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

Michael Bedford Taylor, Walter Lee, Saman Amarasinghe, Anant Agarwal
MIT Laboratory for Computer Science

ABSTRACT

The bypass paths and multiported register files in microprocessors serve as an implicit interconnect to communicate operand values among pipeline stages and multiple ALUs. Previous superscalar designs implemented this interconnect using centralized structures that do not scale with increasing ILP demands. In search of scalability, recent microprocessor designs in industry and academia exhibit a trend towards distributed resources such as partitioned register files, banked caches, multiple independent compute pipelines, and even multiple program counters. Some of these partitioned microprocessor designs have begun to implement bypassing and operand transport 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 such as scalability and deadlock avoidance, they have many unique requirements, including ultra-low latencies (a few cycles versus tens of cycles) and ultra-fast operation-operand matching. This paper discusses the unique properties of scalar operand networks, examines alternative ways of implementing them, and describes 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.*

1. INTRODUCTION

Today's wide-issue microprocessor designs are finding it increasingly 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 [1, 16]. 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 [14].

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 [16].

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

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, the 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 ILP. Instead, the programmer is faced with the unappealing task of explicitly parallelizing these programs. 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 counters. These designs include UT Austin's Grid [15], MIT's Raw [21] and Scale [17], Stanford's Smart Memories [13], Wisconsin's ILDP [9] and Multiscalar [18], and the Alpha 21264 [7]. 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. With these more sophisticated interconnects come more sophisticated hardware or software algorithms to manage them. We label operand transport interconnects and the algorithms that manage them, whether they are centralized or distributed, *scalar operand networks*. Specifically, a scalar operand network is the set of mechanisms that joins the dynamic operands and operations of a program in space to enact the computation specified by a program graph. These mechanisms include the physical interconnection network as well as the operation-operand matching system that coordinates these values into a coherent computation. Scalar operand networks can be designed to have short wire lengths. Therefore, they can scale with increasing transistor counts. Furthermore, because they can be designed to resemble generalized interconnection networks, scalar operand networks can potentially 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 shown that partitioned microprocessors based on point-to-point scalar operand networks can successfully exploit fine-grained ILP. Lee et al. [12] 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. [15] 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 interconnection networks is important for multiprocessors, we believe that the study of scalar operand networks in microprocessors is also important. Although these networks share many of the challenges in designing message passing networks, such as scalability and deadlock avoidance, they have many unique requirements including ultra-low latencies (a few cycles versus tens of cycles) and ultra-fast operation-operand matching (0 cycles versus tens of cycles). This paper identifies five important challenges in designing scalar operand networks. The paper also defines a parameterized 5-tuple model that gives structure to the task of reasoning about the tradeoffs in the design of these networks. To show that large-scale low-latency operand networks are 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 sender and receiver occupancies have a first order impact on ILP performance. For our benchmarks running on a 64-tile microprocessor (i.e., 64 ALUs, 64-way partitioned register file, 64 instruction and data caches, connected by a scalar operand network) we measure a performance drop of up to 25 percent when either the send or the receive occupancy is increased from zero to one cycle. The performance loss due to network contention, on the other hand, was discovered to average only 5 percent for a 64-tile Raw mesh. These results lead us to conclude that whether the network is static or dynamic is less important (at least for up to 64 nodes) than whether it offers efficient support for matching operands with the intended operations.

The rest of this paper proceeds as follows. Section 2 provides background on operand networks and their evolution. Section 3 discusses the ILP computation model and how it maps to partitioned microprocessors based on point-to-point scalar operand networks. Section 4 continues by describing several important challenges in designing scalar operand networks that distinguish these networks from previous multiprocessor interconnects. Section 5 discusses the scalar operand network implementation in the Raw processor. Section 6 quantifies the sensitivity of ILP performance to network properties. Section 7 presents related work. Section 8 concludes.

2. EVOLUTION OF SCALAR OPERAND NETWORKS

The role of a scalar operand network is to join the dynamic operands and operations of a program in space to enact the computation specified by a program graph. A scalar operand network includes both the physical interconnection network and the associated operation-operand matching algorithms (hardware or software) that coordinate operands and operations into a coherent computation. Designing scalar operand networks 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 scalar operand networks – from those that employ early, monolithic register file interconnects to the more recent ones that incorporate routed point-to-point mesh interconnects.

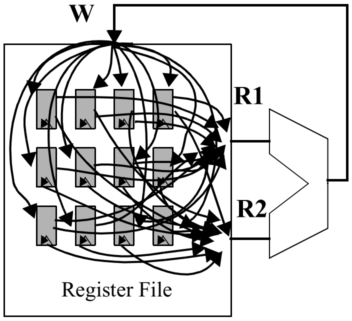


Figure 1: The simplest scalar operand network.

A non-pipelined processor with a register file and ALU contains a simple, specialized 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 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, it is actually routing to itself on each cycle.) 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.

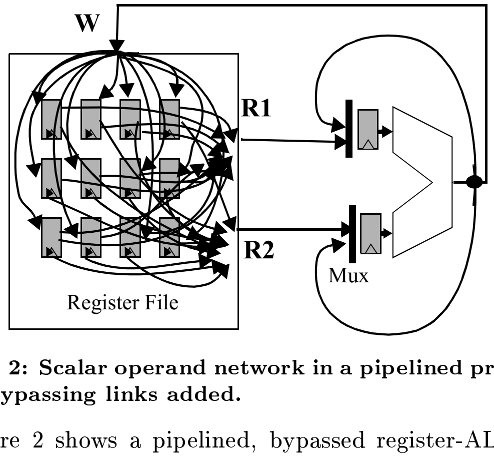


Figure 2: Scalar operand network in a pipelined processor with bypassing links added.

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 a shadowed 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.

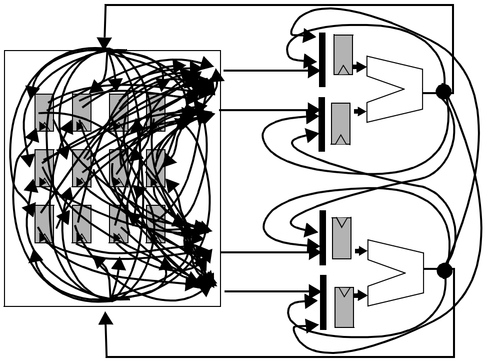


Figure 3: A pipelined processor with bypass links and multiple ALUs.

