm		0.97	0.99	1.19	1.23	1.13	1.13
SHA		0.96	1.18	1.63	1.53	1.44	1.42
MPEG-kernel		0.90	1.36	2.15	3.46	4.48	7.07
Moldyn	array	0.95	1.36	2.38	2.99	4.28	4.38
	struct	0.92	0.94	1.60	2.57	3.11	3.59
Unstruct	array	0.82	1.21	2.35	3.59	5.22	6.12
	struct	0.86	1.29	2.07	3.00	4.10	4.92

Table 3: Benchmark speedup with full distributed static promotion through equivalence class unification and modulo unrolling. Speedup compares the run-time of the Rawcc-compiled code versus the run-time of the code generated by the Machsuif MIPS compiler.

Table 3 shows the speedups attained by the benchmarks for Raw microprocessors for a varying number of tiles. For Moldyn and Unstructured, we present results for two different versions. The original versions are written in a FORTRAN-like style, using array of base types to represent their data. The new versions have better data abstraction; they use a collection of structs to represent the program's objects.

All the benchmarks except ocean are compiled with full static promotion. In ocean, dynamic accesses are used for two purposes. First, affine array accesses in the outer loops are converted to dynamic accesses in order to avoid multi-dimensional unroll as required by static promotion through modulo unrolling. Second, dynamic accesses are used for array accesses which cannot be determined to be affine without inter-procedural analysis and inlining.

Our results in Table 3 show that Rawcc with Maps is able to orchestrate the parallelism available in the applications. To summarize the results, Rawcc is able to attain speedups in the range of 15-20 for four of the of the five dense matrix codes, and speedups of 4-8 for non-dense matrix codes with a reasonable amount of ILP. The dense matrix applications and MPEG-kernel have a lot of parallelism, with loops whose parallelism scale with the amount of unrolling. Moldyn and Unstructured have a modest amount of parallelism. They have parallelism both within loop iterations and across iterations, but loop carried dependences eventually limit the amount of parallelism exposed by unrolling. Note that the speedup for Moldyn is obtained without special handling of the reduction in its time-intensive loop. Its performance should improve further with reduction recognition. Finally, Adpcm and SHA have little parallelism. Their work both within and across iterations is mostly serial.

Comparisons between the two versions of Unstructured and Moldyn show that our promotion techniques are effective in providing memory parallelism for both programming styles. While the struct versions use arrays of structs to represent their data, the array versions use two-dimensional arrays with the second dimension representing the fields of the struct. The opportunities for memory parallelism are identical for both versions, but Maps exposes that parallelism through different means. Memory parallelism in the array versions is exposed through array distribution and modulo unrolling, while parallelism in the struct version is exposed through equivalence class unification. The different speedups for the two versions are accounted for by details concerning address calculation costs and opportunities for optimization of those costs.

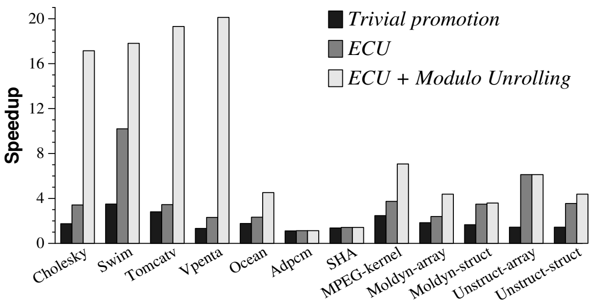

Figure 8: Comparison of 32-tile speedups for trivial static promotion, ECU promotion, and full Maps promotion.

