Research Workshop

1. MLP - Memory Level Parallelism
2. What to do with billion fraudsters
MLP yes! ILP no!

*Memory Level Parallelism, or why I no longer care about Instruction Level Parallelism*

Andrew Glew  
Intel Microcomputer Research Labs and University of Wisconsin, Madison

**Problem Description:** It should be well known that processors are outstripping memory performance: specifically that memory latencies are not improving as fast as processor cycle time or IPC or memory bandwidth.

Thought experiment: imagine that a cache miss takes 10000 cycles to execute. For such a processor instruction level parallelism is useless, because most of the time is spent waiting for memory. Branch prediction is also less effective, since most branches can be determined with data already in registers or in the cache; branch prediction only helps for branches which depend on outstanding cache misses.

At the same time, pressures for reduced power consumption mount.

Given such trends, some computer architects in industry (although not Intel EPIC) are talking seriously about retreating from out-of-order superscalar processor architecture, and instead building simpler, faster, dumber, 1-wide in-order processors with high degrees of speculation. Sometimes this is proposed in combination with multiprocessing and multithreading: tolerate long memory latencies by switching to other processes or threads.

I propose something different: build narrow fast machines but use intelligent logic inside the CPU to increase the number of outstanding cache misses that can be generated from a single program.

**Solution:** First, change the mindset: MLP, Memory Level Parallelism, is what matters, not ILP, Instruction Level Parallelism.

By MLP I mean simply the number of outstanding cache misses that can be generated (by a single thread, task, or program) and executed in an overlapped manner. It does not matter what sort of execution engine generates the multiple outstanding cache misses. An out-of-order superscalar ILP CPU may generate multiple outstanding cache misses, but 1-wide processors can be just as effective.

Change the metrics: total execution time remains the overall goal, but instead of reporting IPC as an approximation to this, we must report MLP. Limit studies should be in terms of total number of non-overlapped cache misses on critical path.

Now do the research: Many present-day hot topics in computer architecture help ILP, but do not help MLP. As mentioned above, predicting branch directions for branches that can be determined from data already in the cache or in registers does not help MLP for extremely long latencies. Similarly, prefetching of data cache misses for array processing codes does not help MLP – it just moves it around.

Instead, investigate microarchitectures that help MLP:

1. **Trivial case – explicit multithreading, like SMT.**
2. Slightly less trivial case – implicitly multithread single programs, either by compiler software on an MT machine, or by a hybrid, such as Wisconsin Multiscalar, or entirely in hardware, as in Intel’s Dynamic Multi-Threading.
3. Build 1-wide processors that are as fast as possible: use circuit tricks, as well as logic tricks such as redundant encoding for numeric computation and memory addressing.
4. Allow the hardware dynamic scheduling mechanisms to use sequential algorithms implemented by this narrow, fast, processor, rather than limiting it to parallel algorithms implementable in associative logic.
5. Build very large instruction windows allowing speculation tens of thousands of instructions ahead. Avoid circuit speed issues by caching the instruction window. Remove small arbitrary limits on the number of cache misses outstanding allowed.
6. Further reduce the cost of very large instruction windows by throwing away anything that can be recomputed based on data in registers or cache.
7. Don’t stall speculation because the oldest instruction in the machine is a cache miss. Let the front of the machine continue executing branches, forgetting data dependent on cache misses.
8. Parallelize linked data structure traversals by building skip lists in hardware – converting sequential data structures into parallel ones. Store these extra skip pointers in main memory.

Call such a processor microarchitecture a “super-non-blocking” microarchitecture.

**Justification:** The processor/memory trend is well known. Theoretically optimal cache studies show only limited headroom.

Barring a revolution in memory technology, the Memory Wall is real, and getting closer. Multithreading and multiprocessing have some hope of tolerating memory latency, but only if there are parallel workloads. If single thread performance is still an issue, the only potentially MLP enhancing technologies are what I describe here, or data value prediction – and data value prediction seems to only do well for stuff that fits in the cache.

“Super-non-blocking” processors extends dynamic, out-of-order, execution to maximize MLP, but simplifies it by discarding superscalar ILP as unnecessary.
Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine

Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna,
Jonathan Babb, Vivek Sarkar, Saman Amarasinghe M.I.T. Laboratory for Computer Science
Cambridge, MA 02139, U.S.A.
{walt,barua,mfrank,chinnama,jbabb,vivek,saman}@lcs.mit.edu
http://www.cog.lcs.mit.edu/raw

Abstract

Increasing demand for both greater parallelism and faster clocks dictate that future generation architectures will need to decentralize their resources and eliminate primitives that require single-cycle global communication. A Raw microprocessor distributes all of its resources, including instruction streams, register files, memory ports, and ALUs, over a pipelined two-dimensional mesh interconnect, and exposes them fully to the compiler. Because communication in Raw machines is distributed, compiling for instruction-level parallelism (ILP) requires both spatial instruction partitioning as well as traditional temporal instruction scheduling. In addition, the compiler must explicitly manage all communication through the interconnect, including the global synchronization required at branch points. This paper describes RAWCC, the compiler we have developed for compiling general-purpose sequential programs to the distributed Raw architecture. We present performance results that demonstrate that although Raw machines provide no mechanisms for global communication the Raw compiler can schedule to achieve speedups that scale with the number of available functional units.

1 Introduction

Modern microprocessors have evolved while maintaining the faithful representation of a monolithic uniprocessor. While innovations in the ability to exploit instruction level parallelism have placed greater demands on processor resources, these resources have remained centralized, creating scalability problem at every design point in a machine. As processor designers continue in their pursuit of architectures that can exploit more parallelism and thus require even more resources, the cracks in the view of a monolithic underlying processor can no longer be concealed. An early visible effect of the scalability problem in commercial architectures is apparent in the clustered organization of the Multiflow computer [19]. More recently, the Alpha 21264 [14] duplicates its register file to provide the requisite number of ports at a reasonable clock speed.

As the amount of on-chip processor resources continues to increase, the pressure toward this type of non-uniform spatial structure will continue to mount. Inevitably, from such hierarchy, resource accesses will have non-uniform latencies. In particular, register or memory access by a functional unit will have a gradation of access time. This fundamental change in processor model will necessitate a corresponding change in compiler technology. Thus, instruction scheduling becomes a spatial problem as well as a temporal problem.

The Raw machine [23] is a scalable microprocessor architecture with non-uniform register access latencies (NURA). As such, its compilation problem is similar to that which will be encountered by extrapolations of existing architectures. In this paper, we describe the compilation techniques used to exploit ILP on the Raw machine, a NURA machine composed of fully replicated processing units connected via a mostly static programmable network. The fully exposed hardware allows the Raw compiler to precisely orchestrate computation and communication in order to exploit ILP within basic blocks. The compiler handles the orchestration by performing spatial and temporal instruction scheduling, as well as data partitioning using a distributed on-chip memory model.

This paper makes three contributions. First, it describes the space-time scheduling of ILP on a Raw machine, borrowing some techniques from the partitioning and scheduling of tasks on MIMD machines. Second, it introduces a new control flow model based on asynchronous local branches inside a machine with multiple independent instruction streams. Finally, it shows that independent instruction streams give the Raw machine the ability to tolerate timing variations due to dynamic events.

