How to build scalable on-chip ILP networks for a decentralized architecture

Abstract

The era of billion transistors-on-a-chip is creating a completely different set of design constraints, forcing radically new microprocessor architecture designs. This paper examines a few of the possible microarchitectures that are capable of obtaining scalable ILP performance. First, we observe that the network that interconnects the processing elements is the critical design point in the microarchitecture. Next, we characterize four fundamental properties that have to be satisfied by the interconnection network. Next, we provide case studies of two different networks that satisfy these properties. Finally, a detailed evaluation of these networks is presented to highlight the scalability and performance of these microarchitectures. We show that by using compile time information, we can build simpler networks and use them efficiently.

1 Introduction

Microprocessor design has become increasingly marked by the desire to leverage increased transistor budgets, reduce verification and design costs, and to address the crumbling of electrical abstractions. A number of contemporary architecture research projects, including IBM's Blue Gene, Raw [11], SCALE, and Smart Memories [8], have addressed these desires. All of these projects exchange the monolithic VLIW or superscalar processor for a distributed, decentralized array of independent, replicated computing elements. These elements are connected by a network with short, local wires that connect only near neighbors. We refer to these architectures as "decentralized architectures." Figure 1 depicts a decentralized architecture.

A key feature of these architectures is that they have properties that scale very well asymptotically with respect to the number of transistors. The Blue Gene project proposes an architecture with literally a million processing elements. Architectures on such a scale will become increasingly commonplace in the near future. Although asymptotic analysis has traditionally been inappropriate for microprocessors, where constants tend to dominate[1], it will become an increasingly valuable tool as we see larger and larger designs.

Decentralized architectures have a number of asympotic advantages:

First, the micro-architectural design and verification cost of these decentralized architectures is essentially constant with respect to the number of transistors. When the designers gain access to more transistors, they simply put down more computing elements.

Second, the wire length of these designs is also constant in size with respect to the number of transistors. As a result, the wire delay and routing congestion of the design, a major problem for monolithic

[1] A factor of two in processor means a difference in billions of dollars of revenue for AMD!

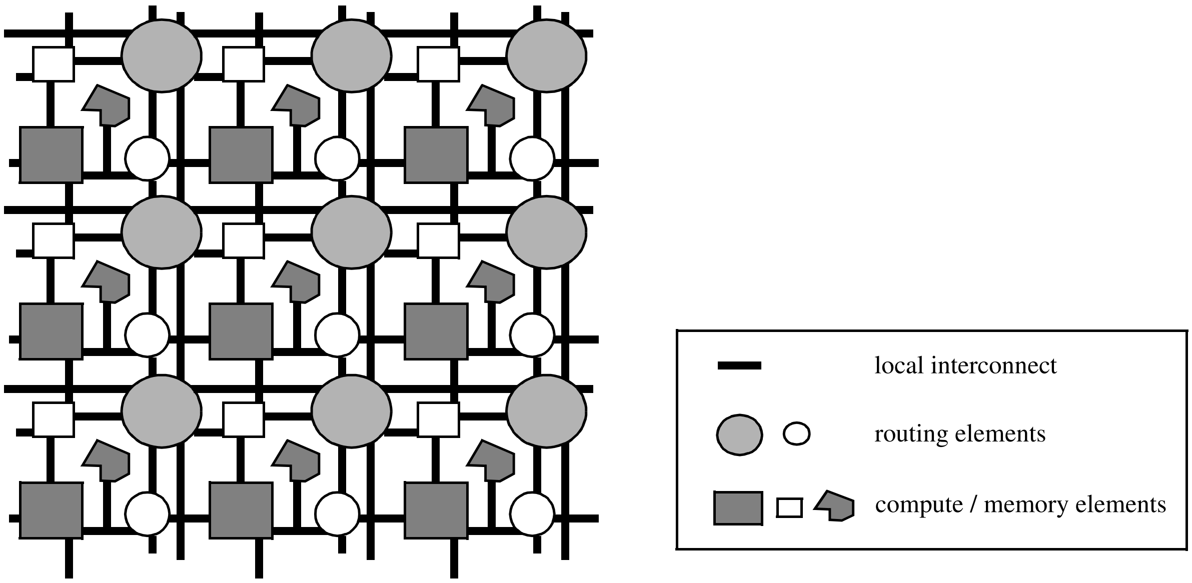

Figure 1: A decentralized architecture

architectures, does not changes with the number of transistors.

Third, decentralized architectures also reduce clock complexity relative to transistor budget. In the first method, we relax the assumption of a global synchronous, skewless clock, and have the computing elements perform the appropriate handshaking when they exchange data. Alternatively, there is promising work [9] in online clock tuning algorithms that can correct for skews in distributed clocks. At the very least, decentralized architectures offer very precise, geometric clock boundaries rather than a large three-dimensional rat's nest of wires and transistors.

Finally, these architectures may offer constant yields per unit silicon with respect to transistor count. Processing elements with point defects could be left unused by the software. However, for particularly nasty cases, this problem may require that a laser be used to isolate the computing element electrically from its neighbors. The components can then be binned for number of inactive processing elements, just as they are for clock speed today.

Now that we have motivated decentralized architectures, we will describe what it is we are going to say about how they are designed.

