Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine

Walter Lee, Rajeev Barua, Devabhaktuni Srikrishna, Jonathan Babb,
Vivek Sarkar, Saman Amarasinghe, Anant Agarwal [*]
M.I.T. Laboratory for Computer Science
Cambridge, MA 02139, U.S.A.

{walt,barua,chinnama,jbabb}@lcs.mit.edu
{vivek,saman,agarwal}@lcs.mit.edu
http://cag-www.lcs.mit.edu/raw

December 3, 1997

Abstract

Advances in VLSI technology will enable chips with over a billion transistors within the next decade. Unfortunately, the centralized-resource architectures of modern microprocessors are ill-suited to exploit such advances. Achieving a high level of parallelism at a reasonable clock speed requires distributing the processor resources – a trend already visible in the dual-register-file architecture of the Alpha 21264. A Raw microprocessor takes an extreme position in this space by distributing all its resources such as register files, memory ports, and ALUs over a pipelined two-dimensional interconnect, and exposing them fully to the compiler. Compilation for instruction-level parallelism (ILP) on such distributed-resource machines requires both spatial instruction scheduling and traditional temporal instruction scheduling. The compiler must also orchestrate data memories to take advantage of the on-chip distributed-memory bandwidth. This paper describes the techniques used by the Raw compiler to handle these issues. Preliminary results from a SUIF-based compiler for sequential programs written in C and Fortran indicate that the Raw approach to exploiting ILP can achieve speedups scalable with the number of processors for applications with parallelism within a basic block. Though research is still ongoing, these results offer positive indications that Raw may provide competitive performance against existing superscalars for applications with small amounts of parallelism, while achieving significantly better performance for applications with a large amount of ILP or coarse-grain parallelism.

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

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

at every design point in a machine. As processor designers continue in their pursuit of an architecture that can exploit more parallelism and thus requires 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 internal organization of the Alpha 21264 [6] [7], which duplicates its register file to provide the requisite number of ports at a reasonable clock speed. A cluster is formed by organizing half of the functional units and half of the cache ports around each register file. Communication within a cluster occurs at normal speed, but it takes an extra cycle to propagate a value from one cluster to either the register file or the bypass logic of the remote cluster.

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. *Instruction scheduling becomes a spatial problem as well as a temporal problem.*

The Raw machine [15] 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, and data partitioning using a distributed on-chip memory model. It provides parallel memory access at a fine granularity, fast support for memory references that are statically analyzable, and a fallback mechanism for dynamic references. We show that the Raw compiler can extract both regular and irregular parallelism from sequential instruction streams, thereby achieving good performance for a large class of applications.

The rest of the paper is organized as follows. Section 2 introduces the concept of a NURA machine, relates the Raw machine with modern processors through this concept, and illustrates its compilation issues. Section 3 describes RAWCC , a specific instance of a compiler for NURA machines. Section 4 examines the optimization problems encountered by compilers of the Raw machine or any NURA machine, and it describes the solutions employed by RAWCC . Section 5 describes the memory model used by Raw to handle distributed memory in a mostly static machine. Section 6 shows the performance of RAWCC . Section 7 presents future work, and Section 8 concludes. Appendix A gives a proof of how a mostly static machine can tolerate skews introduced by dynamic events without changing the behavior of the program.

2 Motivation

This section motivates the Raw architecture and its compiler for exploiting ILP. 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, and we explain the issues concerned with compiling to such architectures.

A problem with scalability Modern processors are not designed to scale. Because superscalars require significant global hardware resources to support parallel instruction execution, architects for such machines face an uncomfortable dilemma. On the one hand, faster machines require more hardware resources such as register ports, caches, functional units, issue logic, instruction window, and renaming logic. On the other hand, the quadratic area complexity of some of these resources and the need to connect these resources with long global wires make it difficult for architects to add more while keeping the cycle time low. Like a superscalar, a VLIW suffers similar scalability problems, but to a lesser extent. It needs not implement hardware logic to discover ILP, but it still must contend with problems on issue bandwidth, multi-ported register files, caches, and wire delays.

A logical solution: impose hierarchy Up to now, microprocessors have maintained the faithful representation of a monolithic uniprocessor. As pressure from all sources demand computers to have more resources and be more powerful, this monolithic view will be difficult to maintain. A crack is already visible in the Alpha 21264. Its dual-ported cache and four functional units require eight read ports and six write ports from the register file. A single register file is unable to provide this many ports without incurring a large penalty in cycle time. Instead, the register file is duplicated, with each physical register file providing 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, but it takes an extra cycle to propagate a value from one cluster to either the register file or the bypass logic of the remote cluster.

This example suggests a logical evolutionary path that resolves the scalability problem: impose a hierarchy on the organization of hardware resources. [13] To address the scalability of hardware resources, 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. Once processing units are replicated and distributed, a final scalability problem remains: the scalability of the interconnect. Initially, a bus or a full crossbar suffices to give acceptable performance. Eventually, as the number of components increases, a point to point network will be necessary to provide the required latency and bandwidth – a progression reminiscent of multiprocessor evolution.

NURA machines 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 register-like latencies that are much lower latencies than that of a multiprocessor.

As the most frequently accessed type of storage element, any change in the register model has profound implications on all levels of the computer system. The presence of distributed functional units and register files in a NURA machine means that instructions should be assigned to the appropriate functional units to preserve locality. Instruction scheduling becomes a spatial problem as well as a temporal problem. This extra dimension in generating code for NURA machines means that current compilation technology for exploiting ILP cannot efficiently support NURA machines. In the Alpha 21264, the assignment of instructions to functional units is completely dynamic. For a one level hierarchy, this solution is reasonable. As the structure grows, however, compile time analysis will be helpful if not

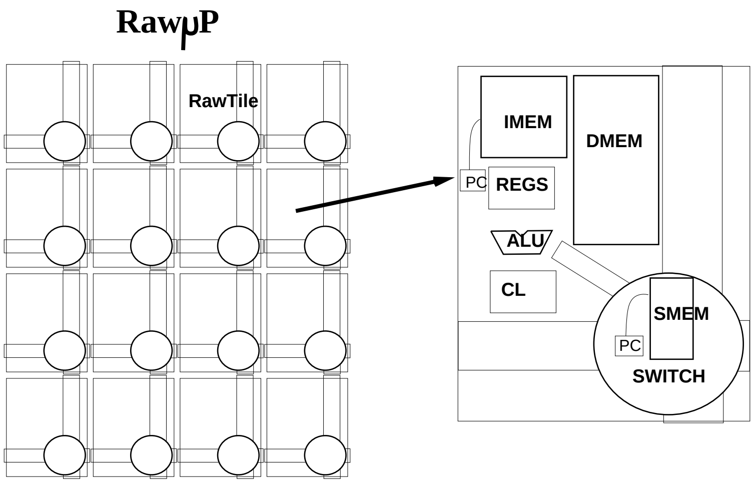

Figure 1: RawμP composition. A typical Raw system might include a Raw microprocessor coupled with off-chip RDRAM and stream-IO devices.

absolutely necessary in order to obtain acceptable performance. We describe the compilation techniques necessary to handle this problem in Section 4.1.