The rest of the paper is organized as follows. Section 2 motivates the need for NURA machines, and it introduces the Raw machine as one such machine. Section 3 overviews the space-time scheduling of ILP on a Raw machine. Section 4 describes RAWCC, the Raw compiler, and it explains the memory and data access model RAWCC implements. Section 5 describes the basic block orchestration of ILP. Section 6 describes the orchestration of control flow. Section 7 shows the performance of RAWCC. Section 8 presents related work, and Section 9 concludes.

2 Architectural motivation and background

Raw machines are made up of a set of simple tiles, each with a portion of the register set, a portion of the on-chip memory, and one of the functional units. These tiles communicate via a scalable point-to-point interconnect. This section motivates the Raw architecture. We examine the scalability problem of modern processors, trace an architectural evolution that overcomes such problems, and show that the Raw architecture is at an advanced stage of such an evolution. We highlight non-uniform register access as an important feature in scalable machines. We then describe the Raw machine, with emphasis on features which make it an attractive scalable machine. Finally, we describe the relationship between a Raw machine and a VLIW machine.

The Scalability Problem Modern processors are not designed to scale. Because superscalars require significant hardware resources

to support parallel instruction execution, architects for these machines face an uncomfortable dilemma. On the one hand, faster machines require additional hardware resources for both computation and discovery of ILP. On the other hand, these resources often have quadratic area complexity, quadratic connectivity, and global wiring requirements which can be satisfied only at the cost of cycle time degradation. VLIW machines address some of these problems by moving the cycle-time elongating task of discovering ILP from hardware to software, but they still suffer scalability problems due to issue bandwidth, multi-ported register files, caches, and wire delays.

Up to now, commercial microprocessors have faithfully preserved their monolithic images. As pressure from all sources demands computers to be bigger and more powerful, this image will be difficult to maintain. A crack is already visible in the Alpha 21264. In order to satisfy timing specification while providing the register bandwidth needed by its dual-ported cache and four functional units, the Alpha duplicates its register file. Each physical register file provides half the required ports. A cluster is formed by organizing two functional units and a cache port around each register file. Communication within a cluster occurs at normal speed, while communication across clusters takes an additional cycle.

This example suggests an evolutionary path that resolves the scalability problem: impose a hierarchy on the organization of hardware resources [22]. A processor can be composed from replicated processing units whose pipelines are coupled together at the register level so that they can exploit ILP cooperatively. The VLIW Multiflow TRACE machine is a machine which adopts such a solution [19]. On the other hand, its main motivation for this organization is to provide enough register ports. Communication between clusters is performed via global busses, which in modern and future-generation technology would severely degrade the clock speed of the machine. This problem points to the next step in the scalability evolution — providing a scalable interconnect. For machines of modest sizes, a bus or a full crossbar may suffice. But as the number of components increases, a point to point network will be necessary to provide the required latency and bandwidth at the fastest possible clock speed — a progression reminiscent of multiprocessor evolution.

The result of the evolution toward scalability is a machine with a distributed register file interconnected via a scalable network. In the spirit of NUMA machines (Non-Uniform Memory Access), we call such machines NURA machines (Non-Uniform Register Access). Like a NUMA machine, a NURA machine connects its distributed storage via a scalable interconnect. Unlike NUMA, NURA pools the shared storage resources at the register level. Because a NURA machine exploits ILP of a single instruction stream, its interconnect must provide latencies that are much lower than that on a multiprocessor.

As the base element of the storage hierarchy, any change in the register model has profound implications. The distributed nature of the computational and storage elements on a NURA machine means that locality should be considered when assigning instructions to functional units. Instruction scheduling becomes a spatial problem as well as a temporal problem.

Raw architecture The Raw machine [23] is a NURA architecture motivated by the need to design simple and highly scalable processors. As depicted in Figure 1, a Raw machine comprises a simple, replicated tile, each with its own instruction stream, and a programmable, tightly integrated interconnect between tiles. A Raw machine also supports multi-granular (bit and byte level) operations as well as customizable configurable logic, although this paper does not address these features.

Each Raw tile contains a simple five-stage pipeline, interconnected with other tiles over a pipelined, point-to-point network.
pended instruction streams which can handle multiple flows of control, is useful for four reasons. First, it significantly enhances the potential amount of parallelism a machine can exploit [18]. Second, it enables asynchronous global branching described in Section 6, a means of implementing global branching on Raw’s distributed interconnect. Third it enables control localization, a technique we introduce in Section 6 to allow ILP to be scheduled across branches. Finally, it gives a Raw machine better tolerance of dynamic events compared to a VLIW machine, as shown in Section 7.

- Simple, scalable means of expanding the register space: Each Raw tile contains a portion of the register space. Because the register set is distributed along with the functional units and memory ports, the number of registers and register ports scales linearly with total machine size. Each tile’s individual register set, however, has only a relatively small number of registers and register ports, so the complexity of the register file will not become an impediment to increasing the clock rate. Additionally, because all physical registers are architecturally visible, the compiler can use all of them to minimize the number of register spills.

- A compiler interface for locality management: The Raw machine fully exposes its hardware to the compiler by exporting a simple cost model for communication and computation. The compiler, in turn, is responsible for the assignment of instructions to Raw tiles. Instruction partitioning and placement is best performed at compile time because the algorithms require a very large window of instructions and the computational complexity is greater than can be afforded at run-time.

- Mechanism for precise orchestration: Raw’s programmable static switch is an essential feature for exploiting ILP on the Raw machine. First, it allows single-word register-level transfer without the overhead of composing and routing a message header. Second, the Raw compiler can use its full knowledge of the network status to minimize congestion and route data around hot spots. More importantly, the compile-time knowledge about the order in which messages will be received on each tile obviates the run-time overhead of determining the contents of incoming messages.

3 Overview of space-time scheduling

The space-time scheduling of ILP on a Raw machine consists of orchestrating the parallelism within a basic block across the Raw tiles and handling of the control flow across basic blocks. Basic block orchestration, in turn, consists of several tasks: the assignment of instructions to processing units (spatial scheduling), the scheduling of those instructions on the tiles they are assigned (temporal scheduling), the assignment of data to tiles, and the explicit orchestration of communication across a mesh interconnect, both within and across basic blocks. Control flow between basic blocks is explicitly orchestrated by the compiler through asynchronous global branching, an asynchronous mechanism for implementing branching across all the tiles using the static network and individual branches on each tile. In addition, an optimization called control localization allows some branches in the program to affect execution on only one tile.

The two central tasks of the basic-block orchestrator are the assignment and scheduling of instructions. The Raw compiler performs assignment in three steps: clustering, merging, and placement. Clustering groups together instructions, such that instructions within a cluster have no parallelism that can profitably be exploited given the cost of communication. Merging reduces the number of clusters down to the number of processing units by merging the clusters. Placement performs a bijective mapping from the merged clusters to the processing units, taking into account the topology of the interconnect. Scheduling of instructions is performed with a traditional list scheduler.

Other functionalities of the basic-block orchestrator are integrated into this framework as seamlessly as possible. Data assignment and instruction assignment are implemented to allow flow of information in both directions, thus reflecting the inter-dependent nature of the two assignment problems. Inter-block and intra-block communication are both identified and handled in a single, unified manner. The list scheduler is extended to schedule not only computation instructions but communication instructions as well, in a manner which guarantees the resultant schedule is deadlock-free.

MIMD task scheduling There are two ways to view Raw’s problem of assigning and scheduling instructions. From one perspective, the Raw compiler statically schedules ILP just like a VLIW compiler. Therefore, a clustered VLIW with distributed registers and functional units faces a similar problem as the Raw machine [6][10][12]. From another perspective, the Raw compiler schedules tasks on a MIMD machine, where tasks are at the granularity of instructions. A MIMD machine faces a similar assignment/scheduling problem, but at a coarser granularity [1][20][26].

