Scalar Operand Networks:
On-Chip Interconnect for ILP in Partitioned Architectures

Michael Bedford Taylor, Walter Lee, Saman Amarasinghe, and Anant Agarwal
Laboratory for Computer Science
Massachusetts Institute of Technology
{mtaylor, walt, saman, agarwal} @ cag.lcs.mit.edu

Abstract

The bypass paths and multiported register files in microprocessors serve as an implicit interconnect to communicate operand values between pipeline stages and multiple ALUs. Previous superscalar designs used centralized structures for this interconnect and do not scale with increasing ILP demands. In search of scalability, recent microprocessor designs in industry and academia reveal a trend towards distributed resources such as partitioned register files, banked caches, multiple independent compute pipelines, and even multiple program counter architectures. Some of these partitioned microprocessor designs have begun to implement the bypassing and operand transport mechanism using point-to-point interconnects rather than centralized networks. We call interconnects optimized for scalar data transport, whether centralized or distributed, *Scalar Operand Networks.* Although these networks share many of the challenges of multiprocessor networks, for example, scalability and deadlock avoidance, they have many unique features including their requirement of ultra-low latencies (a few cycles versus tens of cycles), and provision for ultra-fast operand matching or ultra-fast receive side demultiplexing. This paper discusses these unique properties of scalar operand networks, discusses alternative ways of implementing them, and examines in detail the implementation of one such network in the Raw microprocessor. The paper analyzes the performance of these networks for ILP workloads and the sensitivity of overall ILP performance to network properties.

Introduction

Today's wide-issue microprocessor designs are finding it increasing difficult to convert burgeoning silicon resources into usable, general-purpose, functional units. The problem is not so much that the area of microprocessor structures is growing out of control; after all, Moore's law's exponential growth is easily able to outpace a mere quadratic growth in area. Rather, it is the delay of the interconnect inside the processor blocks that has become unmanageable [Palacharla97,VAgarwal00]. Thus, although we can build almost arbitrarily wide-issue processors, clocking them at high frequencies will become increasingly difficult. A case in point is the Itanium 2 processor, which sports a zero-cycle fully-bypassed 6-way issue integer execution core. Despite occupying less than two percent of the processor die, this unit spends half of its critical path in the bypass paths between the ALUs [ISSCC 2002].

More generally, the pervasive use of global, centralized structures in these contemporary processor designs constrains not just the frequency-scalability of functional unit bypassing, but of many of the components of the processor that are involved in the task of naming, scheduling, orchestrating and routing operands between functional units [Palacharla198].

Building processors that can exploit increasing amounts of ILP continues to be important today. Many existing general purpose applications display larger amounts of ILP than can be gainfully exploited by current architectures. Furthermore, other forms of parallelism such as data parallelism, pipeline parallelism or coarse-grained parallelism, can easily be converted into instruction level parallelism.

Chip multiprocessors, like IBM's two-core Power4, hint at a scalable alternative for codes that can leverage more functional units than a wide-issue microprocessor can provide. Research and commercial implementations have demonstrated that multiprocessors based on scalable interconnects can be built to scale to thousands of nodes. Unfortunately, their high cost of inter-node operand routing (i.e. the cost of transferring the output of an instruction on one node to the input of a dependent instruction on another node) is often too high (tens to hundreds of cycles) for these multiprocessors to exploit instruction-level parallelism available in general purpose sequential programs. Instead, the programmer has to explicitly parallelize programs for multiprocessors, which has limited their appeal. Furthermore, because the difference in cost between local ALU and remote ALU communication is so large (sometimes on the order of 30x), programmers and compilers need to employ entirely different algorithms to leverage parallelism at the two levels.

Seeking to scale ILP processors, recent microprocessor designs in industry and academia reveal a trend towards distributed resources to varying degrees, such as partitioned register files, banked caches, multiple independent compute pipelines, and even multiple program counter architectures. These projects include IBM's Blue Gene [BlueGene], Stanford's Smart Memories[Mai00], UT Austin's Grid[Grid], MIT's Raw [Raw97] and Scale [Scale], Wisconsin's ILDP [ildp] and Multiscalar [Multiscalar], and the Alpha 21264 [alpha-21264]. Such partitioned or distributed microprocessor architectures have begun to replace the traditional centralized bypass network with a more general interconnect for bypassing and operand transport. We label operand transport interconnects, whether they are centralized or distributed, *Scalar Operand Networks.* Specifically, a scalar operand network is the interconnection network in microprocessors that provides the means for operand transport between ALUs and register files. These networks can be designed to have short wire lengths and therefore they can scale with increasing transistor counts. Furthermore, because they can be designed to resemble generalized interconnection networks, they can provide transport for other forms of data including I/O streams, cache misses, and synchronization signals.

Partitioned microprocessor architectures require scalar operand networks that combine the low-latency and low-occupancy operand transport of wide-issue superscalar processors with the frequency-scalability of multiprocessor designs. Several recent studies have also shown that partitioned microprocessors based on point-to-point scalar operand networks can successfully exploit fine-grained ILP. [Lee98] showed that a compiler can successfully schedule ILP on a partitioned architecture that uses a static point-to-point network to achieve speedup that was commensurate with the degree of parallelism inherent in the applications. Nagarajan et al. [grid] showed that the performance of a partitioned architecture using a dynamic point-to-point network was competitive with that of an idealized wide issue superscalar, even when the partitioned architecture counted a modest amount of wire delay.

Much as the study of interconnect networks is important for multiprocessors, we believe that the study of operand transport networks in microprocessors is also important. Although operand networks share many of the challenges in designing message passing networks, for example, scalability and deadlock avoidance, they have many unique challenges including their requirement of ultra-low latencies (a few cycles versus tens of cycles), and provision for ultra-fast operand matching or ultra-fast receive side demultiplexing (0 cycles versus tens of cycles). This paper identifies and discusses several important issues in designing scalar operand networks, and defines a parameterized model of these networks, through which we measure the costs and tradeoffs of a variety of design choices. To show that such an operand network is realizable, we also describe some of the details of the actual 16-way issue scalar operand network designed and implemented in the Raw microprocessor, using the .15 micron IBM SA-27E ASIC process.

One concrete contribution of this paper is that we show that sender and receiver occupancy have a first order impact on ILP performance. For our benchmarks running on a 32-tile microprocessor (i.e., 32 ALUs, 32-way partitioned register file, 32 instruction and data caches, connected by a scalar operand network) we measure a performance drop of between 10 and 30 percent when either the send or the receive occupancy is increased from zero to 1 cycle. Somewhat surprisingly, we discovered that network latency had a smaller impact. For example, performance of a 32-tile microprocessor was adversely affected by 10 to 30 percent when an idealized zero-latency crossbar with infinite buffering was replaced with a mesh network with 1 cycle delay per hop. For 32 tiles, network contention had an even smaller impact (about 10 percent). Our results further indicate that whether the network is static or dynamic is less important (at least for up to 32 nodes) than whether they offer support for receive side demultiplexing of operands.

The rest of this paper proceeds as follows. We first provide some background on operand networks and their evolution. We then discuss briefly the ILP computation model and how it maps to partitioned microprocessors based on point-to-point scalar operand networks. We continue by describing several important challenges in designing scalar operand networks and distinguish them from previous multiprocessor interconnects. We then discuss the scalar operand network implementation in the Raw processor, and examine the sensitivity of ILP performance to network properties. We conclude with a summary of our results.

Evolution of Scalar Operand Networks

The role of an operand transport mechanism is to make the dynamic operands and operations of a program meet in space to enact the computation specified by a program graph. Operand transport was a simple task during the era of non-pipelined processors, but as our demands for parallelism (e.g., multiple ALUs, large register name spaces), clock rate (e.g., deep pipelines), and scalability (e.g., partitioned register files) have increased, this task has become much more complex. This section describes the evolution of operand transport mechanisms -- from early, monolithic register file interconnects to recent, routed point-to-point mesh interconnects.

A non-pipelined processor with a register file and ALU contains a specialized, but very simple form of a scalar operand network. The logical register numbers provide a naming system for connecting the inputs and outputs of the operations. The number of logical register names sets the upper bound on the number of live values that can be held in the operand network. In Figure 1, rather than treating the register file as a black box, we emphasize its role as a device that is capable of performing two parallel routes from any two of a collection of registers to the output ports of the register file, and one route from the input of the register file to any of the registers. Each thin arc in the diagram represents a possible operand route that may be performed on each cycle. (In fact, each register should have a self-arc, since, in the default case, they are actually routing to themselves on each cycle. Every value in the register file is being routed every cycle, it's just that some of them are routed back to the same location.) It may seem absurd to view a register file in this framework, but the delay of a register file is largely interconnect-related; so the interconnect-centric view may not be far off, especially as wire delay worsens in our fabrication processes.