Raw architecture The Raw machine [15] uses a NURA architecture motivated by the need to design simple and highly scalable processors. Raw's features relevant to the exploitation of ILP include the use of a simple, replicated tile, each with its own instruction stream, and a programmable, tightly integrated interconnect between tiles, as depicted in Figure 1. Raw machine also supports multi-granular (bit and byte level) operations as well as customizable configurable logic, but this paper does not address these features.

Each Raw tile contains a simple RISC-like pipeline and is interconnected with other tiles over a pipelined, point-to-point network. Unlike current superscalars, a Raw processor does not bind specialized logic structures such as register renaming logic or dynamic instruction issue logic into hardware. Instead, it focuses on keeping each tile small to maximize the number of tiles that can fit on a chip, thereby increasing the amount of parallelism it can exploit and the clock speed it can achieve. The network in a Raw machine resides between the register files and the functional units to provide fast, register-level communication. Unlike modern superscalars, the interface to this interconnect is fully exposed to the software.

Figure 2 contrasts the internal organization of a Raw machine versus that of a superscalar. In a Raw machine, the switch 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 switch contains two distinct networks, a static and a dynamic one. The static switch is programmable, allowing statically inferable communication pattern 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. Communication instructions (*send*, *receive*, or *route*) 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

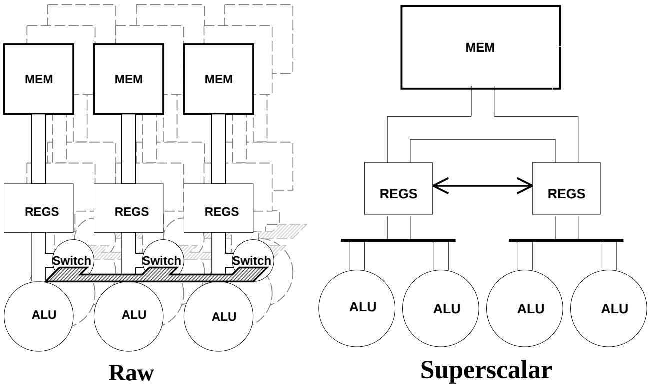

Figure 2: Internal organization of Raw processors versus superscalars. The Raw microprocessor distributes the register file and communicates between ALUs on a software-exposed, point-to-point interconnect. In contrast, a superscalar contains clusters of a register file and several function units, and it communicates between these components via busses hidden from the software.

as cache misses (see Appendix A), 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.

Raw machines draw much of its inspiration from VLIW machines, and they share many common features. Like the VLIW Multiflow TRACE machine [5], Raw machines have a large register name space, a distributed register file, multiple memory ports, and they rely heavily on compiler technology to discover and statically schedule ILP. Unlike traditional VLIWs, however, Raw machines provide multiple instruction streams. Individual instruction streams give a Raw processor significantly more flexibility to perform independent but statically scheduled computations on different tiles, such as loops with different bounds. They also provide better behavior in the face of dynamic events. In VLIW, instructions have to proceed in lock-step, so that a cache miss causes the entire machine to stall. In a Raw processor, the need to explicitly synchronize the instruction streams is obviated by the presence of near-neighbor flow control. The effects of a cache miss are localized to a single instruction stream on a single tile; other tiles can proceed independently. In addition, a Raw machine differs from a VLIW machine in that it explores a software-exposed two-dimensional interconnect, and that it combines instruction and data scheduling over this interconnect. This feature makes the space-time scheduling problem more general for Raw machines than for VLIWs.

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

- *Elimination of the ISA constraint:* Raw resolves several longstanding problems with handling the expansion of register sets. An ISA traditionally imposes troublesome restrictions on register set expansion. Backward compatibility prohibits an ISA from exporting extra registers to the software. Modern architectures circumvent this problem by de-coupling the number of ISA registers from the number of physical registers, and by relying on complex dynamic renaming logic to take advantage of the additional physical registers. This indirection, however, comes at a cost of under-utilizing powerful static compiler analysis that can alleviate or even eliminate the burden of dynamic renaming. Moreover, it leads to code inefficiency that is uncorrectable at run-time when the compiler runs out of ISA registers and is forced to spill to memory. Spill code, which involves a potentially expensive memory reference, cannot be eliminated at run-time even in the presence of available dynamic registers. A Raw machine adds registers by increasing the number of tiles. The ISA of the processors on an individual tile need not change as a result.

- *Establishment of a compiler interface for locality management:* Current ISAs do not provide interface mechanisms to allow the compiler to control computation locality in NURA machines. Raw fully exposes its hardware to the compiler by exporting a simple cost model for communication and computation. In turn, the compiler is responsible for the assignment of instructions to Raw tiles. We believe instruction partitioning should be accomplished at compile time for several reasons: First, it requires sophisticated data dependence graph analysis, for which the necessary information is readily available at compile time but not at run-time, and second, we can ill-afford its complexity at runtime.

- *Mechanism for precise orchestration:* Raw's programmable static switch is an essential feature for exploiting ILP in the Raw machine. First, it allows single-word register-level transfer without the overhead of composing and routing a message header. Second, it gives the Raw compiler the fine-grain control required to orchestrate computation and communication events to eliminate synchronization overhead. Third, the Raw compiler can use its full knowledge of the network status to minimize congestion and route data around hot spots. By considering congestion, the Raw compiler can also predict the latency of communication events precisely, thus enabling it to schedule instructions well.

3 The Raw compiler system

This section describes the Raw system, consisting of the MIT Raw prototype machine and its compiler. Section 3.1 describes the Raw prototype targeted by the compiler. Section 3.2 describes the compiler which exposes the ILP of sequential programs to the hardware. Section 3.3 describes a major component of the Raw compiler, the basic block orchestrater.

3.1 Target machine model — the Raw prototype

To provide a concrete target for studying the compilation issues in Raw specifically and NURA in general, we have designed a Raw prototype at MIT. Our results are obtained from an instruction-level simulator of the prototype. This section describes the features of the prototype relevant to the exploitation of ILP. Multigranular operations and reconfigurable logic are features orthogonal to this goal, so they are not included in this paper.

Like the Raw machine described in Section 2, the Raw prototype consists of simple tiles organized in a mesh interconnect. Each tile contains a processor and a switch. The processor supports the R2000 instruction set. It consists of a conventional five-stage pipeline, single-precision floating point unit, fully bypassed and pipelined functional units, 32 GPRs, and memory. It does not contain any FPRs; floating point operations manipulate GPRs instead. Table 1 lists the latencies of the basic instructions. The switch itself contains a stripped down Mips R2000 with its own instruction sequencer and CPU, but with a smaller register set (8 GPRs) and no data memory. It supports both a static and a dynamic network. Since the compiler mainly generates static communication, we focus on the static network here. The processor and the static switch are connected internally via two ports, one from processor to switch and

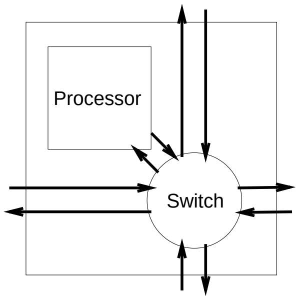

Figure 3: Communication ports for the static network on a prototype tile. Each switch on a tile is connected to its processor and its four neighbors via an input port and an output port. The figure is not drawn to scale.