The Raw compiler leverages research in the rich field of MIMD task scheduling. MIMD scheduling research is applicable to clustered VLIWs as well. To our knowledge, this is the first paper which attempts to leverage MIMD task scheduling technology for the scheduling of fine-grained ILP. We show that such technology produces good results despite having fine-grained tasks (i.e., single instructions).

4 RAWCC

RAWCC, the Raw compiler, is implemented using the SUIF compiler infrastructure [24]. It compiles both C and FORTRAN programs. The Raw compiler consists of three phases. The first phase performs high level program analysis and transformations. It contains Maps [7], Raw’s compiler managed memory system. The memory provided by Maps and the data access model is briefly described below. The initial phase also includes traditional techniques such as memory disambiguation, loop unrolling, and array reshape, plus a new control optimization technique to be discussed.
in Section 6. In the future, it will be extended with the advanced ILP-enhancing techniques discussed in Section 8.

The second phase, the space-time scheduler, performs the scheduling of ILP. Its two functions, basic block orchestration and control orchestration, are described in Section 5 and Section 6, respectively.

The final phase in RAWCC generates code for the processors and the switches. It uses the MIPS back-end developed in Machine SUIF [21], with a few modifications to handle the communication instructions and communication registers.

Memory and data access model Memory on a Raw machine is distributed across the tiles. The Raw memory model provides two ways of accessing this memory system, one for static reference and one for dynamic reference. A reference is static if every invocation of it can be determined at compile-time to refer to memory on one specific tile. We call this property the static resident property. Such a reference is handled by placing it on the corresponding tile at compile time. A non-static or dynamic reference is handled by disambiguating the address at run-time in software, using the dynamic network to handle any necessary communication.

The Raw compiler attempts to generate as many static references as possible. Static references are attractive for two reasons. First, they can proceed without any of the overhead due to dynamic disambiguation and synchronization. Second, they can potentially take advantage of the full memory bandwidth. This paper focuses on results which can be attained when the Raw compiler succeeds in identifying static references. A full discussion of the compiler managed memory system, including issues pertaining to dynamic references, can be found in [7]. In Section 7, we do make one observation relevant to dynamic references: decoupled instruction streams allow the Raw machine to tolerate timing variations due to events such as dynamic memory accesses.

Static references can be created through intelligent data mapping and code transformation. For arrays, the Raw compiler distributes them through low order interleaving, which interleaves the arrays element-wise across the memory system. For array references which are affine functions of loop indices, we have developed a technique which uses loop unrolling to satisfy the static resident property. Our techniques are a generalization of an observation made by Ellis [12]. Details are presented in [8].

Scalar values communicated within basic blocks follow a data-flow model, so that the tile consuming a value receives it directly from the producer tile. To communicate values across basic block boundaries, each program variable is assigned a home tile. At the beginning of a basic block, the value of a variable is transferred from its home tile to the tiles which use the variable. At the end of a basic block, the value of a modified variable is transferred from the computing tile to its home tile.

5 Basic block orchestrator

The basic block orchestrator exploits the ILP within a basic block by distributing the parallelism within the basic block across the tiles. It transforms a single basic block into an equivalent set of intercommunicating basic blocks that can be run in parallel on Raw. Orchestration consists of assignment and scheduling of instructions, assignment of data, and the orchestration of communication. This section first gives the implementation details of how the orchestrator performs these functions, followed by a general discussion of its design.

Figure 2 shows the phase ordering of the basic block orchestrator. Each phase is described in turn below. To facilitate the explanation, Figure 3 shows the transformations performed by RAWCC on a sample program.

![Figure 2: Phase ordering of the basic block orchestrator.](image)

**Initial code transformation** Initial code transformation massages a basic block into a form suitable for subsequent analysis phases. Figure 3a shows the transformations performed by this phase. First, renaming converts statements of the basic block to static single assignment form. Such conversion removes anti-dependencies and output-dependencies from the basic block, which in turn exposes available parallelism. It is analogous to hardware register renaming performed by superscalars.

Second, two types of dummy instructions are inserted. Read instructions are inserted for variables which are live-on-entry and read in the basic block. Write instructions are inserted for variables which are live-on-exit and written within the basic block. These instructions simplify the eventual representation of stitch code, the communication needed to transfer values between the basic blocks. This representation in turn allows the event scheduler to overlap the stitch code with other work in the basic block.

Third, expressions in the source program are decomposed into instructions in three-operand form. Three-operand instructions are convenient because they correspond closely to the final machine instructions and because their cost attributes can easily be estimated. Therefore, they are logical candidates to be used as atomic partitioning and scheduling units.

Finally, the dependence graph for the basic block is constructed. A node represents an instruction, and an edge represents a true flow dependence between two instructions. Each node is labeled with the estimated cost of running the instruction. For example, the node for a floating point add in the example is labeled with two cycles. Each edge represents a word of data transfer.

**Instruction partitioner** The instruction partitioner partitions the original instruction stream into multiple instruction streams, one for each tile. It does not bind the resultant instruction streams to specific tiles — that function is performed by the instruction placer. When generating the instruction streams, the partitioner attempts to balance the benefits of parallelism against the overheads of communication. Figure 3b shows a sample output of this phase.

Certain instructions have constraints on where they can be partitioned and placed. Read and write instructions to the same variable have to be mapped to the processor on which the data resides (see global data partitioner and data and instruction placer below). Similarly, loads and stores satisfying the static resident property must be mapped to a specific tile. The instruction partitioner performs its duty without considering these constraints. They are taken into account in the global data partitioner and in the data and instruction placer.
Figure 3: An example of the program transformations performed by RAWCC. (a) shows the initial program undergoing transformations made by initial code transformation; (b) shows result of instruction partitioner; (c) shows result of global data partitioner; (d) shows result of data and instruction placer; (e) shows result of communication code generator; (f) shows final result after event scheduler.
Partitioning is performed through clustering and merging phases introduced in Section 3. We describe each in turn:

**Clustering:** Clustering attempts to partition instructions to minimize run-time, assuming non-zero communication cost but infinite processing resources. The cost of communication is modeled assuming an idealized uniform network whose latency is the average latency of the actual network. The phase groups together instructions that either have no parallelism, or whose parallelism is too fine-grained to be exploited given the communication cost. Subsequent phases guarantee that instructions with no mapping constraints in the same cluster will be mapped to the same tile.

RAWCC employs a greedy technique based on the estimation of completion time called Dominant Sequent Clustering [26]. Initially, each instruction node belongs to a unit cluster. Communication between clusters is assigned a uniform cost. The algorithm visits instruction nodes in topological order. At each step, it selects from the list of candidates the instruction on the longest execution path. If then checks whether the selected instruction can merge into the cluster of any of its parent instructions to reduce the estimated completion time of the program. Estimation of the completion time is dynamically updated to take into account the clustering decisions already made, and it reflects the cost of both computation and communication. The algorithm completes when all nodes have been visited exactly once.

**Merging:** Merging combines clusters to reduce the number of clusters down to the number of tiles, again assuming an idealized switch interconnect. Two useful heuristics in merging are to maintain load balance and to minimize communication events. The Raw compiler currently uses a locality-sensitive load balancing technique which tries to minimize communication events unless the load imbalance exceeds a certain threshold. We plan to consider other strategies, including an algorithm based on estimating completion time, in the future.