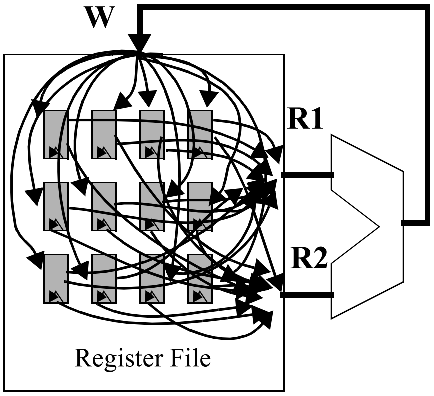


Figure 1: The simplest operand network

Figure 2 shows a pipelined, bypassed, register-ALU pair. The operand network now adds several new paths, multiplexers and pipeline registers. Notice, we have partitioned the operand traffic into two classes: "live" operands that are being routed directly from functional unit to functional unit, and "quiescent-but-live" operands that are being routed "through time" (via self routes in the register file) and then eventually to the ALU. This optimization improves the cycle time because the routing complexity of the live values is far less than the routing complexity of the resident register set. This transformation also changes the naming system -- the registers in the pipeline dynamically shadow the registers in the register file, and any reference to that register name will actually refer to the youngest shadowing register in the pipeline. However, for an in-order pipeline, the total number of live operands in the system does not actually change.

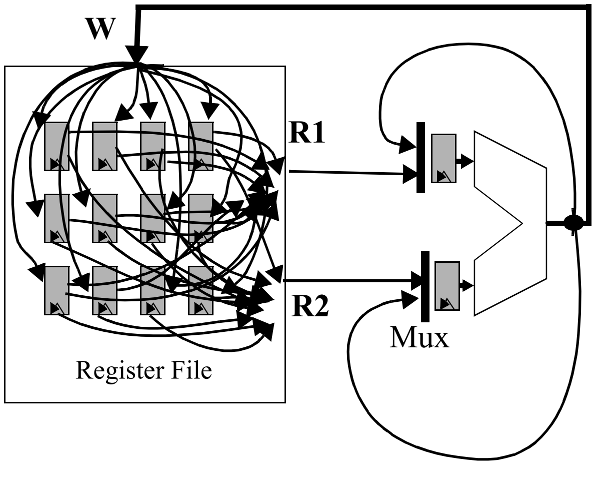


Figure 2: Scalar operand network in a pipelined processor with bypassing links added

Figure 3 shows a pipelined processor with multiple ALUs. Notice that the operand transport system includes many more multiplexers, pipeline registers, and bypass paths, and begins to look much like our traditional notion of a network.

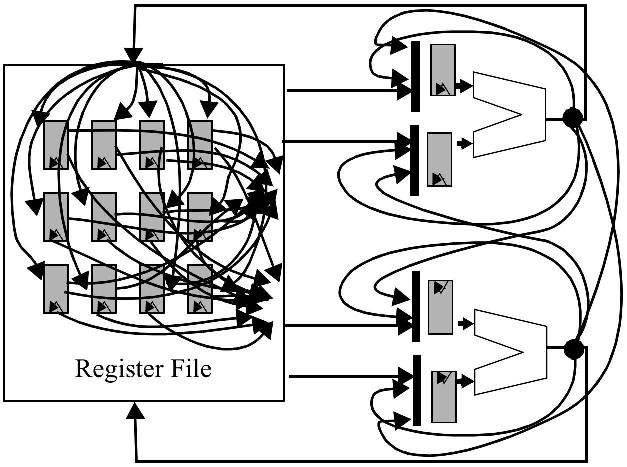


Figure 3: A pipelined processor with bypass links and multiple ALUs

Next, as the number of ALUs increases, and processors begin to support out-of-order issue and register renaming, the number of physical registers, register file ports, and possible bypass paths increases significantly. When out-of-order issue is added to the pipeline without register renaming, the parallelism of the operand network is constrained by the number of logical register names in the operand network, because the processor will need to respect anti- and output- dependences. Register renaming implements a larger number of physical registers and allows the quantity of simultaneous live values in the operand network to be increased beyond the number of named live values that are supported in the original program encoding. The resulting increase in the number of physical registers, coupled with the increase in the number of ports makes it increasingly hard to scale the register file.

Figure 4 depicts the partitioned register file and distributed ALU design of the Multiscalar -- one of the early distributed ILP processors. Notice that the Multiscalar connected the individual ALUs with a pipelined, one-dimensional multi-hop operand network, and pipelined the results along the one-dimensional network. Accordingly, the interconnect between the ALUs in the Multiscalar is an example of an early point-to-point scalar operand network.

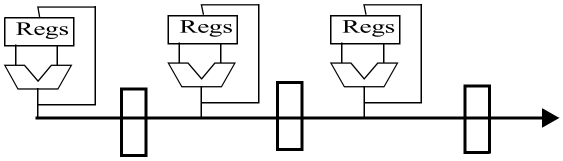


Figure 4: The operand network in the Multiscalar

Figure 5 shows the two-dimensional point-to-point scalar operand network in the Raw microprocessor. Raw implements a set of replicated tiles, and distributes all the physical resources -- ALUs -- floating point and integer, registers, caches, memories, I/O ports. Raw also implements multiple PCs and distributes them to each tile so that instruction fetch and decoding is also parallelized. Both the Multiscalar and Raw (and in fact most distributed microprocessors) exhibit replication in the form of more or less identical units that we will refer to as *tiles.* Thus, for example, we will use the term tile to refer to either the individual ALUs in the Grid processor, or the individual pipelines in the Multiscalar, Raw or the ILDP processors.

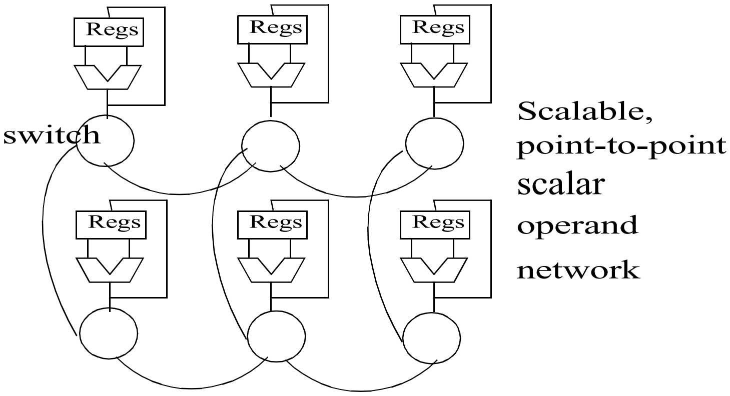


Figure 5: Scalar operand network based on a two dimensional, point-to-point routed interconnect

Exploiting ILP on distributed architectures like the Multiscalar, or Raw, or Grid, is not as straightforward as in zero-cycle bypass networks. The next section provides some background on how ILP gets mapped in such processors, and the section following that discusses the challenges in building scalar operand networks for partitioned microprocessors.

ILP Computation on Partitioned Architectures

Once the processor resources are partially or fully distributed, and connected via a scalar operand network, the mapping of ILP can take many forms. As background, we will discuss a generic distributed ILP architecture here and use it to illustrate some of the issues in mapping ILP to the distributed resources. Depending on the specific methods adopted to map ILP, the demands on the network can differ. We will address these alternatives in the next section.

ILP computations are commonly expressed as a dataflow graph. A dataflow graph is a logical network consisting 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 implies a sequential ordering between the execution of the two operations. Figure 6 shows a code fragment and its dataflow graph.

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 scheme, the application must sequentialize these operations.

```
i = a[j]
q = b[i]
r = q + j
s = q >> 3
t = r * s
b[i] = t
```

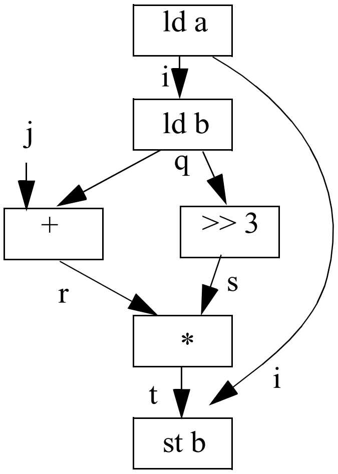


Figure 6: A code fragment and its dataflow graph

In Figure 6, 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 transforming 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 and pipelining.

To execute an ILP computation on a distributed-resource microprocessor containing a scalar operand network (for example, that shown in Figure 5), one needs to find an assignment from the nodes of the dataflow graph to the nodes of the network of ALUs, and route the intermediate values between these ALUs. Figure 7 shows one possible assignment of operations to each of the physical resources. Immediately, several issues relating to scalar operand networks become evident.

First, how the assignment of operations to ALUs should be performed? This assignment of operations can be performed at run-time or compile-time. Superscalar and early dynamic dataflow [Arvind90] are examples of run-time assignment architectures, while VLIW, TTA [Janssen96], Raw and Grid are compile-time assignment architectures.