Int Op	Cycles	Fp Op	Cycles
ADD	1	ADDF	2
SUB	1	SUBF	2
MUL	12	MULF	4
DIV	35	DIVF	12

Table 1: Latency of common operations on the Raw prototype machine.

one from switch to processor. The switch is also connected to each of its four neighboring switches via an input and an output port, for a total of eight external ports. Figure 3 shows the organization of ports on a single tile.

Communication ports are exported to the software as extensions to the register set. They can be used like normal registers as operands to any computation instruction, with the natural restriction that input ports are read-only and output ports are write-only. Like regular registers, communication ports hold 32-bit values. They have blocking semantics, which means that an instruction attempting to read from an empty input port or write to a full output port will stall until the port is ready.

To support the expression of static communication, the instruction set for the switches is augmented with a ROUTE instruction.[1] A ROUTE instruction consists of pairs of source and destination registers, each pair representing a routing path. In a ROUTE instruction, an input port can be listed in more than one route to represent a "multicast" operation, but an output port can be listed in only one route. If a port specified by a ROUTE instruction is not ready (i.e. input is empty, or output is full), the entire routing instruction stalls. Because a ROUTE instruction does not consume any of the resources needed by a computation instruction, a switch can perform both a computation instruction and a ROUTE instruction on the same cycle.

Raw's simple communication cost model is also exposed to the compiler. 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. For example, a single-word message between neighboring processors would take four cycles, as shown in Figure 4. Note, however, that it is possible for both the send and receive instruction to be performing useful computation, so that the *effective* overhead of the

[1] It is not necessary to create new instructions to support sending and receiving of data, since these actions can be expressed by using existing computation instructions with the appropriate communication registers.

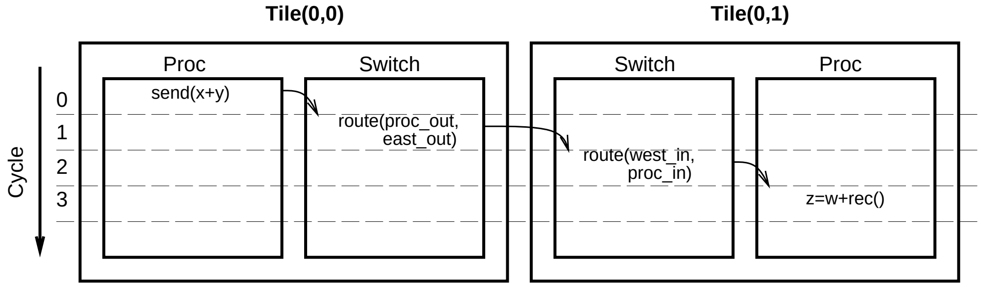

Figure 4: The end-to-end latency of a single-word message between neighbors is four cycles. In this example, both the send and the receive are performing useful computation, so that the effective overhead is only two cycles.

communication can be as low as two cycles.

3.2 RAWCC

RAWCC , the compiler for the Raw prototype, is implemented using SUIF [16], the Stanford University Intermediate Format. It supports both C and Fortran programs. The Raw compiler consists of three phases. The first phase identifies the basic blocks. Currently, the basic block identifier comprises a traditional control flow analysis augmented with loop unrolling. In the future, it will leverage trace scheduling [4] or superblock [8] techniques used by VLIW compilers to increase the size of the basic blocks.

The second phase, the *basic block orchestrater*, exposes the ILP within each basic block and performs the space-time scheduling of the basic block. It is a major component of the compiler and will be discussed in detail below.

The final phase does the code generation for both the processors and the switches. We use the Mips back-end developed in Machine SUIF [12], with a few modifications to handle the extra communication instructions and communication registers.

3.3 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 basic blocks that can be run in parallel on Raw. The orchestrater generates intermediate code for both the processor and the programmable switch on each tile.

Exploiting the ILP within a basic block consists of three steps. First, the orchestrater assigns each instruction in the basic block to a physical processor. Second, for any value which is needed at a processor different from where it is generated, the orchestrater generates the necessary communication instructions to transmit that value. Finally, the orchestrater schedules all events, consisting of computation on the tiles as well as communication on both the tiles and the switches.

The basic access model for program variables is as follows. Every variable is assigned to a home tile, where persistent values of the variable are stored. At the beginning of a basic block, the value of a variable is transferred from its home to the tiles which use the variable. Within a basic block, references

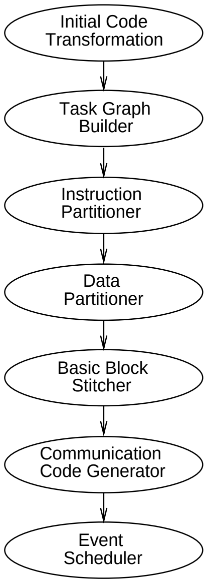

Figure 5: Task composition of the basic block orchestrater.

are strictly on local renamed variables, allocated either in registers or on the stack. At the end of the basic block, the updated value of a modified variable is transferred from the computing tile to its home tile. This access model works well for simple scalar references, requires some compiler analysis for some memory references, and fails for other memory references. For a discussion of the issues and solutions, see Section 5.

The opportunity to exploit ILP across statically-scheduled, MIMD processing units is unique to Raw. However, the Raw compiler problem is actually a composition of problems in two previously studied compiler domains: the task partitioning/scheduling problem and VirtualWires' communication scheduling problem. Research in the abstract task partitioning and scheduling problem has been extensive in the context of mapping directed acyclic task graphs onto MIMD multiprocessors(e.g., [10][18]). The Raw task partitioning and scheduling problem is similar, with tasks defined to be individual instructions. In VirtualWires [3], the pin limitation of FPGAs is alleviated by multiplexing each pin to communicate more than one values. Communication events are scheduled on pins over time. In Raw, the communication ports on the processors and switches serve the same role as pins in FPGAs; the scheduling problems in the two contexts are analogous.

The following are the implementation details of the basic block orchestrater. Figure 5 shows its task decomposition. Each task is described in turn below. To facilitate the explanation, Figure 6 shows the transformations performed by RAWCC on a sample program. The discussion here focuses on the functionality of the basic system. Handing of unpredictable timing variations is discussed in details in Appendix A, while handling of memory references is discussed in Section 5.

Initial code transformation Initial code transformation massages a basic block into a form suitable for subsequent analysis phases. Figure 6a shows the transformations performed by this phase. First, *software renaming* converts statements of the program to static single assignment form. Such conversion removes *anti-dependencies* (write after read) and *output-dependencies* (write after write) from the basic block, which in turn exposes the available parallelism. It is analogous to register renaming performed by superscalars.

Second, 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.

Task graph builder The task graph builder constructs the task graph representing the basic block. In the task graph, each node represents an instruction, and each directed edge represents a dependency between nodes. Figure 6b shows a sample output of this phase. Nodes are labeled with the estimated costs of running the instructions. For example, the node for a floating point add in the example is labeled with two cycles. Edges are labeled with the data size, from which the communication cost can be computed after the instruction mapping is known. For Raw, an edge label is always one word and is implicit.

