Scalar Operand Networks

Michael Bedford Taylor, Walter Lee, Saman P. Amarasinghe, *Member*, *IEEE*, and Anant Agarwal, *Member*, *IEEE*

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 toward 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. 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 latency (a few cycles versus tens of cycles) and ultra-fast operation-operand matching. This paper discusses the unique properties of scalar operand networks (SONs), examines alternative ways of implementing them, and introduces the AsTrO taxonomy to distinguish between them. It discusses the design of two alternative networks in the context of the Raw microprocessor, and presents timing, area, and energy statistics for a real implementation. The paper also presents a 5-tuple performance model for SONs and analyzes their performance sensitivity to network properties for ILP workloads.

Index Terms—Interconnection architectures, distributed architectures, microprocessors.

1 Introduction

Today's wide-issue microprocessor designers are finding it increasingly difficult to convert burgeoning silicon resources into usable, general-purpose functional units. At the root of these difficulties are the centralized structures responsible for orchestrating operands between the functional units. These structures grow in size much faster asymptotically than the number of functional units. The prospect of microprocessor designs for which functional units occupy a disappearing fraction of total area is unappealing but tolerable. Even more serious is the poor frequency-scalability of these designs; that is, the unmanageable increase in logic and wire delay as these microprocessor structures grow [1], [16], [30]. 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].

The scalability problem extends far beyond functional unit bypassing. Contemporary processor designs are pervaded by unscalable, global, centralized structures. As a result, future scalability problems lurk in many of the components of the processor responsible for 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.

Seeking to scale ILP processors, recent microprocessor designs in industry and academia reveal a trend toward 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 [26], [28] and SCALE, Stanford's Smart Memories [13], Wisconsin's ILDP [8] and Multiscalar [20], Washington's WaveScalar [23] and the 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. 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 (SON*[1]*) 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 (referred to hereafter as *the transport network*) as well as the operation-operand matching system (hardware or software) that coordinates these values into a coherent computation. This choice of definition parallels the use of the word "Internet" to include both the physical links and the protocols that control the flow of data on those links.

SONs can be designed to have short wire lengths and to scale with increasing transistor counts. Furthermore, because they can be designed around generalized transport networks,

- *The authors are with CSAIL 32-G742, 32 Vassar Street, Cambridge, MA 02139. E-mail: mbt@mit.edu, {walt, agarwal, saman}@cag.lcs.mit.edu.*

Manuscript received 22 Feb. 2004; revised 15 July 2004; accepted 19 July 2004.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDSSI-0054-0204.

[1] Pronounced ess-oh-en.

1045-9219/05/$20.00 © 2005 IEEE Published by the IEEE Computer Society

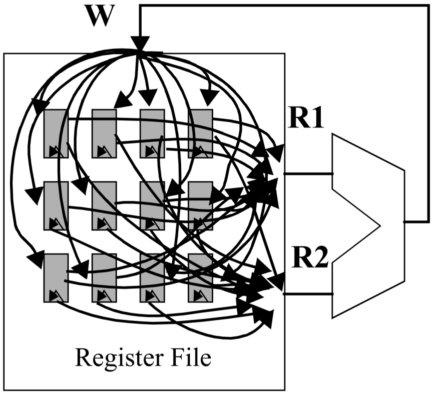


Fig. 1. A Simple SON.

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 SONs can successfully exploit fine-grained ILP. Lee et al. [11] showed that a compiler can successfully schedule ILP on a partitioned architecture that uses a static point-to-point transport 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 transport 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 SONs in microprocessors is also important. Although these SONs share many of the challenges in designing message passing systems also physical interconnect networks, such as scalability and deadlock avoidance, they have many unique requirements including ultra-low latency (a few cycles versus tens of cycles) and ultra-fast operation-operand matching. This paper identifies five important challenges in designing SONs, and develops the AsTrO taxonomy for describing their logical properties. The paper also defines a parameterized 5-tuple model that quantifies performance trade offs in the design of these networks. To show that large-scale low-latency SONs are realizable, we also describe the details of the actual 16-way issue SON designed and implemented in the Raw microprocessor, using the 180 nm IBM SA-27E ASIC process.

One concrete contribution of this paper is that we show sender and receiver occupancy have a first-order impact on ILP performance. In contrast, the performance loss due to network transport contention averages only 5 percent for a 64-tile Raw mesh. These results lead us to conclude that whether the network transport is static or dynamic is less important (at least for up to 64 nodes) than whether the SON offers efficient support for matching operands with the intended operations.

The paper proceeds as follows: Section 2 provides background on scalar operand networks and their evolution. Section 3 describes the key challenges in designing scalable SONs; these challenges derive from a combined ancestry of multiprocessor interconnects and primordial uniprocessor bypass networks. Section 4 discusses operation-operand matching. Section 5 introduces a taxonomy for the logical structure of SONs. Section 6 describes the design of two SONs. Section 7 discusses Raw's VLSI SON implementation. Section 8 quantifies the sensitivity of ILP performance to network properties. Section 9 presents related work and Section 10 concludes this paper.

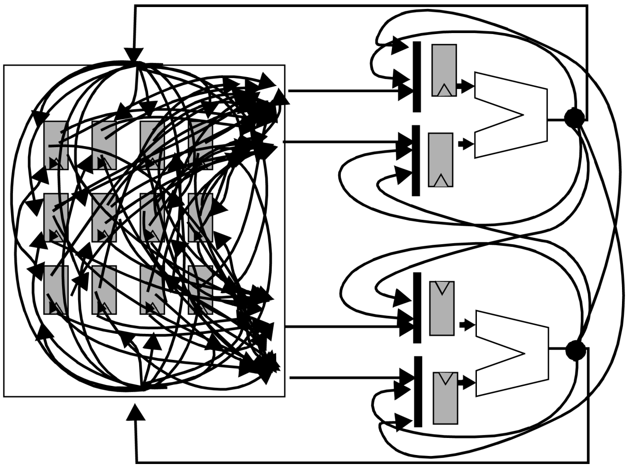


Fig. 2. A pipelined processor with bypass links and multiple ALUs.

2 Evolution of Scalar Operand Networks

The role of an SON is to join the dynamic operands and operations of a program in space to enact the computation specified by a program graph. This section describes the evolution of SONs—from early, monolithic register file interconnects to more recent ones that incorporate routed point-to-point mesh interconnects.

A nonpipelined processor with a register file and ALU contains a simple specialized form of an SON. 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 SON.

Fig. 1 emphasizes the role of a register file as a device capable of performing two parallel routes from any two internal registers to the output ports of the register file, and one route from the register file input to any of the internal registers. Each arc in the diagram represents a possible operand route that may be performed each cycle. This interconnect-centric view of a register file becomes increasingly appropriate as wire delays worsen in VLSI processes.

Fig. 2 shows a pipelined, bypassed processor with multiple ALUs. Notice that the SON 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 SON. First, there is the temptation to support out-of-order issue of instructions, which forces the SON to deal with the possibility of having several live aliases of the same register name. Adding register renaming to the SON allows the network to manage these live register aliases. Perhaps more significantly, register renam-

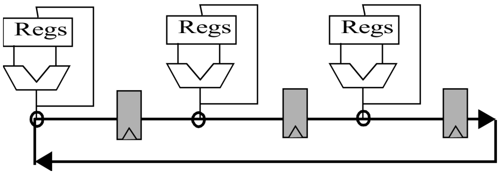


Fig. 3. Multiscalar's SON.

ing 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.

More generally, increasing the number of functional units necessitates more live values in the SON, 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 SONs that employ centralized register files as operand interconnect.

One solution is to partition and distribute the interconnect and the resources it connects. Fig. 3 depicts the partitioned register file and distributed ALU design of the Multiscalar—one of the early distributed ILP processors. This design uses a one-dimensional multihop ring SON to communicate operands to successive ALUs.

Fig. 4 shows the two-dimensional point-to-point SON in the Raw microprocessor. Raw implements a set of replicated tiles and distributes all the physical resources: FPUs, 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 an individual node in Grid, and an individual pipeline in Multiscalar, Raw, or ILDP.

