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, the compiler managed memory system for a Raw architecture. 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 and moves 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 is necessary to scale beyond 16 tiles for applications with irregular access patterns.*

[*] *Proceedings of the Twenty-Sixth International Symposium on Computer Architecture* **(ISCA-26)**, Atlanta, GA, June, 1999. This is the current draft. The camera-ready version will be available after Feb 19, 1999 from http://www.cag.lcs.mit.edu/raw/documents/

[†] This research is funded by Darpa contract # DABT63-96-C-0036.

1 Introduction

Rapidly improving VLSI technology places billion-transistor chips within reach in the next decade. Such large amounts of transistors expands the space of feasible architectural designs from what is possible today. One trend is to use such resources to build a powerful central 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. But such resources makes design and verification difficult. They have quadratic hardware complexity and connectivity, requiring long wires whose performance does not scale with technology. Moreover, the large area they consume provides diminishing return, and they are poorly suited for emerging stream and multimedia applications which demand simple but plentiful amount of computing resources and high-throughput IO.

A Raw microprocessor [16] adopts a different approach. 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 [9]. 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 [3] by moving much of its memory system on chip. For example, a billion-transistor chip with half its area devoted to memory can contain several tens of MBytes of SRAM. Using integrated DRAM allows at least four times that amount. Such an amount of memory makes it possible for the working set of many programs to be kept entirely on chip.

A critical design issue is how to organize such a large on-chip memory. Clearly, it is infeasible to build a fast, single-banked memory of that size. 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 becomes a multi-cycle operation just from the wire delay alone [11].

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 with the number processing elements. It supports fast local memory accesses without the need for 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.

The traditional approach of maintaining a unified memory abstraction on distributed memory machines is to synchronize through the memory system using hardware primitives such as locking. This mechanism, however, is expensive in two respects. In terms of hardware, it relies on cache coherence hardware to avoid making a remote memory access every time a lock is queried. In terms of execution cost, the mechanism is heavyweight, which limits the type of programs it can profitably execute to those with access patterns that require few synchronizations [1] [4]. The approach we present implements the memory abstraction with simple and less expensive hardware primitives through intelligent use of the static network. This enables arbitrary sequential programs to be parallelized across distributed processing elements.

We have implemented a SUIF-based [17] compiler that implements Maps by incorporating static promotion and software serial ordering. We have measured the base costs of various techniques and evaluated the system 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 are able to attain significant performance improvement from the improved bandwidth and locality provided by the distributed memory. Second, though static accesses usually yield superior performance because of their low overheads, selective use of dynamic accesses can benefit applications with irregular access patterns and high memory-bandwidth requirements.

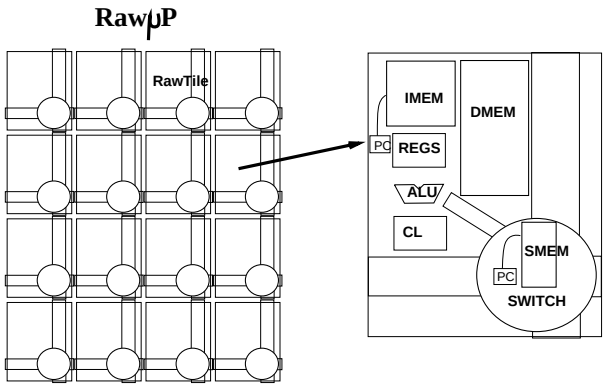

Figure 1: A Raw microprocessor is a mesh of tiles, each with a processing element, some memory and a switch. The processing element contains registers, ALU, and configurable logic (CL). It interfaces with its local instruction and data memory as well as the switch. The switch contains its own instruction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

The rest of this paper is organized as follows. Section 2 briefly describes the Raw architecture and its mechanisms for accessing memory. Section 3 overviews Maps, explaining the issues it faces, the solutions it adopts, and its organization in the context of the Raw compiler. Section 4 describes the traditional analysis techniques leveraged by Maps. Section 5 describes techniques for static promotion. Section 6 describes support for dynamic accesses. Section 7 presents the results. Section 8 discusses the related work, and Section 9 concludes.

