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.cag.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

* *Proceedings of the Eighth International Conference on Architectural Support for Programming Language and Operating Systems* **(ASPLOS-8)**, San Jose, Ca, October, 1998.

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 are 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.

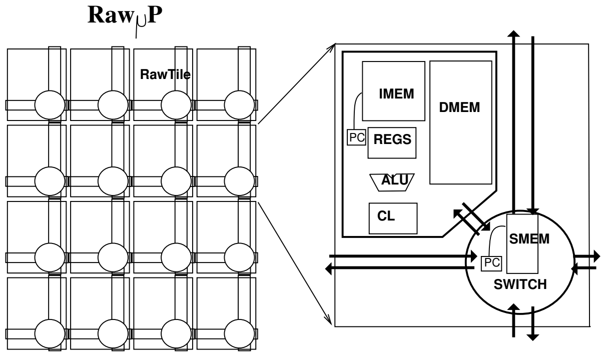

Figure 1: A Raw microprocessor is a mesh of tiles, each with a processor and a switch. The processor contains instruction memory, data memory, registers, ALU, and configurable logic (CL). The switch contains its own instruction memory.

The tile is kept simple and devoid of complex hardware resources in order to maximize the clock rate and the number of tiles that can fit on a chip. Raw's network is tightly integrated with the processor to provide fast, register-level communication. Unlike modern superscalars, the interface to this interconnect is fully exposed to the software.

The network interface on a Raw machine is integrated directly into the processor pipeline to support single-cycle sends and receives of word-sized values. A word of data travels across one tile in one clock cycle. The network supports both static and dynamic routing through the static and dynamic switches, respectively. The static switch is programmable, allowing statically inferable communication patterns to be encoded in the instruction streams of the switches. This approach eliminates the overhead of composing and routing a directional header, which in turn allows a single word of data to be communicated efficiently. 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 dynamic memory references and I/O operations, and it obviates the lock-step synchronization of program counters required by many statically scheduled machines. The dynamic switch is a wormhole router that makes routing decisions based on the header of each message. It includes additional lines for flow control. This paper focuses on communication using the static network.

The Raw prototype uses a MIPS R2000 pipeline on its tile. For the switch, it uses a stripped down R2000 augmented with a *route* instruction. Communication ports are added as extensions to the register space. The bold arrows in Figure 1 shows the organization of ports for the static network on a single tile. Each switch on a tile is connected to its processor and its four neighbors via an input port and an output port. It takes one cycle to inject a message from the processor to its switch, receive a message from a switch to its processor, or route a message between neighboring tiles. A single-word message between neighboring processors would take four cycles. Note, however, that the ability to access the communication ports as register operands allows useful computation to overlap with the act of performing a send or a receive. Therefore, the *effective* overhead of the communication can be as low as two cycles.

In addition to its scalability and simplicity, the Raw machine is an attractive NURA machine for several reasons:

- *Multisequentiality:* Multisequentiality, the presence of independent 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.

Relationship between Raw and VLIW machines The Raw architecture draws much of its inspiration from VLIW machines. They both share the common goal of statically scheduling ILP. From a macroscopic point of view, a Raw machine is the result of a natural evolution from a VLIW, driven by the desire to add more computational resources.

There are two major distinctions between a VLIW machine and Raw machine. First, they differ in resource organization. VLIW machines of various degrees of scalability have been proposed, ranging from completely centralized machines to machines with distributed functional units, register files, and memory [19]. The Raw machine, on the other hand, is the first ILP microprocessor that provides a software-exposed, scalable, two-dimensional interconnect between clusters of resources. This feature limits the length of all wires to the distance between neighboring tiles, which in turn enables a higher clock rate.

Second, the two machines differ in their control flow model. A VLIW machine has a single flow of control, while a Raw machine has multiple flows of control. As explained above, this feature increases available exploitable parallelism, enables asynchronous global branching and control localization, and improves tolerance of dynamic events.

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 orchestrater 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 orchestrater 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 residence 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 residence property. Our technique is a generalization of an observation made by Ellis [12]. Details are presented in [8].

Scalar values communicated within basic blocks follow a dataflow 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 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 orchestrater

The basic block orchestrater 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 orchestrater performs these functions, followed by a general discussion of its design.

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

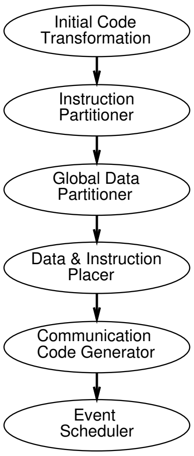

Figure 2: Phase ordering of the basic block orchestrater.

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 residence 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.

