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}@lcs.mit.edu
{saman,agarwal}@lcs.mit.edu
http://cag-www.lcs.mit.edu/raw

Abstract

Microprocessors of the next decade and beyond will be built using VLSI chips employing billions of transistors. In this generation of microprocessors, achieving a high level of parallelism at a reasonable clock speed will require full distribution of machine resources. Raw architectures explore this architectural space by distributing all their processor and memory resources as a 2-D mesh of simple tiles. To provide a simple sequential programming model, a Raw architecture exposes the hardware fully and relies on the compiler or the software run-time system to achieve efficient execution while maintaining the semantics of a single instruction stream and a unified memory system.

Supporting a single view of memory on top of a distributed memory architecture presents a challenging compiler problem. This paper presents a compiler system called Maps that supports a unified view of memory on a Raw architecture. Maps relies on two inter-tile interconnects: a fast, statically schedulable network and a slower dynamic network. Maps attempts to schedule the memory accesses for maximum parallelism and speed while enforcing proper dependence. It optimizes for speed in two ways: by finding accesses that can be scheduled on the static interconnect through a process called *static promotion*, and by minimizing dependence sequentialization for the remaining accesses. Static accesses are discovered through applications of traditional pointer and array analysis, and a new technique called *modulo unrolling*. Maps enforces proper dependence through a combination of explicit synchronization and a technique called *software serial ordering*. We have implemented Maps based on the SUIF infrastructure. This paper presents preliminary results based on compiling several programs using Maps.

1 Introduction

We live in an age where Moore's law is as prophetic as Murphy's law. Bigger and more processors are being placed inside a single chip at a rate consistent with Moore's law. Within the chip boundary, however, shrinking technology makes the speed of basic logic faster and faster, while improvement in

[*]This research is funded in part by ARPA contract # DABT63-96-C-0036 and in part by an NSF Presidential Young Investigator Award.

MIT-LCS-TM-583, July 1998

wire speed lags behind [8]. These forces impose increasing relative penalties on complex logic and global wires. Consequently, locality and simplicity become the key principles of good hardware design.

Locality suggests that processing resources be fully distributed with near neighbor connectivity and the elimination of global buses. Simplicity suggests that complex hardware be eliminated from a processor. But simplicity in the hardware buys little unless simplicity at the programming level is retained as well. Accordingly, the architecture must be able to run existing sequential programs without requiring new programming methodologies. To support sequential programs, traditional processors use complicated hardware to provide features like memory reorder buffers and mechanisms to extract instruction-level parallelism, which violates the simplicity principle of hardware design.

The Raw alternative [12] is to expose fully the raw hardware to the compiler so it can implement the support functions for sequential programs, thereby maintaining hardware simplicity. For locality, a Raw architecture distributes not only the functional units and registers but memory as well, and it couples them with a simple point-to-point network.

Using a distributed processor architecture to run general-purpose sequential programs presents several interesting research issues. The scheduling of instruction-level parallelism, which includes both a spatial and a temporal component, has been addressed in [6]. From the memory's perspective, the distributed organization presents both an opportunity and a challenge. The opportunity is to be able to use compiler knowledge to optimize the references for locality, parallelism, and efficiency. The challenge is to ensure that the parallelized program has a coherent view of the memory system. In particular, potentially dependent memory accesses which may be spread across the processing tiles must be performed in correct sequential program order.

The traditional approach to enforcing dependence on parallelized programs 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] [3]. The approach we present solves the memory coherence problem with much less hardware, and enables arbitrary sequential programs to be parallelized on distributed, scalable machines.

This paper presents *Maps*, a compiler-managed memory system for Raw architectures. The Maps approach consists of a minimal-hardware, compiler-managed memory system built on top of distributed memory coupled by both a static and a dynamic network. Processors are also distributed along with the memory so that each processor is coupled to a local memory. The hardware provides two mechanisms to each processor for accessing memory: a fast path for local accesses or for memory accesses that can be routed using a fixed schedule through the static network; and a slower, fail-safe path for dynamic accesses over the dynamic network. Maps allows parallel accesses for both the static and dynamic mechanisms, while enforcing proper dependence through a combination of compiler analysis, explicit synchronization, and a new technique called *software serial ordering*. To efficiently use the provided mechanisms, the Maps system optimizes the performance of an application in two ways: by finding static accesses using a new process called *static promotion*, and by minimizing dependence sequentialization. These goals are realized through applications of traditional pointer and array analysis. A new technique called *modulo unrolling* further improves the discovery of static accesses.