The current algorithm is as follows. The Raw compiler initializes $N$ empty partitions (where $N$ is the number of tiles), and it visits clusters in decreasing order of size. When it visits a cluster, it merges the cluster into the partition with which it communicates the most, unless such merging results in a partition which is 20% larger than the size of an average partition. If the latter condition occurs, the cluster is placed into the smallest partition instead.

**Global data partitioner** To communicate values of data elements between basic blocks, a scalar data element is assigned a "home" tile location. Within basic blocks, renaming localizes most value references, so that only the initial reads and the final write of a variable need to communicate with its home location. Like instruction mapping, the Raw compiler divides the task of data home assignment into data partitioning and data placement.

The job of the data partitioner is to group data elements into sets, each of which is to be mapped to the same processor. To preserve locality as much as possible, data elements which tend to be accessed by the same thread should be grouped together. To partition data elements into sets which are frequently accessed together, RAWCC performs global analysis. The algorithm is as follows. For initialization, a virtual processor number is arbitrarily assigned to each instruction stream on each basic block, as well as to each scalar data element. In addition, statically analyzable memory references are first assigned dummy data elements, and then those elements are assigned virtual processor numbers corresponding to the physical location of the references. Furthermore, the access pattern of each instruction stream is summarized with its affinity to each data element. An instruction stream is said to have affinity for a data element if it either accesses the element, or it produces the final value for the element in that basic block. After initialization, the algorithm attempts to localize as many references as possible by remapping the instruction streams and data elements. First, it remaps instruction streams to virtualized processors given fixed mapping of data elements. Then, it remaps data elements to virtualized processors given fixed mappings of instruction streams. Only the true data elements, not the dummy data elements corresponding to fixed memory references, are remapped in this phase. This process repeats until no incremental improvement of locality can be found. In the resulting partition, data elements mapped to the same virtual processor are likely related based on the access patterns of the instruction streams.

Figure 3c shows the partitioning of data values into such affinity sets. Note that variables introduced by initial code transformation (e.g., $y_1$ and $tmp_{-1}$) do not need to be partitioned because their scopes are limited to the basic block.

**Data and instruction placer** The data and instruction placer maps virtualized data sets and instruction streams to physical processors. Figure 3d shows a sample output of this phase. The placement phase removes the assumption of the idealized interconnect and takes into account the non-uniform network latency. Placement of each data partition is currently driven by those data elements with processor preferences, i.e., those corresponding to fixed memory references. It is performed before instruction placement to allow cost estimation during instruction placement to account for the location of data. In additional to mapping data sets to processors, the data placement phase also locks the dummy read and write instructions to the home locations of the corresponding data elements.

For instruction placement, RAWCC uses a swap-based greedy algorithm to minimize the communication bandwidth. It initially assigns clusters to arbitrary tiles, and it looks for pairs of mappings that can be swapped to reduce the total number of communication hops.

**Communication code generator** The communication code generator translates each non-local edge (an edge whose source and destination nodes are mapped to different tiles) in the dependence graph into communication instructions which route the necessary data value from the source tile to the destination tile. Figure 3e shows an example of such transformation. Communication instructions include send and receive instructions on the processors as well as route instructions on the switches. New nodes are inserted into the graph to represent the communication instructions, and the edges of the source and destination nodes are updated to reflect the new dependence relations arising from insertion of the communication nodes. To minimize the volume of communication, edges with the same source are serviced jointly by a single multicast operation, though this optimization is not illustrated in the example.

The current compilation strategy assumes that network contention is low, so that the choice of message routes has less impact on the code quality compared to the choice of instruction partitions or event schedules. Therefore, communication code generation in RAWCC uses dimension-ordered routing: this spatial aspect of communication scheduling is completely mechanical. If contention is determined to be a performance bottleneck, a more flexible technique can be employed.

**Event scheduler** The event scheduler schedules the computation and communication events within a basic block with the goal of producing the minimal estimated run-time. Because routing in Raw is itself specified with explicit switch instructions, all events to be scheduled are instructions. Therefore, the scheduling problem is a generalization of the traditional instruction scheduling problem.

The job of scheduling communication instructions carries with it the responsibility of ensuring the absence of deadlocks in the network. If individual communication instructions are scheduled separately, the Raw compiler would need to explicitly manage the buffering resources on each communication port to ensure the ab-
sence of deadlock. Instead, RAWCC avoids the need for such management by treating a single-source, multiple-destination communication path as a single scheduling unit. When a communication path is scheduled, contiguous time slots are reserved for instructions in the path so that the path incurs no delay in the static schedule. By reserving the appropriate time slot at the node of each communication instruction, the compiler automatically reserves the corresponding channel resources needed to ensure that the instruction can eventually make progress.

Though event scheduling is a static problem, the schedule generated must remain deadlock-free and correct even in the presence of dynamic events such as cache misses. The Raw system uses the static ordering property, implemented through near-neighbor flow control, to ensure this behavior. The static ordering property states that if a schedule does not deadlock, then any schedule with the same order of communication events will not deadlock. Because dynamic events like cache misses only add extra latency but do not change the order of communication events, they do not affect the correctness of the schedule.

The static ordering property also allows the scheduler to be stored as compact instruction streams. Timing information needs not be preserved in the instruction stream to ensure correctness, thus obviating the need to insert no-op instructions. Figure 3f shows a sample output of the event scheduler. Note, first, the proper ordering of the route instructions on the switches, and, second, the successful overlap of computation with communication on P0, where the processor computes and writes z while waiting on the value of y.1.

RAWCC uses a single greedy list scheduler to schedule both computation and communication. The algorithm keeps track of a ready list of tasks. A task is either a computation or a communication path. As long as the list is not empty, it selects and schedules the task on the ready list with the highest priority. The priority scheme is based on the following observation. The priority of a task should be directly proportional to the impact it has on the completion time of the program. This impact, in turn, is lower-bounded by two properties of the task: its level, defined to be its critical path length to an exit node; and its average fertility, defined to be the number of dependent nodes divided by the number of processors. Therefore, we define the priority of a task to be a weighted sum of these two properties.

Discussion There are two reasons for decomposing the space-time instruction scheduling problem into multiple phases. First, given a machine with a non-uniform network, empirical results have shown that separating assignment from scheduling yields superior performance [25]. Furthermore, given a graph with fine-grained parallelism, having a clustering phase has been shown to improve performance [11].

In addition, the space-time scheduling problem, as well as each of its subproblems, is NP complete [20]. Decomposing the problem into a set of greedy heuristics enables us to develop an algorithm which is computationally tractable. The success of this approach, of course, depends heavily on carefully choosing the problem decomposition. The decomposition should be such that decisions made be an earlier phase should not inhibit subsequent phases from making good decisions.

We believe that separating the clustering and scheduling phases was a good decomposition decision. The benefits of dividing merging and placement have been less clear. Combining them so that merging is sensitive to the processor topology may be preferable, especially because on a Raw machine some memory instructions have predetermined processor mappings. We intend to explore this issue in the future.

The basic block orchestrator integrates its additional responsibilities relatively seamlessly into the basic space-time scheduling framework. By inserting dummy instructions to represent home tiles, inter-basic-block communication can be represented the same way as intra-basic-block communication. The need for explicit communication is identified through edges between instructions mapped to different tiles, and communication code generation is performed by replacing these edges with a chain of communication instructions. The resultant graph is then presented to a vanilla greedy list scheduler, modified to treat each communication path as a single scheduling unit. This list scheduler is then able to generate a correct and greedily optimized schedule for both computation and communication.