Models of Parallelism Any of the proposed architectures need to take advantage of parallelism in order to scale. The microarchitecture can target different parallelism models such as instruction level parallelsim, pipeline parallelism, thread parallelism, or coarse-grained parallelism. The advantage of instruction level parallelism(ILP) is that it still gives good performance for both pipeline and course-grained parallelism, since these forms of parallelism can be converted into ILP.

In our previous work, we have develop a compiler system that is capable of obtaining scalable ILP performance by performing both spatial instruciton partitioning as well as traditional temporal insturction scheduling[7]. This compiler demonstrate that for a large class of applications, scalable ILP performence is obtainable.

Manifest

There are two key parts to designing a decentralized architecture: the network and the computing element. This paper focuses on the network.

The computing elements are the part of the architecture that are responsible for performing actual computation. These elements can look like in-order processors, small-scale out-of-order processors, FPGA logic, clusters of ALUs, or even single functional units.

The "network" is the entire system that is responsible for routing values in a correct fashion (i.e., respecting program dependences) between the computing elements. In a traditional 5-stage pipeline, the "network" would consist of the bypassing logic and the bypass paths.

Please note that when we refer to an ILP network, we are not strictly implying a monolithic piece of hardware. As RISC principles and principals have taught us, the dividing line between hardware and software is very flexible. Some parts of the ILP network may in fact be implemented in software and with compiler information (as we will demonstrate in the data section of this paper.)

We further show that the compiler is often essential in reducing the cost or improve the performance of these ILP networks.

This paper starts by describing the communication patterns used by programs with general ILP. It continues by proposing a set of fundamental properties that a network needs in order to facilitate ILP, independent of whether the computing elements are FPGA logic, a cluster of ALUs, smart memories, or processors. It narrows its focus on a computation model where the computation on each computation element is totally ordered, and it studies two ILP network implementations in this domain, with an emphasis on how they fulfill our fundamental network properties. The paper continues by giving performance results for the two networks on programs compiled by a parallelizing C/Fortran compiler, and for some hand-coded applications. It finalizes by characterizing the usage of our ILP network design, and it suggests some improvements.

This paper makes three contributions. First, it identifies the fundamental functional requirements for building a scalable ILP network for a decentralized architecture. Second, it presents two implementations of ILP networks. Finally, it offers some insights into how an ILP network can be implemented efficiently.

The rest of this paper is organized as follows. Section 2 characterizes the ILP computation and the communication that arises from it. Section 3 describes the fundamental requirements of an ILP network. Section 4 describes two ILP networks. Section 5 shows the performance of the ILP networks. Section 6 concludes.

2 ILP Computation

This section characterizes the the nature of computations with instruction level parallelism.

The computations are typically expressed as a dataflow graph. A dataflow graph is a logical network, which consists of nodes and arcs. The nodes represent the operations in the dataflow graph, and the arcs represent data values flowing from the output of one operation to the input of the next. The existence of an arc between operations implicates a sequential ordering between the execution of the two operations.

Memory operations fall into a special case. In this case, if the compiler cannot determine that the memory operations will not conflict, then there are probabilistic read-after-write and write-after-write dependences existing between the operations. Short of using a speculation (optimistic concurrency control) scheme, the application must sequentialize these operations.

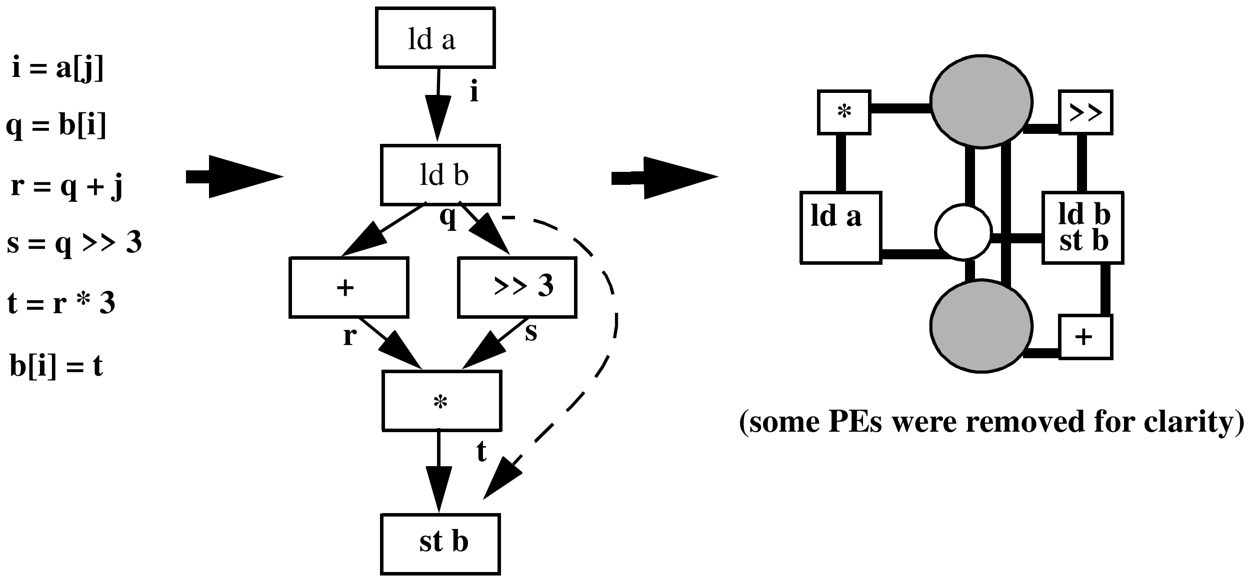

Figure 2: Dataflow Graphs and Operation Assignment

In Figure 2, the memory accesses to b[i] have a possible dependence, which creates a non-deterministic dependence between the nodes. The computation is almost entirely sequential; only the add and shift operations can be performed in parallel.