Mapping ILP to architectures with distributed scalar operand networks is not as straightforward as with early, centralized architectures. ILP computations are commonly expressed as a dataflow graph, where the nodes represent operations, and the arcs represent data values flowing from the output of one operation to the input of the next. To execute an ILP computation on a distributed-resource microprocessor containing an SON, we must first find an *assignment* from the nodes of the dataflow graph to the nodes in the network of ALUs. Then, we need to *route* the intermediate values between these ALUs. Finally, we must make sure that operands and their corresponding operations are correctly matched at the destinations. See [11], [27] for further details on orchestrating ILP in partitioned microprocessors. The next section will address each of these three issues relating to SONs, and discuss how SONs can be built in a scalable way.

3 Challenges in the Design of Scalable Scalar Operand Networks

To make the concept of a scalar operand network more concrete, it is useful to characterize them in terms of some of the key challenges in designing them. Not surprisingly, these five challenges derive from a combined ancestry of multiprocessor interconnections and primordial uniprocessor bypass networks. Thus, for those with a multiprocessor or network background, the challenges of frequency scalability, bandwidth scalability, and deadlock may seem particularly familiar. On the other hand, efficient operation-operand matching and the challenge of exceptional conditions may seem more familiar to those with a microprocessor background.

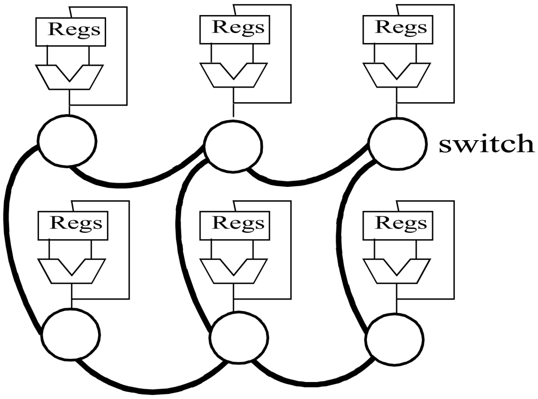


Fig. 4. SON based on a 2D point-to-point routed interconnect.

3.1 Efficient Operation-Operand Matching

Operation-operand matching is the process of gathering operands and operations to meet at some point in space to perform the desired computation.

The focus on efficient operation-operand matching is the fundamental difference between SONs and other network types. If operation-operand matching cannot 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 challenge of efficient operation-operand matching is so fundamental to SONs, we examine it in detail in Section 4.

3.2 Frequency Scalability

Frequency scalability describes the ability of a design to maintain high clock frequencies as that design scales. When an unpipelined, two-dimensional VLSI structure increases in area, relativity dictates that the propagation delay of this structure must increase asymptotically at least as fast as the square root of the area. Practically speaking, the increase in delay is due to both increased interconnect[2] delay and increased logic levels. If we want to build larger structures and still maintain high frequencies, there is no option except to pipeline the circuits and turn the propagation delay into pipeline latency.

3.2.1 *Intracomponent Frequency Scalability*

As the issue-width of a microprocessor increases, monolithic structures such as multiported register files, bypassing logic, selection logic, and wakeup logic grow linearly to cubically in size. Although extremely efficient VLSI implementations of these components exist, their burgeon-

[2] We use this term loosely to refer to the set of wires, buffers, multiplexers, and other logic responsible for the routing of signals within a circuit.

ing size guarantees that intracomponent 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.

Classical solutions to frequency scalability problems incorporate two themes: partitioning and pipelining. Several recently proposed academic microprocessors [8], [13], [15], [20], [26] (and current-day multiprocessors) combine these approaches and compose their systems out of replicated *tiles*. Tiling simplifies the task of reasoning about and implementing frequency 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 tile boundaries are periodically registered so that the cycle time is not impacted. In effect, tiling ensures that the task of reasoning about frequency scalability need only be performed at the intercomponent level.

3.2.2 Intercomponent Frequency Scalability

Frequency 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 and is now becoming a central issue in microprocessor designs. Two recent examples of commercial architectures addressing intercomponent delay are 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 has a one cycle latency cost for results from one integer cluster to be used by the other cluster. 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.

3.2.3 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 trade off 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 nonuniform 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. Appendix 1 (which can be found online at http://www.computer.org/tpds/archives.htm) examines the performance benefit of placing communicating instructions nearby.

3.3 Bandwidth Scalability

While many frequency scalability problems can be addressed by distributing centralized structures and pipelining paths between distant components, there remains a subclass of scalability problems which are fundamentally linked to an SON's underlying architectural algorithms. These *bandwidth scalability* problems occur when the amount of information that needs to be transmitted and processed grows disproportionately with the size of the system.

The bandwidth scalability challenge is making its way from multiprocessor to microprocessor designs. One key red flag of a nonbandwidth scalable architecture is the use of broadcasts that are not directly mandated by the computation. For example, superscalars often rely on global broadcasts to communicate the results of instructions. Because every reservation station output must be processed by every reservation station, the demands on an individual reservation station grow as the system scales. Although, like the Alpha 21264, superscalars can be pipelined and partitioned to improve frequency scalability, this is not sufficient to overcome the substantial area, latency, and energy penalties due to poor bandwidth scalability.

To overcome this problem, we must find a way to decimate the volume of messages sent in the system. We take insight from directory-based cache-coherent multiprocessors which tackle this problem by employing directories to eliminate the broadcast inherent in snooping cache systems. Directories are 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. The resulting reduction in necessary bandwidth allows the broadcast network to be replaced with a point-to-point network of lesser bisection bandwidth.

A directory scheme is one candidate for replacing broadcast in an SON and achieving bandwidth scalability. 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.

In order to be bandwidth scalable, such a directory must be implemented in a distributed, decentralized fashion. There are a number of techniques for doing so. If the system can guarantee that every active dynamic instance of an instruction is always assigned to the same node in the SON, it can store or cache the directory entry at the source node. The entry could be a field in the instruction encoding which is set by the compiler, or it could be an auxiliary data structure dynamically maintained by the architecture or runtime system. The static mapping scheme is quite efficient because the lookup of directory entry does not incur lengthy communication delays. We call architectures [6], [7], [15], [26] whose SONs assign dynamic instances of the same static instruction to a single node *static-assignment architectures*. Static-assignment architectures avoid broadcast and achieve bandwidth scalability by implementing point-to-point SONs.[3]

[3] Nonbroadcast microprocessors can use any point-to-point SON. We leave as interesting open research the relative merits of specific point-to-point SON topologies such as direct meshes, indirect multistage networks, or trees.

In contrast, *dynamic-assignment* architectures like superscalars and ILDP assign dynamic instruction instances to different nodes in order to exploit parallelism. In this case, the removal of broadcast mechanisms is a more challenging problem to address because the directory entries need to be constantly updated as instructions move around. ILDP decimates broadcast traffic by providing intranode bypassing for values that are only needed locally; however, it still employs broadcast for values that may be needed by other nodes. We believe the issue of whether a scalable dynamic assignment architecture can replace the broadcast with a multicast using a distributed register file or directory system is an interesting open research question.

3.4 Deadlock and Starvation

Superscalar SONs use relatively centralized structures to flow-control instructions and operands so that internal buffering cannot be overcommitted. With less centralized SONs, such global knowledge is more difficult to attain. If the processing elements independently produce more values than the SON has storage space, then either data loss or deadlock must occur [5], [21]. This problem is not unusual; in fact, some of the earliest large-scale SON 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. Transport-related deadlock can be roughly divided into two categories; endpoint deadlock, resulting from a lack of storage at the endpoints of messages, and in-network deadlock, which is deadlock inside the transport network itself. Because effective solutions for in-network deadlock have already been proposed in the literature, endpoint deadlock is of greater concern for SONs.

3.5 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 SON design will encounter specific challenges based on the particulars of the design. It is likely that cache misses, branch mispredictions, exceptions, interrupts, and context switches will be among those challenges. For instance, how do context switches work on a dynamic transport SON? 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 branch mispredicts and cache miss requests sent on the SON, or on a separate network?

4 Operation-Operand Matching

The existence of mechanisms for efficient operation-operand matching is the key difference between SONs and other network types. Abstractly, operation-operand matching is the fundamental process of gathering operands and operations to meet at some point in space to perform the desired computation. In this paper, we attempt to concretize the process of operation-operand matching with two constructs, the performance 5-tuple and the AsTrO taxonomy. We start with the discussion of the 5-tuple.