We have implemented a SUIF-based compiler that implements Maps by incorporating software serial ordering and static promotion based on pointer analysis and modulo unrolling. We have been able to compile several programs and run them on a simulator of a Raw machine. Ongoing work is to push

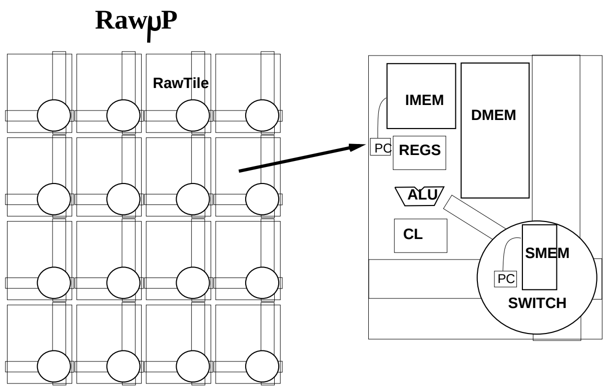


Figure 1: A Raw microprocessor.

several larger benchmarks through the system. This paper discusses the techniques employed by the compiler and presents the basic costs of various operations and a few application results. We demonstrate a high degree of speedup for several regular programs, and we show that software serial ordering is able to achieve moderate speedup in cases where dynamic accesses are necessary.

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 software serial ordering. Section 7 presents the results. Section 8 discusses the related work, and Section 9 concludes.

2 Raw architecture and memory mechanisms

The Raw architecture [12] is a simple, distributed, software-exposed architecture motivated by the desire to maximize the performance per silicon area of a machine. Together, distribution and simplicity enable a fast clock, and they maximize the amount of processing resources that can fit on a chip. Software-exposure allows the compiler to implement features traditionally implemented in hardware, such as memory coherence and the discovery of ILP.

Figure 1 depicts the layout of a Raw machine. The design features a two-dimensional mesh of identical tiles, each with its own processor, memory, and switch. The processor is a simple five-stage 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. The interface to this interconnect is fully exposed to the software.

Each switch contains two distinct networks, a static and a dynamic one. The static switch 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.

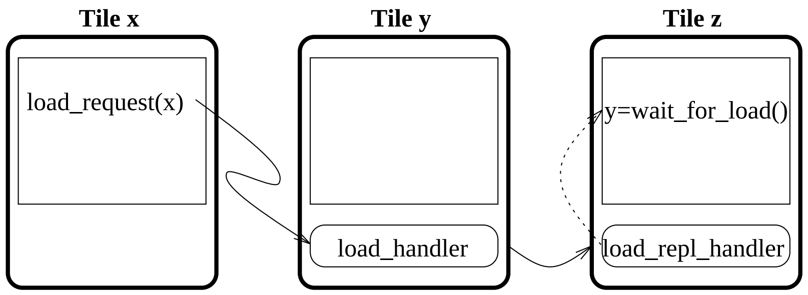


Figure 2: Anatomy of a dynamic load. A dynamic load is implemented with a request and a reply dynamic message. Note that the request for a load needs not be on the same tile as the use of the load.

This specification ensures correctness in the presence of timing variations introduced by dynamic events such as interrupts, 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 fallback 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 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.

3 Maps compiler managed memory system

This section overviews Maps, Raw's compiler-managed memory system. It highlights the two main 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.

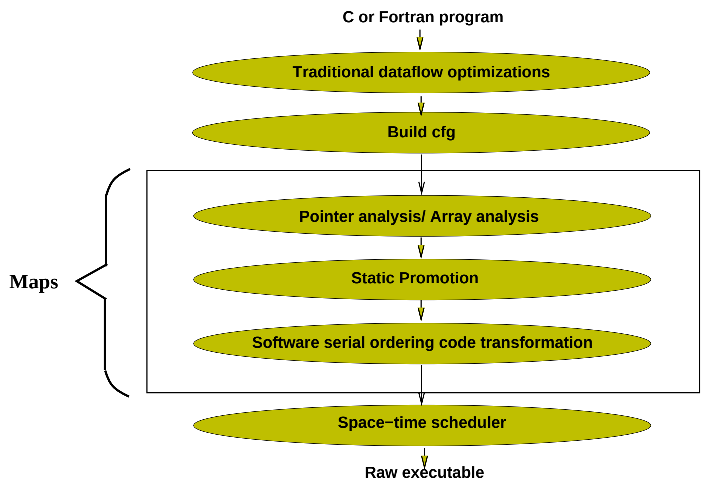

C or Fortran program
Traditional dataflow optimizations
Build cfg
Maps
Pointer analysis/ Array analysis
Static Promotion
Software serial ordering code transformation
Space−time scheduler
Raw executable