Figure 8 measures the benefits of our static promotion techniques on overall speedups. It compares the speedups on 32 tiles for three promotion strategies: no analysis, ECU only, and full promotion using both ECU and modulo unrolling. Without analysis, accesses can be promoted trivially by mapping all data to one tile. Our results, however, show that this promotion strategy leads to low speedups ranging from one to four because it provides no memory parallelism and no data locality. ECU alone yields sufficient memory parallelism to attain the modest overall speedups for the irregular applications such as Adpcm, SHA, Moldyn-struct, and Unstructured-struct. Full Maps promotion, however, is necessary to exploit the large amount parallelism available in the more regular applications. Figure 9 breaks down the utilization of our two techniques in full Maps promotion. It shows for each application the percentage of aggregate objects whose references are promoted through ECU versus those that are promoted through modulo unrolling.

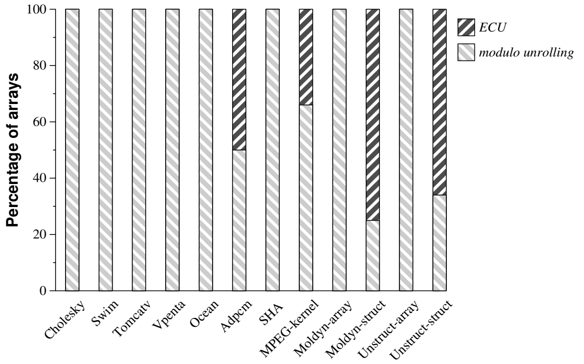

Figure 9: Percentage of arrays whose references are promoted through modulo unrolling versus those that are promoted through equivalence class unification (ECU).

The results in Figure 8 may have implications beyond Raw. They show that applications with a lot of ILP often have high memory bandwidth requirements. These applications would perform poorly on a system with many functional units but limited memory bandwidth. A Raw machine with trivial static promotion fits this architectural description, as do superscalars and centralized VLIWs with centralized memory systems. In addition, the Raw machine with trivial promotion suffers high memory latency due to a lack of locality between the processors and the single memory; this latency is analogous to the multi-cycle on-chip wire delays conventional designs will likely suffer in future VLSI technologies. Faced with similar problems, conventional architectures may well find that a software-exposed distributed memory system combined with a Maps compiler can improve its performance the same way it improves the performance of a Raw machine.

Memory distribution and utilization We measure the effectiveness of our compilation techniques in utilizing the Raw hardware. We consider two metrics: memory distribution and memory bandwidth utilization. In general, balanced data distribution is desirable because it minimizes the per-tile memory needed to run an application, and it alleviates the need to build large and centralized memory which is also fast. Memory bandwidth utilization measures how well an application takes advantage of Raw's independent memory banks. It depends on the amount of memory parallelism exposed by Maps and the amount of parallelism in the application.

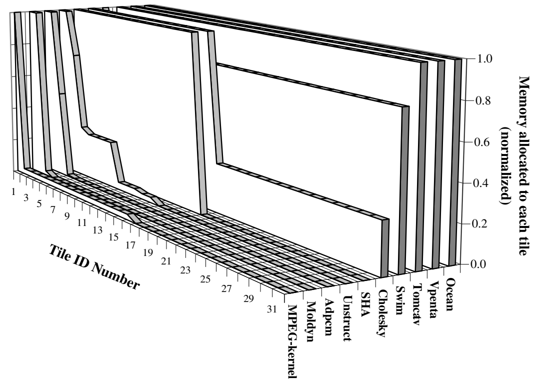

Figure 10: Distribution of primary data on a 32-tile Raw machine. The tiles are sorted in decreasing order of memory consumption. For each benchmark, the graph displays the memory consumption on each tile normalized by the memory consumption of the tile with the largest consumption.

Figure 10 shows the distribution of primary data across tiles for our benchmarks executing on 32 tiles. Most of the dense matrix codes are able to fully distribute their data; Swim and Cholesky are only able to partially distribute their data because of their small problem sizes, but their distributions become balanced with larger problem sizes. MPEG-kernel cannot employ only static memory accesses and still be fully distributed, but the next subsection shows that with a small sacrifice in performance, its data can be fully distributed. The rest of the applications can partially distribute their data across a range of three to sixteen tiles.