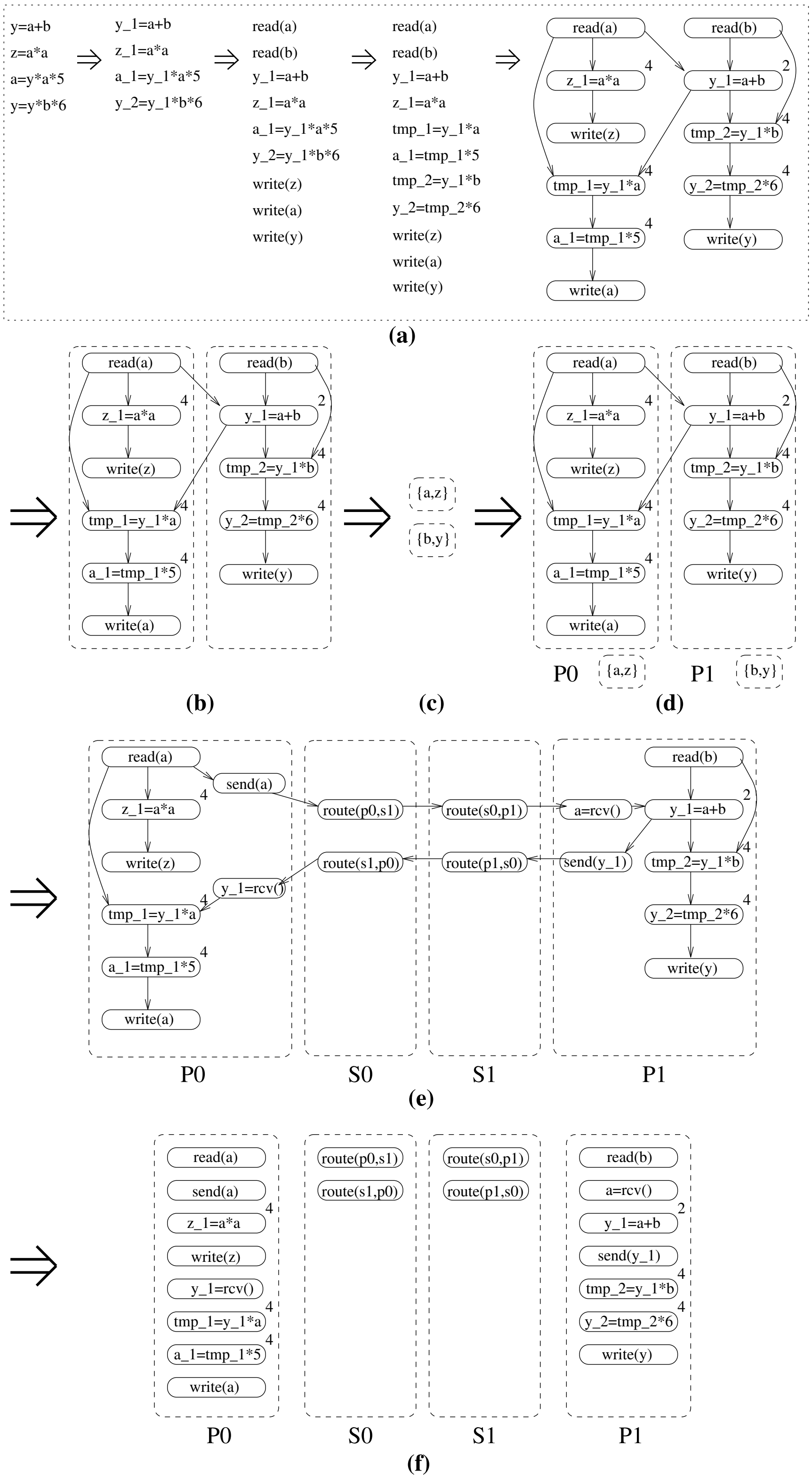

Figure 3: An example of the program transformations performed by RAWCC. (a) shows the initial program undergoing transformations made by *initial code transformation*; (b) shows result of *instruction partitioner*; (c) shows result of *global data partitioner*; (d) shows result of *data and instruction 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. It 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 absence 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 schedule 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 descendent 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 orchestrater 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 fpppp-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 effect 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 orchestrater, 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

Benchmark	Source	Lang.	Lines of code	Primary Array size	Seq. RT (cycles)	Description
fpppp-kernel	Spec92	FORTRAN	735	-	8.98K	Electron Interval Derivatives
btrix	Nasa7:Spec92	FORTRAN	236	15×15×15×5	287M	Vectorized Block Tri-Diagonal Solver
cholesky	Nasa7:Spec92	FORTRAN	126	3×32×32	34.3M	Cholesky Decomposition/Substitution
vpenta	Nasa7:Spec92	FORTRAN	157	32×32	21.0M	Inverts 3 Pentadiagonals Simultaneously
tomcatv	Spec92	FORTRAN	254	32×32	78.4M	Mesh Generation with Thompson's Solver
mxm	Nasa7:Spec92	FORTRAN	64	32×64, 64×8	2.01M	Matrix Multiplication
life	Rawbench	C	118	32×32	2.44M	Conway's Game of Life
jacobi	Rawbench	C	59	32×32	2.38M	Jacobi Relaxation