Figure 3: Structure of the Raw compiler

3.1 Issues of a compiler-managed memory system

The goal of Raw's compiler-managed memory system is to provide efficient use of the hardware memory mechanisms while ensuring correct execution. This goal hinges on two issues, the identification of static accesses and the efficient enforcement of memory dependences. Identifying static accesses is important because static accesses can employ the fast path to the memory system. Compared to dynamic accesses, static accesses eliminate the overheads of synchronization, demultiplexing, and message packetization.

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 two 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. 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. These techniques are described in Section 6. By addressing the two 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.

3.2 Compiler overview

Figure 3 outlines the structure of RAWCC, the Raw compiler built on top of SUIF [13]. 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. Maps uses the information provided by traditional pointer and array analysis to perform static promotion and software serial ordering. The space-time scheduler [6] parallelizes each basic block of the program across the processors, obeying dependence and serialization requirements specified by Maps.

4 Analysis techniques

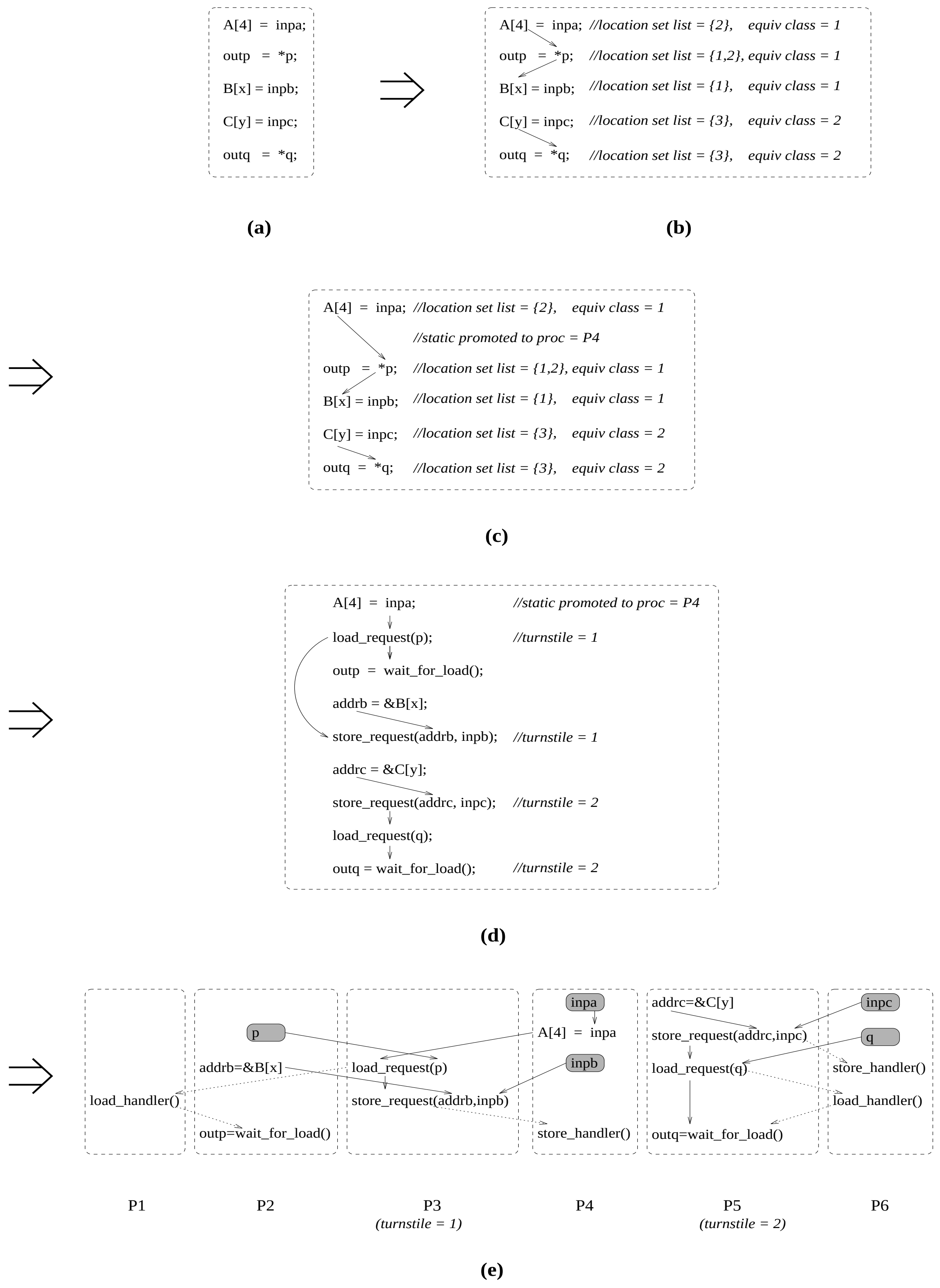
Throughout the memory system, the Raw compiler employs traditional analysis techniques to enhance the effectiveness of its mechanisms. The techniques include pointer analysis and array analysis. This section briefly presents the information provided by these techniques.