Performance 5-tuple. The performance 5-tuple captures the cost of the most basic and essential function of an SON: operation-operand matching. This 5-tuple serves as a simple figure of merit that can be used to compare SON implementations. As the reader will see in the following paragraphs, the 5-tuple demonstrates that networks designed for use as SONs are quantitatively much better at operation-operand matching than other types of networks, such as message-passing or shared-memory systems.

This 5-tuple of costs $<\mathrm{SO}, \mathrm{SL}, \mathrm{NHL}, \mathrm{RL}, \mathrm{RO}>$ consists of:

1. **Send occupancy**: average number of cycles that an ALU wastes in transmitting an operand to dependent instructions at other ALUs.
2. **Send latency**: average number of cycles incurred by the message at the send side of the network without consuming ALU cycles.
3. **Network hop latency**: average transport network hop latency, in cycles, between physically adjacent ALUs.[4]
4. **Receive latency**: average number of cycles between when the final input to a consuming instruction arrives and when that instruction is issued.
5. **Receive occupancy**: average number of cycles that an ALU wastes by using 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 following sections, we examine SONs implemented on a number of conventional systems and describe the components that contribute to the 5-tuple for that system. 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, shared memory, and systolic systems, which have good scalability but poor 5-tuples. The Raw prototype, described in Section 6 and Section 7, falls in between the two extremes, with multiprocessor-like scalability and a 5-tuple that comes closer to that of the superscalar: $<0, 0, 1, 2, 0>$.[5]

[4] When defined this way, 5-tuples for networks whose nodes are embedded uniformly in a given dimensionality of physical space can be directly compared if the internode latencies, in cycles, are roughly proportional to the physical distances between nodes. Unless otherwise specified, 5-tuples are taken to be defined for a 2D mesh.

Our default assumption of 2D packaging is reasonable for on-chip networks in modern VLSI processes, for which 2D node packing minimizes communication latency and for which wire delay is significant in comparison to router delay. This wire delay prevents frequency-scalable hyperdimensional topologies from reducing effective network diameter, in cycles, beyond a constant factor of the underlying physical topology.

Networks with constant communication costs between nodes (e.g., crossbars or multistage networks) can be modeled in our 5-tuple by counting their fixed internode latency as part of the send latency, and setting NHL to 0. Then their 5-tuples can be directly compared to that of all other networks regardless of packaging dimensionality.

[5] In previous work, we used a convention that combined any fixed transport network costs into the SL field. In this paper, we assign the costs more precisely to SL or RL, based on whether the cost is closer to sender or receiver.

4.1 Superscalar Operation-Operand Matching

Out-of-order superscalars achieve operation-operand matching via the instruction window and result buses of the processor's SON. 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 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. Thus, in total, the low-issue superscalars have perfect 5-tuples, i.e., $< 0, 0, 0, 0, 0 >$. We note that network latencies of a handful of cycles have begun to appear in recent clustered superscalar designs.

4.2 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 sender 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*. Barring support for speculative sends and receives (as with a superscalar), the send latency of these networks will be adversely impacted.

The two key multiprocessors communication mechanisms are message passing and shared memory. It is instructive to examine the 5-tuples of these systems. As detailed in [27], the 5-tuple of an SON based on Raw's relatively aggressive on-chip message-passing implementation falls between $<3, 1{+}c, 1, 2, 7>$ and $<3, 2 + c, 1, 2, \ 12>$ (referred to subsequently as *MsgFast* and *MsgSlow*) with c being the commit latency of the processor. A shared-memory chip-multiprocessor SON implementation based on Power4, but augmented with full/empty bits, is estimated to have a 5-tuple of $< 1, 14 + c, 2, 14, 1 >$. For completeness, we also examine a systolic array SON implementation, iWarp, with a 5-tuple of $< 1, 6, 5, 0, 1 >$.

4.3 Message-Passing Operation-Operand Matching

For this discussion, we assume that a dynamic transport network [4] is used to *transport* operands between nodes. 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 must specify the destination(s) of the outgoing 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. Finally, dynamic transport networks typically do not support multicast, so multiple message sends may be required for operands that have nonunit fanout. These factors will impact send and receive occupancy.

Second, receiving nodes must match incoming operands with the appropriate instruction. Because timing variances due to I/O, cache misses, and interrupts can delay nodes arbitrarily, there is no set arrival order for operands sent over dynamic transport. Thus, a tag must be sent along with each operand. When the operand arrives at the destination, it needs to be demultiplexed to align with the *ordering* of instructions in the receiver instruction stream. Conventional message-passing implementations must do this in software [29], or in a combination of hardware and software [12], causing a considerable receive occupancy.

4.4 Shared-Memory Operation-Operand Matching

On a shared-memory multiprocessor, operation-operand matching can be implemented via 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 occupancy 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 is the latency of a store instruction plus the commit latency. The receive latency includes the delay of the load instruction as well as the nonnetwork time required for the cache protocols to process the receiver's request for the line from the sender's cache.

This approach has a 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 [27], we estimate the 5-tuple of this relatively aggressive shared-memory SON implementation to be $< 1, 14 + c, 2, 14, 1 >$.

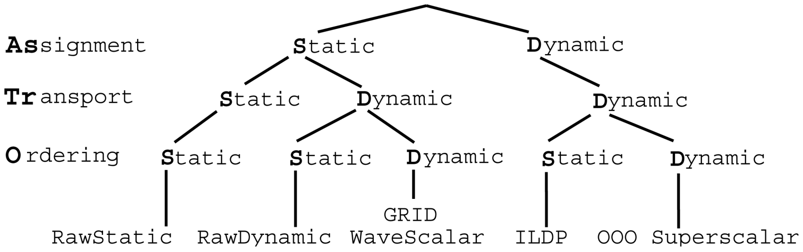


Fig. 5. The AsTrO taxonomy of SONs.

4.5 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 precompiled 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 5-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 one 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 The AsTrO Taxonomy for SONs

In contrast to the 5-tuple, which compares the performance of different SONs, the AsTrO taxonomy captures the key choices in the way SONs manage the flow of operands and operations. Ultimately, these choices will impact the 5-tuple and the cycle time of the SON.

By examining recent distributed microprocessor proposals, we were able to generalize the key differences into the three AsTrO parameters: 1) how operations are **As**signed to nodes, 2) how operands are **Tr**ansported between the nodes, and 3) how operations on the nodes are **O**rdered.

Each of the AsTrO parameters can be fulfilled using a static or dynamic method. Typically, the static methods imply less flexibility but potentially better cycle times, lower power, and better 5-tuples.

An SON uses *dynamic-assignment* if active dynamic instances of the same instruction can be assigned to the different nodes. In *static-assignment*, active dynamic instances of the same static instruction are assigned to a single node. A dynamic assignment architecture attempts to trade implementation complexity for the ability to dynamically load balance operations across nodes. A static assignment architecture, on the other hand, has a much easier task of matching operands and operators. Note that the use of static-assignment does not preclude instruction migration as found in WaveScalar and at a coarser grain, with Raw.

An SON employs *dynamic transport* if the ordering of operands over transport network links is determined by an online process. The ordering of operands across *static transport* links are precomputed. A dynamic transport SON benefits from the ability to reorder operand transmissions between nodes in response to varying timing conditions, for instance, cache-misses or variable latency instructions. Conversely, static transport SONs can prepare for operand routes long before the operand has arrived. More implications of these two properties are discussed in Section 6.

An SON has *static ordering* of operations if the execution order of operations on a node is unchanging. An SON has *dynamic ordering* of operations if the execution order can change, usually in response to varying arrival orderings of operands. A dynamic ordering SON has the potential to be able to reschedule operations on a given node in response to varying time conditions.

Fig. 5 categorizes a number of recent distributed microprocessor proposals into the AsTrO taxonomy. We remark, in particular, that *static assignment* appears to be a key element of all of the current scalable designs. This taxonomy extends that described in [25]. The taxonomy in [19] discusses only operand transport.