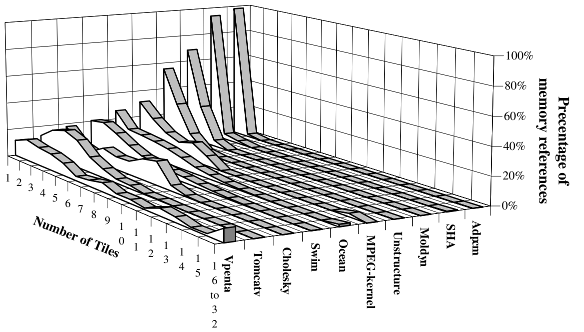

Figure 11: Weighted bandwidth utilization of the memory system on a 32-tile machine. The graph displays the percentage of memory references being issued in a time slot when a given number of tiles is issuing memory requests.

Figure 11 measures the weighted memory bandwidth utilization of a 32-tile machine. It plots the percentage of memory references being issued in a clock cycle when a given number of tiles is simultaneously issuing memory requests. Results show that except for the two highly serial

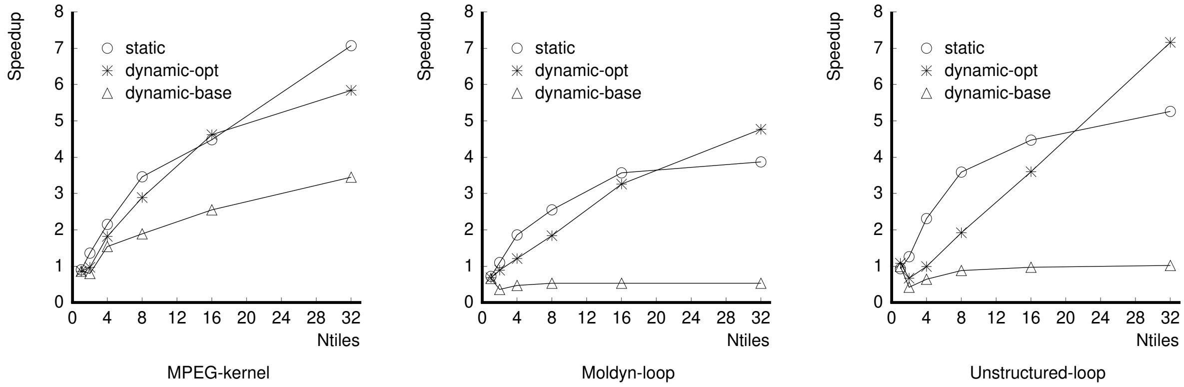

Figure 12: Speedups of benchmarks with optimized dynamic accesses.

benchmarks (Adpcm and SHA), all the benchmarks are able to exploit at least a small amount of parallel memory bandwidth. On the positive end, MPEG-kernel and all the dense matrix applications have at least 20% of their accesses being performed on cycles which issue five or more accesses, with Vpenta enjoying 10-way memory parallelism for over 20% of its accesses.

Exposing memory parallelism through dynamic accesses Static accesses are usually better than dynamic accesses because of their low overhead. Sometimes, however, static accesses can only be attained at the expense of memory parallelism. MPEG-kernel, Unstructured, and Moldyn are benchmarks with irregular accesses which can take advantage of high memory parallelism. This section examines the opportunity of increasing the memory parallelism of these programs by distributing their arrays and using dynamic accesses to implement parallel, irregular accesses.

Benchmark	N=1	N=2	N=4	N=8	N=16	N=32
MPEG-kernel	0.86	0.80	1.54	1.89	2.55	3.45
Moldyn	0.93	0.44	0.54	0.61	0.67	0.68
Unstruct	0.76	0.24	0.37	0.55	0.73	0.83

Table 4: Benchmark speedup with all arrays distributed, with irregular array references implemented through dynamic accesses with software serial ordering.