Typically, these dataflow graphs are enclosed in some sort of control structure: if-statements, loops, etc. ILP compilers increase the amount of parallelism by playing tricks with looping structures to enlarge the size of the dataflow graph and find more things that can execute in parallel. The two most common techniques are loop unrolling [3, 4] and pipelining [6].

To execute a computation on a network of processing elements (PEs), one needs to find an assignment from the nodes of the dataflow graph to the nodes of the network of processing elements, and route the intermediate values between these PEs. This assignment of operations can be performed at run-time or compile-time. Superscalar and early dynamic dataflow [1] are examples of run-time assignment architectures, while TTA [5], VLIW, or Raw [11] are examples of compile-time assignment architectures. If the architecture is a compile-time assignment architecture, the choice of the path that the values take between the PEs can be done either at compile-time or run-time. TTA architectures and Raw choose the routing path at compile-time, while VLIW architectures typically do it at runtime. In Section 4 we will give two examples of systems where the assignment problem is handled at compile time.

The figure shows a sample assignment. Note that the load and store to the array b are located on the same node. This is by necessity, because they both need to access the same piece of memory.

There is a classic tension in the assignment problem. On one hand, we want to spread the computation as far out into space as possible to maximize the amount of PEs that can be used simultaneously (and thus maximize the parallelism). On the other hand, we do not want to have operations performed too far away, because the travel time over the networks will add up and impact the serial performance. For instance, in the diagram, if it takes a cycle to traverse a network link, and the ops all only took one cycle, then it would have been more effective to allocate all of the operations to one processing element (assuming that processing element supported this.)

With an intuition of the structure of ILP programs, and how they are mapped to processing elements, we can now discuss the structure of the ILP networks.

3 ILP Network

When the original ILP networks (i.e. the bypass networks) for VLIW and Superscalar processors were created, the functionality was so implicitly encoded in the structure of the networks that it was difficult or perhaps overly obvious to even mention what the fundamental requirements of the networks were. As we stretch and scale and explore these networks, we find suddenly that design space is so murky that we have to define what exactly these sorts of networks need to be. Certain properties that were once trivial to attain suddenly become very difficult.

The initial ILP networks that we designed failed rather spectacularly because their scalable, decentralized implementations were not capable of providing us with properties that we took for granted in their less scalable cousins.

The properties which we discovered follow:

1. The ILP network must be implemented as a distributed process.

This means that there can be no global wires, global repositories for information, serial operations, unified look-up tables, logic which incorporates state from all of the processing elements, etc, etc, ad infinitum. The view that needs to be taken is that of a number of processing elements, running independent state-machines that are isolated from their non-neighbors. A time cost needs to be assigned for any transferrance of information between non-neighbors.

2. The ILP network must provide operation/operand matching.

This property highlights that fact that, in order to compute, we need the operands and the operation to meet at some point in space to perform the computation.

	Assignment	
Ordering	Run−time	Compile−time
Run−time	Superscalar	Monsoon
Compile−time		Raw, TTA, VLIW

Figure 3: Operation Ordering and Assignment

There is an interesting taxonomy in the way in which architectures perform operation/operand matching. There are two distinction: whether operations are assigned to PEs at compile-time or run-time, and whether the order in which operations on a given PE are executed is determined at compile-time or run-time. Figure 3 depicts shows a matrix which classifies several architectures in this manner.

In compile-time assignment architectures, the compiler assigns each operation to a processing element. In a traditional bus-based VLIW, values are stored in a central repository (the register file), and the instructions snoop the buses into this repository to make sure they have the latest versions to send to the operators.

When we move to a scaleable compile-time assignment architecture, we need to get rid of the central repository. The most effective method of doing this is to route the result of a given operation directly on to the next operation that uses that result. This efficient technique could not be performed if the operands had not been compile-time. The Raw architecture uses exactly this technique.

In run-time assignment architectures, both the operations and operands can move around in the architecture, so there has to be some sort of directory scheme which allows them to find each other. In a traditional superscalar, this directory scheme is implemented through global buses and reservation stations. Unfortunately, these approaches are not scaleable.

The early, seminal work on large-scale run-time assignment architectures is the static and dynamic dataflow machines. They discovered very quickly that the scaleability issues raised by this approach were very challenging. The solved these problems mostly with the use of pipelined sorting networks. One could also imagine using a home-location scheme, not unlike directory-based distributed shared memory protocols. These approaches were plagued by state-explosion (different parts of the program would execute far ahead of the others, filling up all of the machine's execution storage) and deadlock issues. Perhaps most importantly, the latency of operations were greatly multiplied, so serial performance suffered.