6 Operation of the RawDynamic and RawStatic SONs

In this section, we describe RawDynamic, an SDS SON, and RawStatic, an SSS SON. We use the term RawStatic to refer to the SSS SON implemented on top of the hardware resources in the actual Raw processor prototype. This is the SON that the Raw hardware actually uses. RawDynamic refers to an SDS SON implemented with the combination of the existing Raw hardware and a few additional hardware features. These features will be discussed in detail in this section.

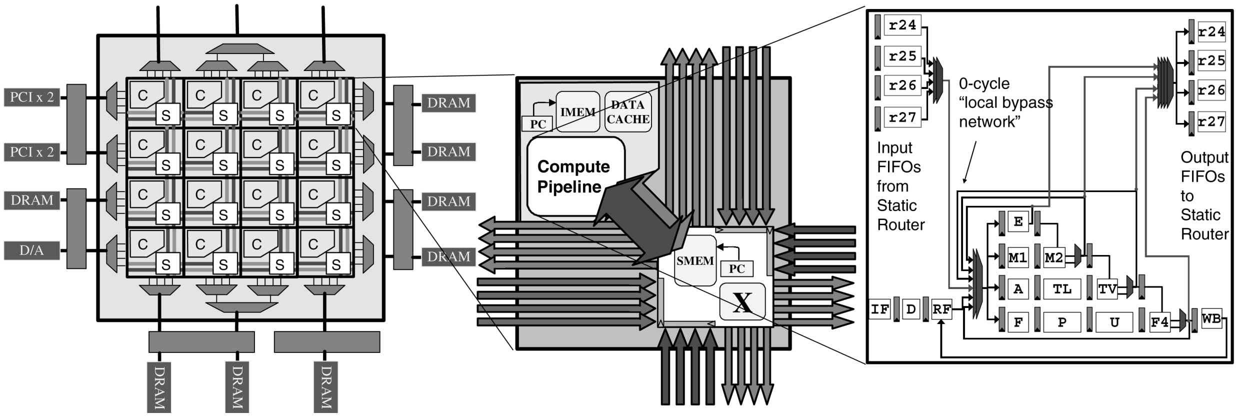


Fig. 6. Architecture of the Raw processor.

Section 7 examines VLSI aspects of RawDynamic and RawStatic.

Commonalities. We discuss the Raw processor, which contains the common components employed by both the RawStatic and RawDynamic SONs. The Raw processor addresses the *frequency scalability* challenge using tiling. A Raw processor is comprised of a 2D mesh of identical, programmable tiles, connected by two types of transport networks. 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. The left and center portions of Fig. 6 show 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, frequency will remain constant as tiles are added. Each Raw tile contains a single-issue in-order compute processor, and a number of routers. In the Raw implementation, the switching portions of the tile (demarkated by the Ss in the left picture) contain two dynamic routers (for two dynamic transport networks); and one static router (for one static transport network). The RawStatic SON employs the static router, while the RawDynamic SON employs one of the dynamic routers; both use a credit system to prevent overflow of FIFOs. Both systems use the second dynamic router for cache-miss traffic; misses are turned into messages that are routed off the sides of the mesh to distributed, off-chip DRAMs.

Both RawDynamic and RawStatic address the *bandwidth scalability* challenge by replacing buses with a point-to-point mesh interconnect. Because all of Raw's point-to-point networks can be 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.

Each of the two SONs relies upon a compiler to assign operations to tiles, and to program the network transports (in their respective ways) to route the operands between the corresponding instructions. Accordingly, both SONs are *static assignment*. Furthermore, because Raw's compute processors are in-order, both RawDynamic and RawStatic can be termed *static ordering* SONs.

Because SONs require low-occupancy, low-latency sends to implement operation-operand matching, Raw employs a register-mapped interface to the transport networks. Thus, the RAW ISA dedicates instruction encoding space in each instruction so that injection of the result operand into the transport network has zero occupancy. In fact, a single 32-bit Raw ISA encoded instruction allows up to two destinations for each output, permitting operands to be simultaneously injected into the transport network and retained locally. Furthermore, to reduce send latency, the Raw processor uses a special inverse-bypass system for both static and dynamic routers. This inverse-bypass system pulls operands from the local bypass network of the processor and into the output FIFOs as soon as they are ready, rather than just at the writeback stage or through the register file [6]. In both cases, the logic must ensure that operands are pulled out of the bypass paths in-order. For the static network, this is because operands must be injected into the network in a known order, and for the dynamic network, because the ordering of words in a message payload must be respected. This interface is shown on the right-hand portion of third component of the Raw processor diagram. Inverse bypassing, combined with an early commit point in the Raw processor, reduces the send latency of operation-operand matching by up to 4 cycles. In effect, we have deliberately designed an early commit point into our processor to eliminate the common multiprocessor communication-commit delay (i.e., $c = 0$) described in the Challenges section.

Register-mapped input FIFOs are used to provide zero-occupancy, unit-latency receives. One cycle of receive latency is incurred because the receive FIFOs are scheduled and accessed in the dispatch stage of the processor. This cycle of receive latency could be eliminated as with a superscalar if the valid bits are routed one cycle ahead of the data bits in the network.

Finally, Raw's use of a zero-cycle bypass for routing operands that a compute processor both produces and uses locally greatly reduces the average cost of operation-operand matching.

Both RawStatic and RawDynamic support *exceptional events*, the final challenge. Branch conditions and jump pointers are transmitted over the transport 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. Tiles that try to use the result of a cache-missing load from another tile will block, waiting for the value to arrive over the transport network. These cache misses are processed over a separate dynamic transport. Raw supports context switches by draining and restoring the transport network contents. This network state is saved into a context block and then restored when the process is switched back in.

6.1 Operation of RawDynamic, an SDS SON

RawDynamic differs from RawStatic because it uses the Raw processor's dynamic dimension-ordered [22] wormhole routed [4] network to route operands between tiles. Thus, it is a *dynamic transport* SON. Dimension-ordered routing is in-network deadlock-free for meshes without end-around connections.

Because RawDynamic uses a dynamic transport network, it needs to extend the existing Raw compute processor ISA by encoding the destination address or offset in the instruction, rather than just the choice of network to inject into. Furthermore, since the instruction may have multiple consumers, it is typical to encode multiple consumers in a single instruction. Finally, because dynamic transport message arrival orderings are impacted by the time at which a message is sent, a system intended to tolerate unpredictable events like cache misses must include a numeric index with each operand that tells the recipient exactly which operand it is that has arrived. Thus, it is likely that dynamic transport SONs will have wider instruction words and wider transport network sizes[6] than the equivalent static transport SONs. Additionally, for efficient implementation of multicast and broadcast operations (such as a jr, or operands with high fanout), the equivalent dynamic transport SON may need more injection ports into the network. In fact, low-occupancy, low-latency broadcast and multicast for dynamic transport networks is an area of active research [17].

We now examine the transport portion of the RawDynamic SON. The Raw dynamic router has significantly deeper logic levels than the Raw static router for determining the route of incoming operands. In our aggressive, speculative implementation, we exploit the fact that most incoming routes in dimension-ordered routers are straight-through-routes. As a result, we were able to optimize this path and reduce the latency of most routes to a single cycle per hop. However, an additional cycle of latency is paid for any turns in the network, and for turns into and out of the compute processor. This overhead, and the fact that a message requires a cycle to go from router to compute processor on the receive side, makes the transport's incremental 5-tuple contribution at least $+ < 0, 1, 1, 2, 0 >$.

The receive end of RawDynamic is where new hardware must be added. Unlike a static transport, the dynamic transport does not guarantee the arrival ordering of operands. The addition of receive-side demultiplexing hardware is required. This hardware uses the operand's numeric index to place the operand into a known location in an incoming operand register file (IRF).[7]

6.1.1 Synchronization and Deadlock

The introduction of an IRF creates two key synchronization burdens. First, receivers need a way to determine whether a given operand has arrived or if it is still in transit. Second, senders need to be able to determine that the destination index is no longer in use when they transmit a value destined for the IRF, or that they are not overflowing the remote IRF.