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

The introduction of multiple ALUs creates additional demands on the naming system of the scalar operand network. First, there is the temptation to support out-of-order issue of instructions, which carries numerous scheduling-related benefits: better utilization of the ALUs, reduced code size, and finally, the ability to change a program's schedule in response to dynamic events or changing microarchitectural properties. On the other hand, out-of-order issue forces the scalar operand network to deal with the possibility of having several live aliases of the same register name.

Adding register renaming to the scalar operand network allows the network to manage these live register aliases. Perhaps more significantly, register renaming also allows the quantity of simultaneous live values to be increased beyond the limited number of named live values fixed in the ISA. An even more scalable solution to this problem is to adopt an ISA that allows the number of named live values to increase with the number of ALUs.

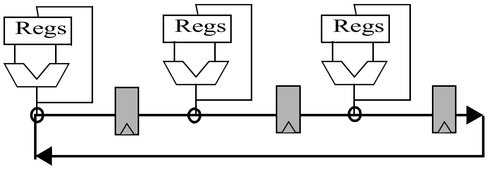


Figure 4: Multiscalar's operand network

More generally, increasing the number of functional units necessitates more live values in the scalar operand network, distributed at increasingly greater distances. This requirement in turn increases the number of physical registers, the number of register file ports, the number of bypass paths, and the diameter of the ALU-register file execution core. These increases make it progressively more difficult to build larger, high-frequency scalar operand networks that employ centralized register files as operand interconnect. One solution is to partition and distribute the interconnect and the resources it connects.

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 pipelined results through individual ALUs with a one-dimensional multi-hop operand network. Accordingly, the interconnect between the ALUs in the Multiscalar distinguishes it as an example of an early point-to-point scalar operand network.

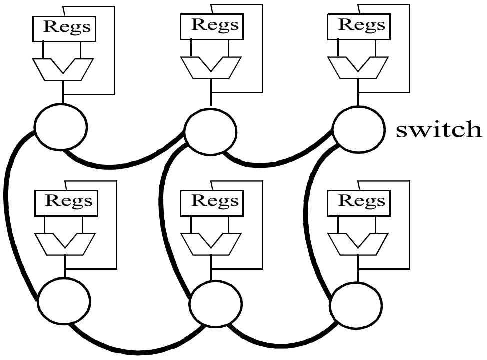


Figure 5: Scalar operand network based on a two dimensional, point-to-point routed interconnect.

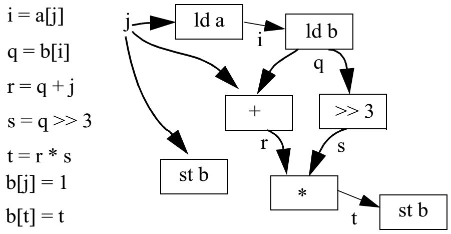
Figure 5 shows the two-dimensional point-to-point scalar operand network in the Raw microprocessor. Raw implements a set of replicated tiles and distributes all the physical resources: floating point and integer ALUs, registers, caches, memories, and I/O ports. Raw also implements multiple PCs, one per tile, so that instruction fetch and decoding are also parallelized. Both 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 an individual ALU in the Grid processor, or an individual pipeline in Multiscalar, Raw, or the ILDP processor.

Exploiting ILP on architectures with distributed scalar operand networks is not as straightforward as with early, centralized architectures. The next section provides some background on how ILP gets mapped in such processors, and the section after next discusses the challenges in building scalar operand networks for partitioned microprocessors.

3. 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. In a dataflow graph, the nodes represent operations, and the arcs represent data values flowing from the output of one operation to the input of the next. The existence of an



```
i = a[j]
q = b[i]
r = q + j
s = q >> 3
t = r * s
b[j] = 1
b[t] = t
```

Figure 6: A code fragment and its dataflow graph.

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. If the compiler cannot determine that the memory operations will not conflict, then there are possible dependences existing between the operations. Short of using a speculation scheme, the application must sequentialize these operations. Thus, in Figure 6, the accesses to b must be done sequentially; only the add and shift instructions 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.

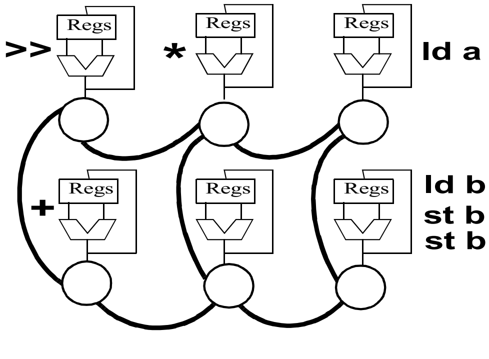


Figure 7: Mapping a dataflow graph on a generic distributed microprocessor containing six tiles.

To execute an ILP computation on a distributed-resource microprocessor containing a scalar operand network (for example, that shown in Figure 5), one first needs to find an assignment from the nodes of the dataflow graph to the nodes of the network of ALUs. Then it needs to 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 should the assignment of operations to ALUs be performed? This assignment of operations can be performed at run-time or compile-time. Superscalar and early dynamic dataflow [2] are examples of run-time assignment architectures, while VLIW, TTA [8], 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 at 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 critical path of the application. For instance, in Figure 7, if it takes a cycle to traverse a network link, and the ops each take only one cycle, then it would have been more effective to allocate all of the operations to one ALU (assuming that the processor supports this mapping).

Finally, what are the constraints on the ordering of the computation and on the ordering of operands? Are the operands guaranteed to arrive in order "just in time," or will they arrive out-of-order and need to be stored until the operation is ready to fire? Is there flow control on the operands coming into a node?

With an intuition of the structure of ILP programs and how they are mapped to distributed microprocessor resources, we can turn to some of the challenges in the designing scalar operand networks.

4. 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, deadlock and starvation, efficient operation-operand matching, and handling exceptional events.

1. Delay Scalability Delay scalability is the term that we use to describe the ability of a design to maintain high clock frequencies as that design scales. When 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 increase in delay 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.