Second, how should the routing be performed? Notice that the value of "q" is available at the node performing the "ld b" operation. This value must be routed to the node performing the "+" operation. If the architecture is a compile-time assignment architecture, the choice of the path that the values take between the ALUs can in turn be done either at compile-time or run-time. Raw chooses the routing path at compile-time, while Grid does it at run-time.

Third, what is the tradeoff between parallelism and communication in the assignment problem? On one hand, we want to spread the computation as far out into space as possible to maximize the number of ALUs 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 network 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 took only one cycle, then it would have been more effective to allocate all of the operations to one ALU (assuming that the design of the ALU and its supporting structures supported this.) Fourth, how should the ordering of the computations be enforced. For example, the “+” and “>>” operations cannot be performed unless the value of “q” is available.

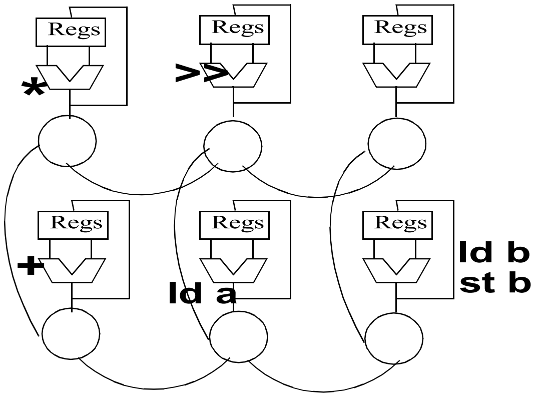


Figure 7: Mapping a dataflow graph on a generic distributed microprocessor containing 6 tiles.

With an intuition of the structure of ILP programs, and how they are mapped to distributed microprocessor resources, we can now discuss the structure of scalar operand networks and how they resolve these and other issues.

Challenges in the Design of Scalar Operand Networks

This section identifies and discusses some of the key challenges in the design of scalar operand networks: delay scalability, bandwidth scalability, efficient operation-operand matching, deadlock, and handling exceptional events.

1. Delay Scalability

As an unpipelined, two-dimensional VLSI structure increases in area, physics dictates that the propagation delay of this structure must increase asymptotically at least as fast as the square root of the area. This is a direct result of the additional distance that signals inside this structure have to travel. If we want to build larger structures and still maintain high frequencies, there is no option except to pipeline the logic and turn the propagation delay into pipeline latency. Delay scalability is the term that we use to describe the ability of a design to trade latency for delay as the system scales. As studies that compare small, short-latency caches with large, long-latency caches have shown, a large number of resources with long latency is not always preferable to a small number of resources with a short latency. This tradeoff between parallelism and locality is becoming increasingly important. On one hand, we want to spread virtual objects - such as cached values, operands, and instructions - as far out as possible in order to maximum the quantities of parallel resources that can be leveraged. On the other hand, we want to minimize communication latency by placing communicating objects close together, especially if they are on the critical path. These conflicting desires motivate us to design architectures with non-uniform costs; so that rather than paying the maximum cost of accessing a object (e.g., the latency of the DRAM), we pay a cost that is proportional to the delay of accessing that particular object (e.g., a hit in the first-level cache). This optimization is further aided if we can exploit locality among virtual objects and place related objects (e.g. communicating instructions) close together.

Inter-component delay scalability

The inherent delay in interconnect is a central issue in multiprocessor designs, and is now becoming a central issue in microprocessor designs. The Alpha 21264 marked the beginning of an architectural movement that recognizes that interconnect delay can no longer be hidden beneath the micro-architectural abstraction layer. Once interconnect delay becomes significant, high-frequency systems must be designed out of components that operate with only partial knowledge of what the rest of the system is doing. In other words, the architecture needs to be implemented as a distributed process. *If a component depends on information that is not generated by a neighboring component, the archi-*

tecture needs to assign a time cost for the transfer of this information. Non-local information includes the outputs of physically remote ALUs, stall signals, branch mispredicts, exceptions, and the existence of memory dependencies. One finds that logical diagrams of interconnect-aware architectures are increasingly more telling about the spatial properties of the design, because necessity has effectively transformed the spatial constraints into logical constraints. There are two clear examples of commercial architectures addressing the delay scalability issue: the Pentium IV, which introduced two pipeline stages that are dedicated to the crossing of long wires between remote components, and the Alpha 21264, which introduces a one cycle latency cost for results from one integer cluster to reach the other cluster.

Intra-component delay scalability

Delay scalability is also an intra-component issue, particularly for components whose size increases as a system scales. A number of common microprocessor structures like multi-ported register files, bypassing logic, selection logic, and wakeup logic grow indirectly, if not directly, with the issue width of the processor. Although designers have realized extremely efficient versions of these components, the burgeoning size of these components guarantees that intra-component interconnect delay will inevitably slow these components down. Thus, these components have an asymptotically unfavorable growth function that is partially obscured by a favorable constant factor.

There are number of solutions to the delay scalability of these structures; but the general themes typically include partitioning and pipelining. A number of recently proposed academic architectures [Grid, ILDP, Raw, SCALE, Multiscalar, SmartMemories] (and current-day multiprocessor architectures) compose their systems out of replicated tiles in order to simplify the task of reasoning about and implementing delay-scalable systems. A system is scaled up by increasing the number of tiles, rather than increasing the size of the tiles. A latency is assigned for accessing or bypassing the logic inside the tile element. The inputs and outputs of the tiles are periodically registered so that the cycle time is not impacted. In effect, they ensure that task of reasoning about delay scalability need only be performed at the intercomponent level.

The replication approach has been used regularly at the VLSI implementation level and is a proven technique for reducing the design complexity of a system. Although designing a system in a monolithic fashion enables *economies of integration* (additional performance due to the ability to optimize across module boundaries), the argument is that tile designs are more leveraged, because an individual tile optimization will result in gains across the entire array of tiles.

2. Bandwidth Scalability

Bandwidth scalability is also a challenge that is making its way from multiprocessor designs to microprocessor designs. Superscalars currently rely on global broadcasts to communicate the results of instructions. This means that every ALU's output is indiscriminately being sent to every waiting instruction that could possibly depend on that ALU. Thus, if RB is the number of result buses of the processor, and WS is the window size of the processor, there are RB*WS individual routes and comparisons that are being made on every cycle. As shown by the Alpha 21264, superscalars can handle the delay scalability of broadcasting by pipelining these broadcast wires. This means that some dependent instructions will incur an extra cycle of delay, but it guarantees that broadcasting of results does not impact cycle time. Unfortunately, the usage of indiscriminate broadcast mechanisms carries additional delay and area penalties as a system is scaled up -- first, the interconnect area (and resulting delay due to the area) of the routing resources, and second, the cost of processing the incoming information.

The architecture community has had previous exposure to the scalability limitations of broadcast-based protocols. Snoopy-cache multiprocessors employ broadcasting; each cycle, R (which is some function of N) broadcasted cache requests have to be compared against the T tags in N processors; typical implementations have an area on the order of N*R*T.

The key to overcoming this problem is to find a way to decimate the volume of messages sent in the system. Directory-based cache-coherent multiprocessors tackle this problem by employing directories: distributed, known-ahead-of-time locations that contain dependence information. The directories allow the caches to reduce the broadcast to a unicast or multicast to only the parties that need the information. Then, the broadcast network is replaced with a point-to-point network that can perform unicast routes in order to exploit the bandwidth savings.

A directory scheme is a potential candidate for replacing broadcast in a scalar operand network. The source instructions can look up destination instructions in a directory, and then multicast output values to the nodes on which the destination instructions reside. If the system can guarantee that every dynamic instance of an instruction is always assigned to the same node in the operand network, it can store or cache the directory entry at the source node. The entry could have been placed there by the compiler, or it could be dynamically annotated by the architecture in a supplementary route table for each source instruction. This would be quite efficient because it does not incur lengthy communication delays in order to discover the destination node for each execution of an instruction. We call architectures [Raw, Grid, iWarp] whose operand networks use fixed assignments of instructions to nodes *fixed-assignment* architectures.

Dynamic-assignment architectures like superscalars and ILDP assign dynamic instruction instances to different nodes in order to load balance. In this case, the removal of broadcast mechanisms is a more challenging problem to address, because the directory can not be co-located with the instruction, which is moving around. ILDP decimates broadcast traffic by providing intra-node bypassing for values that are only needed locally; however it still employs broadcast for values that may be needed by other nodes. It would be interesting to see if one could replace this broadcast with a distributed register file or directory system.

An alternative to the directory approach could involve an approach which is similar to an adhoc mobile network routing protocol, where local nodes forward information about what other nodes they know about. This is beyond the scope of this paper.

3. Efficient Operation/Operand Matching

Operation/operand matching is the process of gathering operands and operations to meet at some point in space to perform a dataflow computation. If operation/operand matching can not be done efficiently, there is little point in scaling the issue-width of a processing system, because the benefits will rarely outweigh the overhead. If microprocessors have something to learn from multiprocessors about scalability, multiprocessors most certainly learn have something to learn from microprocessors about operation/operand matching.