A simple synchronization solution for the receiver is to employ a set of full/empty bits, one per index; instructions can read incoming operands by 1) specifying the numeric indices of input operands and 2) verifying that the corresponding full-empty bits are set to full. Accordingly, the demultiplexing logic can set the full bit when it writes into the register file. A number of bits in the instruction word could be used to indicate whether the full-empty bit of IRF registers should be reset upon read so that a receiving node can opt to reuse an input operand multiple times. Furthermore, support for changing the full-empty bits in bulk upon control-flow transition would facilitate flow-control sensitive operand use. This set of synchronization mechanisms will allow a dynamic transport SON to address the *efficient operation-operand matching* challenge.

The problem of receiver-synchronization falls under the *deadlock* challenge stated in Section 3. The setting of the full/empty bit is of little use for preventing deadlock because there is the greater issue of storing the operands until they can be written into the IRF. If this storage space is exceeded, there will be little choice but to discard operands or stop accepting data from the network. In the second case, ceasing to accept data from the network often results in the unfortunate dilemma that the compute processor cannot read in the very operand that it needs before it can sequence the instructions that will consume all of the operands that are waiting. Data-loss or deadlock ensues.

6.1.2 Deadlock Recovery

Generally, we observe two solutions for deadlock: deadlock recovery and deadlock avoidance. In recovery, buffer space is allowed to become overcomitted, but a mechanism is provided for draining the data into a deep pool of memory. In avoidance, we arrange the protocols so that buffer space is never overcommitted. Reliance on only deadlock recovery for dynamic transport SONs carries two key challenges: First, there is a performance robustness problem due to the slowdown of operations in a deadlock recovery mode (i.e., greater access times to the necessarily larger memories needed for storing large numbers of quiescent operands). Second, the introduction of deadlock recovery can remove the backward flow-control exercised from receiver to sender. As a result, a high data-rate sender sending to a slower receiver can result in the accumulation of huge pools of operands in the deadlock recovery buffer. In this case, the proper solution is to find a way to notify the sender to limit its rate of production [12]. Nonetheless, deadlock recovery is promising because it may allow higher overall resource utilization than is provided by an avoidance scheme. We believe some of the issues of deadlock recovery mechanisms on dynamic-transport SONs are interesting open research questions. A deadlock recovery mechanism for overcommitted buffers is employed in Alewife [10] and in the Raw general network [26].

6.1.3 Deadlock Avoidance

A deadlock avoidance strategy appears to have a more immediate method of satisfying the synchronization con-

[6] If the transport width is wide enough to send all components of an operand in a single message, inverse-bypassing could be modified to allow operands to be transmitted out-of-order. Nonetheless, for resource scheduling and fairness reasons, it makes sense to prefer older senders to newer senders.

[7] A CAM could also be used, simplifying slot allocation at the risk of increasing cycle time.

straints of dynamic transport SONs. We propose one such strategy in which a lightweight, pipelined one-bit barrier is used among senders and receivers to reclaim compiler-determined sets of IRF slots. The system can be designed so that multiple barriers can proceed in a pipelined fashion, with each barrier command in an instruction stream simultaneously issuing a new barrier, and specifying which of the previous barriers it intends to synchronize against. For very tight loops, the barrier distance will increase to include approximately the number of iterations of the loop that will cause the first of any of the IRFs to fill. As a result, processing can continue without unnecessary barrier stalls. This is essentially a double (or more) buffering system for IRF operand slots.

In order to execute small loops without excessive overhead, it may be desireable to incorporate a mechanism that changes the numeric IRF indices based on loop interation to prevent unnecessary unrolling (see parallels to register renaming).

Overall, we optimistically assume that dispatching of operands from the IRF will have the same latency as the dispatching of operands from RawStatic's input FIFOs, or one cycle. We also assume that the synchronization required to manage the IRF incurs no penalty. Combining this with the transport cost of $+ < 0, 1, 1, 2, 0 >$, the 5-tuple for RawDynamic is RawDynamicTuple.

The recent Flit-Reservation System [18] suggests that dynamic transport routes can be accelerated by sending the header word in advance. This optimization allows routing paths to be computed ahead-of-time so that data words can be routed as soon as they arrive. This system has promise, since a compute pipeline may be able to send out the header as soon as it can determine that the instruction will issue, setting up the route as the result operand is being computed. Recent proposals [15] assume that scalable dynamic-transport SONs can be implemented with a 5-tuple of $< 0, -.25, .5, 0, 0 >$.[8] Our experience implementing Raw suggests that the attainability of such a 5-tuple for a high-frequency, scalable dynamic-transport SON is uncertain, even with Flit-Reservation. Implementation of these systems may be the only way to shed light on the issue.

6.2 Operation of RawStatic, an SSS SON

We now describe RawStatic, which is the SON implemented in full in the Raw processor prototype. RawStatic achieves *efficient operation-operand matching* through the combination of the static transport network, an intelligent compiler, and bypass-path integrated network interfaces. This SSS SON affords an efficient implementation that manages and routes operands with a parsimony of hardware mechanisms and logic depths. In contrast to the SDS SON of the previous section, this SSS SON requires no special additional hardware structures in order to support sender synchronization, receiver synchronization and demultiplexing, deadlock avoidance, and multicast.

6.2.1 Sends

RawStatic transmits data merely by having instructions target a register-mapped output FIFO. The static transport automatically knows where the data should go, so there is no need to specify a destination tile or an IRF index in the instruction. Furthermore, because multicast and broadcast are easily performed by the static transport, there is no need to encode multiple remote destinations. However, the inverse-bypass system must ensure the correct ordering of operand injection into the transport network.

Transport. RawStatic's *static transport* maintains a precomputed ordering[9] of operands over individual network links using a *static router*. This router fetches an instruction stream from a local instruction cache; these instructions each contain a simple operation and a number of route fields. The route fields specify the inputs for all outputs of two separate crossbars, allowing two operands per direction per cycle to be routed. The simple operation is sufficient to allow the static routers to track the control flow of the compute processors, or to loop and route independently.

For each operand sent between tiles on the static transport network, there is a corresponding route field in the instruction memory of each router that the operand will travel through. These instructions are generated by a compiler or runtime system. Because the static router employs branch prediction and can fetch the appropriate instruction long before the operand arrives, the preparations for the route can be pipelined, and the operand can be routed immediately when it arrives. This ahead-of-time knowledge of routes allows the transport portion of the RawStatic SON to achieve a partial 5-tuple of $+ < 0, 0, 1, 1, 0 >$.

A static router executes an instruction as follows: In the first couple of pipeline stages, it fetches and decodes the instruction. Then, in the Execute Stage, the static router determines if the instruction routes are ready to fire. To do this, it verifies that the source FIFOs of all of the routes are not empty and that the destinations of all of the routes have sufficient FIFO space. If these conditions are met, all routes proceed. Alternative, less tightly synchronized versions of this system could allow independent routes to occur without synchronizing.

6.2.2 Receives

Once the static transport network has routed an operand to the corresponding receiver, things procedes quite simply. The receiver merely verifies a single "data available" bit coming from one of two input FIFOs (corresponding to the two inputs of an instruction) indicating whether any data is available at all. If there is data available, it is the right data, ipso facto, because the static router provides in-order operand delivery. There is no need to demultiplex data to an IRF[10] with full/empty bits, no need to route an IRF index, and no need to dequeue a destination header. Furthermore, since the words arrive in the order needed, and the system is flow-controlled, there is no need to implement a deadlock recovery or avoidance mechanism to address the *deadlock challenge*. As mentioned earlier, a receiver compute

[8] The negative SL parameter reflects an assumption that half of the first route is free.

[9] We distinguish this type of transport from a static-timing transport, for which both ordering *and* timing of operand transport are fixed. Our early experiments led us away from static-timing transports due to the high cost of supporting the variability inherent in general-purpose computation. Nonetheless, static-timing transport eliminates much of the control and buffering inherent in routing and thus could afford extremely high throughput, low latency communication.

[10] Nonetheless, as shown in Appendix 2, it is useful to have a small register file controlled by the static transport which allows time-delay of operands as they are transmitted between senders and receivers. However, this mechanism does not need full/empty bits, nor does it require conservative usage of buffer space to ensure deadlock avoidance. In fact, our current compiler does not yet take advantage of this feature.