Instruction partitioner The instruction partitioner maps instructions to physical tiles. It generates partitions that balance the benefits of parallelism against the overheads of communication. This problem is decomposed into three sub-problems: clustering, merging, and placement. In clustering, instructions are clustered to eliminate as much as possible the effects of communication overhead on the run-times of critical paths. In merging, clusters of instructions are merged until the number of clusters equals the number of available tiles. Finally in placement, the clusters are mapped onto physical tiles. This last step takes into account the machine configuration. Figure 6c shows a sample output of the instruction partitioner. We discuss the partitioning problem in more detail in Section 4.1.

Data partitioner The data partitioner assigns data values to “home” tile locations. Home locations are needed to store values communicated between basic blocks. 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. Figure 6d shows the assignment of data values to home locations. Note that variables introduced by *initial code transformation* (*e.g.*, y_1 and tmp_1) do not need home locations because they are only referenced within the basic block. Currently the data partitioner uses a round-robin assignment algorithm. A more intelligent algorithm would consider data usage pattern as well. For a discussion on the allocation policy for larger objects such are arrays and structs, see Section 5.2.

Basic block stitcher The basic block stitcher generates the necessary communication, called *stitch code*, between basic blocks. In the unoptimized case, a variable’s value is sent from its home tile to its read-tiles at the beginning of a basic block. Conversely, its value is sent from its write-tile to its home tile at the end of a basic block.[2]

Control flow analysis helps optimize the stitch code. In one optimization, called *one-distance forwarding*, values needed by each tile in a given basic block are automatically forwarded to the right locations by its preceding basic blocks. For values which are produced by the preceding basic blocks, this optimization short circuits the producer-to-home and home-to-consumer communication into a single producer-to-consumer communication.

[2] With renaming, all values are single assignment, so each value has a unique write-tile per basic block.

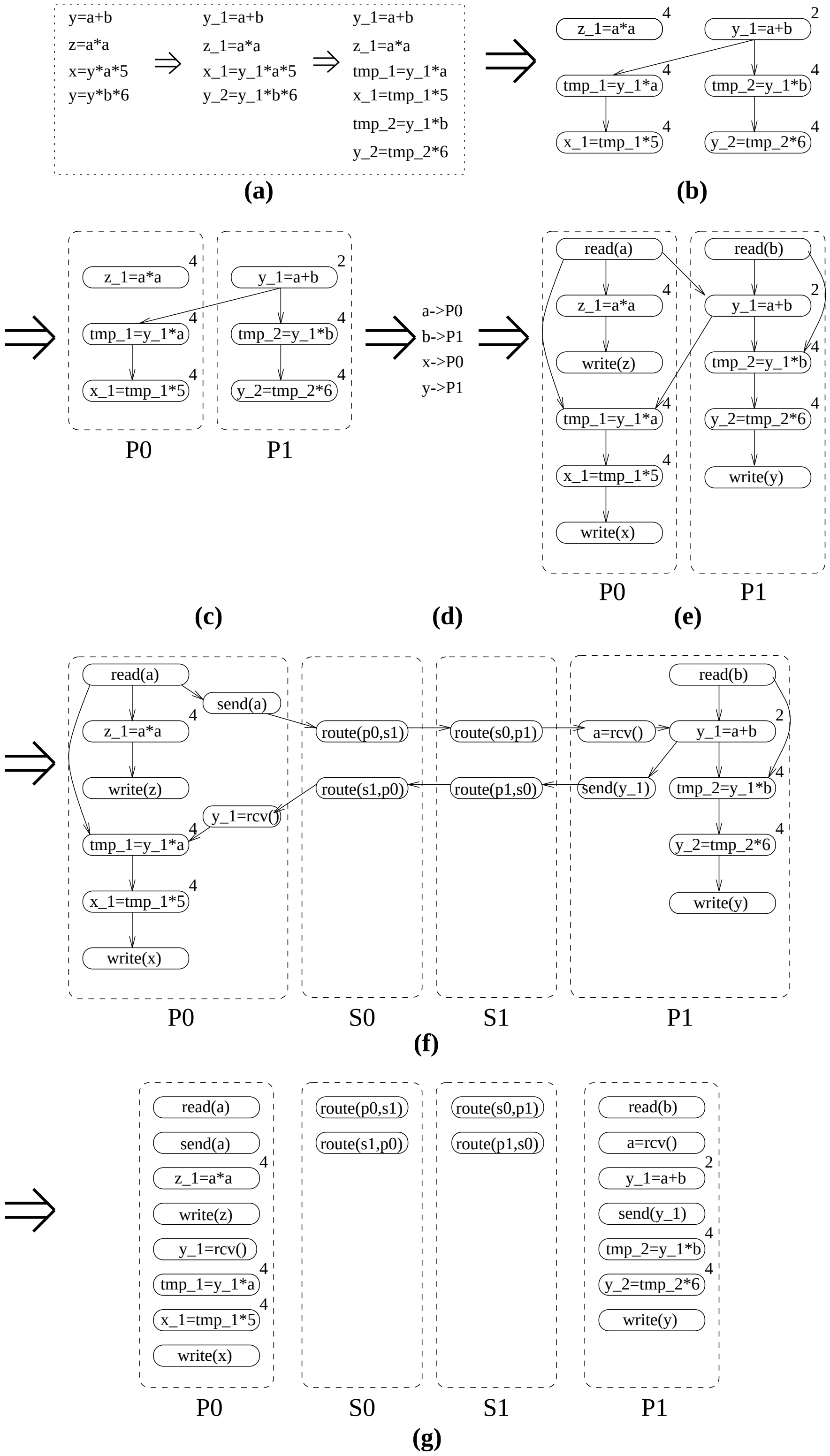

Figure 6: An example of the program transformations performed by RAWCC . (a) shows the initial program undergoing transformations made by *initial code transformation*; (b) shows result of *task graph builder*; (c) shows result of *instruction partitioner*; (d) shows result of *data partitioner*; (e) shows result of *basic block stitcher*; (f) shows result of *communication code generator*; (g) shows final result from *event scheduler*.

In practice, stitch code is inserted into the basic blocks by creating dependency edges between dummy nodes representing program variables and instruction nodes that produce or consume those variables. The stitch code is then scheduled by the event scheduler like any other communication arising from the basic block body. As a result, the overhead of distinctive, synchronizing initialization and finalization phases is avoided. Figure 6e shows a sample output of the stitcher. The example assumes no optimization, so that variable values are communicated to and from their home locations. Other basic blocks not shown in the figure follow the same convention, so that the persistent value of each variable is always available at its home location at the beginning of each basic block.

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 instruction task graph into communication instructions which route the necessary data value from the source tile to the destination tile. Figure 6f shows an example of such transformation. To minimize the volume of communication, edges with the same source are serviced jointly by a single multicast operation. 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.

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 can be specified as a generalization of the traditional instruction scheduling problem.

The task of scheduling communication instructions carries with it the responsibility of ensuring the absence of deadlocks in the network. The Raw compiler satisfies this condition by scheduling the instructions on the same communication path all at once in an end-to-end fashion. By reserving the appropriate time slot at the node of each communication instruction, the compiler 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 should remain deadlock-free and correct even in the presence of dynamic events such as cache misses. Raw 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 effect the correctness of the schedule. Appendix A gives an informal proof of the static ordering property.

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 6g 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. Section 4.2 discusses the event scheduling problem in more detail.