Intra-component delay scalability 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 extremely efficient versions of these components exist, their burgeoning size guarantees that intra-component interconnect delay will inevitably slow them down. Thus, these components have an asymptotically unfavorable growth function that is partially obscured by a favorable constant factor.

There are a number of solutions to the delay scalability of these structures; the general themes typically include partitioning and pipelining. A number of recently proposed academic architectures [9, 13, 15, 17, 18, 21] (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, tiling ensures that the task of reasoning about delay scalability need only be performed at the intercomponent level.

Inter-component delay scalability Delay scalability is a problem not just within components, but between components. Components that are separated by even a relatively small distance are affected by the substantial wire delays of modern VLSI processes. This inherent delay in interconnect is a central issue in multiprocessor designs that is now becoming a central issue in microprocessor designs. There are two clear examples of commercial architectures addressing the interconnect delay 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. The Alpha 21264 marked the beginning of an architectural movement that recognizes that interconnect delay can no longer be ignored by the microarchitectural 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 architecture needs to assign a time cost for the transfer of this information.* Nonlocal information includes the outputs of physically remote ALUs, stall signals, branch mispredicts, exceptions, and the existence of memory dependencies.

Managing latency As studies that compare small, short-latency caches with large, long-latency caches have shown, a large number of resources (e.g., cache lines) 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 maximize 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.

2. Bandwidth Scalability Bandwidth scalability is the ability of a design to scale without inordinately increasing the relative percentage of resources dedicated to interconnect. 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. The output of every ALU is indiscriminately being sent to every waiting instruction that could possibly depend on that value. 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 pipelining causes some dependent instructions to incur an extra cycle of delay but guarantees that broadcasting of results does not directly 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. The fixed mapping scheme is quite efficient because the lookup of directory entry does not incur lengthy communication delays. We call architectures [6, 15, 21] 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.

3. Deadlock and starvation Superscalar operand networks use relatively centralized structures to flow control instructions and operands so that internal buffering cannot 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 problem is not unusual; 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 [2]. Alternatively, priorities in the operand network may lead to a lack of fairness in the execution of instructions, which may severely impact performance.

4. 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.

Because the cost of operation-operand matching is one of the most important figures of merit for a scalar operand network, we define a parameterization of these costs that allows us to compare the operation-operand matching systems of different architectures. This five-tuple of costs <SO, SL, NHL, RL, RO> consists of:

Send occupancy number of cycles that an ALU wastes in transmitting an operand to dependent instructions at other ALUs.

Send latency number of cycles of delay incurred by the message at the send side of the network without consuming ALU cycles.

Network hop latency number of cycles of delay, per hop, incurred travelling through the interconnect; more generally, this is the cost that is proportional to the physical distance between the sender and receiver.

Receive latency number of cycles of delay between when the final input to an ALU instruction arrives and when the consuming instruction is issued.

Receive occupancy number of cycles that an ALU wastes because it is employing a remote value.

For reference, these five components typically add up to tens to hundreds of cycles [10] on a multiprocessor. In contrast, all five components in conventional superscalar bypass networks add up to zero cycles! The challenge is to explore the design space of efficient operation-operand matching systems that also scale.

In the follow subsections, we examine scalar operand networks implemented on a number of existing systems and describe the components that contribute to the 5-tuple for that system. For these systems, we make estimates of 5-tuples for aggressive but conventional implementations. At one end of the spectrum, we consider superscalars, which have perfect 5-tuples, <0,0,0,0,0>, but limited scalability. On the other end of the spectrum, we examine message passing and shared memory systems, which have good scalability but poor 5-tuples. The Raw prototype, described in Section 5, serves as an example of a design that falls in between the two extremes, with multiprocessor-like scalability and a 5-tuple that comes closer to that of the superscalar.

Superscalar operation-operand matching Out-of-order superscalars achieve operation-operand matching via the instruction window and result buses of the processor's operand network. 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 a functional unit loses 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 result buses 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. Thus, in total, the low-issue superscalars have perfect 5-tuples, i.e., <0,0,0,0,0>. Network latencies of a handful of cycles have appeared in clustered superscalar designs.

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 deferral 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. We call the time it takes for an instruction to commit the *commit latency.* 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; two flavors are message passing and shared memory.

Message-passing operation-operand matching For the purposes of discussing message-passing operation-operand matching, we will assume that a dynamic network [5] is being employed. Implementing operation-operand matching using a message-passing style network has two key challenges.

First, nodes need a processor-network interface that allows low-overhead sends and receives of operands. In an instruction-mapped interface, special send and receive instructions are used for communication; in a register-mapped interface, special register names correspond to communication ports. Using either 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 specifying this information. 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 send to only 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 send and receive occupancies.

Second, receiving nodes need to be able to gather incoming operands and match them with the appropriate instruction. Because timing variances due to I/O, cache misses, and interrupts can delay nodes arbitrarily, one cannot assume a set arrival order for operands sent over the dynamic network. Thus, a tag will also need to be sent along with each operand. When the operand arrives at the destination, it needs to be demultiplexed and delivered to the correct instruction. Conventional message-passing implementations typically would have to do this in software, creating a considerable receive occupancy. Section 6 of this paper measures the performance impact of performing this operation in software. In that section, we estimate the 5-tuple of this message-passing implementation of a scalar operand network as <3,3+c,1,1,12> with c being the commit latency of the processor.

Shared-memory operation-operand matching On a shared-memory multiprocessor, one could imagine implementing operation-operand matching by implementing a large software-managed operand buffer 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 when executing loops, a base register could be incremented on loop iteration. 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, then returns the written value. Every so often, all of the processors would synchronize so that they can reuse the operand 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 occupancies of this approach are 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 on the number of outstanding loads and stores. The send latency will be the latency of a store instruction plus the commit latency. The receive latency would include the delay of the load 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. In Section 6, we estimate the 5-tuple of this relatively aggressive shared-memory scalar operand network implementation to be <1,14+c,2,14,1>.

Systolic array operation-operand matching Systolic machines like iWarp [6] were some of the first systems to achieve low-overhead operation-operand matching in large-scale systems. iWarp sported register-mapped communication, although it 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 operation-operand matching. It needed only execute an instruction to change the current input 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 it 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 must incur the commit latency cost before it releases data into the network.