Pointer analysis is leveraged for three purposes: equivalence class unification, software serial ordering, and the minimization of dependence edges. The Raw compiler uses the SPAN, a state-of-the-art pointer analysis package [9]. It contains a notion of abstract data objects, and it associates a *location set number* to each data object. As output, the analysis phase marks each pointer reference with the list of location sets it can refer to, termed its *location set list*. The Raw compiler uses this information to derive two pieces of information, pointer equivalence classes and dependence information. A *pointer equivalence class* is a group of pointer accesses. An access within the class can refer to data referred by accesses in the same class, but it cannot refer to data referred by pointers outside of the class. The Raw compiler derives equivalence classes by first finding equivalence classes on location sets[1]. Based on this, the equivalence class of any access is simply the equivalence class of any of its location sets, all which are always the same. Dependence relations is derived by noting that only pointers in the same location set can interfere with each other. Therefore, dependence edges are only inserted between memory references if the intersection of their location set lists is non-empty.

The abstract data objects used in pointer analysis is sometimes too coarse grain. In particular, the technique does not distinguish between references to different elements in the same array. Therefore, we employ array data dependence analysis to gather dependence information on array accesses. As for equivalence classes, an array is usually a single equivalence class, because one array reference often refer to data in an entire array. Section 5.2 will discuss a novel transformation technique for increasing equivalence classes for array accesses.

Figure 4 presents a running example we will use in the rest of this paper to give a step-by-step illustration of the Maps compiler-managed memory system. Figure 4(a) shows the initial code fragment, which contains five memory accesses $A[4]$, $*p$, $B[x]$, $C[y]$ and $*q$. Figure 4(b) shows the information derived after pointer and array analysis, which includes location set lists, equivalence classes, and data dependence. The location set list for each access was generated by pointer analysis, which derived the actual values using the assumed context of the program (not shown). The dependence edges are introduced whenever the intersection of the location set lists was non-empty. The equivalence classes were derived using connected components, as noted earlier. The remaining parts of the figure, from 4(c) onwards will be explained later as we go along.

[1]Equivalence classes on location sets can be shown to be the connected components on a graph whose nodes are location set numbers and whose edges connect location sets which can be referred to by the same access.


```
A[4]  =  inpa;
outp   =  *p;
B[x] = inpb;
C[y] = inpc;
outq   =  *q;
```

(a)

```
A[4]  =  inpa;  //location set list = {2},    equiv class = 1
outp   =  *p;   //location set list = {1,2}, equiv class = 1
B[x] = inpb;    //location set list = {1},    equiv class = 1
C[y] = inpc;    //location set list = {3},    equiv class = 2
outq  =  *q;    //location set list = {3},    equiv class = 2
```

(b)

```
A[4]  =  inpa;  //location set list = {2},    equiv class = 1
                //static promoted to proc = P4
outp   =  *p;   //location set list = {1,2}, equiv class = 1
B[x] = inpb;    //location set list = {1},    equiv class = 1
C[y] = inpc;    //location set list = {3},    equiv class = 2
outq  =  *q;    //location set list = {3},    equiv class = 2
```

(c)

```
A[4]  =  inpa;                    //static promoted to proc = P4
load_request(p);                  //turnstile = 1
outp  =  wait_for_load();
addrb = &B[x];
store_request(addrb, inpb);       //turnstile = 1
addrc = &C[y];
store_request(addrc, inpc);       //turnstile = 2
load_request(q);
outq = wait_for_load();           //turnstile = 2
```

(d)

(e)

Figure 4: Example of compilation through 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.