Run-time assignment is especially attractive because it effectively performs run-time load balancing of processing elements, which is very useful for adjusting to variable latencies in the computation.

Ultimately, some dataflow work switched to a hybrid assignment model, which had run-time assigned threads, composed of compile-time assigned operations. These threads are created and communication through explicit instructions, rather than as an automatic function of the ILP network. In a sense, they moved the run-time assignment part up to the compiler and os level, out of the hardware. The latency cost of this run-time thread assignment were mitigated because the communication at the thread level is more course.

In the Monsoon dataflow project, a given function could be dynamically assigned to a given PE, but all of the operations inside the function would be compile-time assigned to the same processing element. They credited this approach with providing them with improved locality.

In compile-time operation ordering, the compiler determines the order in which operations will execute on a given processing element. Both Raw and VLIWs employ compile-time operation ordering.

In run-time operation ordering, the ordering of instructions on a given processing element is determined by the arrival of the operands. Superscalars employ run-time operation ordering. Interestingly, Monsoon performs compile-time assignment but run-time ordering. Run-time ordering helps tolerate memory access latency and other events with unpredictable timing.

3. The ILP network must gaurantee finite arrival time.

This property highlights the fact that the ILP network has to gaurantee delivery of operands to processing elements so that the computation is robust. Currently, our architectures still have the luxury of assuming physically reliability (although that changes with the Blue Gene machine!). However, it has become a difficult question to determine if a ILP network is logically reliable.

The original ILP architectures used buses to communicate values between processing elements, and had a centralized way in which operands could be flowed controlled so that the buses would not be overcommitted. When we switch to a decentralized architecture, such global knowledge becomes impossible. As a result, controlling the occupancy of the network becomes very difficult. If the processing elements independently produce more values than the network has storage space, then either data-loss or deadlock must occur. This is a frightening prospect; something that the non-scaleable ILP network designers seldom had to worry about.

In particular, if dynamic networks are being employed in the chip, very careful measures have to be exerted in order to gaurantee that deadlock cannot happen, or that if it does happen, that it can be recovery from. The early dataflow machine papers demonstrated that they had tremondous problems with the overcommitment of storage and deadlock.

4. The ILP network must be able to tolerate timing variations.

This property recognizes the fact that computations often have timing characteristics that can be unpredictable at compile time. Their timing or cache behaviour may vary with the input dataset, or synchronization with an external device (like an interrupt) may change the timing behaviour of a given processing element. If the ILP network cannot tolerate timing variation, it will either fail or severely reduce the sorts of computations that can be performed.

In early machines, it was possible to run a global stall signal across the chip in order to keep processing elements synchronized when a timing variation event occurred. This is clearly not an alternative in a scaleable machine. As a result, a more distributed mechanism for handling timing variation needs to be developed.

Note that this timing variation, like all of the concepts in ILP networks, does not need to be handled strictly in hardware. The PEs, for instance, could handle timing variation through the use of periodic handshakes in software. Given this note, a number of research projects started with software timing variation tolerancy and moved it in to hardware for performance and system-complexity reasons. In particular, both Raw and Warp started with cycle-counted non-flow-controlled interconnects between processing elements and switched to distributed flow-controlled versions.

In the next section, we will describe our implementation of a pair of ILP networks, and show it meets the properties that we described in this section.

4 Two case studies

This section presents two case studies of ILP networks. The case studies are performed on the Raw microprocessor, a distributed microprocessor with compile-time operation ordering and assignment on each of its computing elements. The Raw microprocessor is an interesting platform for our purpose because it contains both a general dynamic network and a static network that provide register level communication. We consider how both networks can support ILP computation. For each network, we discuss how it supports variation tolerance, operation/operand matching, and reliable delivery.

4.1 Compilation framework

Compiler support for exploiting ILP on the Raw microprocessor is provided by Rawcc. Rawcc extracts ILP out of sequential programs and orchestrates it across the independent processing nodes of the Raw microprocessor. It performs loop unrolling and control flow localization to expand the scope in which it looks for parallelism [7]. For convenience, we refer to this scope as a *basic block*, even though it technically can span more than one basic block. Orchestration of parallelism is done in two phases:

instruction partitioning and cross-tile instruction scheduling. Instruction partitioning statically maps each instruction to a processing node. The compiler enables a memory instruction to be mapped to a single processing node by ensuring that the portion of memory it can potentially access is mapped onto a single processing node. [2]. Cross-tile instruction scheduling provides a total ordering of computations on each processing node while taking into account the effects of inter-node communication latencies.

4.2 Supporting ILP communication on the Raw dynamic network

The Raw dynamic network is a general dynamic network supporting multi-word messages. Each message on the dynamic network is preceded by a header that contains the length and destination of the message. Routing decisions are made as follows. When a message header arrives at a switch, the switch interprets it to determine how that message should be routed. The latency for routing a word usually one cycle per switch. Because of the interpretation cost, the message header incurs an extra cycle of latency when it makes a turn.