5. Handling Exceptional Events Exceptional events, despite not being the common case, tend to occupy a fair amount of 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?

Having explored some of the challenges of implementing scalar operand networks, we now look at the Raw scalar operand network and how it addresses these five challenges.

5. IMPLEMENTATION OF THE RAW SCALAR OPERAND NETWORK

This section describes the scalar operand network of the Raw microprocessor. We received one hundred and twenty of the .15 micron, 6-layer Cu, 330 mm^2, 1657 pin, 16-tile Raw prototypes from IBM's fabrication facility in October 2002.

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-style 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 connected to nearest neighbors using 4 separate networks, two static and two dynamic. These networks consist of over 1024 wires per tile. The static router controls 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.

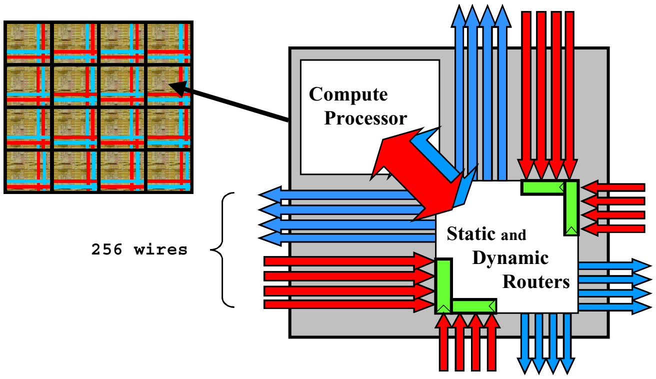


Figure 8: The Raw microprocessor.

The Raw processor handles the *delay scalability* challenge through tiling. Each tile is sized so that a signal can travel through a small amount of logic and across the tile in 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. Modulo building a good clock tree, we do not have to worry about the frequency decreasing as we add more tiles.

Raw's scalar operand network addresses the *bandwidth scalability* challenge by replacing buses with a point-to-point mesh interconnect. Because Raw's point-to-point static network is programmed to route operands only to those tiles that need them, the bandwidth required for operand transport is decimated relative to a comparable bus implementation.

Raw's scalar operand-network achieves *efficient operation-operand matching* through the combination of the static network, an intelligent compiler, and bypass-path integrated network interfaces.

The static network is used to route the outputs of instructions on one tile to the inputs of the dependent instructions on other tiles. It provides single-cycle-per-hop latencies and can route two values in each direction per cycle. If an instruction output is required on the same tile, then the 0-cycle latency internal bypass paths are used. Live but not active values can be stored in the static router register file, compute-processor register file, or in the FIFOs of the network itself.

The 5-stage static router controls two routing crossbars and thus two physical networks. Each crossbar routes values between seven entities – north, east, south, west, the static router pipeline, the compute processor, and the other crossbar – and supports multicast. Each cycle, the static router fetches a 64-bit instruction word from an 8 K-entry instruction cache. 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 uses its register file to recast data values into the compute processor in the event that they are required multiple times.

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 they 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. 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. This ahead-of-time knowledge of routes could enable implementations of the static network that have lower latencies and higher frequencies than the equivalent dynamic network.

The static router is flow controlled. It will not proceed to the next instruction until all of the routes in a particular instruction have completed. This behavior 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 Raw tile compute processor register maps all of Raw's networks in order to interface with the on-chip networks with minimal latencies and occupancies. Because the static network has been pre-programmed with the instructions for routing the data, sender compute processors do not have to specify destinations, and receivers do not have to demultiplex incoming operands. Thus Raw's send and receive occupancies are zero, and the per-hop cost is one cycle. Because a message must go through the local switch on a route, the send latency due to the network is 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 integration makes the network ports truly first-class citizens in the architecture. Figure 9 shows the bypass-integrated network ports. Registers 24 through 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 that allows an instruction to specify two outputs: one network or register and the network implied by r24 (the first static network). This feature gives the tile the option of keeping local copies of transmitted values.

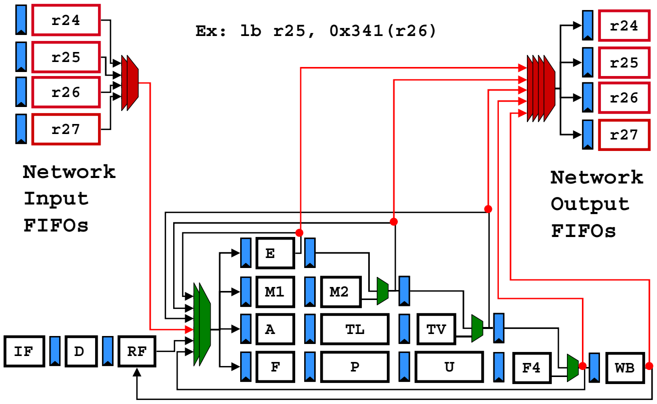


Figure 9: Direct integration of the network interfaces into the bypass paths of the compute processor.

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 [6]. This optimization 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.

Because of the early-commit optimization, the compute processor incurs no additional send latency (i.e., the commit latency $c = 0$). There is one cycle of receive latency because the receive FIFOs are accessed in the dispatch stage of the processor. The valid bits of these FIFOs are also required for the stall logic of the pipeline. This cycle of receive latency could be eliminated as with a superscalar if we route the valid bits one cycle ahead of the data bits in the network.

The combination of static router, compiler, 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 the static network satisfies the *deadlock and starvation* challenge.

Raw's operand network also supports *exceptional events*, the final challenge. 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. 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 static network. 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 [19] and [21]. Details on how the compiler programs the static switch are available in [12].

6. RESULTS

This section presents results on the impact of scalar operand network properties on performance. We focus our evaluation on the 5-tuple cost model. First, we consider the impact each element of the 5-tuple has on performance. We find that occupancy costs have the highest impact. Then, we consider the impact of some factors not modeled directly in the 5-tuple. We find that the impact of contention and multicast on performance is not significant, which reassures us that our cost model successfully captures first order effects. Finally, we use our model to study operand networks based on traditional multiprocessor communication mechanisms. Though these mechanisms are scalable, we show that they do not provide adequate performance for operand delivery – which in turns justifies the study of scalable scalar operand networks as a distinct area of research.