Like a traditional uniprocessor compiler, RAWCC faces a phase ordering problem with event scheduling and register allocation. Currently, the event scheduler runs before register allocation; it has no register consumption information and does not consider register pressure when performing the scheduling. The consequence is two-fold. First, instruction costs may be underestimated because they do not include spill costs. Second, the event scheduler may expose too much parallelism, which cannot be efficiently utilized but which comes at a cost of increased register pressure. The experimental results for fppp-kernel in Section 7 illustrate this problem. We are exploring this issue and have examined a promising approach which adjusts the priorities of instructions based on how the instructions affect the register pressure. In addition, we intend to explore the possibility of cooperative inter-tile register allocation.

6 Control orchestration

Raw tiles cooperate to exploit ILP within a basic block. Between basic blocks, the Raw compiler has to orchestrate the control flow on all the tiles. This orchestration is performed through asynchronous global branching. To reduce the need to incur the cost of this global orchestration and expand the scope of the basic block orchestrator, the Raw compiler performs control localization, a control optimization which localizes the effects of a branch in a program to a single tile.

Asynchronous global branching The Raw machine implements global branching asynchronously in software by using the static network and local branches. First, the branch value is broadcasted to all the tiles through the static network. This communication is exported and scheduled explicitly by the compiler just like any other communication, so that it can overlap with other computation in the basic block. Then, each tile and switch individually performs a branch without synchronization at the end of its basic block execution. Correct execution is ensured despite introducing this asynchrony because of the static ordering property.

The overhead of global branching on a Raw machine is explicit in the broadcast of the branch condition. This contrasts with the implicit overhead of global wiring incurred by global branching in VLIWs and superscalars. Raw’s explicit overhead is desirable for three reasons. First, the compiler can hide the overhead by overlapping it with useful work. Second, this branching model does not require dedicated wires used only for branching. Third, the approach is consistent with the Raw philosophy of eliminating all global wires, which taken as a whole enables a much faster clock speed.

Control localization Control localization is the technique of treating a branch-containing code sequence as a single unit during assignment and scheduling. This assignment/scheduling unit is called a macro-instruction. The technique is a control optimization made possible though Raw’s independent flows of control, which allows a Raw machine to execute different macro-instructions concurrently on different tiles. By localizing the effects of branches to individual tiles, control localization avoids the broadcast cost of asynchronous
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Source</th>
<th>Lang.</th>
<th>Lines of code</th>
<th>Primary Array size</th>
<th>Seq. RT (cycles)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fppp-kernel</td>
<td>Spec92</td>
<td>FORTRAN</td>
<td>735</td>
<td>15x15x15x15x15</td>
<td>8.96K</td>
<td>Electron Interval Derivatives</td>
</tr>
<tr>
<td>brix</td>
<td>Nasa7:Spec92</td>
<td>FORTRAN</td>
<td>236</td>
<td>3x3x32</td>
<td>28.7M</td>
<td>Vectorized Block Tri-Diagonal Solver</td>
</tr>
<tr>
<td>cholesky</td>
<td>Nasa7:Spec92</td>
<td>FORTRAN</td>
<td>126</td>
<td>3x3x32</td>
<td>34.3M</td>
<td>Cholesky Decomposition/Substitution</td>
</tr>
<tr>
<td>tomcatv</td>
<td>Spec92</td>
<td>FORTRAN</td>
<td>157</td>
<td>32x32</td>
<td>21.0M</td>
<td>Inverts 3 Pentadiagonals Simultaneously</td>
</tr>
<tr>
<td>mac</td>
<td>Nasa7:Spec92</td>
<td>FORTRAN</td>
<td>254</td>
<td>32x32</td>
<td>78.4M</td>
<td>Mesh Generation with Thompson's Solver</td>
</tr>
<tr>
<td>life</td>
<td>Rawbench</td>
<td>C</td>
<td>118</td>
<td>32x32</td>
<td>2.44M</td>
<td>Conway's Game of Life</td>
</tr>
<tr>
<td>jacobit</td>
<td>Rawbench</td>
<td>C</td>
<td>59</td>
<td>32x32</td>
<td>2.38M</td>
<td>Jacoby Relaxation</td>
</tr>
</tbody>
</table>

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

![Diagram](image)

Figure 4: An illustration of control localization. (a) shows a control flow graph before control localization. Each oval is an instruction, and the dashed box marks the code sequence to be control localized. (b) shows the control flow graph after control localization.

global branching.

Figure 4 shows an example of control localization. In the figure, 4a shows a control flow graph before control localization, with the dashed box marking the sequence of code to be control localized. 4b shows the control flow graph after control localization, where the original branch has been hidden inside the macro-instruction. Note that control localization has merged the four original basic blocks into a single macro-extended basic block, within which ILP can be orchestrated by the basic block orchestrater.

To control localize a code sequence, the Raw compiler does the following. First, the Raw compiler verifies that the code sequence can in fact be placed on a single tile, which means that either (1) all its memory operations refer to a single tile, or (2) enough memory operations and all their preceding computations can safely be separated from the code sequence so that (1) is satisfied. Next, the compiler identifies the input variables the code sequence requires and the output variables the code sequence generates. These variables are computed by taking the union of their corresponding sets over all possible paths within the code sequence. In addition, given a variable for which a value is generated on one but not all paths of the program, the variable has to be considered as an input variable as well. This input is needed to allow the code sequence to produce a valid value of the variable independent of the path traversed inside it. The result of identifying these variables is that the code sequence can be assigned and scheduled like a regular instruction during basic block orchestration.

In practice, control localization has been invaluable in allowing RAWCC to use unrolling to expose parallelism in inner loops containing control flow. Currently, RAWCC adapts the simple policy of localizing into a single macro-instruction every localizable forward control flow structure, such as arbitrary nestings of IF-THEN-ELSE constructs and case statements. This simple policy has enabled us to achieve the performance reported in Section 7. A more flexible approach which varies the granularity of localization will be exploited in the future.

7 Results

This section presents some early performance results of the Raw compiler. We show the performance of the Raw compiler as a whole, and then we measure the portion of the performance due to high level transformations and advanced locality optimizations. In addition, we study how multisequentiality can reduce the sensitivity of performance to dynamic disturbances.

Experiments are performed on the Raw simulator, which simulates the Raw prototype described in Section 2. Latencies of the basic instructions are as follows: 2-cycle load, 1-cycle store, 1-cycle integer add or subtract; 12-cycle integer multiply; 35-cycle integer divide; 2-cycle floating add or subtract; 4-cycle floating multiply; and 12-cycle floating divide.