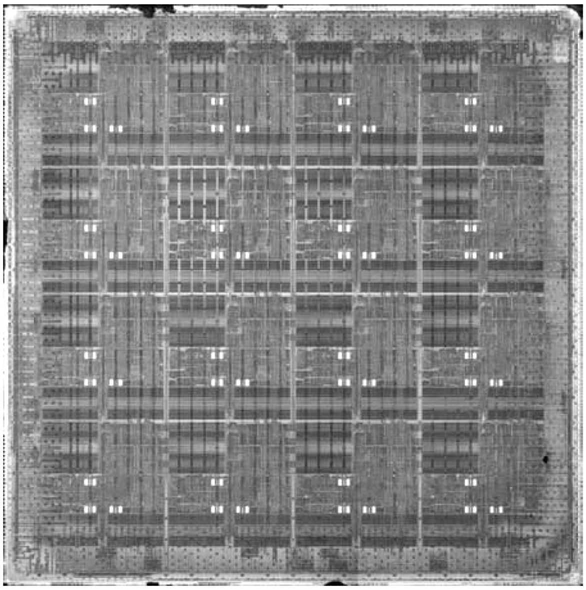


Fig. 7. A die photo of the 16-tile Raw chip. The tiles and RAMs are easy to spot. A tile is 4 $mm \times 4$ mm.

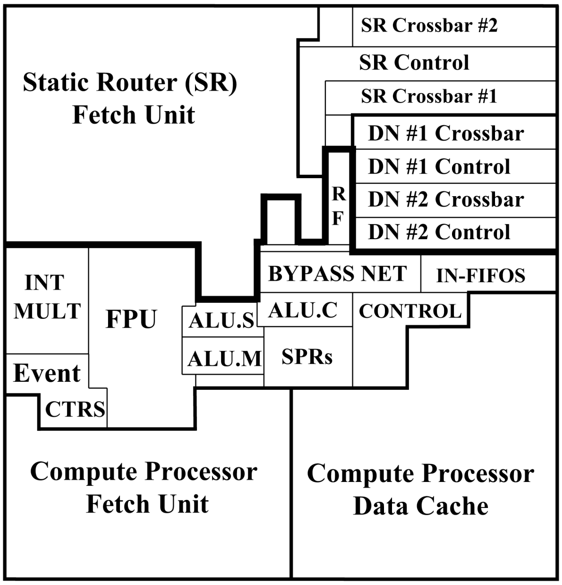


Fig. 8. Floorplan of a Single Raw Tile. DN = dynamic network. SR = static router. Everything below the bold border is part of the compute processor. RF = register file.

processor requires one cycle of latency to issue an instruction after its operand has arrived. This, combined with the partial transport tuple of $+ < 0, 0, 1, 1, 0 >$ makes the overall 5-tuple for RawStatic $< 0, 0, 1, 2, 0 >$. See Section 7 for in-depth derivation of this 5-tuple.

This brief summary of RawStatic is expanded further in [24], [11], and [26].

7 VLSI Implementation of an SSS SON

Having overviewed the operation of RawStatic and RawDynamic, we can now consider aspects of their VLSI implementations, using data from the actual Raw SSS SON implementation.

We received 120 of the 180 nm, 6-layer Cu, 330 mm^2, 1,657 pin Raw prototypes from IBM's CMOS7SF fabrication facility in October 2002. The chip core has been verified to operate at 425 MHz at the nominal voltage, 1.8V, which is comparable to IBM PowerPC implementations in the same ASIC process. Fig. 7 shows a die photo.

The Raw prototype divides the usable silicon area into an array of 16 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 routers—static and dynamic, and 96 KB of instruction cache. These tiles are connected to nearest neighbors using four separate networks, two static and two dynamic. These networks consist of more than 1,024 wires per tile.

7.1 Physical Design

The floorplan of a Raw tile is shown in Fig. 8. Approximately 40 percent of the tile area is dedicated to the dynamic and static transports. The local bypass network is situated in the center of the tile because it serves as a clearing house for the inputs and outputs of most of the components. The distance of a component from the bypass networks is an inverse measure of the timing criticality of the component. Components that have ample timing slack can afford to be placed further away and suffer greater wire delay. Thus, we can see that the static network paths were among the least critical, while the single-cycle ALU, control, and dynamic network components were most critical. The Event Counters, FPU, and Integer Multiply were placed "wherever they fit" and the corresponding paths were pipelined (increasing the latency of those functional units) until they met the cycle time. Finally, the fetch units and data cache are constrained by the sizes of the SRAMs they contain (occupying most of their area) and, thus, have little flexibility beyond ensuring that the non-RAM logic is gravitated toward the bypass paths.

Of particular interest are the relative sizes of the dynamic and static transport components, which are roughly equal in raw transport bandwidth. The crossbars of the two transports are of similar size. Raw's static router has an additional fetch unit, for which there is no equivalent in the dynamic transport SON. On the other hand, a dynamic transport SON like RawDynamic requires additional hardware structures beyond the transport, which are not represented in the Raw floorplan. We estimate the area of these structures as follows. First, the IRF will occupy 8-12x more area than the existing Register File due to the number of ports (>= 2W, 2R) and increased register count (we estimate 128) necessary to implement deadlock avoidance. Second, the dynamic router must transport headers and IRF indices with the data words, which we estimate results in 50 percent wider paths in the crossbars in order to provide the same operand bandwidth (as opposed to raw bandwidth) and the same scalability (1,024 tiles). Finally, to support multicast, a dynamic transport SON needs to encode multiple destinations in the compute processor instructions themselves. Thus, the required compute processor fetch unit RAM size for a given miss ratio may double or even triple. The implementation of complete bandwidth-scalable dynamic transport SONs will shed more light on this subject.

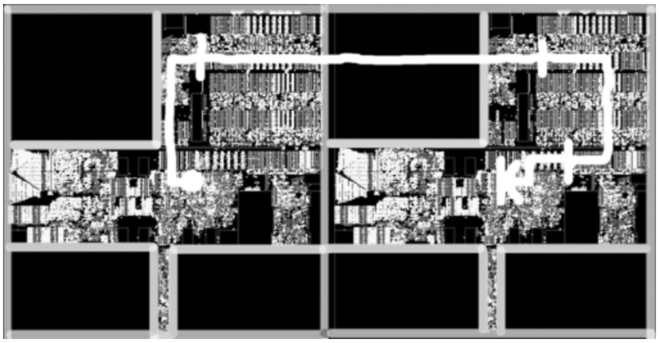


Fig. 9. The path of an operand route between two tiles. Path is superimposed over an EDA screenshot of the placed cells. Grayish cells are collections of single-bit registers.

7.2 Energy Characteristics of SONs

By measuring the current used by the 16-tile Raw core using an ammeter, we were able to derive a number of statistics for the use of Raw's transport networks. First, the static transport is around 40 percent more power efficient than the dynamic transport for single word transfers (such as found in SONs), but the dynamic network is almost twice as efficient for 31-word transfers. We found that there is a vast difference between the power of worst-case usage (where every routing channel is being used simultaneously), and typical usage. Worst-case usage totals 38W @ 425 MHz, consisting of 21 percent static transport, 15 percent dynamic transport, 39 percent compute processors, and 28 percent clock. Typical usage is 17W @ 425 MHz, with around 5 to 10 percent going to static transport, and 55 percent to clock. Our data indicates that clock-gating at the root of the clock tree of each tile (in order to reduce the energy usage of the clock tree when tiles are unused) is an architectural necessity in these systems. More detail on these results can be found in [9]. A recent paper examines power trade offs in on-chip networks [31].

7.3 Examination of Tile-Tile Operand Path

In this portion, we specifically examine the path an operand takes travelling in four clock cycles. We further annotate the parameters of the 5-tuple $<\mathrm{SO}, \mathrm{SL}, \mathrm{NHL}, \mathrm{RL}, \mathrm{RO}>$, $<0, 0, 1, 2, 0>$, as they are determined. First, the operand travels through the sender's rotate operator inside the ALU, and is latched into a FIFO attached to the sender's static router. Because the FIFO is register mapped, SO = 0. In the next cycle, the value is routed through the local static router across to the next tile's FIFO. This process takes one cycle per hop. (SL = 0, NHL = 1). A cycle later, the operand is routed through the receiver's static router to the receiver's compute processor input FIFO. On the final cycle, the value is latched into the input register of the receiver ALU (RL = 2). Since instructions have register-mapped interfaces to the network, there is no instruction overhead for using values from the network (RO = 0).