Most of our experiments focus on results for 64 tiles. We have performed all our experiments for 32 tiles as well. The results are qualitatively similar, and due to space limitations we do not present them here.

Experimental setup Our apparatus includes a simulator, a memory model, a compiler, and a set of benchmarks.

Simulator Our experiments were performed on Beetle, a validated cycle-accurate simulator of the Raw microprocessor [19]. In our experiments, we used Beetle to simulate up to 64 tiles. Data cache misses are modeled faithfully; they 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.

Beetle has two modes: one mode simulates the Raw prototype's actual static scalar operand network; the other mode simulates a parameterized scalar operand network based on the 5-tuple cost model. The parameterized network correctly models the occupancy and latency costs, but it does not model contention. The network maintains an infinite buffer for each destination tile, so an operand arriving at its destination buffer is stored until an ALU operation consumes it.

Memory model The Raw compiler maps each piece of program data to a specific home tile. This home tile becomes the tile that is responsible for caching the data on chip. The distribution of data to tiles is provided by Maps, Raw's compiler managed memory system [4, 11]. Using compiler analysis, Maps attempts to select a distribution that guarantees that any load or store refers to data corresponding to exactly one home tile. Dense matrix arrays, for example, usually get distributed element-wise across the tiles. The predictability of the accesses allows memory values to be forwarded from the caches of the home tiles to other tiles using the static operand network, which is much faster than the analogous dynamic mechanism. See [4] for details.

	2	4	8	16	32	64
cholesky	1.622	3.234	5.995	9.185	11.898	12.934
vpenta	1.714	3.112	6.093	12.132	24.172	44.872
mxm	1.933	3.731	6.207	8.900	14.836	20.472
fpppp-kernel	1.511	3.336	5.724	6.143	5.988	6.536
sha	1.123	1.955	1.976	2.321	2.536	2.523
swim	1.601	2.624	4.691	8.301	17.090	28.889
jacobi	1.430	2.757	4.953	9.304	15.881	22.756
life	1.807	3.365	6.436	12.049	21.081	36.095

Table 1: Performance of Raw's static scalar operand network <0,1,1,1,0> for two to 64 tiles. Speedups are relative to that on one tile.

Compiler Code is generated by Rawcc, the Raw parallelizing compiler [12]. 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.

Before scheduling, 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.

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

Even though the Raw hardware has zero receive occupancy cost, Rawcc inserts a move instruction to buffer an incoming operand that is needed more than once in a destination tile. In this select case, because the operand is not consumed directly from the network, the receive occupancy is one. For convenience, we will continue to use <0,1,1,1,0> to represent Raw's static scalar operand network.

Benchmarks The benchmarks we used for our experiments are Cholesky, Vpenta, Mxm, Swim, Fpppp-kernel, Sha, Jacobi and Life. Cholesky, Vpenta, and Mxm are from Nasa7 of Spec92. Swim is from Spec95. Fpppp-kernel consumes 50% of the run-time of Spec95's Fpppp. Sha is an implementation of Secure Hash Algorithm. Jacobi and Life are from the Raw benchmark suite [3]. Fpppp-kernel and Sha are irregular codes, while the rest are dense matrix codes. The problem sizes of the dense matrix applications have been reduced to cut down on simulation time. To improve parallelism via unrolled loop iterations, we manually applied an array reshape transformation to Cholesky, and a loop fusion transformation to Mxm. Both transformations can be automated.

Impact of each 5-tuple parameter on performance We evaluated the impact of each 5-tuple parameter on performance. We used the Raw static operand network as the baseline for comparison, and we recorded the performance as we varied each individual 5-tuple parameter.

Baseline performance First, we measured the absolute performance attainable with the actual, implemented Raw static operand network with 5-tuple <0,1,1,1,0>. Table 1 shows the speedup attained by the benchmarks as we vary the number of tiles from two to 64. Speedup for a benchmark is computed relative to its execution time on a single tile.

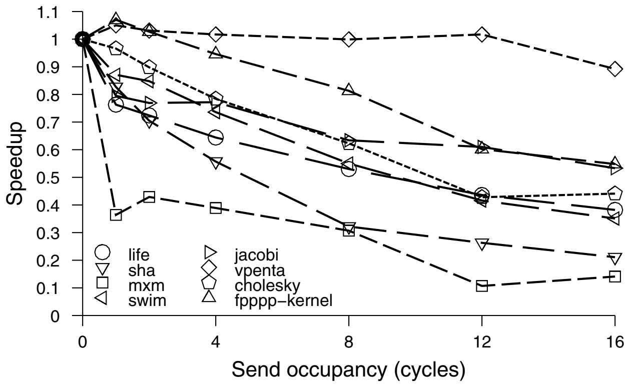


Figure 10: Effect of send occupancy on performance on 64 tiles, *i.e.,* <n,1,1,1,0>.

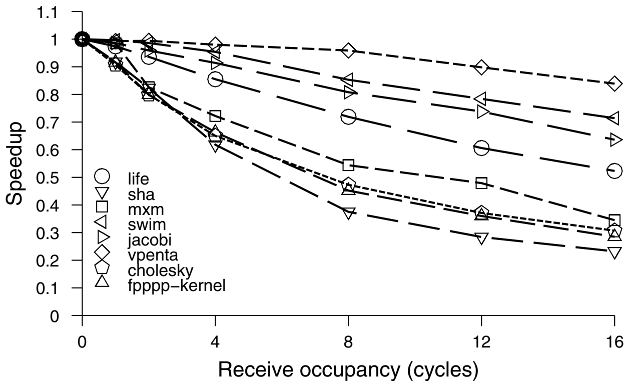


Figure 11: Effect of receive occupancy on performance on 64 tiles, *i.e.,* <0,1,1,1,n>.

All of the benchmarks are able to gain additional performance with additional tiles. On 64 tiles, Sha has the least parallelism and attains a 2.5 fold speedup. Fpppp-kernel and Cholesky contain modest amounts of parallelism and attain 7 and 13 fold speedup, respectively. The remaining five benchmarks (Vpenta, Mxm, Swim, Jacobi, and Life) have enough parallelism for the speedup to scale well up to 64 tiles, with speedups ranging from 20 to 45. The presence of sizable speedup validates our experimental setup for the study of operand networks – without such speedups, it would be moot to explore operand networks that are scalable.