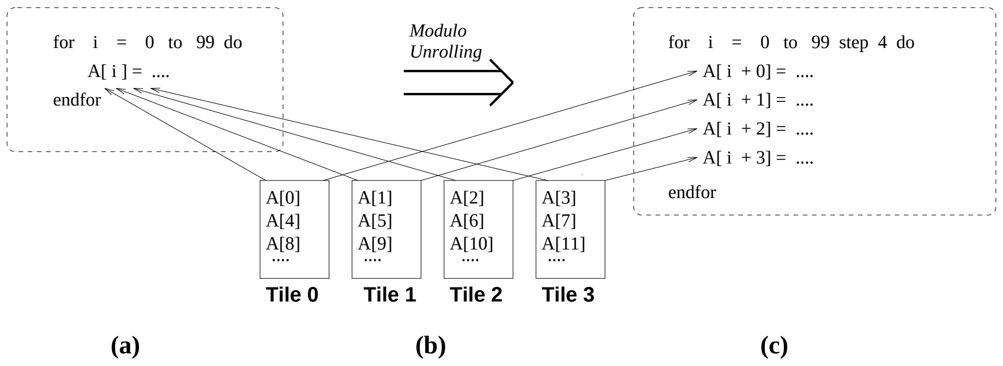


Figure 5: Modulo Unrolling Example.(a) shows the original code; (b) shows the distribution of array A on a 4 processor Raw machine; (c) shows the code after unrolling. After unrolling, each access refers to locations in only one processor.

5 Static promotion of memory accesses

Static references are references that always refer to memory on the same compile-time known tile location. 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. The Raw compiler aims to make most memory references static. Without analysis, all memory references carry 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

Standard pointer analysis can help guide the placement of data such that static promotion of accesses become possible. Equivalence classes can be used for static promotion as follows. All the data in an equivalence class can be mapped uniformly to the memory on a single tile. Then, references to that equivalent class will refer exclusively to that tile and can be statically promoted. By mapping every equivalent class in such a manner, all memory references can be statically promoted.

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 layed 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 which can statically promote all array references inside a loop. We illustrate this technique through a simple example. Consider the source code in Figure 5(a). Using low-order interleaving, the data layout for array A on a four-processor Raw machine is shown in Figure 5(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 5(b) point to the program accesses which refer to that tile. As we can see, the A[i] access in Figure 5(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 5(c) shows the result of unrolling the code in Figure 5(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. This technique is always applicable for loops with array accesses having indices which are affine functions of enclosing loop induction variables. For a detailed explanation and the derivation of the unrolling factor, see [2].

5.3 The need for dynamic references

A compiler can statically promote all accesses through equivalence-class unification alone, and modulo unrolling helps improve the distribution of data 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, resulting in excessive code expansion. In addition, static promotion may inhibit parallelism. Indirect array accesses of the form $A[B[i]]$, for example, cannot be promoted unless the array $A[]$ is mapped to a single tile. This mapping creates a memory hot-spot which inhibits exploiting parallelism in a program. If the array were distributed instead and accessed dynamically, we would lose the static references, but gain parallel access to the A array. The decision of what accesses in a program should be promoted is based on whole-program analysis, which looks at the requirements of different accesses weighted by their frequency, and attempts to minimize the total runtime.

Therefore, it is important to have a good fall-back mechanism for dynamic references. More importantly, it is important for such mechanisms to integrate well with the static mechanism. The next section explains how these goals are accommodated.

Continuing our running example, Figure 4(c) shows the result of static promotion. Only the $A[4]$ reference is promoted, as it is a simple affine function. For the sake of illustration, we assume that the compiler chooses not to promote the other four references.

6 Software serial ordering

The software system for implementing dynamic accesses is responsible for ensuring that all possible dependences involving dynamic accesses are satisfied. Such dependence can be between a static access and a dynamic access or between two dynamic accesses. This section describes the mechanisms for handling each type of dependence.