Fig. 9 superimposes the path taken by the operand on its journey, across two tiles. The white line is the path the operand takes. The tick-marks on the line indicate the termination of a cycle. The floorplan in Fig. 8 can be consulted to understand the path of travel.

The reference critical path of the Raw processor was designed to be the delay through a path in the compute processor fetch unit. This path consists of an $8\mathrm{K} \times 32$ bit SRAM and a 14-bit 2-input mux, the output routed back into the address input of the SRAM. All paths in the tile were optimized to be below this delay. This path times at approximately 3.4 ns, excluding register overhead.[11] The total delay of a path may vary from this number, because the register and clock skew overhead varies slightly. Furthermore, because none of the paths that an operand travels along are close to the cycle time of the chip, CAD optimizations like buffering, gate reordering, and gate sizing may not have been run. As a result, some of the delays may be greater than necessary. In Tables 1 and 2, we examine the timing of signals travelling through the most critical paths of the operand route. We examine both the propagation of the actual operand data as well as the single bit valid signal that interacts with the control logic to indicate that this data is a valid transmission. These are the key signals that are routed the full distance of an operand route.

This process is somewhat complicated by the fact that a given signal may not be on the direct critical path. For instance, it may arrive at the destination gate earlier than the other inputs to the gate. In this case, it may appear as if the gate has greater delay than it really does, because the input-to-output time includes the time that the gate is waiting for the other inputs.

We address this issue as follows: First, for each cycle of each signal (data and valid), we give the *output slack* and the *input slack*. The input slack is the true measure of how much later the input signal could have arrived, without any modification to the circuit, without impacting the cycle time of the processor. The output slack measures how much earlier the output signal is latched into the flip-flop ending the cycle than is needed for the given cycle time. In general, these two numbers are not exclusive; one cannot simply calculate the combined slack that could be exploited without examining the full path of gates between input and output. However, if the output slack of one stage and the input slack of the next stage sum to greater than the cycle time, they can be combined to eliminate a register. The elimination of such a register on this path would cause a reduction in the communication latency and thus in the 5-tuple.

Of interest in Table 1 are Cycles 3 and 4, where considerable adjacent Output and Input slack exists, on the Data Signal, totaling 3.68 ns, and on the Valid signal, totaling 2.4 ns. The Data signal is already fast enough to remove the register. However, the Valid signal requires an additional 1 ns of slack. With some optimization, it seems likely that one cycle of latency could be removed from the 5-tuple; reducing Raw's 5-tuple to $<0, 0, 1, 2, 0>$. Two possibilities are: 1) ANDing the Valid line later on into its consuming circuit, the stall calculation of the compute processor in cycle 4, or 2) adjusting the static router to begin processing the Valid signals (and, thus, most of the control

[11] IBM requires tapeout on worst-case rather than nominal process corners. Because of this, the numbers are pessimistic relative to the actual frequency (425 MHz) of average parts produced from the fab and used in typical conditions. All numbers are worst-case numbers, extracted using CAD tools with parasitics derived from the chip wiring.

TABLE 1
Slack in Critical Paths of Tile-Tile Operand Path (in ns)

Cycle	1		2		3		4	
Path	Local ALU		Local Router		Remote Router		Remote Dispatch	
Slack Type	Input	Output	Input	Output	Input	Output	Input	Output
Data (32b)	0	.44	1.01	1.44	1.34	**1.86**	**1.82**	.34
Valid (1b)	0	**1.90**	**.42**	1.24	.46	**1.20**	**1.23**	.094

logic for a route) a half cycle earlier, taking advantage of the early arrival (2.3 ns) of the Valid signal into Cycle 2.

It is not altogether suprising that the control, including the Valid signal, is the bottleneck. In our experiences implementing multiple versions of both dynamic and static on-chip transport networks, the data paths are *always* less critical than the corresponding control paths, to the extent that the Data path can almost be ignored. Furthermore, it is easy to underestimate the number of logic levels required after the Valid and Data signals are already sent out toward the destination tile—there are FIFOs to be dequeued, "space available" registers to be updated, and any other logic required to get the routers ready for the next route.

In Table 2, we show detailed timings for the tile-tile operand path. Because the valid signal is not actually propagated through the operand path, but rather is consumed and regenerated at each step, the control paths are relatively complex to describe. Instead, we show the arrival time of the data portion of the operand as it travels through the circuit. This time includes the calculation of the control signals for the muxes that manage the flow of the operand. This gives us an intuition for the delays incurred, especially due to the transport of the data over long distances. Of particular interest is that wire (and repeater) delay is almost 40 percent of the delay in a single-cycle network hop.

8 Analytical Evaluation of SONs

This section examines the impact of SON properties on performance. We focus our evaluation on the 5-tuple cost model. First, we consider the performance impact of each element of the 5-tuple. Then, we consider the impact of some factors not modeled directly in the 5-tuple. Finally, we compare RawStatic with SONs based on traditional multiprocessor communication mechanisms. Though multiprocessors are scalable, we show that they do not provide adequate performance for operand delivery—which in turn justifies the study of scalable SONs as a distinct area of research. Appendices 1 and 2 (which can be found online at http://www.computer.org/tpds/archives.htm) examine the locality and usage properties of SONs.

8.1 Experimental Setup

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

8.1.1 Simulator

Our experiments were performed on Beetle, a validated cycle-accurate simulator of the Raw microprocessor [24]. In our experiments, we used Beetle to simulate up to 64 tiles. Data cache misses are modeled faithfully; they are satisfied over a dynamic transport network that is separate from the SON. All instructions are assumed to hit in the instruction cache.

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

8.1.2 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 [3].

8.1.3 Compiler

Code is generated by Rawcc, the Raw parallelizing compiler [11]. Rawcc takes sequential C or Fortran programs and schedules their ILP 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 operands.

TABLE 2
Detailed Timings for Tile-Tile Operand Path

Cycle 1 **Local ALU**	ns	**Cycle 2** **Local Router**	ns	**Cycle 3** **Remote Router**	ns	**Cycle 4** **Remote Dispatch**	ns
Rotate Unit	1.84	FIFO Access	.39	FIFO Access	.89	FIFO Access	.92
ALU Mux	.19	Xbar	.94	Xbar, Drive to FIFO	.78	Slack	.77
Local Bypass Routing	.41	Drive to Neighbor Tile	.76			Processor Bypassing	.88
Inv. Bypass, Drive to FIFO	.57					Branch Compare	.63
Output Slack	.44	Output Slack	1.44	Output Slack	1.86	Output Slack	.34

TABLE 3
Performance of Raw's Static SON <0,0,1,2,0> for 2 to 64 Tiles

	2	4	8	16	32	64
Sha	1.1	1.8	1.8	2.0	2.4	2.4
Aes	1.9	2.9	3.0	3.6	3.2	3.8
Fpppp-kernel	1.5	3.2	5.9	6.6	7.5	7.6
Adpcm	0.8	1.2	1.4	1.4	1.2	0.9
Unstructured	1.5	2.5	2.6	2.5	1.9	1.7
Moldyn	1.2	1.7	1.7	1.6	1.0	0.9
Btrix	1.6	4.8	11.7	24.5	46.1	-
Cholesky	1.9	4.3	7.3	8.4	9.2	9.1
Vpenta	1.6	4.9	11.0	24.1	49.2	81.4
Mxm	1.8	4.2	7.1	9.3	10.6	11.9
Tomcatv	1.2	2.9	5.0	7.8	9.0	-
Swim	1.0	2.1	4.2	8.1	16.5	24.3
Jacobi	2.1	4.3	9.5	18.6	36.1	62.1
Life	0.9	2.5	5.4	10.8	21.9	48.7

Speedups are relative to that on one tile.

To make intelligent instruction assignment and scheduling decisions, Rawcc models the communication costs of the target network accurately. Rawcc's compilation strategy seeks to minimize the latency of operand transport on critical paths of the computation. Accordingly, it performs operation assignment in a way that gravitates sections of those critical paths to a single node so that that node's 0-cycle local bypass network can be used, minimizing latency.