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

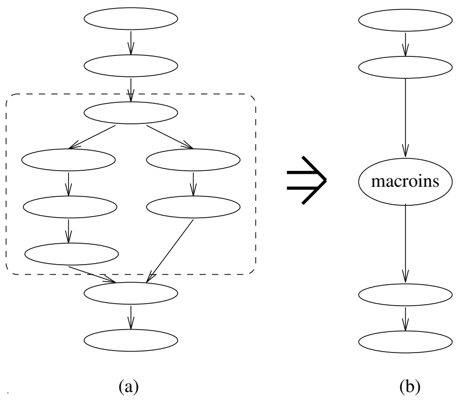

Figure 4: An illustration of control localization. (a) shows a control flow graph before control localization. Each oval is an instruction, and the dashed box marks the code sequence to be control localized. (b) shows the control flow graph after control localization.

Benchmark	N=1	N=2	N=4	N=8	N=16	N=32
fpppp-kernel	0.48	0.68	1.36	3.01	6.02	9.42
btrix	0.83	1.48	2.61	4.40	8.58	9.64
cholsky	0.88	1.68	3.38	5.48	10.30	14.81
vpenta	0.70	1.76	3.31	6.38	10.59	19.20
tomcatv	0.92	1.64	2.76	5.52	9.91	19.31
mxm	0.94	1.97	3.60	6.64	12.20	23.19
life	0.94	1.71	3.00	6.64	12.66	23.86
jacobi	0.89	1.70	3.39	6.89	13.95	38.35

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

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 fpppp of Spec92. Since the Raw prototype currently does not support double-precision floating point, all floating point operations in the original benchmarks are converted to single precision. Table 1 gives some basic characteristics of the benchmarks.

Speedup We compare results of the Raw compiler with the results of a MIPS compiler provided by Machsuif [21] targeted for a MIPS R2000. Table 2 shows the speedups attained by the benchmarks for Raw machines of various number of tile. The results show that the Raw compiler is able to exploit ILP profitably across the Raw tiles for all the benchmarks. The average speedup on 32 tiles is 19.7.

All the benchmarks except fpppp-kernel are dense matrix applications. These applications perform particularly well on a Raw machine because arbitrarily large amount of parallelism can be ex-

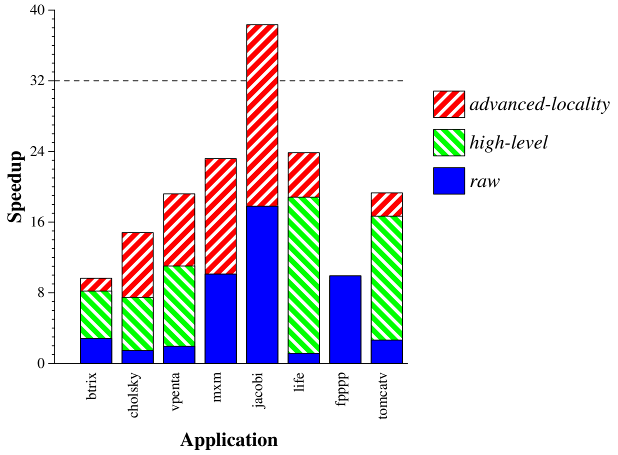

Figure 5: Breakdown of speedup on 32 tiles into *base* component, *high-level* component, and *advanced-locality* component.

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 btrix. Its inner loops handle array dimensions of either five or fifteen. Therefore, the maximum parallelism exposed to the basic block orchestrater 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 tomcatv 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 other 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.

Fpppp-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 fpppp-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.

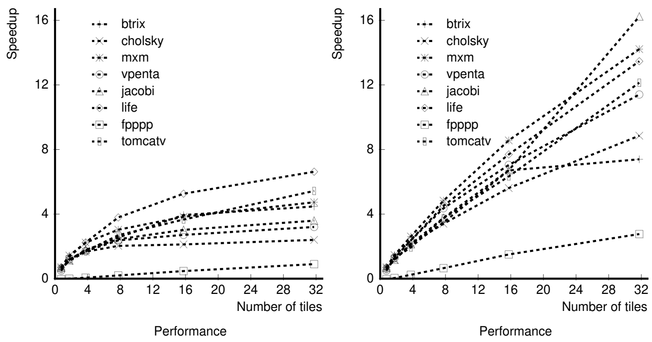

Figure 6: Speedup of applications in the presence of dynamic disturbances for two machine models. The left graph shows results for a machine with a single pc which must stall synchronously; the right graph shows results for a Raw machine with multiple pcs which can stall asynchronously.