Send and receive occupancies Next, we measured the effects of send and receive occupancies on performance. We emphasize that these occupancies are visible to the compiler so that it can account for them as best as it can when it schedules ILP. Figures 10 and 11 graph these effects. All data points are based on 64 tiles. A performance of 1.0 corresponds to the performance of the actual Raw static operand network with 5-tuple <0,1,1,1,0>.

The trends of the two occupancy curves are similar. Overall, results indicate that occupancy impacts performance significantly. Except for the anomaly for Mxm explained below, performance drops by as much 25% even when the send occupancy is increased from zero to one cycle. Increasing the send occupancy from one to two yields between a 0% to 10% loss, with the rest of the curves following linear trends. For receive occupancy, the performance curves look like those of the send occupancy shifted by one, with the datapoint for receive occupancy p corresponding to the datapoint for send occupancy $p + 1$. As noted in the experimental setup, the base receive occupancy is somewhere between zero and one, which helps explain this correspondence.

The slope of each curve is primarily a function of the amount of slack in the schedule. Benchmarks with large number of cache misses, such as Vpenta, have higher slacks and are able to better tolerate increase in send occupancies.

The large drop off in performance for Mxm when going from zero to one in send occupancy is due mainly to an issue with register allocation and not the occupancy increase itself. The nature of the accesses in Mxm requires that its loop be unrolled along two dimensions in order for Maps to get predictable memory accesses. This unrolling leads to a large basic block with many live ranges that are difficult to register allocate. The quality of the register allocation is somewhat due to luck, and this factor can have a large effect on performance. It happens that the code for send occupancy one register allocates much better than the code for send occupancy zero.

It is clear from these results that scalar operand networks should implement zero cycle send and receive occupancy.

Send and receive latencies We switched to the parameterized scalar operand network to measure the effects of send and receive latencies on performance. For this experiment, we set the network hop latency to zero, with 5-tuple <0,n,0,0,0> or <0,0,0,n,0>. Due to simulator constraints, the minimum latency we can simulate is one. Note that because the parameterized network does not model contention, the effect of n cycles of send latency is the same as the effect of n cycles of receive latency, so we need not collect data for both. Also, note that each of these 5-tuples also happens to represent an n cycle contention-free crossbar.

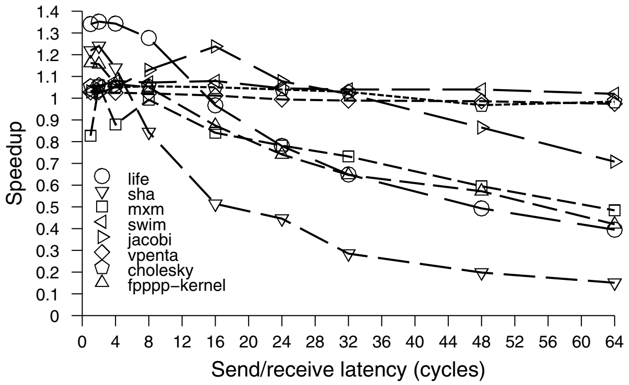


Figure 12: Effects of send or receive latencies on performance on 64 tiles, *i.e.,* <0,n,0,0,0> or <0,0,0,n,0>.

Figure 12 graphs these latency effects on performance on 64 tiles. As with the occupancy experiments, a speedup of 1.0 represents the performance of the Raw static operand network with 5-tuple <0,1,1,1,0>.

Intuitively, we expect benchmark sensitivity to latency to depend on its granularity of available parallelism. The finer the grain of parallelism, the more sensitive the benchmark is to latency. Granularity of parallelism does not necessary correspond to amount of available parallelism: it is possible to have a large amount of fine-grained parallelism, or a small amount of coarse grained parallelism.

We found that the sensitivity to latency varies greatly between the benchmarks. Cholesky, Swim, and Vpenta exhibit “perfect” locality and thus coarse grained parallelism, so they are not sensitive to latency at all. Note that in our compilation framework, perfect locality not only requires that the computation exhibit good locality, but that memory accesses also exhibit good locality. The rest of the benchmarks have finer grained parallelism and incur slowdown to some degree. Sha, which has the least amount of parallelism, also has the finest grained parallelism and suffers the worst slowdown.

There is an anomaly with Jacobi for latency eight and 16, where the application actually speeds up as latency increases. This behavior is due to a factor not accounted for in the compiler. The compiler schedules each scheduling region individually, assuming that they all start and end at the same time, but the actual execution of scheduling regions may overlap. It just happens that the schedules for latency eight and 16 overlap much better than the schedules with lower latency.

Overall, we observe that benchmark performance is less sensitive to send/receive latencies than to send/receive occupancies. Even restricting ourselves to consider only benchmarks that are latency sensitive, a latency of 64 causes about the same degradation of performance as an occupancy of 16.

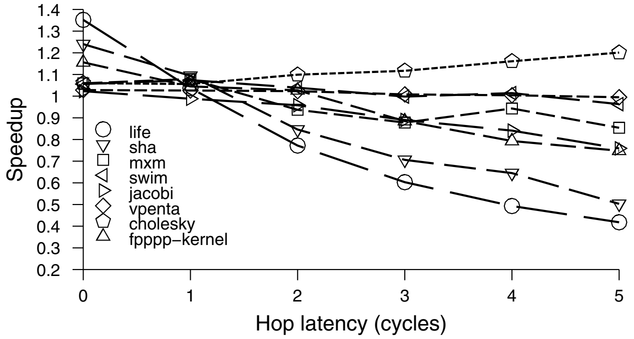


Figure 13: Effects of network hop latency on performance on 64 tiles, *i.e.*, <0,0,n,1,0>.

Network hop latency We also measured the effect of network hop latency on performance. The 5-tuple that describes this experiment is <0,0,n,1,0>, with n varying from zero to five. The range of hop latency is selected to match roughly with the range of latency in the send/receive latency experiment: when hop latency is five, it takes 70 cycles (14 hops) to travel between opposite corner tiles.