In a multiprocessor system, the cost of communicating an operand from the output of one instruction to the input of a dependent instruction can be broken down into five components: send occupancy, send latency, network latency, receive latency, and receive occupancy. For reference, these five components typically add up to tens to hundreds of cycles [Anatomy93] on a multiprocessor. Contrastingly, all five components in conventional superscalar bypass networks add up to zero cycles! The challenge is to design an efficient operation/operand matching system that also scales. In order to do this, we first define the five-tuple that describes operation/operand communication costs, and then we examine a number of implementations and their associated costs.

The five-tuple of costs <SO, SL, NHL, RL, RO> consists of:

Send Occupancy - # cycles that an ALU wastes in transmitting an operand to dependent instructions at other ALUs.
Send Latency - # cycles of delay incurred by the message at the send side of the network
without consuming ALU cycles.
Network hop latency - # cycles of delay, per hop, incurred travelling through the interconnect
Receive latency - # cycles of delay between when the final input to an ALU instruction arrives
and when the consuming instruction is issued.
Receive Occupancy - the number of cycles that an ALU loses because it is employing a remote value.

Microprocessor operation/operand matching

Superscalar processors achieve operation/operand matching via the bypassing network and the instruction window hardware of the processor. The routing information required to match up the operations is inferred from the instruction stream and routed, invisible to the programmer, with the instructions and operands. Beyond the occasional move instruction (say in a software-pipelined loop, or between the integer and floating point register files, or to/from functional-unit specific registers), superscalars do not typically incur send or receive occupancy. Superscalars tend not to incur send latency, unless they lose out in a result bus arbitration. Receive latency is often eliminated by waking up the instruction before the incoming value has arrived, so that the instruction can grab its inputs from the bypass paths as it enters the ALU. This optimization requires that wakeup information be sent earlier than the result values. If more instructions are woken up in a cycle than the select logic can simultaneously handle, one could say that there is an effective receive latency. However, this is an arguable point. Network latencies of a handful of cycles have appeared in clustered superscalar designs.

TTA processor instruction streams explicitly specify the routes between functional units and the register files. Because the route are encoded as part of each instruction, they also do not incur send or receive occupancies.

Multiprocessor operation/operand matching
One of the unique issues with multiprocessor operation/operand matching is the tension between commit point and communication latency. Uniprocessor designs tend to execute early and speculatively and defer commit points until much later. When these uniprocessors are integrated into multiprocessor systems, all potential communication must be deferred until the relevant instructions have reached the commit point. In a modern-day superscalar, this means that there could be tens or hundreds of cycles that pass between the time that a communication instruction executes and the time at which it can legitimately send its value on to the consuming node. Until these networks support speculative sends and receives (as with a superscalar!), the send latency of these networks will be adversely impacted.

Multiprocessors employ a variety of communication mechanisms; the two flavors are message passing and shared memory.

For the purposes of discussing message-passing operator/operand matching, we will assume that a dynamic network [Dally] is being employed. Implementing operator/operand matching using a message-passing style network has two key challenges. The first is simply to provide an processor-network interface that allows the processor to perform low-overhead sends and receives of operands. In an instruction-mapped interface, there are special send and receive instructions that are used for communication; in a register-mapped interface, instructions can target special registers that correspond to communications.

Using this interface, the sender somehow needs to specify the destination(s) of the out-going operands. (Recall that the superscalar uses indiscriminate broadcasting to solve this problem.) There are a variety of methods for doing this. For instruction-mapped interfaces, the send instruction can leave encoding space (the log of the maximum number of nodes) or take a parameter to specify the destination node. For register-mapped interfaces, an additional word may have to be sent to specify the destination. If a node can only be sent to a limited number of neighboring locations, then multiple registers can be used to denote those directions. Finally, dynamic networks typically do not support multicast, so multiple message sends may be required for operands that have non-unit fanout. These parameters will impact the sender and receiver-side occupancy.

The receiver needs to be able to gather in-coming operands and match them with the appropriate instruction. Because timing variances due to cache misses and interrupts can delay nodes arbitrarily, one can not assume a set arrival order for values sent over the dynamic network. This means that a tag will also need to be sent along with each operand. When the tag arrives at the destination, it needs to be *demultiplexed*, and delivered to the correct instruction at hand. Conventional message-passing implementations typically would have to do this in software. This creates a considerable receiver-side occupancy. The results section of this paper measures the performance impact of performing this operation in software.

On a shared-memory multiprocessor, one could imagine implementing operator/operand matching by implementing a large software-managed physical register file in cache RAM. Each communication edge between sender and receiver could be assigned a memory location that has a full/empty bit. In order to support multiple simultaneous dynamic instantiations of an edge, for loops, a base register could be incremented on every iteration of the loop. The sender processor would execute a special store instruction that stores the outgoing value and sets the full/empty bit. The readers of a memory location would execute a special load instruction that blocks until the full/empty bit is set, and returns the written value. Every so often, all of the processors would synchronize so that they can reuse the communication buffer. A special mechanism could flip the sense of the full/empty bit so that the bits would not have to be cleared.

The send and receive occupancy of this approach is difficult to evaluate. The sender's store instruction and receiver's load instruction only occupy a single instruction slot; however, the processors may still incur an occupancy cost due to limitations of the number of outstanding loads and stores that they can have. The send latency will be the latency of a store instruction, plus the time for the instruction to commit. The receive latency would include the delay of the store instruction, as well as the non-network time required for the cache protocols to process the receiver's request for the line from the sender's cache.
This approach has number of benefits: First, it supports multicast (although not in a way that saves bandwidth over multiple unicasts). Second, it allows a very large number of live operands due to the fact that the physical register file

is being implemented in the cache. Finally, because the memory address is effectively a tag for the value, no software instructions are required for demultiplexing.

Systolic array operation/operand matching

Systolic machines like iWarp [iWarp] were some of the first systems to achieve low-overhead operand/operation matching in large-scale systems. iWarp sported register-mapped communication, and is optimized for transmitting streams of data rather than individual scalar values. The programming model supported a small number of pre-compiled communication patterns (no more than 20 communications streams could pass through a single node.) For the purposes of operation/operand matching, each stream corresponded to a logical connection between two nodes. Because values from different sources would arrive via different logical streams, and values sent from one source would be implicitly ordered, iWarp had efficient receive-side demultiplexing. It needed only execute an instruction to change the current stream if necessary, and then use the appropriate register designator. Similarly, for sending, iWarp would optionally have to change the output stream and then write the value using the appropriate register designator. Unfortunately, the iWarp system is limited in its ability to facilitate ILP communication by the hardware limit on the number of communication patterns, and by the relatively large cost of establishing new communication channels. Thus, the iWarp model works well for stream-type bulk data transfers between senders and receivers, but is less suited to ILP communication. With ILP, large numbers of scalar data values must be communicated with very low latency in irregular communication patterns. iWarp's five-tuple can modeled as (1,6,5,0,1) - one cycle of occupancy for sender stream change, six cycles to exit the node, four or six cycles per hop, approximately 0 cycles receive latency, and 1 cycle of receive occupancy. An on-chip version of iWarp would probably incur a smaller per-hop latency, but a larger send latency; because, like a multiprocessor, it waits for instruction commit to transmit things out of the network. However, the fact that iWarp's communication resources could not be cheaply virtualized beyond the limit of 20 channels (some of which were reserved for the operating and runtime systems) creates obstacles to its use as a scalar operand network.

4. Deadlock and starvation

Superscalar operand networks use relatively centralized structures to flow control instructions and operands so that internal buffering can not be overcommitted. With less centralized operand networks, such global knowledge is more difficult to attain. If the processing elements independently produce more values than the operand network has storage space, then either data-loss or deadlock must occur. This is not an unusual problem; in fact some of the earliest large-scale operand network research -- the dataflow machines -- encountered serious problems with the overcommitment of storage space and resultant deadlock. Alternatively, priorities in the operand network may lead to a lack of fairness in the execution of instructions, which may severely impact performance.

5. Handling Exceptional Events

When designing any microprocessor, there is a certain quantity of exceptional events that, despite not being the common case, tend to occupy a fair amount of our design time. Whenever designing a new architectural mechanism, one needs to think through a strategy for handling these exceptional events. Each operand network design will encounter specific challenges based on the particulars of the design. It is a good bet that cache misses, branch mispredictions, exceptions, interrupts and context switches will be among those challenges. For instance, if an operand network is being implemented using a dynamic network, how do context switches work? Is the state drained out and restored later? If so, how is the state drained out? Is there a freeze mechanism for the network? Or is there a roll back mechanism that allows a smaller representation of a process's context? Are the branch mispredicts and cache miss requests sent on the operand network, or on a separate network?