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. The reason is that otherwise a store completion on the dynamic network cannot be guaranteed. If the dynamic reference is a load, its reply guarantees completion. 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 processors. Furthermore, it would not suffice to synchronize the issues of the requests on different processors. 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, that is to wait for the earlier access to send back a dynamic acknowledgment from the remote memory tile, before latter request is issued. While correct, complete serialization is expensive because it serializes the slow round-trip dynamic requests.

We propose a technique called *software serial ordering* to efficiently ensure dependence. 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* processor. 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 processors must necessarily refer to different memory locations, so there is no memory dependence which needs to be enforced. Second, requests destined for the same processor are delivered in order by the dynamic network, as required by the network's in-order delivery guarantee. Finally, the memory processor 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.

Figure 4(d) shows the result of the software serial ordering transformation on the code in 4(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 reply directly points to the wait_for_load. The figure also shows the turnstile assignments. Finally, the dependence edges are inherited from Figure 4(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 4(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 and 2, assigned to processors P3 and P5. The $A[4]$ static access is placed on the processor it was promoted to, namely P4. All other computation is partitioned 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 processors resolved at run-time and unknown at compile-time. The dotted

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

Figure 6: Cost of memory accesses.

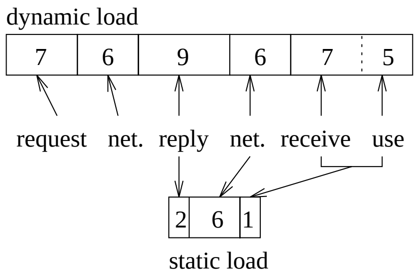


Figure 7: Breakdown of the cost of static and dynamic loads.

edges represent dynamic messages. Note that the partitioner can schedule the two turnstile nodes P3 and P5 with other associated computation relatively in parallel. On turnstile 1 on P3, the store_request does not have to wait for the previous load to reply before proceeding. Hence most of the latency in the load of p and the store of $addrb$ is overlapped with each other. Furthermore, all other computation, including the address computation, wait_for_loads, memory handlers, and unrelated computation, can be potentially scheduled in parallel.

7 Results

This section presents some initial results evaluating the performance of Maps and the Raw compiler. It first presents the end-to-end cost of static and dynamic memory accesses. It then measures the performance of dynamic memory accesses. Finally, it presents some preliminary application performance results.

Cost of memory accesses Figure 6 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. The result shows the considerable benefit of a static access over a dynamic one. Figure 7 breaks down the cost of static and dynamic loads for a tile distance of four. It shows that the overhead of a dynamic access is dominated by the protocol cost of sending and processing messages, as well the need to request data. A static access, on the other hand, leverages compiler-orchestration to eliminate most of that overhead. Note that much of the cost can be alleviated by prefetching.

Performance of dynamic accesses We evaluate the effectiveness of software serial ordering in supporting parallelism. Table 1 shows the ability to overlap successive loads issued on the same turnstile processor. It assumes that messages are zero-latency and free of contention overhead for both the network and the processor; thus it illustrates the maximal performance allowed by this approach. Our measurements show that of the 28 cycles, seven for the request are serialized and 21 can be overlapped. The maximal throughput of a turnstile is thus one request every seven cycles.

Number of issues	1	2	3	4	5	6
Cost/issue	28	17.5	14	12.25	11.2	10.5

Table 1: Average cost of memory references issued through one turnstile.

Serialization	P=1	P=2	P=4	P=8	P=16	P=32
Single turnstile	1.00	0.28	0.55	0.84	1.04	1.23
Multiple turnstiles	1.00	0.31	0.62	1.17	1.52	2.59
No turnstile	1.00	0.45	0.86	1.92	3.63	7.00

Table 2: Speedups of program kernel with varying degrees of serializations.

We measure the basic effectiveness of the dynamic memory system with a program kennel. The main body of the kernel is the following loop:

```
for (i=0; i<SIZE; i++) {
  A[i] = A[i] + B[XB[i]] + C[XC[i]] + D[XD[i]]
              + E[XE[i]] + F[XF[i]];
}
```

All the arrays are laid out in a low-order interleaved manner. A and the X arrays can be statically promoted through modulo unrolling. The arrays B through F, however, are dynamic because they can refer to any part of the interleaved arrays. Table 2 measures the performance benefit of reducing serialization through compiler optimizations. For each configuration, we parallelize the kernel for varying number of tiles, denoted by P. Each speedup figure is computed by comparing the run-time against the uniprocessor case, where all accesses are local. From Table 2, when accesses are sequentialized and issued on a single turnstile, performance is poor, with negligible speedup on 32 tiles. When dynamic accesses are divided into five turnstiles, one for each dynamic array, we attain a modest speedup of 2.59. Finally, when the dynamic accesses are recognized to be loads and thus require no sequentialization through any turnstile, we attain a good speedup of 7.00. These results show that it is indeed possible to attain overall speedup even when a program has a substantial number of dynamic accesses.