4 Optimization problems

Ignoring the implementation details, parallelizing a basic block in a Raw machine is mainly a three-step process: instruction partitioning, communication code generation, and event scheduling. For a NURA machine with dynamic routing between register files, communication code generation would be rendered unnecessary, and the task simplifies to instruction partitioning and event scheduling.

The current compilation strategy assumes that network contention is low, so that the choice of message routes has negligible 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 and is completely mechanical. This section focuses on the remaining, more important optimization problems that are also those faced by the compiler of any NURA machines: instruction partitioning and event scheduling.

4.1 Instruction partitioning

The instruction partitioning problem in NURA machines mirrors that of the task partitioning problem for mapping a directed acyclic task graph onto MIMD machines. The Raw compiler leverages work in this domain [10][18].

The input to the task partitioning problem is a machine specification and a directed acyclic task graph. The machine specification specifies the number of tiles, the network configuration between the tiles, and the latency and bandwidth of the communication channels. For Raw, the network configuration is a mesh, with a latency of one cycle for every hop and bandwidth of one word per cycle. In the task graph, nodes represent tasks and edges represent communication between the tasks. Each node is labeled with its cost of computation; each edge is labeled with its communication volume, from which the cost of communication can be estimated. In Raw, every edge has a unit label, representing one word of transfer.

The MIMD task scheduling problem differs from the classical task scheduling problem in that the cost of non-local communication is non-zero. This difference leads to the interesting observation that even the problem of finding the optimal partition for infinite number of processors becomes non-trivial. A good partition must balance the benefits of parallelism against the overheads of communication. Driven by this motivation, RAWCC uses the following decomposition of the problem:

Clustering The purpose of clustering is to group together instructions that either have no parallelism, or whose parallelism is too small to exploit relative to the communication cost. Subsequent phases guarantee that instructions in the same cluster will be mapped to the same tile. The clustering technique approximates communication cost by assuming an idealized fully-connected switch with uniform latency.

RAWCC uses the Dominant Sequent Clustering heuristic [18], an efficient, one-pass greedy algorithm that works as follows. Initially, each instruction node belongs to a unit cluster. 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 length of the critical path. Estimation of the critical path length takes into account the cost of computation as well as communication. The algorithm completes when all nodes have been visited.

Merging This phase merges 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. Currently, RAWCC relies on clustering to reduce communication events; it uses only the load balance heuristics during merging.

RAWCC generates load balanced partitions as follows. It 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 the smallest number of instructions.

Placement The placement phase maps the merged clusters to physical tiles, and removes the assumption of the idealized interconnect and takes into account the non-uniform network latency. In this phase, RAWCC tries to minimize 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. This greedy algorithm can be replaced by one with simulated annealing for better performance.

4.2 Event scheduling

In the event scheduling problem, the inputs are a set of computation and communication instructions, a directed acyclic graph representing the dependencies between the instructions, a machine dimension specification, and the mapping from instruction to either processor or switch. The phase outputs a schedule of the constituent instructions for each tile and each switch, optimized for parallel run-time. In other words, the scheduling problem consists of placing instructions into a matrix, with processor/switch numbers on one axis and time on another.

From one perspective, the event scheduling problem in Raw is a generalized instruction scheduling problem. However, keeping track of the ready status of a communication instruction is tricky and requires explicit management of communication buffers. Instead, we treat a single-source, multiple-destination communication path as an atomic event to be scheduled. 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 reserves the corresponding channel resources needed to ensure that the instruction can eventually make progress.

In a NURA machine with dynamic routing, the compiler schedules computation but not communication. The event scheduling problem then maps directly to the traditional MIMD task scheduling problem. In Raw, by considering a communication path as a single task to be scheduled, the RAWCC scheduling problem becomes a composition of MIMD processor's task scheduling problem [17] and VirtualWires' communication scheduling problem [11]. We employ a hybrid solution leveraging research in both areas. We propose a greedy task scheduling algorithm that uses a priority scheme to select tasks to be scheduled, where a task is either a computation instruction or a communication path. A communication path is scheduled as a unit at the earliest time slot when it can proceed without intermediate stalls.

The greedy algorithm keeps track of a ready list of tasks. As long as the list is not empty, it selects and schedules the task on the ready list with the highest priority. At the heart of the algorithm is the priority scheme that determines the order in which the ready tasks should be processed. This priority scheme is derived as follows. Clearly, tasks whose scheduling contribute most toward minimizing the length of the schedule should have the highest priorities. On the other hand, minimizing the schedule length is equivalent to maximizing the efficiency of the processors. To maximize efficiency, two types of tasks should get high scheduling priorities. First, tasks on the critical path should get high priorities to reduce the length of time during which processors are to be kept busy. The reduction in time in turn reduces the amount of work needed to keep the processors busy. Second, tasks with a large number of descendent tasks should get high priorities in order to expose as much parallelism as early as possible. To capture the two effects, we define the priority of a task as a weighted sum of two quantities: its *level*, defined to be the maximum distance between the task and any of its exit nodes; and its *fertility*, defined to be the number of descendent tasks.

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 are 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 6 illustrate this problem. We intend to explore the interaction between event scheduling and register allocation in the future.

5 Memory model

In the spirit of the Raw philosophy of keeping resources decentralized and therefore scalable, Raw distributes its memory across the tiles. Distributed memory allows parallel access to memory and makes the on-chip cache and memory bandwidth scale automatically with the number of tiles.[3] A distributed memory model has major implications on the compilation of sequential programs. The Raw compiler has to partition program data, and it must be able to support arbitrary memory operations, even those that cannot be statically analyzed. This section discusses the issues and the solution employed by the Raw compiler.

5.1 Overview

The RAWCC approach to the distributed memory problem is to provide fast access of memory references that are statically analyzable, and to provide adequate mechanisms for statically unanalyzable ones.[4] The compiler classifies a memory reference based on two attributes: whether the location (tile number) of the data is a statically known constant, and whether its dependence relation with other memory references can be statically determined. Knowledge about the location of the data determines whether the reference can be fetched using the static network; knowledge about its dependence relation determines how accurately the instruction task graph can be constructed.

Data location A memory reference whose data location is known at compile-time can be seamlessly handled under the Raw compilation framework. The compiler simply assigns the memory reference instruction to its known home tile. Once the instruction is appropriately placed, any necessary communication to propagate values needed or supplied by the memory instruction will be automatically generated in the same way as communication generated for other instructions. When the data location is known,

[3] Although bandwidth to external memory is still limited by the number of pins, a large amount of on-chip memory mitigates this problem.

[4] We use “memory references” to refer to both loads and stores. To keep the discussion concrete and avoid excessive verbiage, we describe only the details for loads whenever convenient.