Figure 13 shows the result of this experiment. As expected, the graph exhibits the same qualitative properties as that of the send/receive latency experiment. Cholesky, Vpenta, and Swim are not adversely effected by the latency (The Cholesky speedup anomaly is due to the same effect that caused Jacobi to speed up with send/receive latency). The rest of the benchmarks incur varying degree of slowdowns.

Summary These experiments indicate that the most performance critical components in the 5-tuple for 64 tiles are the send and receive occupancies, followed closely by the per-hop latency, followed more distantly by send and receive latencies. The 5-tuple framework gives structure to the task of reasoning about the tradeoffs in the design of scalar operand networks.

Impact of other factors on performance We now consider some factors not accounted for in our 5-tuple cost model: network contention and multicast.

Contention We measured contention by comparing the performance of the actual Raw static scalar operand network with the performance of the parameterized operand network with the same 5-tuple: <0,1,1,1,0>. The former models contention while the latter does not. Figure 14 plots this comparison. Each data point is a ratio of the execution time of the realistic network over that of the idealized contention-free network. Thus, we expect all data points to be 1.0 or above.

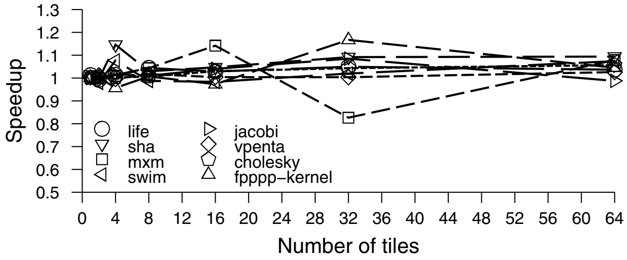


Figure 14: Impact of removing contention.

The figure shows that the cost of contention is modest and does not increase much with number of tiles. On 64 tiles, the average cost of contention is only 5%. The anomaly with Mxm is caused by the register allocation nonlinearity explained earlier.

Multicast Raw's scalar operand network supports multicast, which reduces send occupancy and makes better use of network bandwidth. We evaluated the performance benefit of multicast on the Raw static network for 64 tiles. For the case without multicast, we assume the existence of a broadcast mechanism that can transmit control flow information over the static network with reasonable efficiency.

Somewhat surprisingly, we find that benefit of multicast is small. On 64 tiles, Cholesky benefits by 5%, Mxm by 1%. The rest of the applications do not benefit at all.

Summary The results of the two experiments in this section provide some validation that our 5-tuple cost model is useful. It suggests that effects not modeled directly by the 5-tuple are not significant, and that the model captures to first order the performance of operand networks.

Performance of multiprocessor-based operand networks Using the 5-tuple cost model, we model various realistic operand networks based on traditional multiprocessor communication mechanisms, and we analyze how our benchmarks perform on those networks. These results motivate the differences in performance requirements between scalar operand networks and multiprocessor networks.

Dynamic operand network We first derive the 5-tuple parameters for an operand network based on traditional message passing via a dynamic network. To derive our parameters, we design an implementation of an operand network using Raw's dynamic network. Note that the Raw dynamic network is already more suitable to operand delivery than traditional dynamic networks because of its low latency, but it still lacks direct support for operation-operand matching and efficient deadlock avoidance. As described below, our implementation performs operation-operand matching in software, but it does not account for the cost of deadlocks, which is often significant. Thus, the 5-tuple parameters in this study are likely to be far better than those of existing implementations.

We implement operand delivery on the dynamic network as follows. An operand message includes three words: the message header with the addressing information, an operand tag, 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, timing variations associated with cache misses and interrupts can cause operands to be delivered out of order. To handle the case of unexpected operands, the receiver performs the following: upon receiving an operand, it checks the tag of the operand, and it buffers the operand in memory if the tag does not match with the expected tag. Before checking the network, the receive code first checks this buffer to see if the operand has already been buffered.

The 5-tuple parameters are computed as follows. The send occupancy is the time it takes to send the operand message: three cycles. Assuming cache hits, the receive occupancy is 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, for a total of 12 cycles. We use the send plus receive latency cost to capture the component of the latency that is independent of the number of hops. This latency includes the same latency incurred by the static network (two cycles), as well as the number of "turns" a message takes, which is always either one or two. Thus, send plus receive latency is either three or four. The network hop latency is one.

We define two 5-tuples to bound the performance of this implementation. RawDynSlow uses all maximum parameter values, while RawDynFast uses all minimum parameter values. They are <3,3+c,1,1,12> and <3,2+c,1,1,7>, respectively. Recall that c is the commit latency.

Dynamic operand network with hardware support We next consider a dynamic operand network with additional hardware support to eliminate occupancy costs. To eliminate send occupancy, we can provide a register-mapped network interface, with the operand tag and operand destinations encoded in the instruction itself. On the receive side, messages can be received into dedicated hardware buffers that are indexed by the message tag. This buffer eliminates the receive occupancy, but it does add one cycle of receive latency. In actual use, the buffer must also be flushed periodically to allow reuse of message ids, but we assume that this cost is negligible. We name this implementation RawDyn+Hard, with 5-tuple <0,2+c,1,2,0>.

Shared memory operand network For an on-chip shared memory multiprocessor, we estimate the cost of a software-managed operand buffer discussed in Section 4. We make the most aggressive assumptions to get a bound on the best performance possible. We assume that values in the operand buffer are communicated through the L2 caches, likely the fastest available communication between tiles. We model operand delivery as having the equivalent latency of two cache misses, one on the sender side and one on the receive side. We assume that a cache miss takes 14 cycles, which is commensurate with the latency of L2 cache misses for chip-multiprocessors today.

The 5-tuple for this operand network is computed as follows. Sends use a store instruction, with occupancy one. Receives use a load instruction, also with occupancy one. The sends and receive latencies are modeled as cache misses, with latency 14. Because operand delivery involves both a request and a reply through the network, the net network hop latency is two to account for the round-trip. Thus, the 5-tuple is <1,14+c,2,14,1>. We use SMP to refer to this network and its 5-tuple.