2 Raw architecture and memory mechanisms

The Raw architecture [16] 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 com-

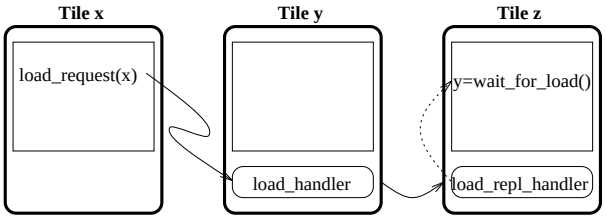

Figure 2: Anatomy of a dynamic load. A dynamic load is implemented with a request and a reply dynamic message. Note that the request for a load needs not be on the same tile as the use of the load.

ponents; 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.

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, every dynamic instance of the reference must refer to memory on the same tile. 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 places the data value onto the output port of its static switch. The precompiled instruction streams of the static network route the data value through the network to the processor needing the data. That processor accesses its static input port to get the data.

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 fol-

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

Table 1: Cost of memory operations.

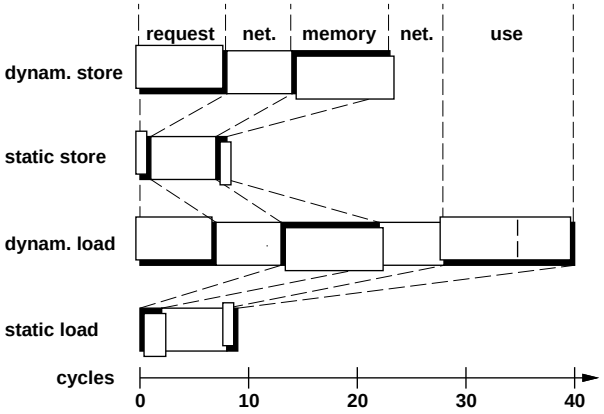

Figure 3: Breakdown of the cost of memory operations between tiles four units apart. Highlighted portions represent processor occupancy, while unlifted portions portion represents network latency.

lowed 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 1 lists the end-to-end cost of memory operations as a function of the tile distance. It includes both the processing cost and the network latency. Figure 3 breaks down these costs for a tile distance of four. 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, compile-time ordering and scheduling of static messages via blocking ob-

viates the need for such mechanisms.

3 Overview of Maps

This section overviews Maps, Raw's compiler-managed memory system. It highlights the issues in the design of such a system and summarizes Raw's solutions to these issues. It also gives an overview of the Raw compiler with a focus on its functionality related to Maps.

3.1 Issues in a compiler-managed memory system

The goal of Maps is to provide efficient use of the hardware memory mechanisms while ensuring correct execution. This goal hinges on three issues, the identification of static accesses, the support for memory parallelism, and the efficient enforcement of memory dependences. Identifying static accesses is important because static accesses are much faster than dynamic accesses. Compared to dynamic accesses, static accesses eliminate the overheads of synchronization, demultiplexing, and message packetization.

Adequate support for memory parallelism is defined with respect to instruction level parallelism. For any program, Maps should provide enough memory parallelism to exploit the program's ILP. Memory parallelism can be increased by distributing across tiles data which are frequently accessed together. However, data distribution should be done in a manner which facilitates easy static promotion.

For the correct serial execution of memory dependences, three types of dependences must 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 processors 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. The challenge is to provide sufficient mechanism to ensure this serialization while inhibiting performance as little as possible.

The Maps system focuses on effective solutions to these problems. It actively converts memory references into static references in a process called *static promotion*. Static promotion employs two new techniques: *equivalence class unification*, which promotes references through intelligent placement of data objects guided by traditional pointer analysis; and *modulo unrolling*, which promotes references through loop unrolling and intelligent placement of arrays. These techniques are described in Section 5. Maps provide memory parallelism by mapping data to different tiles whenever doing so does not interfere with static promotion. To efficiently enforce dependences involving dynamic references, the compiler employs a new technique termed *software serial ordering* to enforce dependences between dynamics, and it uses explicit synchronization through the static network to enforce a dependence between a static and a dynamic access. Optimizations on baseline software serial ordering are provided by *epochs* and memory update operations. These supports for dynamic accesses are described in Section 6. By addressing these central issues, the Raw compiler enables fast accesses in the common case, while allowing efficient and parallel accesses for both the static and dynamic mechanisms.