The benchmarks we select include programs from the Raw benchmark suite [4], program kernels from the nasa7 benchmark of Spec92, tomcatv of Spec92, and the kernel basic block which accounts for 50% of the run-time in fppp of Spec92. Since the Raw prototype currently does not support double-precision floating point, all floating point operations in the original benchmarks are converted to single precision. Table 1 gives some basic characteristics of the benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>N=1</th>
<th>N=2</th>
<th>N=4</th>
<th>N=8</th>
<th>N=16</th>
<th>N=32</th>
</tr>
</thead>
<tbody>
<tr>
<td>fppp-kernel</td>
<td>0.48</td>
<td>0.58</td>
<td>1.36</td>
<td>3.01</td>
<td>6.02</td>
<td>9.42</td>
</tr>
<tr>
<td>brix</td>
<td>0.83</td>
<td>1.48</td>
<td>2.61</td>
<td>4.40</td>
<td>8.58</td>
<td>9.64</td>
</tr>
<tr>
<td>cholesky</td>
<td>0.85</td>
<td>1.58</td>
<td>3.38</td>
<td>5.48</td>
<td>10.30</td>
<td>14.81</td>
</tr>
<tr>
<td>tomcatv</td>
<td>0.70</td>
<td>1.76</td>
<td>3.31</td>
<td>6.38</td>
<td>10.59</td>
<td>19.20</td>
</tr>
<tr>
<td>mac</td>
<td>0.92</td>
<td>1.64</td>
<td>2.76</td>
<td>5.52</td>
<td>9.91</td>
<td>19.31</td>
</tr>
<tr>
<td>life</td>
<td>0.94</td>
<td>1.97</td>
<td>3.60</td>
<td>6.64</td>
<td>12.20</td>
<td>23.19</td>
</tr>
<tr>
<td>jacobit</td>
<td>0.89</td>
<td>1.70</td>
<td>3.39</td>
<td>6.89</td>
<td>13.93</td>
<td>38.35</td>
</tr>
</tbody>
</table>

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

The Raw compiler is able to exploit ILP profitably across the Raw tiles for all the benchmarks. The average speedup on 32 tiles is 19.7.

All the benchmarks except fppp-kernel are dense matrix applications. These applications perform particularly well on a Raw machine because arbitrarily large amount of parallelism can be ex-
posed to the Raw compiler by unrolling the loop. Currently, the Raw compiler unrolls loops by the minimum amount required to guarantee the static residence property referred to in Section 4, which in most of these cases expose as many copies of the inner loop for scheduling of ILP as there are number of processors. The only exception is brix. Its inner loops handle array dimensions of either five or fifteen. Therefore, the maximum parallelism exposed to the basic block orchestrator is at most five or fifteen.

Many of these benchmarks have been parallelized on multiprocessors by recognizing do-all parallelism and distributing such parallelism across the processors. Raw detects the same parallelism by partially unrolling a loop and distributing individual instructions across tiles. The Raw approach is more flexible, however, because it can schedule do-across parallelism contained in loops with loop carried dependences. For example, several loops in tomatcv contain reduction operations, which are loop carried dependences. In multiprocessors, the compiler needs to recognize a reduction and handle it as a special case. The Raw compiler handles the dependence naturally, the same way it handles any arbitrary loop carried dependences.

The size of the datasets in these benchmarks is intentionally made to be small to feature the low communication overhead of Raw. Traditional multiprocessors, with their high overheads, would be unable to attain speedup for such datasets [2].

Most of the speedup attained can be attributed to the exploitation of ILP, but unrolling plays a beneficial role as well. Unrolling speeds up a program by reducing its loop overhead and exposing scalar optimizations across loop iterations. This latter effect is most evident in the jacobi and life benchmarks, where consecutive iterations share loads to the same array elements that can be optimized through common subexpression elimination.

Fppp-kernel is different from the rest of the applications in that it contains irregular fine-grained parallelism. This application stresses the locality/parallelism tradeoff capability of the instruction partitioner. For the fppp-kernel on a single tile, the code generated by the Raw compiler is significantly worse than that generated by the original MIPS compiler. The reason is that the Raw compiler attempts to expose the maximal amount of parallelism without regard to register pressure. As the number of tiles increases, however, the number of available registers increases correspondingly, and the spill penalty of this instruction scheduling policy reduces. The net result is excellent speedup, occasionally attaining more than a factor of two speedup when doubling the number of tiles.

Figure 6: Speedup of applications in the presence of dynamic disturbances for two machine models. The left graph shows results for a machine with a single pc which must stall synchronously; the right graph shows results for a Raw machine with multiple pcs which can stall asynchronously.

**Speedup breakdown** Figure 5 divides the speedup for 32 tiles for each application into three components: base, high-level, and advanced-locality. The base speedup is the speedup from a base compiler which uses simple unrolling and moderate locality optimization.

High-level shows the additional speedup when the base compiler is augmented with high level transformations, which include control localization and array reshape. Array reshape refers to the technique of tailoring the layout of an array to avoid hotspots in memory accesses across consecutive iterations of a loop. It is implemented by allocating a tailored copy of an array to a loop when the loop has a layout preference which differs from the actual layout. This technique incurs the overhead of array copying between the global and the tailored array before and after the loop, but most loops do enough computation on the arrays to make this overhead worthwhile. Brix, cholesky, and vpena benefit from this transformation, while life and tomatcv get their speedup improvements from control localization.

Advanced-locality shows the performance gain from advanced locality optimizations. These optimizations include the use of locality sensitive algorithms for data partitioning and for the merging phase during instruction partitioning, as described in Section 5. The figure shows that all applications except brix and fppp attain sizable benefits from these optimizations. The average performance gain of all the applications is 60%.

**Effects of dynamic events** The Raw compiler attempts to statically orchestrate all aspects of program execution. Not all events, however, are statically predictable. Some dynamic events include I/O operations and dynamic memory operations with unknown tile locations. We study the effects of run-time disturbances such as dynamic memory operations on a Raw machine. We model the disturbances in our simulator as random events which happen on loads and stores, with a 5% chance of occurrence and an average stall time of 100 cycles. We examine the effects of such disturbances on two machine models. One is a faithful representation of the Raw machine; the other models a synchronous machine with a single instruction stream. On a Raw machine, a dynamic event only directly affects the processor on which the event occurs. Other tiles can proceed independently until they need to communicate with the blocked processor. On the synchronous machine, however, a dynamic event stalls the entire machine immediately. This behavior is similar to how a VLIW responds to a dynamic event.\(^1\)

---

\(^1\)Array reshape is currently hand-applied; it is in the process of being automated.

\(^2\)Many VLIW's support non-blocking stores, as well as loads which block on use instead of blocking on miss. These features reduce but do not eliminate the adverse effects of stalling the entire machine, and they come with a potential penalty in clock
Figure 6 shows the performance of each machine model in the presence of dynamic events. Speedup is measured relative to the MIPS-compiled code simulated with dynamic disturbances. The results show that asynchrony on Raw reduces the sensitivity of performance to dynamic disturbances. Speedup for the Raw machine is on average 2.9 times better than that for the synchronous machine. In absolute terms, the Raw machine still achieves respectable speedups for all applications. On 32 tiles, speedup on fppp is 3.0, while speedups for the rest of the applications are at least 7.6.

8 Related work

Due to space limitations, we only discuss past work that is closely related to the problem of space-time scheduling of ILP, which is the focus of this paper. For a comparison of Raw to other architectures, please refer to [23].

The MIMD task scheduling problem is similar to Raw's space-time instruction scheduling problem, with tasks at the granularity of instructions. RAWCC adapts a decomposed view of the problem influenced by MIMD task scheduling. Sarkar, for example, employs a three step approach: clustering, combined merging/placement, and temporal scheduling [20]. Similarly, Yang and Gerasoulis uses clustering, merging, and temporal scheduling, without the need for placement because they target a machine with a symmetric network [25]. Overall, the body of work on MIMD task scheduling is enormous; readers are referred to [1] for a survey of some representative algorithms. One major distinction between the problems on MIMD and on Raw is that on Raw certain tasks (static memory references) have predetermined processor mappings.