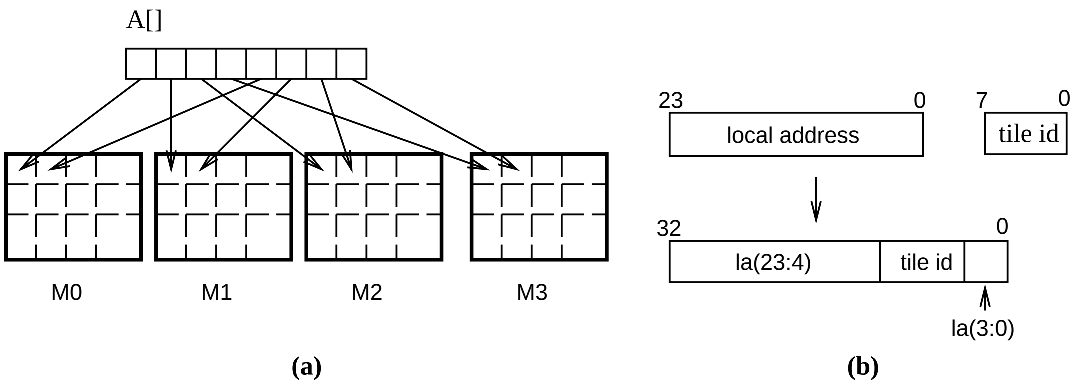

Figure 7: Low order interleaving. (a) An array object is interleaved element-wise across four memories. (b) A 24 bit local address combines with with an 8 bit tile id to form an address to a low-order interleaved memory. The interleaving granularity determines the point at which the symbolic address is split. $(sizeof(low\_order\_chunk) = log_2(16))$

a remote memory reference turns into a local memory reference followed by a send through the static network.

A memory reference whose data location is not known at compile-time is handled by the dynamic network. The run-time address for this reference contains both its home tile number and the local address within the tile. To service the reference, a dynamic message containing the local address is sent to the tile identified by the run-time address. A remote-memory message handler at the home tile receives the local address, performs the memory operation, and sends back the requested data.

Dependence relations Knowing full dependence relations between a memory reference and other memory references in the same basic block gives the compiler maximal freedom to reorder the reference with respect to other references without changing the behavior of the program. If the dependence information is incomplete, the compiler has two choices. It can either make conservative assumptions that ensure correctness but may be over-restrictive about memory reordering, or it can aggressively reorder the references, relying on the run-time system to detect speculation failure and jump to the appropriate patch-up code. The Raw compiler currently takes the former approach, but it will adapt the latter approach in the future.

5.2 Low-order interleaving

Exploitation of fine-grained parallelism requires parallel access of memory at a fine granularity. Raw provides this access by using *low-order interleaving* of memory. Low-order interleaving distributes small logically contiguous blocks of memory across the tiles. Figure 7 shows a low-order interleaved object and the bit layout of a low-order interleaved address.

For objects whose access patterns are regular and statically analyzable, the compiler can tailor the layout of the object, and thus low-order interleaving is not pertinent. Many objects, however, have access patterns that are either irregular or unanalyzable. For these objects, low-order interleaving is a best-effort attempt at achieving parallelism in the absence of any specific information. It caters to the spatial locality of memory accesses in programs.

5.3 Staticizing affine function accesses of arrays in loops

For obvious reasons, loops consume a large fraction of the runtime in many programs. Performance of programs can greatly improve if every memory reference inside a loop can be communicated via the static network. The basic requirement is the *static reference property*, which states that every memory access in the final code sequence of the loop must have the same home tile every time it is executed. This property usually does not hold for even the simplest of cases. For example, consider the following:

```
For (i=0; i<100; i++)
   A[i] = i;
EndFor
```

If the first 100 elements of $A[]$ do not reside on the same tile, the reference $A[i]$ will not satisfy this property.

Assuming that arrays are low-order interleaved element-wise, we have devised a technique to transform a common class of loops to a form that satisfies the static reference requirement. The technique applies to FOR loops with array accesses whose indices are arbitrary affine functions of the loop induction variables. The key observation is that for affine function accesses, the home location of the element accessed by a memory-referencing instruction follows a repetitive pattern across iterations. This repetitive pattern is compile-time determinable. For example, consider the loop below:

```
For (i=0; i<100; i++)
   A[i] = A[2i];
EndFor
```

Given low-order interleaving over four tiles, consecutive accesses of A[i] produce the tile numbers [0, 1, 2, 3, 0, 1, 2, 3 ...], yielding a repetition distance of 4. For A[2i], the pattern is [0, 2, 0, 2 ...], with a repetition distance of two. Similar patterns can be found for any affine access. If the loop is unrolled by a factor equal to the least common multiple of the repetitive distances of every memory operation in the loop, then every access through a given memory operation in the unrolled code will have the same home processor.

Through analysis, we have obtained symbolic expressions for the unroll factor required in the case of a k-dimensional loop nest with d-dimensional array accesses. The derivation is beyond the scope of this paper; we summarize the major results instead. The unroll factor per loop dimension is always at most N, the number of tiles on the machine. Hence, the overall code expansion for a k-dimensional loop is at most a factor of N^k. For simple affine index functions of the form $\pm i \pm constant$, however, the overall code expansion is at most N irrespective of k. This rule of thumb covers common cases such as A[i,j], A[j,i] and A[100-i,j+2].

Our technique works even in the presence of irregular control flow and is fully integrated into the control-flow handling mechanism described in Section 3.2. Unknown loop bounds cause no difficulty either. For many programs, the compiler is able to identify FOR loops even when the input program contains while loops.

Benchmark	Source	Lang.	Lines of code	Array size	Seq. RT (cycles)	Description
life	Rawbench	C	118	32×32	1.08M	Conway's Game of Life
vpenta	Nasa7:Spec92	Fortran	157	32×32	2.56M	Inverts 3 Pentadiagonals Simultaneously
cholesky	Nasa7:Spec92	Fortran	126	$3\times15\times15$	1.79M	Cholesky Decomposition/Substitution
tomcatv	Spec92	Fortran	254	32×32	214M	Mesh Generation with Thompson's Solver
fpppp-kernel	Spec92	Fortran	735	-	8.98K	Electron Interval Derivatives
mxm	Nasa7:Spec92	Fortran	64	32×64, 64×8	5.98M	Matrix Multiplication
jacobi	Rawbench	C	59	32×32	0.17M	Jacobi Relaxation

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

Benchmark	N=1	N=2	N=4	N=8	N=16	N=32
life	0.91	1.2	1.6	1.8	1.9	*
vpenta	0.92	1.2	1.8	2.2	2.6	3.0
cholesky	0.90	1.3	2.1	3.3	5.3	*
tomcatv	0.97	1.7	2.7	3.8	5.6	7.8
fpppp-kernel	0.51	0.92	1.9	4.0	8.1	13.7
mxm	0.92	1.8	3.3	6.3	10.2	*
jacobi	0.97	1.6	3.4	5.6	15	22

Table 3: 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.

6 Results

This section presents some performance results of the Raw compiler. We show that Raw can achieve good speedups for programs with basic blocks containing *either* regular or irregular parallelism. We use selective Spec92 benchmarks as well as programs from the Raw benchmark suite [2]. Experiments are performed on the Raw simulator, which simulates the Raw prototype described in Subsection 3.1. The simulator also assumes that every memory reference into local memory is a cache hit with a two-cycle latency.

The benchmarks we select include programs from the Raw benchmark suite, 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 does not support double-precision floating point, all floating point operations in the original benchmarks are converted to single precision. Table 2 gives some basic characteristics of the benchmarks.