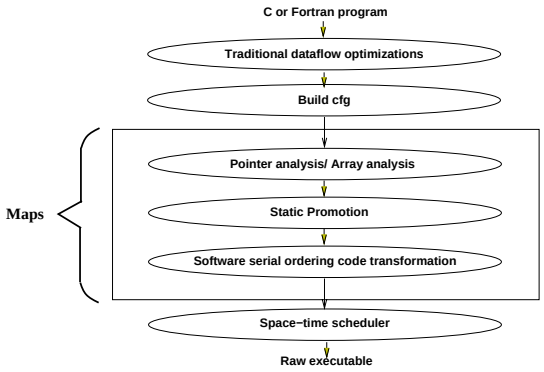

Figure 4: Structure of the Raw compiler

3.2 Compiler overview

Figure 4 outlines the structure of Rawcc, the Raw compiler built on top of SUIF [17]. 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 [9]. Maps uses the information provided by traditional pointer and array analysis to perform static promotion and software serial ordering. The space-time scheduler parallelizes each basic block of the program across the processors, obeying dependence and serialization requirements specified by Maps.

4 Analysis techniques

Throughout the memory system, Maps employs traditional analysis techniques to enhance the effectiveness of its mechanisms. The techniques include pointer analysis and array analysis. This section briefly presents the information provided by these techniques.

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 [13]. Pointer analysis is used to provide *location set lists* for every pointer in the program, where every location set corresponds to an abstract data object in the program. A pointer's location set list is a list of abstract data objects to which it can reference. We use this information to derive *alias equivalence classes*, which are groups of pointers related through their location set lists. [1] Pointers in the same alias equivalence class can potentially alias to the same object, while pointers in different equivalence classes can never reference the same object.

[1] More formally, alias equivalence classes are the connected components of a graph whose nodes are pointers and whose edges are between nodes whose location set lists have at least one common element.

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 accesses through pointers with common elements in their location sets 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.

Figure 5 presents a running example we will use in the rest of this paper to give a step-by-step illustration of the Maps compilation process. Figure 5(a) shows the initial code fragment, which contains five memory accesses $A[4]$, $*p$, $B[x]$, $C[y]$ and $*q$. Figure 5(b) shows the information derived after pointer and array analysis, which includes location set lists, alias equivalence classes, and data dependences. The location set lists are generated by pointer analysis in the context of a full program which is not shown. The remaining parts of the figure will be explained in subsequent sections.

5 Static promotion of memory accesses

Static references are references that can be determined at compile time to always refer to memory on the same tile. These references can employ the fast path to the memory system, either as local memory references or remote references which can be routed through the static network. Maps aims to make most memory references static. Without analysis, memory references yield no information and must by default be dynamic. This section describes two techniques for inducing *static promotion*, the process of making a memory reference static via careful data layout and code transformation. The result of static promotion is transformed code that has a fixed known processor number for each promoted access. Section 5.1 describes equivalence class unification, a general promotion technique based on the use of pointer analysis to guide the placement of data. Section 5.2 describes modulo unrolling, a code transformation technique applicable to most array references in the loops of scientific applications. Section 5.3 explains the limitation of static promotion and motivates the need for an efficient dynamic fall-back mechanism.

5.1 Equivalence class unification

Equivalence class unification (ECU) is a static promotion technique based on pointer analysis. It uses the alias equivalence classes provided by pointer analysis to help guide the placement of data. ECU promotes all pointers 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 larger objects such as structs and arrays are often accessed close together in the same program. Distribution and static promotion of arrays are addressed in Section 5.2. For structs, Maps breaks them up into their sub-components before running the program through pointer analysis. This technique allows pointer analysis to provide information on finer grained objects, often yielding more equivalence classes which can be mapped to different tiles, thus improving memory parallelism.[2]

5.2 Modulo unrolling

The major limitation of equivalence class unification is that arrays are treated as single objects belong to a single equivalence class. Mapping an entire array to a single processor 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 the array are interleaved in a round-robin manner across successive tiles on the Raw machine. 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-processor Raw machine is shown in Figure 6(b). In the loop, successive A[i] accesses go to processors 0, 1, 2, 3, 0, 1, 2, 3 ... . The edges out of tiles in Figure 6(b) point to the program accesses which refer to that tile. As we can see, the A[i] access in Figure 6(a) refers to memory 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 processor. Specifically, A[i] always refers to processor 0, A[i+1] to processor 1, A[i+2] to processor 2, and A[i+3] to processor 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 [2].