Table 4 shows the performance of the aforementioned benchmarks when all arrays are distributed. Irregular accesses are implemented through dynamic accesses, with software serial ordering to ensure correctness. Results for Moldyn and Unstructured are poor, with slowdowns for all configurations. MPEG-kernel attains speedup but is twice as slow as its purely static speedup. This result is not surprising: dynamic accesses serialized through a turnstile is provably slower than corresponding static accesses serialized through a memory node. To reap benefit from the exposed memory parallelism, serialization of dynamic accesses has to be reduced through epoch and update optimizations. Currently, epoch generation has not been automated, so our evaluation of these techniques uses a hand-coded implementation of epochs. To simplify this task, we apply our optimizations on a selected loop from each of Moldyn and Unstructured, in addition to the full MPEG-kernel. The loop we select from Moldyn accounts for 86% of the run-time. In Unstructured, many of the loops with irregular accesses have similar structure; we select one such representative loop. Figure 12 shows the performance of dynamic references when epoch and update optimizations are applied to these applications, compared with the unoptimized dynamic performance and the static performance. It shows that the dynamic optimizations are effective in reducing serialization and attaining speedup. All three benchmarks benefit from epochs, while Moldyn and Unstructured benefit from updates as well. Together, the optimizations completely eliminate the turnstile serialization for these applications.

The speedup trends of these applications reflect the amount of available memory parallelism. For static accesses, the amount of memory parallelism that can be exposed through ECU is limited to the number of alias equivalence classes. Depending on the access patterns, the amount of useful memory parallelism may be less than that. This level of memory parallelism does not scale with the number of tiles. For a small number of tiles, ECU is able to expose enough parallelism to satisfy the number of processing elements. But for larger number of tiles, insufficient memory parallelism causes the speedup curve to level off.

In contrast, the use of dynamic accesses allow arrays to be distributed, which in turn exposes memory parallelism scalable with the number of tiles. As a result, the speedup curve for optimized dynamic scales better than that for static. For up to 16 tiles, static outperforms optimized dynamic; for 32 tiles, optimized dynamic actually outperforms static, and the trend suggests that optimized dynamic will increasingly outperform static for even larger number of tiles. Note that for the dynamic experiment, only the irregular accesses were selectively made dynamic, the affine array accesses and all scalar data were still accessed on the static network.

Why do we need software serial ordering? As discussed in the previous section, dynamic accesses using software serial ordering can never perform better than static accesses promoted through ECU. This section shows how software serial ordering can be useful, using an example from Unstructured.

Unstructured contains an array $X[]$ which is accessed in only two loops, an initialization loop ($init$) and a usage loop (use). The initialization loop makes irregular accesses to $X[]$ and is executed only once. The usage loop makes affine accesses to $X[]$ and is executed many times. For best performance, Maps should optimize the placement of $X[]$ for the usage loop.

Array mapping	Loop	Access type	Speedup
centralized	init	static serial	1.89
	use	static serial	3.86
	total	–	3.85
distributed	init	dynamic serial	0.59
	use	static parallel	4.43
	total	–	4.42

Table 5: An example of overall performance improvement through the use of software serial ordering. Software serial ordering enables Maps to distribute a critical array, which optimizes for static parallel access in the critical use loop in exchange for dynamic accesses with software serial ordering in the non-critical $init$ loop. Performance is measured for 32 tiles.

Table 5 compares the performance of the loops when $X[]$ is placed on one tile to when it is distributed across 32 tiles. When the array is centralized, both $init$ and use attain speedups because they enjoy fast static accesses. When the array is distributed, however, $init$ suffers slowdown because it has dynamic serial accesses going through a turnstile, while use attain better speedup compared to the centralized case. For the full program, however, the performance of use matters much more. Thus, distributing $X[]$ provides the better overall performance, despite the overhead $init$ incurs from software serial ordering.