Preliminary application performance The RAWCC compiler has been evaluated on a set of regular benchmarks shown in Table 3. We compare the results of the Raw compiler with the results of a MIPS compiler provided by Machsuif [11] targeted for an R2000. Table 4 shows the speedups attained by the benchmarks for Raw machines of various sizes. For all these benchmarks, Maps is able to statically promote 100% of the accesses. This result enables the Raw compiler to profitably exploit the significant amount of parallelism available in the benchmarks. Consequently, the average speedup attained on 32 tiles is a promising 17.2.

Note that the speedups are attained from sequential code using automatic parallelization, and not for code tailored to any high-performance architecture.

Benchmark	Source	Lang.	Lines of code	Primary Array size	Seq. RT (cycles)	Description
btrix	Nasa7:Spec92	Fortran	236	15×15×15×5	287M	Vectorized Block Tri-Diagonal Solver
cholesky	Nasa7:Spec92	Fortran	126	3×32×32	34.3M	Cholesky Decomposition/Substitution
vpenta	Nasa7:Spec92	Fortran	157	32×32	21.0M	Inverts 3 Pentadiagonals Simultaneously
tomcatv	Spec92	Fortran	254	32×32	78.4M	Mesh Generation with Thompson's Solver
mxm	Nasa7:Spec92	Fortran	64	32×64, 64×8	2.01M	Matrix Multiplication

Table 3: 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
btrix	0.83	1.48	2.61	4.40	8.58	9.64
cholsky	0.88	1.68	3.38	5.48	10.30	14.81
vpenta	0.70	1.76	3.31	6.38	10.59	19.20
tomcatv	0.92	1.64	2.76	5.52	9.91	19.31
mxm	0.94	1.97	3.60	6.64	12.20	23.19

Table 4: Benchmark Speedup. Speedup compares the run-time of the RAWCC-compiled code versus the run-time of the code generated by the Machsuif MIPS compiler.

8 Related work

Both the Raw architecture and Maps are influenced by research in several areas. Due to space limitations, we do not present related work on the architectural aspects of Raw. For a detailed comparison to other architectures, see [12].

Software distributed shared memory schemes on multiprocessors (DSMs) [10] [4] 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 [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 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 [7]. Relative memory disambiguation [7] 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 [5]. He observes that unrolling can sometimes help disambiguate accesses, but he does not attempt to formalize the observation into a theory or algorithm. Instead, his technique 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 point-to-point interconnects. Thus, each memory unit is small and close to its processing element, thereby allowing low latency access. The distributed memory system has no central bottlenecks and can sustain a high aggregate memory system bandwidth. 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 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 minimize the synchronization and sequentialization necessary for correct program behavior, and it strives to allow as many parallel accesses as possible. Through static promotion of memory references, Maps attempts to maximize its utilization of the fast static network. It seamlessly integrates support for dynamic accesses – memory references that it fails to promote – over the dynamic network using software serial ordering, incurring minimal synchronization.

We show that Maps is able to statically promote a large number of memory accesses, which can then executed in an efficient manner. The overhead and register-like latencies of the statically promoted memory operations are lower than any hardware-based techniques such as coherent caches. We also show that software serial ordering based on turnstiles and the dynamic network provides an efficient fallback mechanism for memory references that the compiler cannot statically promote. Software serial ordering tries to minimize the sequentialization of dependent dynamic memory references. For example, it is able to overlap three-fourths of the individual latency when multiple dependent dynamic memory operations are issued from separate tiles. Furthermore, Maps extracts parallelism by using a distinct turnstile for each equivalence class of dynamic references.

We are encouraged by the results of our compiler-Maps approach to memory orchestration for the small set of benchmarks we have executed on the system. We are currently pushing larger and more general programs through the system. We demonstrate a high degree of speedup for several regular programs, and we show that software serial ordering is able to achieve moderate speedup in cases where dynamic accesses are necessary. If the results for more general 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.

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

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

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

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

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

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

[10] 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.

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

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

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