5.3 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 is undesirable to promote all references. First, in rare cases modulo unrolling may require unrolling of multi-dimensional loops, which results in excessive code expansion. Some accesses in these loops can be made dynamic to reduce the unrolling require-

[2] This technique is currently hand applied and is in the process of being automated.

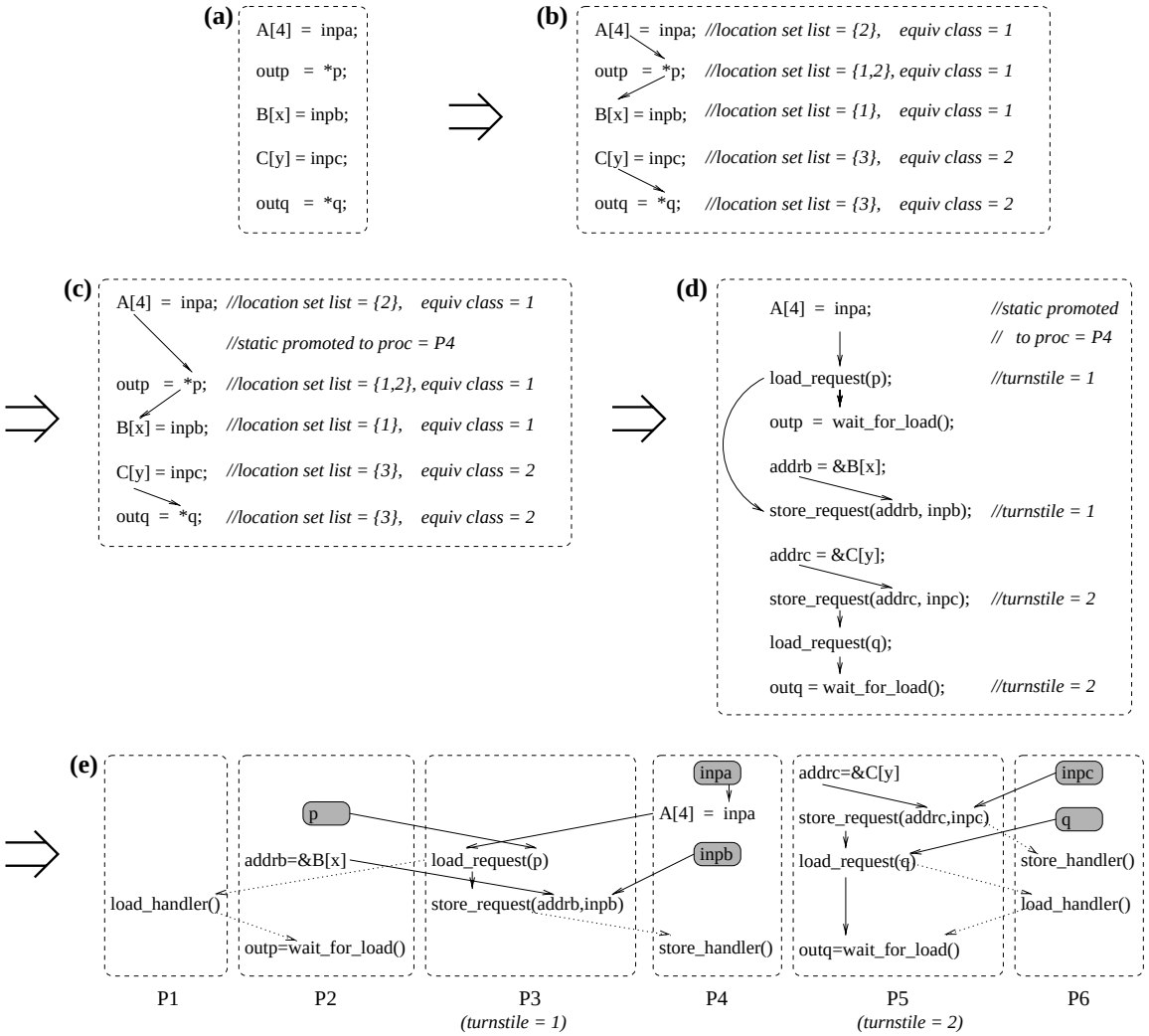

Figure 5: Example of memory system compilation. (a) initial code; (b) after analysis; (c) after static promotion; (d) after software serial ordering, which includes turnstile assignment, split-phase code generation and dependence inheritance; (e) one possible outcome after partitioning.