Implementation of the Raw Scalar Operand Network

The Raw prototype divides the usable silicon area into an array of 16 identical, programmable tiles. A tile contains an 8-stage in-order single-issue MIPS-derived compute processor, a 4-stage pipelined FPU, a 32 KB data cache, two types of communication routers -- static and dynamic, and 96 KB of instruction cache. These tiles are interconnected by to neighboring tile using 4 networks, two static and two dynamic. These networks consist of over 1024 wires per tile. The static routers control the static network, which is used as point-to-point transport for Raw's operand network.

The dynamic routers and networks are used for all other traffic such as memory, interrupts and user-level message passing codes.

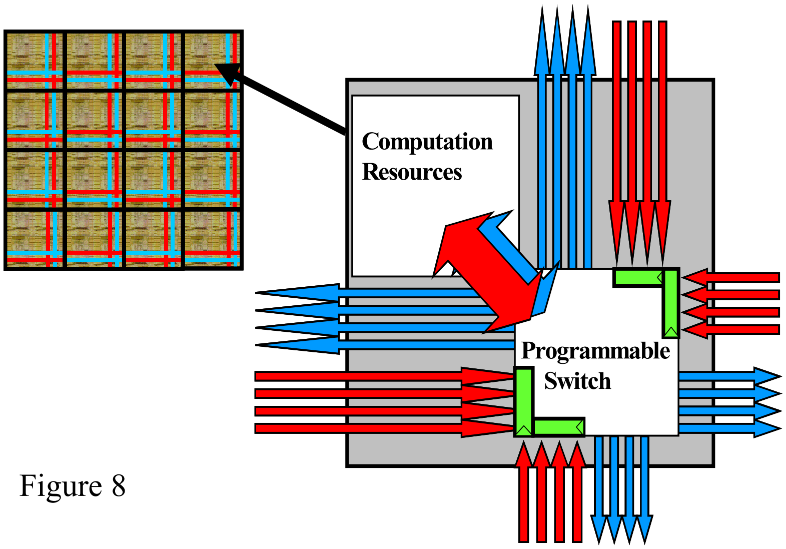


Figure 8

Each tile is sized so that the amount of time for a signal to travel through a small amount of logic and across the tile is one clock cycle. Larger Raw systems can be designed simply by stamping out more tiles. Figure 8 shows the array of Raw tiles, an individual Raw tile and its network wires. Notice that these wires are registered on input. This tiling approach is how we address the operand network delay-scalability challenge. Modulo building a good clock tree, we do not have to worry about the frequency decreasing as we add more tiles.

The static router is used to route the outputs of instructions on one tile to the inputs of the dependent instructions on other tiles. If the output is required on the same tile, then the 0-cycle latency interal bypass paths can be used. Live but not active values can be stored in the switch register file, compute-processor register file, or in the FIFOs of the network itself.

The static router is a 5-stage pipeline that controls two routing crossbars and thus two physical networks. Each crossbar routes values between seven entities - the static router pipeline, north, east, south, west, the compute processor, and the other crossbar. Thus, the Raw static network also supports multicast. The static router uses the same fetch unit design as the compute processor, except it fetches a 64-bit instruction word from the 8 K-entry instruction memory. This instruction simultaneously encodes a small command (conditional branches with/without decrement, accesses to a small register file) and thirteen routes, one for each crossbar output, for a total of fourteen operations per cycle per tile. The static router also has a small register file so that it can broadcast data values into the compute processor in the event that they are required multiple times and the compute processor does not want to waste a move instruction to its local register file.

For each word sent between tiles on the static network, there is a corresponding instruction in the instruction memory of each router that the word will travel through. These instructions are typically programmed at compile time, and are cached just like the instructions of the compute processor. Thus, the static routers collectively reconfigure the entire communication pattern of the network on a cycle-by-cycle basis. Further, because the router program memory is large and also cached, there is no practical architectural limit on the number of simultaneous communication patterns that can be supported in a computation. Because the static router knows what route will be performed long before the word arrives, the preparations for the route can be pipelined, and the data word can be routed immediately when it arrives. We believe that it is likely that this ahead-of-time knowledge of routes enables implementations of the static network that have lower latencies and higher frequencies than the equivalent dynamic network.

The static router is flow controlled, and will not proceed to the next instruction until all of the routes in a particular instruction have completed. This ensures that destination tiles receive incoming words in a known order, even when tiles suffer cache misses, interrupts, branch mispredicts, or other unpredictable events. The static router provides single-cycle-per-hop latencies and can route two values in each direction per cycle. Because Raw's network is point-to-point, and we route operands only to those tiles that need them, we effectively decimate the bandwidth required for operand transport relative to a comparable superscalar implementation.

Let us examine the Raw tile's compute processor. The compute processor register maps all of Raw's networks. Because the static network has been pre-programmed with the instructions for routing the data, the Raw compute processor does not have to specify destinations, and does not have to perform receive-side demultiplexing. Thus Raw's send and receive occupancies are zero, and the per-hop cost is one cycle. Because a message must goes through the local switch on a route, the send latency due to the network is also one cycle.

In order to ascertain the receive and send latencies, let us examine the Raw compute processor pipeline. Our design takes network integration one step further: the networks are not only register-mapped but also integrated directly into the bypass paths of the processor pipeline. This makes the network ports truly first-class citizens in the architecture. Figure 9 shows how this works. Registers 24..27 are mapped to the four physical networks on the chip. For example, a read from register 24 will actually pull an element from an input FIFO, while a write to register 24 will send the data word out onto that network. If data is not available on an input FIFO, or if an output FIFO does not have enough room to hold a result, the processor will stall in the register fetch (RF) stage. The instruction format also provides a single bit in the instruction which allows the instruction to specify two output destinations: one network or register and the network implied by r24 (the first static network). This gives the tile the option of keeping local copies of transmitted values.

The interesting activity occurs on the output FIFOs. Each output FIFO is connected to each pipeline stage. The FIFOs pull the oldest value out of the pipeline as soon as it is ready, rather than just at the writeback stage or through the register file [iWarp]. This decreases the latency of an ALU-to-network instruction by as much as 4 cycles for our 8-stage pipeline. This logic is exactly like the standard bypass logic of a processor pipeline except that it gives priority to older instructions rather than newer instructions. In effect, we've deliberately designed an early commit point into our processor in order to eliminate the common multiprocessor communication-commit delay that was described in the challenges section.

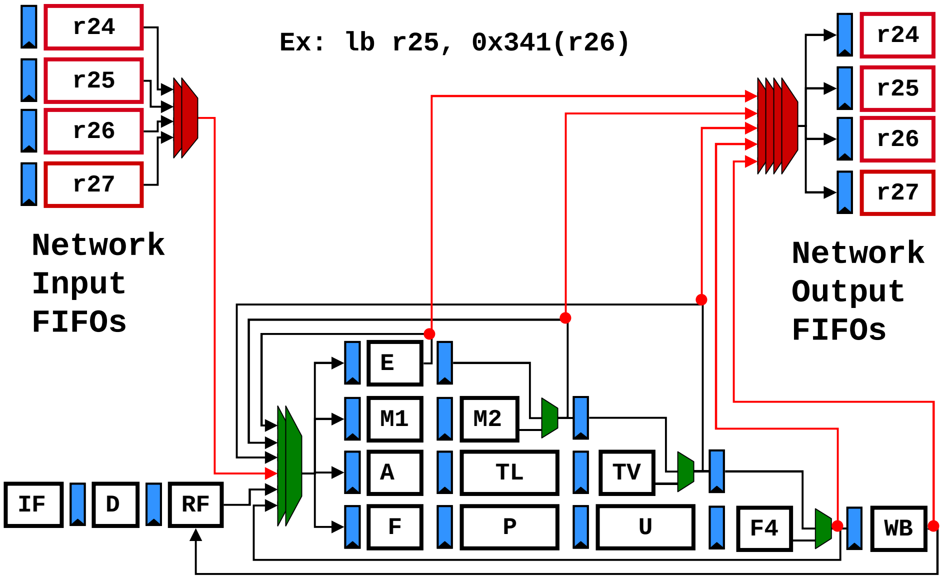


Figure 9: Directly integrating the network interfaces into the bypass paths

Because of the early-commit optimization, the compute processor incurs no additional send latency. On the receive side, register-mapping ensures that there is no receive occupancy. There is one cycle of receive latency because the receive FIFOs are accessed in the dispatch stage of the processor. The valid bits of this FIFOs are also required for the stall logic of the pipeline. It is possible that this cycle of receive latency could be eliminated if we route the valid bits one cycle ahead of the data bits in the network, but circuit studies will have to show if this will impact the critical path.

This combination of static router and efficient processor-network interconnect allows Raw to perform operand-operator matching with a 5-tuple of (0,1,1,1,0). Furthermore, because routes are specified at compile time, the compiler can easily guarantee that there are no deadlock conditions in the static network. For the user-level messaging via the dynamic network, Raw provides a deadlock-recovery mechanism. The memory network obeys a deadlock-avoidance credit algorithm which prevents deadlock from occurring at the cost of conservative buffer usage.