The dynamic switch employs dimension ordered routing to avoid network level deadlocks. Wormhole routing allows the head of a message to be routed to the next node before the end of a message has been received. Routing resources are allocated on a first-come, first-serve basis; if two messages arrive simultaneously, arbitration is resolved arbitrarily. The network has blocking semantics: a switch blocks and waits if it tries to route from an empty input port or route to a full output port. Messages can be received via either interrupt or polling.

The dynamic network provides natural support for variation tolerance, because messages can be arbitrarily delayed without effecting its routability. This flexibility, however, leads to uncertainty, which in turn complicates destination message ordering (the act of creating an ordering of values corresponding to the compile-time operation ordering) and reliable delivery. We discuss each issue in detail. Unless explicitly stated otherwise, the discussion applies whether messages are received via interrupt or polling.

Destination message ordering

A general way to handle potentially out of order messages in a totally ordered computation model is as follows. The network contains an associative buffer into which messages are received, which we term a *demultiplexing buffer*. Each element of the associative buffer has a full-empty bit. Every message in this system is tagged with an id. This id is known a priori to both the sender and the receiver. When a node receives a message, it uses the id to index into the associative buffer, store the message in the corresponding element, and set the corredponding full bit. It is an error to write to a full buffer element. The main computation process these messages in its static total order, resettting the full-empty bit whenever it no longer needs a message. This mechanism decouples message receive from message processing, allowing messages to arrive in a different order than the processor expects. We call this scheme the *buffer demultiplexing scheme*.

In the Raw dynamic protocol, both the associative buffer and its full-empty bits are implemented in memory.

We define several terms concerning the buffer demultiplexing scheme. A message is said to be *retired* when its buffer element can be reused; a buffer element is said to be *free* if all messages written to it has been retired. Message retirement needs not, and usually does not, correspond to the point when a message is originally received.

Normally, there are more messages in the programs than there are number of elements in the demultiplexing buffer. Thus the buffers need to be shared among many messages. This requirement leads to two issues: the assignment of messages to buffer elements and buffer reuse synchronization. The latter is needed to ensure that two messages destined for the same buffer element do not clobber each other.

Buffer assignment The primary goal in assigning messages to buffer elements is to minimize the amount of buffer reuse synchronization required. This goal, in turn, suggests that messages reusing the same buffer should be as far away as possible. Here is one possible assignment mechanism. The compiler assigns a distinct virtual buffer element to each message in a basic block. A message gets its virtual buffer element from those on its destination node, and the pools of virtual buffers on each node are distinct. Basic blocks that contain more virtual buffer elements than there are physical buffer elements need to be divided into smaller basic blocks before assignment.

From these virtual buffer assignments, physical buffers can be determined either statically or dynamically. In static allocation, each virtual buffer is simply mapped directly to a physical buffer. Dynamic allocation works analogous to the way stack allocation is performed. On each node, a buffer pointer keeps track of the first buffer element that has not been used. At the beginning of a basic block, the pointer is incremented by the maximum number of messages that can be sent to any one node in that basic block.[2] If the buffer has overflowed, all the nodes synchronize and the buffer pointer is reset and reincremented. Regardless, a physical buffer number is determined by subtracting the virtual buffer number from the buffer pointer.

The tradeoff between static and dynamic allocation are as follows. Dynamic allocation only performs buffer reuse synchronization when the demultiplexing buffer runs out of space, but it pays some overhead in buffer pointer manipulation and bookkeeping. Static allocation, on the other hand, avoids the overheads of the buffer pointer, but it must perform buffer reuse synchronization between every basic block. In general, dynamic allocation favors large demultiplexing buffers, while static allocation favors small ones.

Buffer reuse synchronization Before a sender can send a message, it needs to know that the receiver has retired all previous messages mapped to the same buffer element. In our environment, this condition can be enforced as follows. Let m_{t-1} and m_t be two consecutive messages that use the same id i during the execution of a program. For correctness, the protocol needs to guarantee that there exists a sequence of messages from the receiver to the sender that occurs between the retirement of m_{t-1} and the sending of message m_t.

It would be prohibitly expensive to explicitly synchronize each pair of consecutive messages designated for the same buffer element. Instead, a compiler can optimize the buffer reuse synchronization in several ways. Synchronization can be piggy-backed onto existing communication with zero overhead. When explicit synchronization is needed, selective placement of a single message may be able to satisfy the synchronization requirements of multiple message ids.

We propose two mechanisms that can be run at the beginning of a basic block to allow buffer reuse for either our static or dynamic allocation scheme. A simple mechanism is a barrier. This scheme would be sufficient for systems with efficient barrier support, or if there are enough buffers to tolerate infrequent synchronization.

If synchronization is frequent on a system without efficient barrier support, however, the following mechanism, which we term *explicit batch synchronization*, can be more efficient. In this scheme, a

[2]This pointer update mechanism loses some buffer space due to non-uniform distribution of messages across the destination nodes, but it keeps the overhead of the the pointer bookkeeping tolerable.

receive node reserves a buffer element with each potential sender for synchronization purposes. For each processing node s, the compiler examines the list of messages to be sent and received in the order they are scheduled. It keeps track of nodes from which s has received as well as the nodes to which s has sent. When it encounters a message m with source node s and destination node d, it checks whether there exists a preceding message between the two nodes, either form s to d or vice versa. If this message does not exist, a synchronization message is inserted from d to s before m. The synchronization messages inserted by this algorithm ensures that our condition for buffer reuse is satisfied. Note, however, that this algorithm scales quadratically with the number of pairs of communicating node, so that if many nodes communicate with each other, a software barrier could still be more efficient.