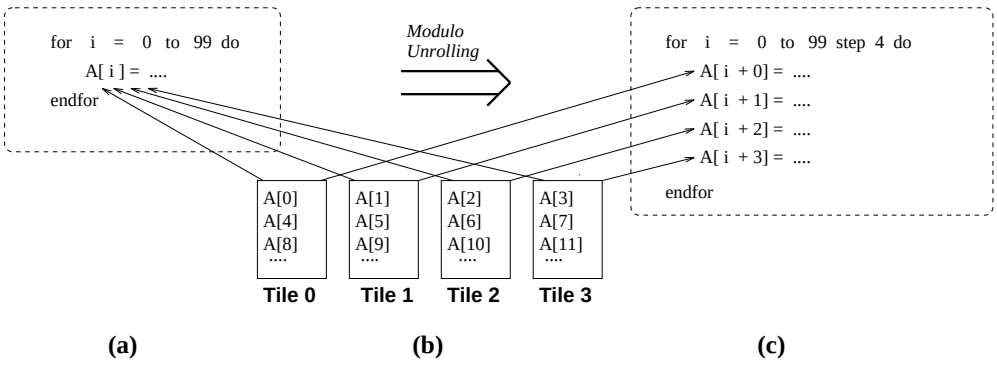

Figure 6: Example of modulo unrolling. (a) shows the original code; (b) shows the distribution of array A on a 4 processor Raw machine; (c) shows the code after unrolling. After unrolling, each access refers to locations in only one processor.

ment. 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 distribute the array and implement these indirect accesses dynamically, which yields better parallelism at the cost of higher access latency. Moreover, dynamic accesses can be useful not for speeding up the portion of the program which execute them, but for enabling static memory parallelism in a more critical part of the program. For example, arrays with mostly affine accesses but few irregular accesses benefit from dynamic accesses for this reason. Finally, dynamic accesses can increase the power 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, which enables better data distribution and improves 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 all accesses static by default. Automatic detection of situations which can benefit from dynamic accesses is still ongoing research. However, Section 7 shows two programs whose performance can be improved when dynamic accesses can be selectively employed.

Continuing our running example, Figure 5(c) shows the code fragment after static promotion. For the sake of illustration, we assume that only $A[4]$, a simple affine access, is promoted. Other references either belong to alias equivalence class which did not go through equivalence class unification, or they are non-affine array accesses to distributed arrays.

6 Support for dynamic accesses

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

6.1 Enforcing dynamic dependences

This section describes the mechanisms used by Maps to enforce possible memory dependences involving dynamic accesses. Maps handles these dependences with two separate mechanisms, one for dependences between a static access and a dynamic access, and one for 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 dependence relation orders a static before a dynamic, a synchronization message is sent at the completion of the static memory operation to the issue of the dynamic operation. When a dependence relation orders a dynamic before a static, a synchronization message is sent at the completion of the dynamic reference to the static reference. If the dynamic reference is a store, this synchronization requires a store acknowledgment message. Completion of the dynamic reference then corresponds to either a load reply or the store acknowledgment. The important feature of this mechanism is that the source and destination of the synchronization message is known at compile-time, so that 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, by waiting for the earlier access to send back a dynamic acknowledgment from the remote memory tile before latter request is issued. While correct, this solution is expensive because it serializes the slow round-trip dynamic requests.

We propose *software serial ordering* to efficiently ensure such dependences. The technique leverages the in-order delivery of messages on the dynamic network between any source-destination pair of tiles. It works as follows. As explained in Section 2, a dynamic load is converted into a split-phase transaction with distinct load-request and load-reply operations, while a dynamic store is converted into a store-request. 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, the 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 computation, latency of memory request and reply, and processing time of memory requests to different tiles. Furthermore, different equivalence classes can employ different turnstile processors 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 run-time

addresses. Note that while software serial ordering handles dynamic accesses, it leverages the fast static network on Raw to handle synchronization and data transfer for good performance.