Raw's operand network also supports exceptional events. Branch conditions and jump pointers are transmitted over the static network, just like data. Raw's interrupt model allows each tile to take and process interrupts individually. Compute processor cache misses stall only the compute processor that misses; however, eventually over time switches and tiles executing instructions that attempt to route in the result of a cache-missing load from a neighboring tile will block, waiting for the value to be transmitted over the switch. These cache misses are processed over a separate, dynamic network. Raw supports context switches by draining and restore the contents of its networks. This network state is saved off into a context block and then restored when the process is switched back in.

This brief summary of Raw is expanded further in [Taylor2002] and [RawSpec99]. Details on how the compiler programs the static switch are available in [Lee98].

Results

This section evaluates the performance impact of several of the properties of operand networks using a few scalar applications.

Experimental setup

Our experiments are performed on Beetle, a cycle accurate simulator of the Raw microprocessor [RawSpec99]. A 16-tile Raw chip has been taped out to IBM for fabrication and parts are expected in October 2002. Beetle implements faithfully Raw's static operand network. We also augmented Beetle with a parameterized scalar operand network for these experiments. As discussed earlier, Raw contains a MIPS-like single-issue 8-stage pipeline. Data cache misses are modeled faithfully. Data cache misses from each tile are satisfied over a dynamic network that is separate from the static, scalar operand network. All instructions are assumed to hit in the instruction cache.

The code is generated by Rawcc, the Raw parallelizing compiler [Lee98]. Rawcc takes sequential C or Fortran programs and parallelizes them across the Raw tiles. Rawcc operates on individual scheduling regions, each of which is a single-entry, single-exit control flow region. The mapping of code to Raw tiles includes the following tasks: assigning instructions to tiles, scheduling the instructions on each tile, and managing the delivery of any non-local operands. For the Raw machine, remote operand delivery includes generating the necessary switch code. Assignments of instructions to tiles require a delicate balance between parallelism and communication. Only parallelism that can overcome the communication cost should be mapped to different tiles. Furthermore, the compiler observes communication locality by placing communicating operations on the same tile or on neighboring tiles. For scheduling, Rawcc coordinates the instruction streams on both the processors and the switches to implement Raw's static operand matching execution model. For performance, its scheduler tries to overlap computation and communication as much as possible, without causing too much register pressure.

Rawcc's memory model is implemented by Maps, its compiler managed memory system [Barua99][Larsen02]. Maps uses compiler analysis to distribute the data of the sequential program across the tiles. The distribution is performed so that individual loads and stores are still predictable to be on a particular tile. This means that dense matrix arrays are often distributed element-wise across the tiles.

Rawcc performs unrolling for two reasons. First, it unrolls to expose more parallelism. Second, unrolling is performed in conjunction with Maps to allow the compiler to distribute arrays, while at the same time keeping the accesses to those arrays predictable. Additional details are presented in [Barua99].

We use the following benchmarks for our experiments. Cholesky, Vpenta, and Mxm are from Nasa7 of Spec92. Swim is from Spec95. Fpppp-kernel is the inner loop of Fpppp from Spec 95 that consumes 50% of the run-time. Sha is an implementation of Secure Hash Algorithm. Jacobi and Life are from the Raw benchmark suite [Babb]. Fpppp-kernel and Sha are irregular codes, while the rest are dense matrix codes.

The following hand modifications are made on the benchmarks: array reshape to Cholesky and loop fusion to Mxm. Both transformations can be automated. They are performed to improve the amount of parallelism with an unrolled loop iteration. The problem sizes of the dense matrix applications have been reduced to cute down on simulation time.

We note two additional points about our experimental setup. Even though the Raw hardware has zero receive occupancy cost, the compiler inserts a move instruction to receive an operand when the remote operand is needed more than once on the destination tile, instead of direct consumption by the ALU. Thus, in some cases, Raw incurs a receive occupancy of one. Note that the receive occupancy of the operand network in superscalars is always zero because results are broadcast to all destinations including ALUs and register file, and the register file is multiported.

To make intelligent instruction assignment and scheduling decisions, Rawcc needs to model the communication cost accurately. Accordingly, in all our experiments, Rawcc's model reflects the actual network simulated. Our memory placement algorithm, however, is currently insensitive to the latencies of the scalar network. (The compiler, however, does attempt to place operations close to the memory bank they access.) Therefore, when dense matrix arrays are distributed, they are always distributed across all the tiles. As communication cost increases, it may better for the arrays to be distributed across fewer tiles, but our experiments do not vary this parameter.

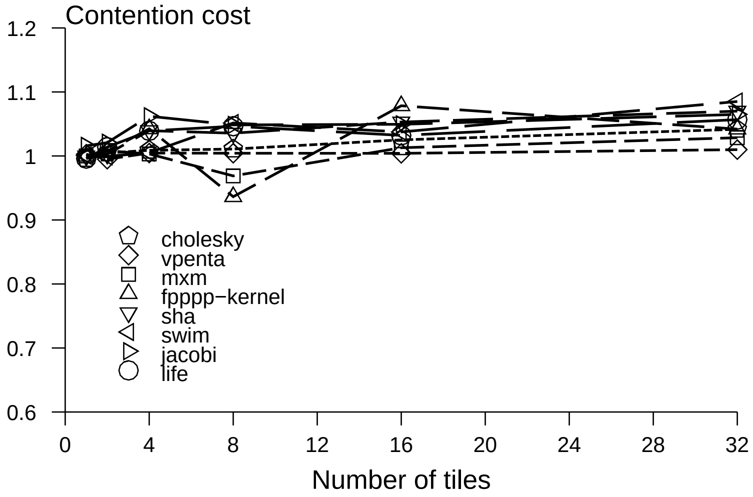
Basic experiments

First, let us get an idea of the amount of exploitable parallelism in the applications. Table 1 shows the performance of the benchmarks on the Raw machine. We vary the number of tiles from one to 32, and we measure the performance relative to one tile. These results indicate that the set applications represents a fairly wide range of parallelism. Sha has the least parallelism and obtains a 2.5 fold speedup on a 32-tile Raw chip versus a single tile Raw processor. fpppp and cholesky contain modest amounts of parallelism and obtain 7 and 8 fold speedup respectively. The other applications contain greater amounts of parallelism: vpenta obtains 15 fold speedup, matrix multiply 13.2 fold, swim 16.6 fold, jacobi 14.6 fold, and life obtains 21.3 fold speedup.

Table 1: Speedups for 1,2,4,8,16,32 tiles

	01	02	04	08	16	32
cholesky	1.00	1.53	2.75	4.95	7.13	8.15
vpenta	1.00	1.99	3.47	6.22	10.27	15.45
mxm	1.00	1.82	3.55	6.45	8.18	13.2
fpppp-kernel	1.00	1.47	2.96	5.13	6.47	7.10
sha	1.00	1.11	2.05	1.94	2.29	2.53
swim	1.00	1.70	2.86	4.84	8.72	16.62
jacobi	1.00	1.40	2.62	4.54	7.84	14.56
life	1.00	1.83	3.28	6.36	12.33	21.36

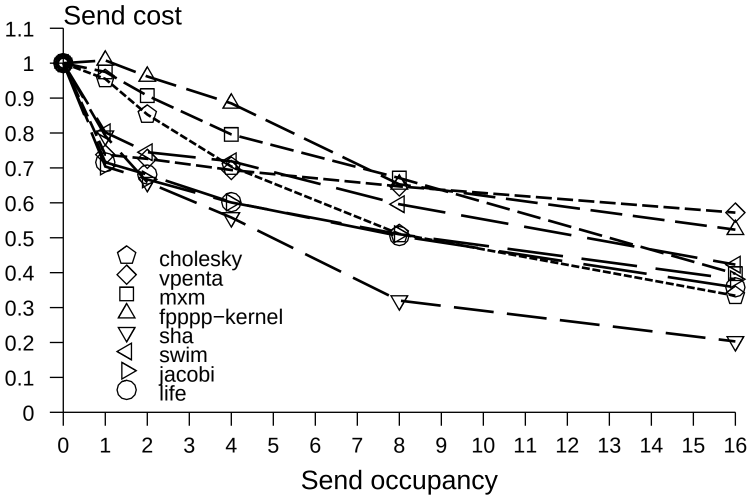
Next, we consider the cost of network contention on performance as the number of tiles increases. The figure labelled "Contention cost" considers this result. Each data point is a ratio of two performances; the execution-time of the Raw static mesh network versus the execution-time of an infinite bandwidth and infinitely buffered version of the same network. Each data point shows the speedup a benchmark achieves on the contention-free network over that of the contention-accurate network. The figure shows that the cost of contention is modest, about 10 percent for 32 tiles. For 32 or less tiles, this result indicates that reduced contention is not the best motivator for whether a scalar operand network should be static or dynamic.



The occasional anomalous behavior of some of the benchmarks (their performance increases slightly as the latency increases) arises from the effects of scheduling on register pressure. In effect, as the compiler schedules the code around the longer latencies, it is inadvertently reducing the live ranges of those values at the destination tile. When this value is not immediately needed, the net effect is always beneficial.