In the domain of ILP scheduling, the Bulldog compiler faces a problem which most resembles that of the Raw compiler, because it targets a VLIW machine which distributes not only functional units and register files but memory as well, all connected together via a partial crossbar [12]. Therefore, it too has to handle memory references which have predetermined processor mappings. Bulldog adopts a two-step approach, with an assignment phase followed by a scheduling phase. Assignment is performed by an algorithm called Bottom-Up Greedy (BUG), a critical-path based mapping algorithm that uses fixed memory and data nodes to guide the placement of other nodes. Like the approach adopted by the clustering algorithm in RAWCC, BUG visits the instructions topologically, and it greedily attempts to assign each instruction to the processor that is locally the best choice. Scheduling is then performed by greedy list scheduling.

There are two key differences between the Bulldog approach and the RAWCC approach. First, BUG performs assignment in a single step which simultaneously addresses critical path, data affinity, and processor preference issues. RAWCC, on the other hand, divides assignment into clustering, merging, and placement. Second, the assignment phase in BUG is driven by a greedy depth-first traversal that maps all instructions in a connected subgraph with a common root before processing the next subgraph. As observed in [19], such a greedy algorithm is often inappropriate for parallel computations such as those obtained by unrolling parallel loops. In contrast, instruction assignment in RAWCC uses a global priority function that can intermingle instructions from different connected components of the data dependence graph.

Other work has considered compilation for several kinds of clustered VLIW architectures. A LC-VLIW is a clustered VLIW with limited connectivity which requires explicit instructions for inter-cluster register-to-register data movement [9]. Its compiler performs scheduling before assignment, and the assignment phase uses a min-cut algorithm adapted from circuit partitioning which tries to minimize communication. This algorithm, however, does not directly attempt to optimize the execution length of input DAGs.

Three other pieces of work discuss compilation algorithms for clustered VLIWs with full connectivity. The Multiflow compiler uses a variant of BUG described above [19]. UAS (Unified Assign-and-Schedule) performs assignment and scheduling of instructions in a single step, using a greedy, list-scheduling-like algorithm [6]. Desoli describes an algorithm targeted for graphs with a large degree of symmetry [10]. The algorithm bears some semblance to the Raw partitioning approach, with a clustering-like phase and a merging-like phase. One difference between the two approaches is the algorithm used to identify clusters. In addition, Desoli's clustering phase has a threshold parameter which limits the size of the clusters. This parameter is adjusted iteratively to look for the value which yields the best execution times. The Raw approach, in contrast, allows the graph structure to determine the size of the clusters.

The structure of the Raw compiler is also similar to that of the Virtual Wires compiler for mapping circuits to FPGAs [5], with phases for partitioning, placement, and scheduling. The two compilation problems, however, are fundamentally different from each other, because a Raw machine multiplexes its computational resources (the processors) while an FPGA system does not.

Many ILP-enhancing techniques have been developed to increase the amount of parallelism available within a basic block. These techniques include control speculation [16], data speculation [22], trace/superblock scheduling [13] [15], and predicated execution [3]. With some adjustments, many of these techniques are applicable to Raw.

Raw's techniques for handling control flow are related to several research ideas. Asynchronous global branching is similar to autonomous branching, a branching technique for a clustered VLIW with an independent i-cache on each cluster [6]. The technique eliminates the need to broadcast branch targets by keeping a branch on each cluster.

Control localization is related to research in two areas. It resembles hierarchical reduction [17] in that they both share the idea of collapsing control constructs into a single abstract node. They differ in motivation and context. Control localization is used to enable parallel execution of control constructs on a machine with multiple instruction streams, while hierarchical reduction is used to enable loops with control flow to be software pipelined on a VLIW. In addition, control localization is similar with Multiscalar execution model [22], where tasks with independent flows of control are assigned to execute on separate processors.

It is useful to compare control localization to predicated execution [3]. Control localization enables the Raw machine to perform predicated execution without extra hardware or ISA support. A local branch can serve the same role as a predicate, permitting an instruction to execute only if the branch condition is true. Control localization, however, is more powerful than predicated execution. A single branch can serve as the predicate for multiple instructions, in effect amortizing the cost of performing predicated execution without the ISA support for predicates. Moreover, control localization utilizes its fetch bandwidth more efficiently than predicated execution. For IF-THEN-ELSE constructs, the technique fetches only the path which is executed, unlike predicated execution which has to fetch both paths.

9 Conclusion

This paper describes how to compile a sequential program to a next generation processor that has asynchronous, physically distributed hardware that is fully-exposed to the compiler. The compiler partitions and schedules the program so as to best utilize the hardware. Together, they allow applications to use instruction-level parallelism to achieve high levels of performance.
We have introduced the resource allocation (partitioning and placement) algorithms of the Raw compiler, which are based on MIMD task clustering and merging techniques. We have also described asynchronous global branching, the method which the compiler uses for orchestrating sequential control flow across a Raw processor's distributed architecture. In addition, we have introduced an optimization, which we call control localization, which allows the system to avoid the overheads of global branching by localizing the branching code to a single tile.

Finally, we have presented performance results which demonstrate that for a number of sequential benchmark codes, our system can find and exploit a significant amount of parallelism. This parallel speedup scales with the number of available functional units, up to 32. In addition, because each Raw tile has its own independent instruction stream, the system is relatively tolerant to variations in latency that the compiler is unable to predict.

Although the trend in processor technology favors distributing resources, the resulting loss of the synchronous monolithic view of the processor has prevented computer architects from readily adapting this trend. In this paper we show that, with the help of novel compiler techniques, a fully distributed processor can provide scalable instruction level parallelism and can efficiently handle control flow. We believe that compiler technology can enable very efficient execution of general-purpose programs on next generation processors with fully distributed resources.

Acknowledgments

The authors wish to thank the other members of the MIT Raw team, especially Anant Agarwal. Without their contributions of expertise, ideas and infrastructure this study would not be possible. This research was funded by ARPA grant #DBT63-96-C-0036.

References


Maps: A Compiler-Managed Memory System for Raw Machines

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

Abstract

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

1 Introduction

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

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

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

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

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

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

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

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

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

2 Background

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

![RawP](image)

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

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

Communication on the Raw machine is handled by two distinct networks, a fast, compiler-scheduled static network and a traditional dynamic network. The interfaces to both of these networks are fully exposed to the software. Each switch on the static network is programmable, allowing statically inferrable 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 synchro-
nization 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-nationally inferable communication. A processor handles dynamic messages via either polling or interrupts.

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

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

<table>
<thead>
<tr>
<th>Distance</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic store</td>
<td>17</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>Static store</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>Dynamic load</td>
<td>28</td>
<td>34</td>
<td>36</td>
<td>38</td>
<td>40</td>
</tr>
<tr>
<td>Static load</td>
<td>3</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 1: Cost of memory operations.

![Figure 3: Breakdown of the cost of memory operations between tiles two units apart. Highlighted portions represent processor occupancy, while unshaded portions represent network latency.](image)

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

The goal of Maps is to provide efficient use of hardware memory mechanisms while ensuring correct execution. This goal hinges on three issues, identification of static accesses, support for memory parallelism, and efficient enforcement of memory dependences. The primary goal of Maps is to identify static accesses. As shown in Table 1, static accesses are several times faster than dynamic accesses, and they enable locality optimization by the space-time scheduler to co-locate data with the computation which access it. In addition, Maps attempts to provide memory parallelism by distributing data across tiles. Not only does it distribute different objects to different tiles, it also divides up aggregate objects such as arrays and structs and distributes them across the tiles. This distribution is important as it enables parallel accesses to different parts of the aggregate objects.
For correctness, Maps must ensure that the memory accesses occurring on different tiles obey the dependences implied by the original serial program. Three types of memory dependences need to be considered: those between static accesses, those between dynamic accesses, and those between a static and a dynamic access. Dependences between static accesses are easily enforced. References mapped to different memory banks are necessarily non-conflicting, so the compiler only needs to avoid reordering potentially dependent memory accesses on each tile. The real difficulty comes from dependences involving dynamic accesses, because accesses made by different tiles may potentially be aliased and require serialization. Maps uses a combination of explicit synchronization and a new technique called software serial ordering to enforce these dependences.