Figure 5(d) shows the result of the software serial ordering transformation on the code in 5(c). It shows the four dynamic accesses (*p, B[x], C[y] and *q) converted to split-phase transactions using a request/reply model. For simplicity, the load_reply handler is not shown; instead the request directly points to the wait_for_load. The figure also shows the turnstile assignments. Finally, the dependence edges are inherited from Figure 5(c) in the obvious manner. Additional dependences are placed to serialize the requests assigned on the same turnstile as required by software serial ordering.

Figure 5(e) shows one possible outcome after space-time scheduling. The computation is distributed on six processors, P1 through P6. The dynamic requests are serialized on turnstiles 1 on P3 and 2 on P5. The $A[4]$ static access has been promoted to P4 as required. All other computation is partitioned in a manner which exploits parallelism while respecting dependences. The shaded nodes represent loads of input variables at their latest locations. The interrupt-driven remote memory handlers are run on tiles resolved at run-time and unknown at compile-time. The dotted edges represent dynamic messages. Note that much of the work between different memory operations can proceed in parallel, including the address computations, wait_for_loads, and memory handlers on different tiles. Note in particular that potentially dependent memory operations, such as load to p and store to $addrb$ on turnstile 1, can proceed in parallel once they go through the turnstile.

6.2 Dynamic optimizations

Epochs Without optimizations, all dynamic memory requests in a single alias equivalence class has to go through the turnstile for the entire duration of the program. The reason is the following. To schedule a dynamic memory request on a tile, the compiler has to guarantee that all its preceding dependent dynamic accesses have completed from the view of that tile. The only tile on which such information is readily available is the turnstile of the memory request.

Sometimes, however, Maps can determine that the dynamic memory accesses to an alias equivalence class in a region of the program are all independent. Loads are trivially independent of each other. When stores are present, pointer and array analysis may help prove the independence. Pointer analysis can determines two pointers to be independent if they have non-intersecting location set lists. Relative memory disambiguation can determine that $*p$ and $*(p+1)$ always access different locations if the value of p does not change between the accesses. Additionally, array analysis can disambiguate accesses to the same array that pointer analysis cannot disambiguate.

Epochs allow Maps to take advantage of such independence information. Epochs refer to sub-regions in a program across which no dynamic accesses can cross. They are enforced by explicit checking the completion of all accesses through load-replys or store-acknowledgments. When Maps identifies program regions which can benefit from parallel accesses in an alias equivalence class, it can make it an epoch and allow completely parallel accesses within the region.

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 improves 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 collapse two expensive and serial dynamic memory operations into one. Finally, the collapsing of the load and the store also eliminates the memory dependence between them. This elimination can help increase the utility of epochs by finding more regions with independent accesses to an alias equivalence class.

7 Results

Maps has been implemented on Rawcc, the Raw compiler based on the SUIF compiler infrastructure [17]. 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 [15] 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 [9]. Inner loops are unrolled as many times as there are number of tiles.

Table 2 gives the characteristics of the benchmarks used for the evaluation. Benchmarks include four 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, Mxm, and Vpenta are derived from Nasa7 of Spec92. Adpcm-encode is the compression part of the compression/decompression pair in Adpcm. MPEG-kernel is the portion of MPEG which takes up 70% of the total runtime. Because the Raw simulator currently does not support double-precision floating point, all floating point operations are converted to single precision.

Table 3 shows the speedups attained by the benchmarks for Raw microprocessors of a varying number of tiles. It shows that Rawcc with Maps is able to orchestrate the parallelism available in the applications. 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 modest amount of parallelism. They have parallelism both within loop iterations and

Benchmark	Type	Source	Lines of code	Seq. RT (cycles)	Primary Array size	Description
Cholesky	Dense Mat.	Nasa7:Spec92	126	34.3M	$3\times32\times32$	Cholesky Decomposition/Substitution
Mxm	Dense Mat.	Nasa7:Spec92	64	2.0M	32×64, 64×8	Matrix Multiplication
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
Adpcm-encode	Multimedia	Mediabench	133	7.1M	1000	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	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.