Raw's programmable static network allows the compiler to synthesize virtually arbitrary communication patterns including multicast when a value is needed on multiple remote tiles. Multicast reduces sender occupancy and makes better use of network bandwidth. We evaluated the performance benefit of multicast on the Raw static network for 32 tiles. For the case without multicast, we assume the existence of a broadcast -- to transmit control flow information over the static network with reasonable efficiency. For five of the benchmarks, the performance is around 0 percent. On the other hand, Mxm, Fpppp-kernel, and Jacobi gain a benefit of 6-8 percent.

The figures labelled "Send Cost" and "Receive Cost" show the performance of the benchmarks as we vary the send and receive occupancies. We emphasize that the increased latencies are visible to the compiler so that it can account for them as best as it can when it schedules ILP on the tiles. All data points are based on 32 tiles. A performance of 1.0 corresponds to the performance of the Raw static network: zero cycles of send occupancy, zero cycles of send latency, single-cycle hop, zero cycles of receive latency and zero to one cycle of receive occupancy. (Recall, when the operand needs to be stored in



the registers for later use, Raw incurs a receive occupancy of one. Thus Raw is plotted showing a receive occupancy of one). The results indicate that occupancy impacts performance significantly -- performance drops by 10 to 30 percent even when the occupancy is increased by 1 cycle. It is clear from these results that scalar operand networks must implement zero cycle send and receive occupancy. Because it has the least parallelism, notice also that Sha is most affected by higher occupancies.

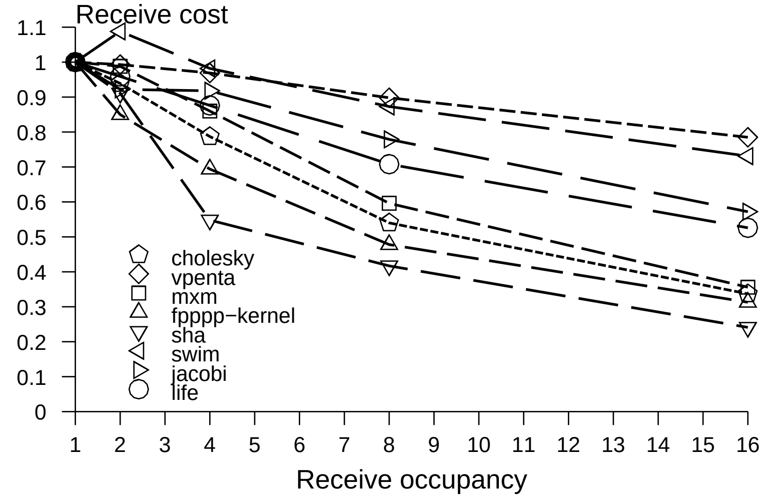
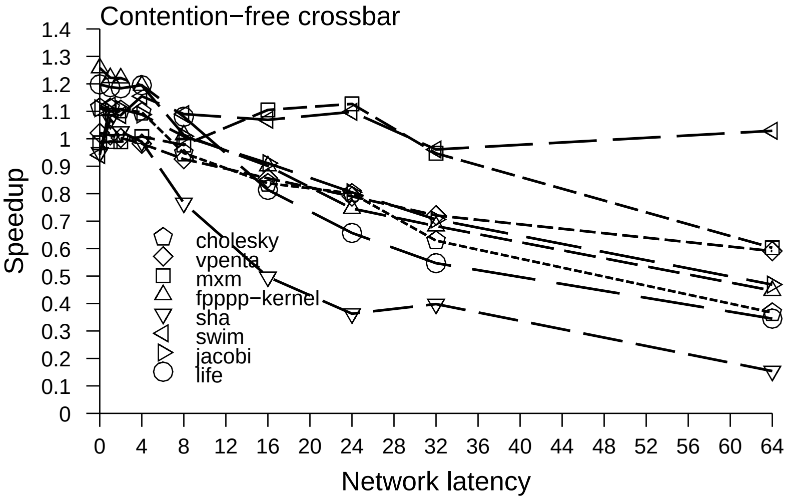
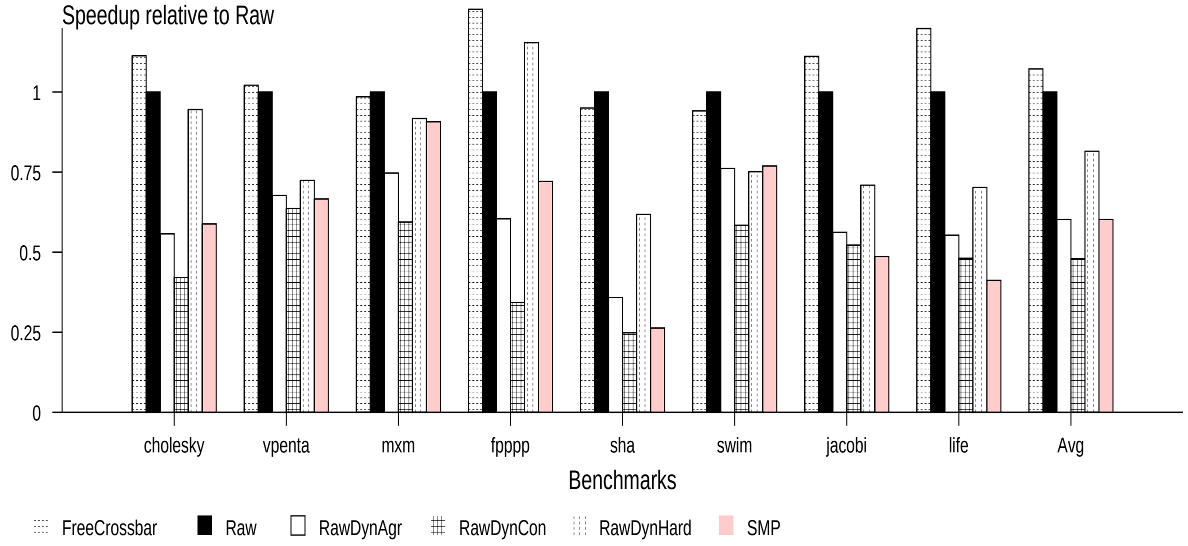


Figure "Contention-Free Crossbar" compares the performance of contention-free crossbar networks relative to that on the Raw static network on 32 tiles, as the latency of the crossbar network is changed. The crossbar network maintains an infinite buffer for each destination tile, so operands arrive at their destination buffers in zero time and are stored until an ALU operation consumes them. Observe that benchmark performance is slightly less sensitive to network latency than it is to send/receive occupancy. The performance of the Raw static network is about 30 percent worse than that of an idealized zero-cycle crossbar. As suggested earlier, the anomalous behavior of Mxm is due to the effect of scheduling on register pressure. The fact that the idealized crossbar has infinite buffers further magnifies this effect.





Network Models

Modeling other operand networks

Using our five-parameter network model, we model various realistic operand networks and analyze how our benchmarks perform on those networks. Figure "Network Models" shows these results on 32 tiles. The six network models are: FreeCrossbar (0,0,0,1,1), Raw (0,1,1,1,1), RawDynAgr (3,2,1,1,7), RawDynCon (3,3,1,1,12), RawDyn+Hard (1,2,1,2,1), and SMP (1,28,2,0,1). All performance bars are normalized to Raw. FreeCrossbar is the zero-cycle, contention-free crossbar displayed in Figure "Contention-free crossbar."

The RawDyn family of networks are operand networks based on the Raw dynamic network. RawDynCon and RawDynAgr are based on an actual implementation of operand transmission on the Raw dynamic network. An operand message includes three words: the message header, which includes the addressing information, an operand id, and the operand itself. The receiving tile receives the operand through polling. Although the compiler performs best-effort scheduling to match up the timing of operand delivery and consumption even on the dynamic network, timing vagaries associated with cache misses and interrupts can cause operands to be delivered out of order. So the receiver must be able to handle the case of unexpected operands. This case is handled by checking the id of the operand, and buffering the operand in memory if the id does not match with the expected id. Before checking the network, the receive code first checks this buffer to see if the operand has already been buffered.

In this implementation, an operand send takes three words. Assuming cache hits, an operand receive takes at last seven cycles -- if all operands arrive in the expected order on a destination tile. When an operand arrives out of order, it costs an extra five cycles to perform the buffering.

Our dynamic operand network implementation currently does not account for the cost of deadlocks. Deadlock handling mechanisms are often expensive -- in hardware, software, or both. Thus, the parameters in this implementation are a conservative estimate of the cost of operand delivery on a general 32-bit dynamic network when receive side demultiplexing and operand matching hardware is not provided.

Within this framework, we make a conservative and an aggressive estimate of the Raw dynamic operand network parameters. RawDynAgr is an aggressive estimate. It assumes that each message travels only along one dimension, and that all operands are delivered in the expected order. RawDynCon is a conservative estimate: it assumes that each message travels in both the X and Y direction (the cost of the extra routing decision is reflected in the extra send latency cycle), and it assumes that every operand is delivered out of order.