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. [1] 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. Btrix, cholesky, and vpenta benefit from this transformation, while life and tomcatv 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 btrix and fpppp 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 effects 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. [2]

[1] Array reshape is currently hand-applied; it is in the process of being automated.

[2] Many VLIWs 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 fpppp 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 aapproach. 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.

speed.

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

[1] I. Ahmad, Y. Kwok, and M. Wu. Analysis, Evaluation, and Comparison of Algorithms for Scheduling Task Graphs on Parallel Processors. In *Proceedings of the Second International Symposium on Parallel Architectures, Algorithms, and Networks*, pages 207–213, June 1996.

[2] S. Amarasingle, J. Anderson, C. Wilson, S. Liao, B. Murphy, R. French, and M. Lam. Multiprocessors from a Software Perspective. *IEEE Micro*, pages 52–61, June 1996.

[3] D. August, W. Hwu, and S. Mahlke. A Framework for Balancing Control Flow and Predication. In *Proceedings of the 30th International Symposium on Microarchitecture*, Dec. 1997.

[4] J. Babb, M. Frank, V. Lee, E. Waingold, R. Barua, M. Taylor, J. Kim, S. Devabhaktuni, and A. Agarwal. The RAW Benchmark Suite: Computation Structures for General Purpose Computing. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, Apr. 1997.

[5] J. Babb, R. Tessier, M. Dahl, S. Hanono, D. Hoki, and A. Agarwal. Logic emulation with virtual wires. *IEEE Transactions on Computer Aided Design*, 16(6):609–626, June 1997.

[6] S. Banerjia. Instruction Scheduling and Fetch Mechanism for Clustered VLIW Processors. In *Ph.D Thesis, North Carolina State University*, 1998.

[7] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal. Maps: A Compiler-Managed Memory System for Raw Machines. Technical Memo TM-583, MIT LCS, July 1998.

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

[9] A. Capitanio, N. Dutt, and A. Nicolau. Partitioned Register Files for VLIWs: A Preliminary Analysis of Tradeoffs. In *Proceedings of the 25th International Symposium on Microarchitecture*, Dec. 1992.

[10] G. Desoli. Instruction Assignment for Clustered VLIW DSP Compilers: A New Approach. Technical report, HP HPL-98-13, Feb. 1998.

[11] M. Dikaiakos, A. Rogers, and K. Steiglitz. A Comparison of Techniques Used for Mapping Parallel Algorithms to Message-Passing Multiprocessors. In *Proceedings of the 6th IEEE Symposium on Parallel and Distributed Processing*, Oct. 1994.

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

[13] J. A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, 7(C-30):478–490, July 1981.

[14] L. Gwennap. Digital 21264 Sets New Standard. *Microprocessor Report*, pages 11–16, Oct. 1996.

[15] W. Hwu, S. Mahlke, W. Chen, P. Chang, N. Warter, R. Bringmann, R. Ouellette, R. Hank, T. Kiyohara, G. Haab, J. Holm, and D. Lavery. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *The Journal of Supercomputing*, 7(1), Jan. 1993.

[16] V. Kathail, M. Schlansker, and B. R. Rau. HPL PlayDoh Architecture Specification: Version 1.0. Technical report, HP HPL-93-80, Feb. 1994.

[17] M. S. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proc. Int'l Conf. on Programming Language Design and Implementation (PLDI)*, pages 318–328, June 1988.

[18] M. S. Lam and R. P. Wilson. Limits of Control Flow on Parallelism. In *Proceedings of 19th Annual International Symposium on Computer Architecture*, pages 46–57, May 1992.

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

[20] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Pitman, London and The MIT Press, Cambridge, Massachusetts, 1989. In the series, Research Monographs in Parallel and Distributed Computing.

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

[22] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, 1995.

[23] E. Waingold, M. Taylor, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, S. Devabhaktuni, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring It All To Software: Raw Machines. *Computer*, pages 86–93, Sept. 1997.

[24] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S. Liao, C.-W. Tseng, M. Hall, M. Lam, and J. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *ACM SIGPLAN Notices*, 29(12), Dec. 1996.

[25] T. Yang and A. Gerasoulis. List Scheduling With and Without Communication. *Parallel Computing Journal*, 19:1321–1344, 1993.

[26] T. Yang and A. Gerasoulis. DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, 1994.