Benchmark	N=1	N=2	N=4	N=8	N=16	N=32
Cholesky	0.88	1.68	3.38	5.48	10.30	14.81
Mxm	0.94	1.97	3.60	6.64	12.20	23.19
Tomcatv	0.92	1.64	2.76	5.52	9.91	19.31
Vpenta	0.70	1.76	3.31	6.38	10.59	19.20
Adpcm-encode	0.94	1.05	1.26	1.33	1.26	1.15
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	0.75	1.14	1.82	2.43	3.21	3.42
Unstruct	0.88	1.34	2.63	4.12	5.34	5.96

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.

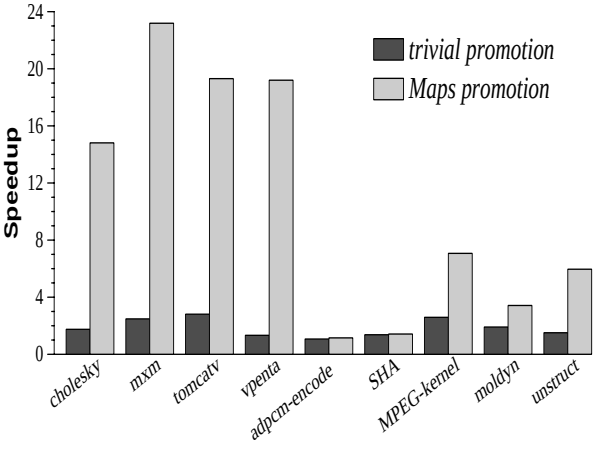

Figure 7: Comparison of 32-tile speedups for trivial static promotion vs Maps static promotion.

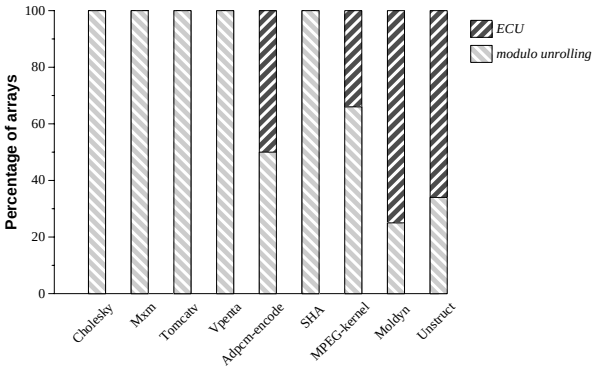

Figure 8: Percentage of arrays whose references are promoted through modulo unrolling versus those that are promoted through equivalence class unification (ECU).

across iterations, but loop carried dependences eventually limit the amount of parallelism exposed by unrolling. Note that the speedup for Moldyn was obtained without special handling of the reduction in its time-intensive loop. With reduction recognition, the speedup is likely to go up. Finally, Adpcm and SHA have little parallelism. Their work both within and across iterations are mostly serial.

Figure 7 compares the performance of static memory accesses with and without our static promotion analysis techniques. Without analysis, accesses can be promoted trivially by mapping all data to one tile. Results, however, show that such trivial promotion leads to poor performance. This is because trivial promotion provides fast static access at the expense of memory parallelism, which prevents the technique from attaining scalable performance. The Maps techniques, on the other hand, is able to perform static promotion while preserving the memory parallelism. Modulo unrolling exposes memory parallelism within array objects; equivalence class unification exposes memory parallelism between equivalence classes of data objects. Note that the performance of trivial static promotion is also analogous to the performance for a machine with a centralized memory system. They share the same fundamental problem of having memory bandwidth which does not scale with the number of functional units.

Figure 8 shows the percentage of arrays whose references are promoted through modulo unrolling versus those

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	0.69	0.41	0.52	0.58	–	–
Unstruct	0.84	0.51	0.71	0.96	1.06	–

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

that are promoted through ECU. All applications benefit from modulo unrolling. The dense matrix applications, as well as SHA, make regular memory accesses which can all be promoted through modulo unrolling. On the other hand, the other multimedia applications and the irregular scientific applications make both regular and irregular accesses, requiring both modulo unrolling and ECU for static promotion.

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.

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 – ranging from 2.5 slowdown to negligible speedups 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. Figure 9 shows the performance of dynamic references when these optimizations are applied, compared with the unoptimized dynamic performance and the static performance. Because automatic epoch generation is in progress and not yet complete, we obtain our results with hand-coded implementation of epochs. To simplify the hand implementation, we select a loop out of both Moldyn and Unstructured to apply the optimizations. 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. The figure 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. The amount of useful memory parallelism may be less than that, depending on the access patterns in the program. This amount 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.