We compare results of the Raw compiler with the results of a basic Mips compiler provided by Machsuif [12]. Table 3 shows the speedups attained by the benchmarks for Raw machines of various sizes. The results show that the Raw compiler is able to exploit ILP profitably across the Raw tiles for all the benchmarks. Speedup figures range from a low 1.9 on 16 tiles for life to a decent 22 on 32 tiles for jacobi. The results can be classified into three categories:

Low speedup Life and vpenta have low speedups caused by the same underlying problem: loop unrolling is unable to create large basic blocks. Larger basic blocks expose more parallelism to the Raw compiler, and they reduce the communication overhead between basic blocks. The failure to create large basic blocks has two causes. First, although unrolling removes control flow at the boundaries of the unrolled loop, it is unable to remove any control flow inside the body of the loop. (This is the reason life is unable to achieve the same levels of massive speedups as in our early multi-FPGA system [15].) Second, the affine function theory discussed in Section 5.3 sometimes requires unrolling the outer loop to meet the static reference property. Unrolling the outer loop replicates the inner loop, but it does not create larger basic blocks.

A solution to both of these problems is an optimization that allows higher-level control flow structures to be treated as schedulable macro-instructions by the basic block orchestrater. Such an optimization allows basic blocks to extend across control flow statements. We plan to implement this optimization in the near future.

Modest speedup Cholesky and tomcatv have modest speedups. These applications, along with mxm and vpenta, have traditionally been parallelized for multiprocessors by distributing loop iterations with no loop-carried dependence across the processors. Raw detects the same parallelism by partially unrolling the loop and distributing individual instructions across the tiles. A mature and fully cooperative instruction and data partitioner, along with the macro-instructions optimization described above, would be able to perform at least as well as the traditional parallelization method. On the other hand, Raw can be viewed as a multiprocessor with extremely low communication latency. As such, it can profitably exploit parallelism at a much finer grain and for much smaller problem sizes than traditional multiprocessors. This advantage of the Raw architecture is demonstrated by the results for cholesky and mxm from the nasa7 benchmark, both of which have relatively small problem sizes. Traditional multiprocessors require larger problem sizes to amortize their overheads; they are unable to exploit parallelism of such granularity [1].

Good speedup Fpppp-kernel, mxm and jacobi attain good speedups. They are applications with large basic blocks and ample instruction-level parallelism. These results highlight the exciting prospects of a processor whose hardware can absorb as much parallelism as an application can produce.

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 our current compiler attempts to expose the maximal amount of parallelism without regard to register pressure. As the number of tiles increase, 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.

The fpppp-kernel also demonstrates an interesting aspect of the Raw compilation technique because it contains large amount of irregular, ILP-type parallelism that the compiler can gainfully exploit. It is an application for which speedup have traditionally been difficult to attain. Superscalars perform poorly on such an application because they lack enough registers. Multiprocessors cannot attain speedup because the application does not contain loop level parallelism [1]. In contract, the Raw compiler is able to attain performance improvement that scales well for up to 32 tiles.

Like a VLIW compiler, the Raw compiler expects to use trace scheduling to obtain large basic blocks with significant ILP parallelism. Fpppp-kernel is an application with a large basic block with such

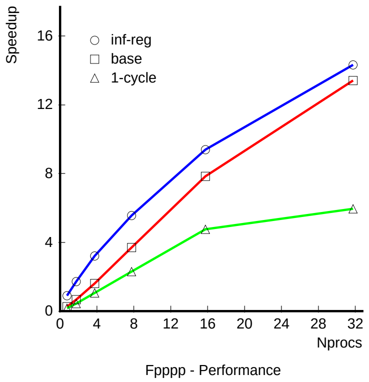

Figure 8: **Performance of fpppp-kernel under various machine configurations.** *Base* is the baseline machine; *inf-reg* is a machine with infinite number of registers per tile; *1-cycle* is a machine with single-cycle instructions. The cycle count for baseline number of *base* and *inf-reg* is 7478, and the cycle count of the baseline number of *1-cycle* is 3998.

irregular parallelism. Therefore, we use it to better understand the performance benefits derived from Raw for this type of applications. Figure 8 shows the performance of fpppp-kernel under various machine configurations. The plot contains speedup curves for the baseline machine with 32 registers per tile and instruction latencies from Table 1 (*base*), a machine with infinite registers (*inf-reg*), and a machine with single-cycle instructions (*1-cycle*). *Inf-reg* shows the potential performance improvement Raw can attain if it can reduce its problem with register spills. *1-cycle* reduces the computation to communication ratio compared to *base*, and thus increases the relative weight of communication overhead on performance. Therefore, the curve for this processor is a lower bound on the performance of fpppp-kernel. Although the speedup for *1-cycle* is lower than that for *base* (13.7 versus 6.2 on 32 tiles), performance scales reasonably for up to 32 processors.

7 Future Work

The Raw compiler is a fertile ground for future research. The most immediate extension to RAWCC is trace scheduling, a technique used by VLIW compilers to merge basic blocks in commonly executed program paths to produce large basic blocks [4]. Because RAWCC exploits basic block parallelism, larger basic blocks can dramatically improve performance.

Many traditional compiler research problems have their variants in RAWCC . RAWCC faces several interface issues between related compiler phases. As in a traditional uniprocessor, RAWCC has to solve the classic interference problem between instruction scheduling and register allocation. The Raw compiler also needs to reconcile instruction partitioning with data partitioning – a problem shared with multiprocessors. In order for the latter problem to be resolved satisfactorily, instruction partitioning between basic blocks must also cooperate to preserve data locality.

The Raw compiler can perform more powerful static analysis if supplemented with run-time information generated by a software run-time system. As in the Multiscalar [13], RAWCC can aggressively speculate on memory operations, relying on the run-time software system to detect speculation failure and jump to the appropriate patch-up code. Raw can also deploy a fast code partitioner and scheduler in the run-time system to generate instruction streams for run-time traces, similar to the DIF-cache technique of dynamic code generation for VLIW machines [9] and the instruction-trace technique in [14].

8 Conclusion

This paper presents the Raw architecture and compiler system. Together, they exploit instruction-level parallelism in applications to achieve high levels of performance by physically distributing resources and exposing them fully. The resulting non-uniform access times creates many challenging compiler problems. Physically distributed register files result in non-uniform register access times causing instruction scheduling to become a problem in both space and time. Non-uniform memory access times further requires the compiler to partition data as well as instructions. The ability to use the fast static network to communicate remote memory references in Raw adds yet another challenge: identifying memory operations with static home locations.

This paper describes the Raw compiler's approach to handling the above problems. First, the Raw compiler detects and orchestrates instruction level parallelism within basic blocks by adapting techniques from MIMD task partitioning and scheduling, as well as from VirtualWires communication scheduling. Second, it uses low order interleaving as the default memory allocation policy to exploit fine-grain parallelism. Finally, it introduces a technique to transform loops that access arrays using affine functions of loop induction variables into loops that execute with only static communication.

A Raw compiler based on SUIF has been implemented. The compiler accepts sequential C or Fortran programs, discovers instruction-level parallelism, and schedules the parallelism across the Raw substrate. Reflecting our ILP focus, the compiler currently does not exploit coarse-grain parallelism through SUIF's loop-level parallelization passes, although it can be easily extended to do so. This paper presents some promising results on the potential of the Raw compiler. The compiler is capable of realizing 14-way parallelism on 32 tiles for the fpppp-kernel, an irregular application that has historically resisted attempts at speeding it up. Since a Raw microprocessor is already suitable for many other forms of parallelism, including coarse-grain loop level parallelism, stream processing, and static pipelines, its ability to exploit ILP is an important step in demonstrating that the Raw machine is an all-purpose parallel machine.