Optimizing the fast path with polling Buffer demultiplexing through memory has a large overhead and decreases the benefits of having a register mapped communication network. We can, however, design a fast path for message handling that writes the message value directly into a register. The idea is to take advantage of compile-time information to insert polling code at places where the compiler expects a message to be received. Included in the polling code is a fast path that is optimized for the message that the compiler predicts the receive node will receive. The fast path is inlined alongside the computation code so that it gets good register allocation and does not incur any procedural overhead. If the expected message is processed by its optimized polling code, its value is written directly into a register and the corresponding buffer element can be immediately retired.

Reliably delivery The Raw dynamic network does not drop messages. Therefore, reliable delivery reduces to the ability to handle deadlocks. Deadlocks occurs if all the receivers are blocked on message sends that cannot complete due to blockage in the network.

There are two general approaches to deal with deadlocks. Deadlock avoidance avoids deadlocks at all times; deadlock recovery allows transient deadlocks that eventually trigger a recovery system. This section will only discuss deadlock avoidance mechanisms.

The specifics of a deadlock avoidance mechanism depends on the way in which messages are received. There are three ways in which messages can be received on the dynamic network: dedicated receive hardware, interrupt, or polling.

A simple way to avoid deadlocks is to be able to guarantee that all receive nodes are always sinking messages. One way to provide this guarantee is to provide hardware support for a receive mechanism that runs independently of the processing node. Similarly, if a processing node receives a message via interrupt, deadlocks can be avoided by always keeping the interrupt on. In the general case, interrupts need to be turned off during message sends, because an interrupt handler itself may need to send a message that would interfere with any partially constructed messages. If we assume that a network is used exclusively for ILP communication, however, the interrupt for that network needs not be turned off because we know its interrupt handlers do not need to send messages. This approach can also be used with a hybrid protocol that uses interrupt for the slow path and polling for fast path.

Deadlock avoidance for a protocol that relies exclusively on polling is more complicated. Here, the uncertainty of message arrival is not very compatible with the certainty of polling. One correct but conservative scheme operate as follows. Before a sending node sends a message, it checks that its output buffer has enough room for it to complete the message send without blocking. If the query succeeds, the message is sent. Otherwise, the sending node starts sinking any incoming messages until its send buffer frees up enough space. Compiler analysis can reduce the amount of polling code that needs to be inserted. For example, if the compiler can prove that no message can arrive at a node that is trying to send a message, that node needs not check the output buffer before sending a message.

Interrupt vs polling The tradeoff between interrupt and polling is as follows. Polling is advantageous because its code can be inlined and register allocated. Interrupt is advantageous because it simplies deadlock avoidance. A good receive mechanism is to use a combination of both – polling for a fast path that uses compiler information to predict when and what message will arrive; interrupt for the slow path that guarantees the absence of deadlock.

4.3 Supporting ILP communication on the Raw static network

The Raw static network is a network whose routing information resides with the switch rather than with each message. Each switch has its own instruction memory; routing decisions are made based on the switch instructions. The switch is pipelined to allow words to be routed in a single cycle; near-neighbor communication latency is three cycles. Switch instructions are generated by the compiler during global instruction scheduling. Each switch allows any number of sources to route to any number of destinations, as long as each destination is specified by at most one source. This scheme supports multicasts (single source to multiple destination) as well as multi-routes (multiple independent messages to multiple destination). The switch have blocking semantics: it blocks if its routing instruction specifies an input port that is empty or an output port that is full. The processor interfaces to the network through registers; accesses to those registers likewise have blocking semantics.

In addition to routing instructions, the switch instruction set also contains control flow instructions. For our purpose, however, the conditions for those control flow instructions will always be generated by the its processor, so that its flow of control tracks that of the processor. In the global view, all processor and switch cooperate to exploit the parallelism in a basic block,

The Raw static network maintains an interesting contract with the compiler in order to support variation tolerance, destination message ordering, and reliable delivery. For each basic block, the compiler orchestrates a static communication schedule based on its compile-time knowledge about the dependences and latencies of operations. It guarantees that the schedule is correct and deadlock free given those latencies. At run-time, however, dynamic events such as cache misses and interrupts can cause the timings of operations to be different from that of the static schedule. Through the blocking sementics of the switches, however, the static network guarantees that the *order* of events on each processor and switch remains as specified by the static schedule. Thus, the order of message arrival on each processing node to remain consistent with that expected by the processor. In addition, the order of resource allocation on the switches remains consistent with that specified by the compiler schedule. Thus, the property that the schedule is deadlock is invariant over timing variations. In summary, the Raw static network provides the following guarantee within a basic block: it ensures a static total ordering of communication events on every processor and every switch. This property, in turn, provides destination message ordering and reliable delivery without any protocol overhead.

5 Results

This section presents the ILP performance on both the static and the dynamic networks of the Raw microprocessor. We present data on the communication pattern generated by the Raw compiler to help understand the usefulness of various network features in this communication space.