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.

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 function accesses 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.

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 $X[]$

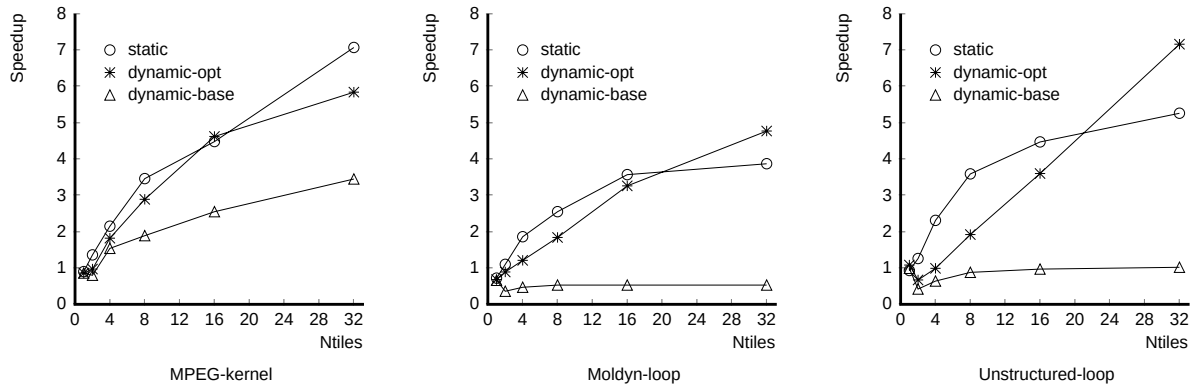

Figure 9: Speedups of benchmarks with optimized dynamic accesses.

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.

8 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 [8]. In addition, some irregular scientific applications can be parallelized on multiprocessors using the inspector-executor method [5]. Typically these techniques involve user-inserted calls to a runtime library such as CHAOS [12], 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 completely automatic. Its generality stems from its exploitation of ILP, rather than the coarse-grain parallelism targeted by [8] and [5]. Multiprocessors are mostly restricted to such coarse-grain parallelism because of their high costs of communication and synchronization. Unfortunately, finding such coarse grain parallelism often requires whole program analysis by the compiler, which works well only in restricted domains. Raw 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) [14] [6] 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 static memory bank disambiguation, a term used by Ellis [7] 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 an non-disambiguated access is sent over a slower "back door." Most VLIWs today, however, use global buses rather than point-to-point networks for communication. The lack of point-to-point VLIWs seems to explain the dearth of work on memory bank disambiguation for compiling for VLIWs.

A different type of memory disambiguation is relevant on the more typical bus-based VLIW machines such as the Multiflow Trace [10]. Relative memory disambiguation [10] aims to discover 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.

The modulo unrolling scheme used in Raw [2] is related to an observation made by Ellis [7]. 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 relies on user-identified array accesses and user annotations to provide the unroll factors needed. In contrast, modulo unrolling is a fully automated and formalized technique 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.

9 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 hard-

ware 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.

We further demonstrate that static promotion must be augmented with support for dynamic references when the number of tiles exceeds 16. Support for dynamic references allows arrays with non-affine accesses to be distributed, thereby exposing more memory parallelism and attaining better speedups. Software serial ordering is introduced as an efficient software mechanism for enforcing proper serialization of accesses to a distributed array in those parts of a program in which the accesses cannot be disambiguated, while benefiting from static promotion (or epoch optimization) in those parts of the program in which static (or relative) memory disambiguation can be performed on the same array.

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.

References

[1] 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.

[2] 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/.

[3] 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.

[4] R. Cytron. Doacross: Beyond vectorization for multiprocessors. Aug. 1986.

[5] 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.

[6] 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.

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

[8] 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. *COMPUTER*, 29(12):84–89, Dec. 1996.

[9] 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.

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

[11] D. Matzke. Will physical scalability sabotage performance gains? *Computer*, pages 37–39, Sept. 1997.

[12] 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.

[13] R. Rugina and M. Rinard. Span: A shape and pointer analysis package. Technical report, M.I.T. LCS-TM-581, June 1998.

[14] 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.

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

[16] 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.

[17] 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.