Analysis Figure 15 shows the average benchmark performance of the operand networks on 64 tiles. The figure includes the four networks discussed in this section, with the optimistic assumption that the commit latency is zero. Since all these networks are modeled without contention, we use the Raw scalar operand network without contention as baseline, and we normalize all performance bars to that baseline. For reference, the figure also includes the performance of RawActual, the actual Raw static network, and UnitCrossbar, a single-cycle, contention-free crossbar with 5-tuple <0,0,0,1,0>. Note that RawActual is the only configuration that models contention. Our results show that traditional communication mechanisms are inefficient for scalar operand delivery. Operand delivery via dynamic network has high occupancy costs, while operand delivery via shared memory has high latency costs. The average performance is 39% to 53% worse than the baseline. With additional hardware support for operation-operand matching, we can eliminate the large occupancies of a dynamic operand network, reducing the performance gap to 4% relative to baseline.

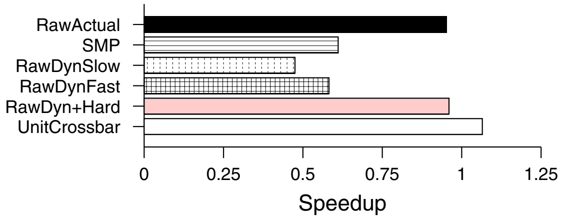


Figure 15: Performance spectrum of operand networks.

7. RELATED WORK

Table 2 contrasts a number of commercial and research systems and the approaches that they use to overcome the challenges of implementing scalable scalar operand networks. We employ tabular form rather than sentence form to facilitate easy comparisons of the systems. The first section of the table summarizes the 5-tuples, and the number of ALUs and fetch units supported in each of the scalar operand networks. The second, third, and fourth sections give the way in which the networks address the delay scalability, bandwidth scalability, and operation/operand matching challenges, respectively. The solutions to the deadlock and exceptions challenges are not listed because they vary between implementations of superscalars, distributed shared memory machines and message passing machines, and they are not specified in full detail in the Grid and ILDP papers. Note that the ILDP and Grid papers examine a range of estimated costs; more information on the actual costs will be forthcoming when those systems are implemented.

This paper extends an earlier framework for operand networks that was given in [20].

8. 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

	Superscalar	Distributed Shared Memory	Message Passing	Raw	Grid	ILDP
Tuple (c = commit time)	<0,0,0,0,0>	<1,14+c,2,14,1>	<3,3+c,1,1,12>	<0,1,1,1,0>	<0,0,n/8,0,0> $0 \leq n \leq 8$	<0,n,0,1,0> n = 0, 2
# ALUs	4	Many	Many	4x4 to 32x32	8x8	8
# Fetch units for N Tiles	1	N	N	N	1	1
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 ordering	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

Table 2: Survey of scalar operand networks

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 several families of scalar operand networks based on this 5-tuple. Our early results show that send and receive occupancies have the biggest impact on performance. For our benchmarks, performance decreases by up to 25 percent even if the occupancy on the send or receive side increases by just 1 cycle. The per-hop latency follows closely behind in importance. Other parameters such as send and receive latencies, presence of multicast, and network contention have smaller impact.

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 processors.

Avenues for further research on scalar operand networks are plentiful. 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 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) mechanisms for fast exception handling and context switching, (6) a thorough analysis of the tradeoffs between commit point, exception handling capability, and send latency, and (7) low energy scalar operand networks.

[1] Agarwal, Hrishikesh, Keckler, and Burger. Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. In *International Symposium on Computer Architecture* (2000).

[2] Arvind, and Brobst. The Evolution of Dataflow Architectures from Static Dataflow to P-RISC. *International Journal of High Speed Computing 5*, 2 (June 1993).

[3] Babb et al. The RAW Benchmark Suite: Computation Structures for General Purpose Computing. In *IEEE Symposium on Field-Programmable Custom Computing Machines* (1997).

[4] Barua, Lee, Amarasinghe, and Agarwal. Maps: A Compiler-Managed Memory System for Raw Machines. In *International Symposium on Computer Architecture* (1999).

[5] Dally. *A VLSI Architecture for Concurrent Data Structures.* Kluwer Academic Publishers, 1987.

[6] Gross, and O'Halloron. *iWarp, Anatomy of a Parallel Computing System.* The MIT Press, Cambridge, MA, 1998.

[7] Gwennap. Digital 21264 Sets New Standard. *Microprocessor Report* (October 28, 1996).

[8] Janssen, and Corporaal. Partitioned Register File for TTAs. In *International Symposium on Microarchitecture* (1996).

[9] Kim, and Smith. An Instruction Set Architecture and Microarchitecture for Instruction Level Distributed Processing. In *International Symposium on Computer Architecture* (2002).

[10] Kubiatowicz, and Agarwal. Anatomy of a Message in the Alewife Multiprocessor. In *International Supercomputing Conference* (1993).

[11] Larsen, and Amarasinghe. Increasing and Detecting Memory Address Congruence. In *International Conference on Parallel Architectures and Compilation Techniques* (2002).

[12] Lee et al. Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine. In *Conference on Architectural Support for Programming Languages and Operating Systems* (1998).

[13] Mai, Paaske, Jayasena, Ho, Dally, and Horowitz. Smart Memories: A Modular Reconfigurable Architecture. In *International Symposium on Computer Architecture* (2000).

[14] Naffziger, and Hammond. The Implementation of the Next-Generation 64b Itanium Microprocessor. In *IEEE International Solid-State Circuits Conference* (2002).

[15] Nagarajan, Sankaralingam, Burger, and Keckler. A Design Space Evaluation of Grid Processor Architectures. In *International Symposium on Microarchitecture* (2001).

[16] Palacharla, Jouppi, and Smith. Complexity-Effective Superscalar Processors. In *International Symposium on Computer Architecture* (1997).

[17] Scale. http://www.cag.lcs.mit.edu/scale/overview.html/, 2002.

[18] Sohi, Breach, and Vijaykumar. Multiscalar Processors. In *International Symposium on Computer Architecture* (1995).

[19] Taylor. The Raw Processor Specification. ftp://ftp.cag.lcs.mit.edu/pub/raw/documents/RawSpec99.pdf.

[20] Taylor, Lee, Frank, Agarwal, and Amarasinghe. How to Build Scalable On-Chip ILP Networks for a Decentralized Architecture. Tech. Rep. 628, MIT, April 2000.

[21] Taylor et al. The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Prgrams. *IEEE Micro* (March 2002).