References

[1] 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.

[2] 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.

[3] 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.

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

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

[6] B. Gieseke and et al. A 600MHz Superscalar RISC Microprocessor with Out-Of-Order Execution. *1997 IEEE International Solid-State Circults Conference. Digest of Technical Papers*, pages 176–178, 1997.

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

[8] W. mei 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.

[9] R. Nair and M. E. Hopkins. Exploiting Instruction Level Parallelism in Processors by Caching Scheduled Groups. In *Proceedings of the 24nd Annual International Symposium on Computer Architecture*, pages 13–25, 1997.

[10] 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.

[11] C. Selvidge, A. Agarwal, M. Dahl, and J. Babb. TIERS: Topology independent pipelined routing and scheduling for VirtualWire compilation. In *1995 ACM International Workshop on Field-Programmable Gate Arrays*, pages 12–14, Berkeley, CA, February 1995. ACM.

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

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

[14] S. Vajapeyam and T. Mitra. Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences. In *Proceedings of the 24nd Annual International Symposium on Computer Architecture*, pages 1–12, 1997.

[15] 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.

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

[17] T. Yang and A. Gerasoulis. List scheduling with and without communication. *Parallel Computing Journal*, 19:1321–1344, 1993.

[18] 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.

A Static Ordering Property

Dynamic events such as cache misses prevent one from statically analyzing the precise timing of a schedule. The Raw compiler relies on the static ordering property of the Raw architecture to generate correct code in the presence these dynamic events. The static ordering property states that the result produced by a static schedule is independent of the specific timing of the execution. Moreover, it states that whether a schedule deadlocks is a timing independent property as well. Either the schedule always deadlocks, or it never does.

To generate a correct instruction schedule, Raw orders the instructions in a way that obeys the instruction dependencies of the program. In addition, it ensures that the schedule is deadlock free assuming one set of instruction timings. Static ordering property then ensures that the schedule is deadlock free and correct for any execution of the schedule.

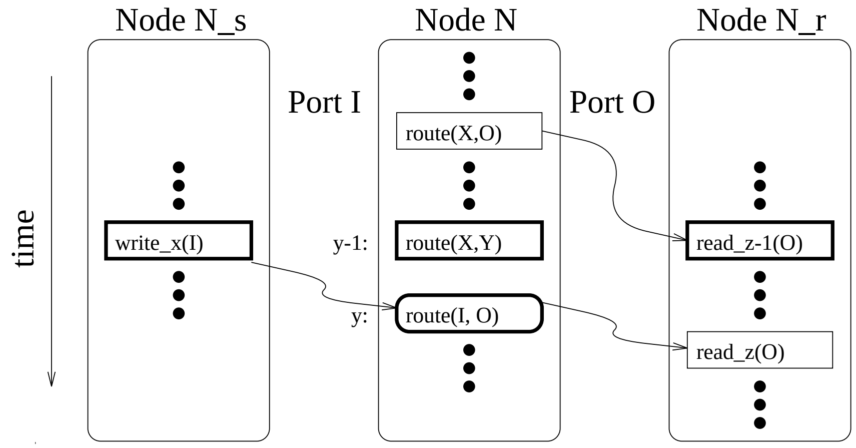

Figure 9: **Dependent instructions of a communication instruction.** Each long rectangle represents an execution node, and each wide rectangle represents an instruction. Spaces between execution nodes are ports. Edges represent flow of data. The focal instruction is the thick rounded rectangle. Its dependent instructions are in thick regular rectangles.

We provide an informal proof of the static ordering property. We restrict the static ordering property to the practical case: given a schedule that is deadlock free for one set of instruction timings, then for any set of instruction timings,

1. it is deadlock free.
2. it generates the same results.

First, we show (1). A deadlock occurs when at least one instruction stream on either the processor or the switch has unexecuted instructions, but no instruction stream can make progress. A non-empty instruction stream, in turn, can fail to make progress if it is attempting to execute a blocked communication instruction. A communication instruction blocks when either its input port is empty, or its output port is full. Computation instructions do not use communication ports; they cannot cause deadlocks and are only relevant in this discussion for the timing information they represent.

Consider a communication instruction c. We derive the conditions under which it can execute. Three resources must be available: its input value, its execution node (processor or switch), and its output ports.[5] The resource requirements can also be represented by execution of a set of instructions. First, note that ports are dedicated connections between two fixed nodes, so that each port has exactly one reader node and one writer node. Let instruction c be the x^{th} instruction that reads its input port I, the y^{th} instruction that executes on its node N, and the z^{th} instruction that writes its output port O. Then the resources for instruction c become available after the following instructions have executed:

1. the x^{th} instruction that writes port I.
2. the $y - 1^{th}$ instruction that executes on node N.
3. the $z - 1^{th}$ instruction that reads (and *flushes*) port O.

See Figure 9.

The key observation is that once a resource becomes available for instruction c, it will *forever* remain

[5] The three resources need not all be applicable. A SEND instruction only requires an output port and a node, while a RECEIVE instruction only requires the input value and a node.

available until the instruction has executed. The value on the input port cannot disappear; the execution node cannot skip over c to run other instructions; the output port cannot be full after the previous value has been flushed. The reservation of the resources is based on three properties: the single-reader/single-writer property of the ports, the blocking semantics of the communication instructions, and the in-order execution of instructions.

Therefore, a communication instruction can execute whenever its dependent instructions, defined by the enumeration above, have executed.

Now, consider the schedule that is deadlock-free for one known set of timings. Plot the execution trace for this set of timings in a two dimensional grid, with node-id on the x-axis and time on the y-axis. Each slot in the execution trace contains the instruction (if any) that is executed for the specified node at the specified time. The plot is similar to Figure 9, except that real execution times, rather than the static schedule orders, are used to place the instructions temporally.

Finally, consider a different set of timings for the same schedule. Let t_{new} be a point in time for the new timings when the schedule has not been completed, and let $E_{new}(t_{new})$ be the set of instructions that have executed before time t_{new}. We use the above deadlock-free execution trace to find a runnable instruction at time t_{new}. Find the smallest time t in the deadlock-free execution trace that contains an instruction not in $E_{new}(t_{new})$. Call the instruction c. The dependent instructions of c must necessarily be contained in $E_{new}(t_{new})$.[6] Therefore, c must be be runnable at time t_{new} for the new set of timings.

Having found a runnable instruction for any point in time when the schedule is not completed, the schedule must always make progress, and it will not deadlock.

The second correctness condition, that a deadlock-free schedule generates the same results under two different sets of timings, is relatively easy to demonstrate. Changes in timings do not affect the order in which instructions are executed on the same node, nor do they change the order in which values are injected or consumed at individual ports. The blocking semantics of communication instructions ensures that no instruction dependence can be violated due to a timing skew between the sender and the receiver. Therefore, the values produced by two different timings must be the same.

[6] This statement derives from two facts:

1. All dependent instructions of c must execute before c in the deadlock-free execution trace.

2. Since c executes at time t and all instructions executed before time t are in $E_{new}(t_{new})$, all instructions executed before c in the deadlock-free execution trace are in $E_{new}(t_{new})$.