Benchmark	Source	Lines of code	Seq. RT (cycles)	Description
Cholesky	Nasa7:Spec92	126	34.3M	Cholesky Decomposition/Substitution
Fpppp-kernel	Spec92	735	8.98K	Electron Interval Derivatives
Jacobi	Rawbench	59	2.38M	Jacobi Relaxation
Life	Rawbench	118	2.44M	Conway's Game of Life
Vpenta	Nasa7:Spec92	157	21.0M	Inverts 3 Pentadiagonals Simultaneously
Moldyn	CHAOS	805	63M	Molecular Dynamics
Unstructured	CHAOS	850	150M	Computational Fluid Dynamics

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

5.1 ILP Performance on Raw

Experimental setup Our experiments are performed on beetle, a cycle accurate simulator for the Raw microprocessor [10]. Beetle contains both a static and a dynamic network that provide communication at the register level. Integer operation latencies are one cycle for simple operations, two-cycle integer multiplies, 36-cycle integer divides, single cycle stores, three-cycle loads. Floating point operation latencies are three-cycle adds and subtracts, three-cycle multiplies, ten-cycle divides, and three-cycle converts. A branch has a single delay slot. In this study, we assume that all memory accesses are cache hits.

Compiler supported is provided by Rawcc, as described in Section 4. In order to stress the communication network for this study, we use a non-zero but small communication cost when determining what granularity of parallelism to partition up. The real cost of communication is then used during instruction scheduling.

For the purpose of contrasting ILP performance on the static and dynamic network, we select several micro-benchmarks that have enough parallelism to profitably exploit 32 tiles. For communication characterization, we add two less parallel applications. Table 1 describes these benchmarks.

Our protocol for performing ILP communication on the Raw dynamic network is as follows. The mechanism performs static buffer assignment in every basic block. For buffer reuse synchronization at the beginning of a basic block, the processing nodes either perform a global tree barrier on the dynamic network or an explicit batch synchronization, whichever is cheaper. Messages are received via polling, with a fast path that optimizes for correct compile-time prediction of the identification of the next message. Deadlocks are avoided by checking the condition of the output buffer before sending a message, and sinking outstanding messages if the output buffer has insufficient space for the send.

The static protocol for ILP communication is exactly as described in Section 4.

Performance results

Table 2 and Table 3 show the end-to-end performance for the dynamic and static protocol, respectively. The dynamic protocol is only able to achieve modest speedup for several applications on 32 tiles. The static protocol, however, is able to attain speedup for all the applications. For 32 tiles, the performance difference ranges from a factor of four for fpppp-kernel to a factor of ten for jacobi. These results show that a network requires more than register level communication in order to efficiently exploit ILP.

Table 4 considers the end-to-end overhead of various components of our dynamic protocol. *Static* is the static performance, with single-cycle sends and receives, three-cycle latencies between neighbors,

Benchmark	N=1	N=2	N=4	N=8	N=16	N=32
jacobi	1.0	0.52	0.60	0.79	1.03	1.65
life	1.0	0.41	0.60	0.84	1.84	2.47
cholesky	1.0	0.52	0.62	0.55		
vpenta	1.0	1.38	1.41	1.17		
fpppp-kernel	1.0	1.12	0.54	0.94	1.46	1.48

Table 2: ILP performance on dynamic network. Each entry is speedup with respect to one tile.

Benchmark	N=1	N=2	N=4	N=8	N=16	N=32
jacobi	1.0	1.39	2.95	5.37	10.33	16.85
life	1.0	1.61	2.90	4.93	9.67	19.22
cholesky	1.0	1.53	2.81	4.62	6.37	7.22
vpenta	1.0	1.98	2.80	3.32		
fpppp-kernel	1.0	1.64	2.66	4.46	6.90	5.96

Table 3: ILP performance on static network. Each entry is speedup with respect to one tile.

and zero overhead for destination message ordering and reliable delivery. The remaining rows are various dynamic implementations, in increasing order of functionality. *Dyn-ordered* is the dynamic performance when we include only the protocol cost of message sends, receives, and destination message ordering. *+Expl-batch-sync* adds the overhead of explicit batch synchronization between buffer reuses. *+Smart-reuse-sync* uses a more intelligent synchronization scheme that determines the cost of explicit batch synchronization versus tree barrier at compile-time, and selects the one that is the cheapest. *+Reliable* takes *+smart-reuse-sync* and adds the overhead of deadlock avoidance.

Results suggest that the base cost of our dynamic network implementation contributes most to the overhead and will benefit most from additional hardware support. No component, however, is cheap enough to be overlooked.

5.2 Communication characterization

This section characterizes the communication pattern extracted by the compiler, in order to help us evaluate the costs and benefits of the various network features.

Table 5 shows some benchmark communication statistics when Rawcc targets an 8x4 Raw machine. The statistics are collected from the version of Rawcc targeting the static network, but the timing independent statistics apply to the dynamic version also. These data suggest the following. The relatively small number route per cycle across the machine suggests that in the absence of spatial or temporal hotspots, the network has sufficient communication bandwidth. The non-unit *Rec/Send* suggests that the communication has modest fanout. *Route/Send* reflects routing distance, and it indicates a large amount of non-near-neighbor communication. The maximum number of receives per basic block is a useful guideline when determining hardware support for buffer demultiplexing in the dynamic protocol.