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

3 Analysis techniques

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

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

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

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

4 Static promotion of memory accesses

Static promotion is the act of making a reference satisfy the static residence property. Without analysis, the Raw compiler is faced with two unsatisfactory choices: map all the data to a single tile, which makes all memory accesses trivially static at a cost of no memory parallelism; or distribute all data, which enables memory parallelism but requires expensive dynamic accesses. This section describes two compiler techniques for static promotion which preserve some memory parallelism. Section 4.1 describes equivalence class unification, a general promotion technique based on the use of pointer analysis to guide the placement of data. Section 4.2 describes modulo unrolling, a code transformation technique applicable to most array references in the loops of
scientific applications. Section 4.3 explains the limitations of static promotion and motivates the need for an effective dynamic fall-back mechanism.

4.1 Equivalence class unification

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

Elements in aggregate objects such as arrays and structs are often accessed close together in the same program. Distribution and static promotion of arrays are addressed in Section 4.2. For structs, SPAN differentiates between accesses to different fields, so that fields of a struct can be in different alias equivalence classes and distributed across the tiles. Figure 3(c) shows how equivalence class unification is applied to our sample program. Note that aggregate objects distributed across more than one tile have the same memory address on each tile. This property allows a single pointer to refer to the entire object, and it enables address computation to be mapped to any arbitrary tile. The alignment requirement is ensured by doing the appropriate padding on the distributed objects, stack, and heap.

4.2 Modulo unrolling

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

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

Intelligent unrolling, however, can enable static promotion. Figure 6(c) shows the result of unrolling the code in Figure 6(a) by a factor of four. Now, each access always refers to elements on the same memory bank. Specifically, A[i] always refers to tile 0, A[i+1] to tile 1, A[i+2] to tile 2, and A[i+3] to tile 3. Therefore, all resulting accesses can be statically promoted. It can be shown that this technique is always applicable for loops with array accesses having indices which are affine functions of enclosing loop induction variables. These accesses are often found in dense matrix applications and multimedia applications. For a detailed explanation and the symbolic derivation of the unrolling factor, see [1].

4.3 Uses for dynamic references

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

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

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

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

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

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

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

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

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

5.2 Dynamic optimizations

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

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

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

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

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

6 Results

We have implemented Maps on Rawcc, the Raw compiler based on the SUIF compiler infrastructure [16]. This section presents evaluation of Maps. Evaluation is performed on a cycle-accurate simulator of the Raw microprocessor. The simulator uses a MIPS R2000 as the processing element on each tile. It faithfully models both the static and dynamic networks, including any contention effects. Application speedup is derived from comparison with the performance of code generated by the Machsuif Mips compiler [14] executed on the R2000 processing element of a single Raw tile. To expose instruction level parallelism across basic blocks, Rawcc employs loop unrolling and control localization [7]. Inner loops are usually unrolled as many times as there are number of tiles.
Table 2: Benchmark characteristics. Column Seq. RT shows the run-time for the uniprocessor code generated by the Machsuif MIPS compiler.

Table 2 gives the characteristics of the benchmarks used for the evaluation. Benchmarks include five dense matrix applications, three multimedia applications, and two scientific applications with irregular memory access patterns. They are all sequential programs. Some benchmarks are full applications; others are key kernels from full applications. Cholesky and Vpenta are extracted from Nas7 of Spec92. MPEG-kernel is the portion of MPEG which takes up 70% of the total run-time. Because the Raw simulator currently does not support double-precision floating point, all floating point operations are converted to single precision.

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

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

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

Our results in Table 3 show that Rawcc with Maps is able to orchestrate the parallelism available in the applications.

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

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

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

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

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

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

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

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

Figure 11 measures the weighted memory bandwidth utilization of a 32-tile machine. It plots the percentage of memory references being issued in a clock cycle when a given number of tiles is simultaneously issuing memory requests. Results show that except for the two highly serialized
benchmarks (Adpcm and SHA), all the benchmarks are able to exploit at least a small amount of parallel memory bandwidth. On the positive end, MPEG-kernel and all the dense matrix applications have at least 20% of their accesses being performed on cycles which issue five or more accesses, with Vpenta enjoying 10-way memory parallelism for over 20% of its accesses.

Exposing memory parallelism through dynamic accesses
Static accesses are usually better than dynamic accesses because of their lower 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>
<thead>
<tr>
<th>Benchmark</th>
<th>N=1</th>
<th>N=2</th>
<th>N=4</th>
<th>N=8</th>
<th>N=16</th>
<th>N=32</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPEG-kernel</td>
<td>0.86</td>
<td>0.80</td>
<td>1.54</td>
<td>1.89</td>
<td>2.35</td>
<td>3.45</td>
</tr>
<tr>
<td>Moldyn</td>
<td>0.93</td>
<td>0.44</td>
<td>0.54</td>
<td>0.61</td>
<td>0.67</td>
<td>0.83</td>
</tr>
<tr>
<td>Unstructured</td>
<td>0.76</td>
<td>0.24</td>
<td>0.37</td>
<td>0.55</td>
<td>0.73</td>
<td>0.83</td>
</tr>
</tbody>
</table>

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

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

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

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

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

Unstructured contains an array X[i] 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>
<thead>
<tr>
<th>Array mapping</th>
<th>Loop</th>
<th>Access type</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>centralized</td>
<td>init</td>
<td>static serial</td>
<td>1.89</td>
</tr>
<tr>
<td></td>
<td>use</td>
<td>static serial</td>
<td>3.86</td>
</tr>
<tr>
<td></td>
<td>total</td>
<td></td>
<td>3.85</td>
</tr>
<tr>
<td>distributed</td>
<td>init</td>
<td>dynamic serial</td>
<td>0.59</td>
</tr>
<tr>
<td></td>
<td>use</td>
<td>static parallel</td>
<td>4.43</td>
</tr>
<tr>
<td></td>
<td>total</td>
<td></td>
<td>4.42</td>
</tr>
</tbody>
</table>

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 usage loop in exchange for dynamic accesses with software serial ordering in the non-critical init loop. Performance is measured for 32 tiles.

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

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

7 Related work

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

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

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

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

A different type of memory disambiguation, relative memory disambiguation, is relevant on the more typical bus-based VLIW machines such as the Multiflow Trace [8]. Relative memory disambiguation aims to discover whether two memory accesses never refer to the same memory location. Successful disambiguation implies that accesses can be executed in parallel. Hence, relative memory disambiguation is more closely linked to dependence and pointer analysis techniques.

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

8 Conclusion

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

This paper addresses the challenging compiler problem of orchestrating distributed memory and communication resources to provide a uniform view of the memory system. We present a compiler-managed memory system called
Maps that provides a sequential memory abstraction to the programmer. The Maps solution attempts to maximize both memory parallelism and its use of the static interconnect.

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

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

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

Acknowledgments

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

References