This example illustrates the general use of software serial ordering. It is a way of enforcing dynamic dependences which is more efficient than other mechanisms such as complete serialization or placing barriers between the dependent accesses. It is used not to improve the performance of the code segment employing it, but as an enabling mechanism to allow the compiler to improve the parts of the program that really affect performance. It provides a universal and efficient handling of dynamic accesses in the absence of applicable optimizations. The overall utility of dynamic accesses remains to be seen, but its use with software serial ordering provides a reasonable starting point on which further optimizations can be explored.

7 Related work

Other researchers have parallelized some of the benchmarks in this paper. Automatic parallelization has been demonstrated to work well for dense matrix scientific codes [6]. In addition, some irregular scientific applications can be parallelized on multiprocessors using the inspector-executor method [3]. Typically these techniques involve user-inserted calls to a runtime library such as CHAOS [11], and are not automatic. The programmer is responsible for recognizing cases amenable to such parallelization, namely those where the same communication pattern is repeated for the entire duration of the loop, and inserting several library calls.

In contrast, the Rawcc approach is more general and requires no user intervention. Its generality stems from its exploitation of ILP rather than coarse-grain parallelism targeted by [3] and [6]. Multiprocessors are mostly restricted to such coarse-grain parallelism because of their high communication and synchronization costs. Unfortunately, finding coarse grain parallelism often requires whole program analysis by the compiler, which works well only in restricted domains. A Raw machine can successfully exploit ILP because of the register-like latencies of the static network. Of course, Raw can exploit coarse-grain parallelism as well.

Software distributed shared memory schemes on multiprocessors (DSMs) [4] [13] are similar in spirit to Map's software approach of managing memory. They emulate in software the task of cache coherence, one which is traditionally performed by complex hardware. In contrast, Maps turns sequential accesses from a single memory image into decentralized accesses across Raw tiles. This technique enables the parallelization of sequential programs on a distributed machine.

Static promotion is related to memory bank prediction, a term used by Fisher [5] for a point-to-point VLIW model. For such VLIWs, he shows that successful disambiguation allows an access to be executed through a fast "front door" to a memory bank, while a non-disambiguated access is sent to a slower "back door." Most VLIWs today, however, use global buses rather than point-to-point networks. The lack of point-to-point VLIWs seems to explain the dearth of work on memory bank disambiguation for VLIW compilation.

A different type of memory disambiguation, relative memory disambiguation, is relevant on the more typical bus-based VLIW machines such as the Multiflow Trace [8]. Relative memory disambiguation aims to discover whether two memory accesses never 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.

Modulo unrolling is related to an observation made by Fisher [5]. He observes that unrolling can sometimes help disambiguate accesses. Based on this observation, his compiler relies on user annotations to determine the unrolling factor needed for such disambiguation. In contrast, modulo unrolling is a fully automated and formalized technique which computes the necessary unrolling factors needed to perform such disambiguation for dense matrix codes. It includes a precise specification of the scope of the technique and a theory to predict the minimal required unroll factor [1].

8 Conclusion

Raw microprocessors are designed for aggressive on-chip memory performance. They distribute their memory and processing resources over a large number of on-chip tiles coupled with a point-to-point interconnect. To retain hardware simplicity, the distributed memory system is exposed to the compiler, so it can provide the abstraction of a unified memory system to support traditional sequential programming models.

This paper addresses the challenging compiler problem of orchestrating distributed memory and communication resources to provide a uniform view of the memory system. We present a compiler-managed memory system called

Maps that provides a sequential memory abstraction to the programmer. The Maps solution attempts to maximize both memory parallelism and its use of the static interconnect.