RawDyn+Hard is a hypothetical implementation of the operand network on the Raw dynamic network with extra hardware support. Sending takes one cycle: message destination and identification have been encoded in the instruction and do not require explicit injection. On the receive side, messages are received into dedicated hardware buffers that are indexed by a message id. Accessing this buffer incurs an additional cycle of latency on the receive end.

Finally, SMP is an estimate of the cost of delivering operands through shared memory on a single chip multiprocessor. We make the reasonable assumption that the SMP nodes can communicate through their L2 caches. The fixed latency cost is modeled as two cache misses -- one on the sender side and one on the receiver side. Because operand delivery involves both a request and a reply through the network, the network latency is doubled to account for the round-trip.

Related Work

This section contrasts a number of commercial and research systems and the approaches that they use to overcome the challenges of implementing scalar operand networks. We employ tabular form rather than sentence form to facilitate easy comparisons of the systems.

Table 2: Survey of scalar operand networks

	Superscalar	Distributed Shared Memory	Message Passing	Raw	Grid	ILDP
Delay scalability mechanism	None	Tiling	Tiling	Tiling	Partial Tiling	Partial Tiling
Operand transport mechanism	broadcast	point-to-point	point-to-point	point-to-point	point-to-point	broadcast
Operand Matching Mechanism	Associative Instruction Window	Full-empty bits on table in cached ram	Software Demultiplexing	Compile-time scheduling	Distributed, associative instruction window	Full-empty bits on distributed register file
Intra-node instruction order	runtime ordering	runtime ordering	compile-time ord’ing	compile-time ordering	runtime ordering	compile time ordering
free intra-node bypassing	yes	yes	yes	yes	no	yes
Instruction distribution	dynamic assignment	fixed assignment, compiler	fixed assignment, compiler	fixed assignment, compiler	fixed assignment, compiler	dynamic assignment of instruction groups
# Fetch Units for N nodes	1	N	N	N	1	1

In the following table, we summarize the five costs of operation/operand matching for each of these networks, as well as the number of ALUs that the systems are intended to support. Note that the ILDP and Grid papers examine a range of estimated costs; more information will be forthcoming when those systems are implemented.

Table 3: Operation/Operand matching costs five-tuples

	Super scalar	Distributed Shared Memory	Message Passing	Raw	ILDP	Grid
Send Occupancy	0	1	3	0	0	0
Fixed Send Latency (c = commit time)	0	26 + c	2 + c	1	0, 2	0
Per-Hop Latency	0	2	1	1	0	0, 1/8, 1/4, 3/8, ... ,8/8
Receive Latency	0	2	1	1	1	0
Receive Occupancy	0	1	12	0	0	0
Number of ALUs	4	many	many	4x4 to 32x32	8	8x8

This paper extends an earlier framework for operand networks that was given in [Taylor2000].

Conclusions

As we approach the scaling limits of wide-issue superscalar processors, researchers are seeking alternatives that are based on partitioned microprocessor architectures. Partitioned architectures distribute ALUs and register files over scalar operand networks that must somehow account for communication latencies. Even though the microarchitectures are distributed, ILP can be exploited on these operand networks because their latencies are extremely low. This paper makes several contributions: it introduces the notion of scalar operand networks, and discusses the challenges in implementing scalable forms of these networks. The challenges include dealing with both the increasing delay and limited bandwidth related to scalable networks, implementing ultra-low cost operation/operand matching, avoiding deadlock and starvation, and achieving correct operation in the face of exceptional events. The paper looks at several recently proposed distributed architectures that exploit ILP and discusses how each addresses the challenges. This paper also describes the implementation of an operand network in the Raw microprocessor and discusses how the implementation deals with each of the challenges in scaling scalar operand networks.

This paper breaks down the latency of operand transport into five components <SO, SL, NHL, RL, RO>: send occupancy, send latency, network hop latency, receive latency, and receive occupancy. The paper evaluates various network parameter alternatives and our early results include the following: Send and receive occupancy have the biggest impact on performance. For our benchmarks, performance decreases by 10 to 30 percent even if the occupancy on the send or receive side increases by just 1 cycle. Other parameters such as network latency, presence of multicast, and network contention have smaller impact.

Finally, in the past, the design of scalar operand networks was closely tied in with the design of other mechanisms in a microprocessor -- for example, register files and bypassing. In this paper, we attempt to carve out the generalized scalar operand network as an independent architectural entity that merits its own research. We believe that research focused on the scalar operand network will yield significant simplifications in future scalable ILP processor. Avenues for further research on scalar operand networks are plentiful and a partial list includes: (1) Designing networks that achieve a high clock rate while minimizing the five components related to performance, (2) evaluating the performance for much larger numbers of tiles and a wider set of programs, (3) generalizing the operand networks so that they support other forms of parallelism exploited by microprocessors such as stream parallelism and thread parallelism in SMT-style processing, (4) complete designs and evaluation of both dynamic and compile-time schemes for operation/operand assignment and scheduling, (5) generalized algorithms for multicasting and its performance, (6) mechanisms for fast exception handling and context switching, (7) a thorough analysis of the tradeoffs between commit point, exception handling capability, and network latency, (8) and low energy scalar operand networks.

References

Arvind90 **The evolution of dataflow architectures from static dataflow to p-risc**. Arvind and S. Brobst. *Proceedings of the Workshop on Massive Parallelism: Hardware, Programming, and Applications*, 1990.

Anatomy93 **The Anatomy of a Message in the Alewife Multiprocessor**. Kubiatowicz and Agarwal. *Proceedings of the International Conference on Supercomputing (ICS) 1993 July 1993.*

alpha-21264 **Digital 21264 Sets New Standard.** Linley Gwennap. *Microprocessor Report*, October 28, 1996.

Babb **The RAW Benchmark Suite: Computation Structures for General Purpose Computing**. J. Babb, et al. *Proceedings of the Symposium on Field-Programmable Custom Computing Machines*, April 1997.

Barua99 **Maps: A Compiler-Managed Memory System for Raw Machines,** Rajeev Barua et al, *Proceedings of the 26th International Symposium on Computer Architecture (ISCA),* June, 1999.

BlueGene **http://www.research.ibm.com/bluegene/comsci.html**, Mark Snir. 2000.

Dally **A VLSI Architecture for Concurrent Data Structures**, Dally, Kluwer Academic Publishers, 1987

Grid **A Design Space Evaluation of Grid Processor Architectures**. R. Nagarajan, K. Sankaralingam,et al *Proceedings of the International Symposium on Microarchitecture*, December 2001.

ildp **An Instruction Set Architecture and Microarchitecture for Instruction Level Distributed Processing**. H. Kim and J. E. Smith. Proceedings of ISCA 29, May 2002.

iWarp **iWarp, Anatomy of a Parallel Computing System**. Thomas Gross and David R. O'Halloron. The MIT Press. Cambridge, MA 1998.

Janssen96 **Partitioned register file for ttas**. J. Janssen and H. Corporaal. *Proceedings of the 28th International Symposium on Microarchitecture*. 1996.

Larsen2002 **Techniques for Increasing and Detecting Memory Alignment**. S. Larsen, et al. *MIT/LCS Technical Memo 621*, MIT-LCS-TM-621, November 2001.

Lee98 **Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine**. Walter Lee et al. *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII),* October 4-7, 1998.

Mai00 **Smart Memories: A Modular Reconfigurable Architecture**. Kenneth Mai, et al. *Proceedings of ISCA 27*, June 2000.

Multiscalar **Multiscalar Processors**, G.S. Sohi, S. Breach, T.N. Vijaykumar. *Proceedings of ISCA 22*, 1995.

Palacharla97 **Complexity-Effective Superscalar Processors**. S. Palacharla, N. Jouppi, and J. Smith. *Proceedings of ISCA 24*, June 1997.

Raw97 **Baring It All to Software: Raw Machines**. Waingold et al. IEEE Computer, September 1997.

Scale http://www.cag.lcs.mit.edu/scale/overview.html, 2000.

RawSpec99 **The Raw Processor Specification**. Michael Taylor. ftp://ftp.cag.lcs.mit.edu/pub/raw/documents/RawSpec99.pdf

Taylor2000 **How to build scalable on-chip ILP networks for a decentralized architecture**. Michael Taylor, Walter Lee, Matthew Frank, Anant Agarwal, Saman Amarasinghe. Massachusetts Institute of Technology's Laboratory for Computer Science Technical Memo MIT-LCS-TM-628. April 2000.

Taylor2002 **The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Prgrams**. Michael Bedford Taylor et al. IEEE Micro, March 2002.

VAgarwal00 **Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures.** Vikas Agarwal, M. S. Mrishikesh, Stephen Keckler and Doug Burger, Proceedings of the 27th ISCA, June 2000.

This research was funded by DARPA, the National Science Foundation and MIT's Project Oxygen.