Table 6 shows the usefulness of multicast on the Raw static network. Five of the applications use multicasts over 10% of the time, with a maximum of over 40% for cholesky.

Benchmark	N=1	N=2	N=4	N=8	N=16	N=32
static	1.0	1.38	2.95	5.37	10.32	16.85
dyn+ordered	1.0	0.80	1.06	1.56	3.02	3.64
+expl-batch-sync	1.0	0.73	0.86	1.08	1.05	0.84
+smart-reuse-sync	1.0	0.73	0.86	1.08	1.46	2.63
+reliable	1.0	0.52	0.61	0.81	1.05	1.69

Table 4: Jacobi performance breakdown. Each entry is speedup with respect to one tile.

Benchmark	Cycle	Send/Cycle	Rec/Cycle	Route/Cycle	Rec/Send	Route/Send	Max Rec/BB
jacobi	483660	0.84	1.38	3.26	1.64	3.89	34
life	256361	0.96	1.47	3.30	1.53	3.45	33
fpppp-kernel	321	0.57	1.14	3.52	2.00	6.17	36
cholesky	646464	0.07	0.52	0.83	7.37	11.86	68
vpenta	975848	0.14	0.37	1.04	2.56	7.25	25
moldyn	26535036	0.32	0.46	1.78	1.43	5.49	256
unstructured	9602683	0.51	0.65	2.82	1.26	5.51	258

Table 5: Benchmark communication statistics when Rawcc targets an 8x4 Raw machine. The statistics are execution counts from the static network. BB stands for basic block.

Rawcc currently does not take advantage of multi-routes. However, there are two reasons why we believe multi-routes are not performance critical. First, the low utilization of the network suggests that route crossings are infrequent. Second, unlike multicasts, the lack of multi-route does not increase the processor occupancy cost of communication.

6 Conclusion

This paper examines the general problem of providing a scalable interconnect for exploiting ILP on a decentralized architecture. We introduce a classification of execution model that is based on whether instruction assignment and instruction ordering are performed at compile-time or run-time. We present the fundamental requirements that a network has to satisfy in order to exploit ILP in this broad domain. We narrow our focus on compile-time assignment and ordering of instructions, and we present two ILP network implementations in this domain in the context of the Raw architecture, a decentralized architecture.

We draw several insights from this exercise. First, we are able to leverage compile-time knowledge about instruction assignment and operand ordering to improve either the performance or the hardware requirement of our protocols. Examples of this principle include the simplification of instruction issue logic through a total ordering of instructions on each processing node, optimizing buffer reuse synchronization on the dynamic network, and providing a fast path for message receives on a general dynamic ILP network that relies on the compiler's ability to predict the next incoming message. Second, we show that the Raw static network is able to satisfy the fundamentals of an ILP network by guaranteeing a static ordering of communication events on every processor and switch, with little protocol overhead at the cost of some run-time routing flexibility.

Benchmark	1	2	3	4
life	0.876	0.093	0.031	0.000
jacobi	0.880	0.086	0.025	0.010
moldyn	0.934	0.057	0.007	0.002
vpenta	0.841	0.116	0.030	0.013
cholesky	0.569	0.327	0.101	0.003
fpppp-kernel	0.855	0.128	0.017	0.000
unstructured	0.963	0.029	0.007	0.001

Table 6: Distribution of the number of output ports accessed by a route instruction for an 8x4 Raw machine.

References

[1] Arvind and S. Brobst. The evolution of dataflow archiectrues from static dataflow to p-risc. In *Proceedings of Workshop on Massive Parallelism: Hardware, Programming, and Applications*, 1990.

[2] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal. Maps: A Compiler-Managed Memory System for Raw Machines. In *Proceedings of the 26th International Symposium on Computer Architecture*, Atlanta, GA, May 1999.

[3] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comput.*, C-30(7):478–490, July 1981.

[4] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau. Parallel processing: A smart compiler and a dumb machine. In *Proceedings fo the ACM SIGPLAN 84 on Compiler Construction, SIGPLAN Notices*, pages 37–47. ACM, June 1984. Vol. 19, No. 6.

[5] J. Janssen and H. Corporaal. Partitioned register file for ttas. In *Proceedings of the 28th International Symposium on Microarchitecture*, pages 303–312, 1996.

[6] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, Atlanta, Georgia, June 22–24, 1988.

[7] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe. Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine. In *Architectural Support for Programming Languages and Operating Systems*, pages 46–57, San Jose, CA, Oct. 1998.

[8] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz. Smart memories: A modular reconfigurable architecture. In *Proceedings of the 27th International Symposium on Computer Architecture*, 2000.

[9] G. Pratt and J. Nguyen. "Distributed Synchronous Clocking. *IEEE Transactions on Parallel and Distributed Systems*, pages 314–328, Mar. 1995.

[10] M. Taylor. Design Decisions in the Implementation of a Raw Architecture Workstation. Master's thesis, MIT, Department of Electrical Engineering and Computer Science, September 1999.

[11] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring It All to Software: Raw Machines. *IEEE Computer*, 30(9):86–93, Sept. 1997. Also available as MIT-LCS-TR-709.