Through the application of equivalence class unification and modulo unrolling, we demonstrate that Maps is able to statically promote the memory references in our regular scientific applications while obtaining ample amounts of memory parallelism, as evidenced by the speedups of about 20 on 32 tiles. Surprisingly, we find that the same techniques are also able to statically promote the memory references in our irregular applications and achieve sufficient memory parallelism to yield speedups of about 5 on 16 or more tiles. There are two reasons for this result: first, even irregular applications contain a modest amount of affine memory accesses, and they usually contain several distinct equivalence classes, each of which can be unified on a different tile. Second, the register-like latency of the static interconnect makes it possible to extract meaningful speedups on applications with small amounts of parallelism. This is an important result because it suggests the feasibility of 8-tile or 16-tile general purpose microprocessors using an all-static interconnect.

Further, we show that selective use of dynamic references may be helpful in certain cases to augment static promotion, as described in section 4.3. One example is when dynamic support allows arrays with non-affine accesses to be distributed, possibly exposing more memory parallelism and attaining better speedups. Another is to use dynamic accesses for infrequent irregular references to arrays, allowing more frequently accessed portions to be static promoted via modulo unrolling. Finally using dynamic accesses for a few "bad references" may prevent excessive merging of equivalence classes, yielding higher memory parallelism. Software serial ordering is introduced as an efficient method of enforcing dependences between dynamic accesses.

We are encouraged by the results of the Maps approach to memory orchestration for both the regular and the irregular benchmarks we have executed on the system. We demonstrate a high degree of speedup for regular programs and modest speedups for irregular applications. If the results for more programs continue to be positive, our software-based Maps approach will be a viable competitor to hardware supported coherent memory systems for single chip micros.

Acknowledgments

We thank Martin Rinard and Radu Rugina for their pointer analysis package, Michael Taylor and Elliot Waingold for implementing the Raw simulator and debugger, Matthew Frank for related discussions and providing several benchmarks, and other members of the Raw group for their support. We also thank the anonymous reviewers for their constructive comments. This research was funded by ARPA grant #DBT63-96-C-0036.

References

[1] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal. Memory Bank Disambiguation using Modulo Unrolling for Raw Machines. In *Proceedings of the ACM/IEEE Fifth Int'l Conference on High Performance Computing(HIPC)*, Dec 1998. Also http://www.cag.lcs.mit.edu/raw/.

[2] D. Burger, J. Goodman, and A. Kagi. Memory Bandwidth Limitations of Future Microprocessors. In *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA'96)*, pages 78–101, May 1996.

[3] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architectures. *Journal of Parallel and Distributed Computing*, 22(3), September 1994.

[4] S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 186–197, Cambridge, Massachusetts, October 1–5, 1996.

[5] J. A. Fisher. Very Long Instruction Word Architectures and the ELI-512. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 140–150, Stockholm, Sweden, June 1983.

[6] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, 29(12):84–89, Dec. 1996.

[7] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe. Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine. In *Proceedings of the Eighth ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 46–57, San Jose, CA, Oct. 1998.

[8] 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.

[9] D. Matzke. Will Physical Scalability Sabotage Performance Gains? *Computer*, pages 37–39, Sept. 1997.

[10] D. E. Maydan, J. L. Hennessy, and M. S. Lam. Efficient and Exact Data Dependence Analysis. In *Proceedings of the SIGPLAN '91 Conference on Program Language Design and Implementation*, Toronto, June 1991.

[11] S. Mukherjee, S. Sharma, M. Hill, J. Larus, A. Rogers, and J. Saltz. Efficient Support for Irregular Applications on Distributed-Memory Machines. In *Principles and Practice of Parallel Programming (PPoPP) 1995*, pages 68–79, Santa Clara, CA, July 1995. ACM.

[12] R. Rugina and M. Rinard. Pointer Analysis for Multithreaded Programs. In *Proceedings of the SIGPLAN '99 Conference on Program Language Design and Implementation*, Atlanta, May 1999.

[13] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, Cambridge, Massachusetts, October 1–5, 1996.

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

[15] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring It All to Software: Raw Machines. *IEEE Computer*, 30(9):86–93, Sept. 1997. Also available as MIT-LCS-TR-709.

[16] 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.