8.1.4 Benchmarks

Table 3 lists our benchmarks. The problem sizes of the dense matrix applications have been reduced to improve 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.

8.2 Performance Impact of 5-tuple Parameters

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

8.2.1 Baseline Performance

First, we measured the absolute performance attainable with the actual, implemented Raw static SON with 5-tuple $< 0, 0, 1, 2, 0 >$. Table 3 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.

The amount of exploited parallelism varies between the benchmarks. Sha, Aes, Adpcm, Moldyn, and Unstructured have small amounts of parallelism and attains between one to four-fold speedup. The others contain modest to high amounts of ILP and attain between 7 to 80 fold-speedups. Note that speedup can be superlinear due to effects from increased cache capacity as we scale the number of tiles. The presence of sizable speedup validates our experimental setup for the study of SONs—without such speedups, it would be moot to explore SONs that are scalable.

8.2.2 Send and Receive Occupancy

Next, we measured the performance impact of send and receive occupancy, as shown in Figs. 10 and 11. We emphasize that the Rawcc compiler accounts for occupancy when it schedules ILP. All data points are based on 64 tiles. Performance is normalized to that of the actual Raw static SON with 5-tuple $< 0, 0, 1, 2, 0 >$.

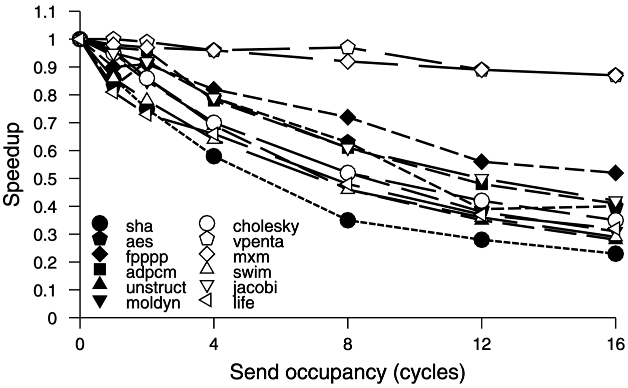


Fig. 10. Send occupancy versus performance on 64 tiles, i.e., $< \mathrm{n}, 0, 1, 2, 0 >$.

The trends of the two occupancy curves are similar. Overall, results indicate that occupancy impacts performance significantly. Performance drops by as much as 20 percent, even when the send occupancy is increased from zero to one cycle.

The slope of each curve is primarily a function of the amount of communication and slack in the schedule. Benchmarks that communicate infrequently and have a large number of cache misses, such as Vpenta, have higher slacks and are able to better tolerate increases in send occupancy. Benchmarks with fine-grained parallelism such as Sha and Fpppp-kernel have much lower tolerance for occupancy.

This data suggests that SONs should strive for zero-cycle send and receive occupancy.

8.2.3 Send and Receive Latency

We switched to the parameterized SON to measure the impact of send and receive latency on performance. For this experiment, we set the network hop latency to zero, with 5-tuple $< 0, \mathrm{n}, 0, 0, 0 >$ or $< 0, 0, 0, \mathrm{n}, 0 >$. Due to simulator constraints, the minimum latency we can simulate is one. Note that because the parameterized network does not model contention, n cycles of send latency is indistinguish-

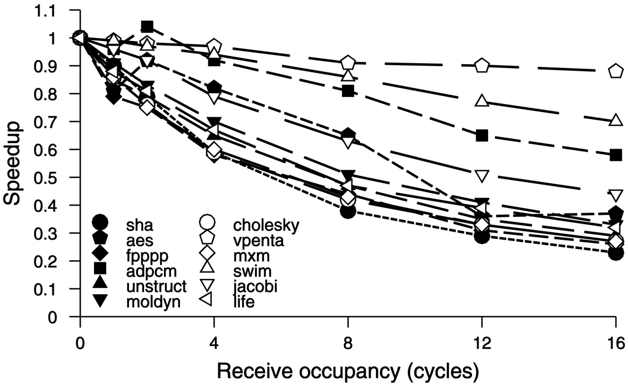


Fig. 11. Receive occupancy versus performance on 64 tiles, i.e., $< 0, 0, 1, \; 2, \mathrm{n} >$.

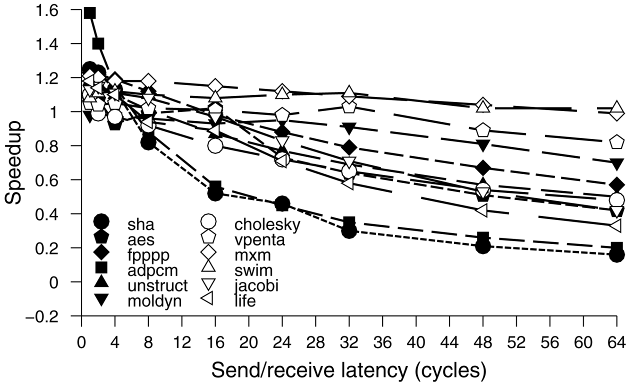


Fig. 12. 64-tile performance versus send or receive latency, i.e., $<0,\mathrm{n},0,0,0>$ or $<0,0,0,\mathrm{n},0>$.

able from the effect of n cycles of receive latency. Also, note that each of these 5-tuples also happens to represent an n cycle contention-free crossbar.

Fig. 12 graphs the performance impact of latency for 64 tiles. A speedup of 1.0 represents the performance of the Raw SSS SON with 5-tuple $<0,0,1,2,0>$.

Intuitively, we expect benchmark sensitivity to latency to depend on the granularity of available parallelism. The finer the grain of parallelism, the more sensitive the benchmark is to latency. Granularity of parallelism does not necessarily 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.

The benchmarks have varying sensitivities to latency. Mxm, Swim, and Vpenta exhibit "perfect" locality and thus coarse grained parallelism, so they are latency insensitive. The other benchmarks have finer grained parallelism and incur slowdown to some degree. Sha and Adpcm, irregular apps with the least parallelism, also have the finest grained parallelism and the most slowdown.

Overall, benchmark performance is much less sensitive to send/receive latency than to send/receive occupancy.

8.2.4 Network Hop Latency

We also measured the effect of network hop latency on performance. The 5-tuple that describes this experiment is $<0,0,\mathrm{n},1,0>$, with $0 \leq \mathrm{n} \leq 5$. 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.

Fig. 13 shows the result of this experiment. In general, the same benchmarks that are insensitive to send/receive latency (Mxm, Swim, and Vpenta to a lesser degree) are also insensitive to network hop latency. Of the remaining latency-sensitive benchmarks, a cycle of network hop latency has a higher impact than a cycle of send/receive latency, which suggests that those applications have at least some communication between nonneighboring tiles.

8.3 Impact of Other Factors on Performance

We now consider some factors not directly modeled in our 5-tuple: multicast and network contention.

8.3.1 Multicast

Raw's SON supports multicast, which reduces send occupancy and makes better use of network bandwidth. We evaluated the performance benefit of this feature. 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.

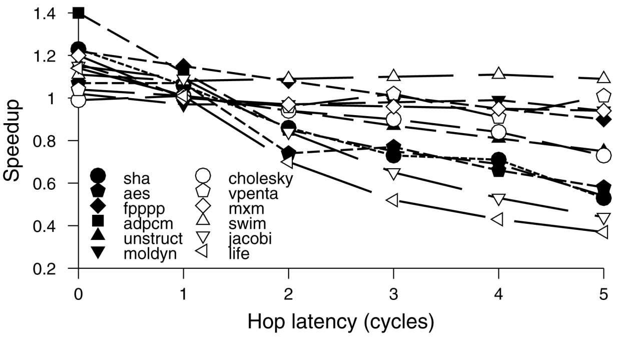


Fig. 13. Effect of network hop latency on performance on 64 tiles, i.e., $<0,0,\mathrm{n},1,0>$.

Somewhat surprisingly, we find that the benefit of multicast is small for up to 16 tiles: on 16 tiles, it is 4 percent. The importance of multicast, however, quickly increases for larger configurations, with an average performance impact of 12 percent on 32 tiles and 23 percent on 64 tiles.

8.3.2 Contention

We measured contention by comparing the performance of the actual Raw static SON with the performance of the parameterized SON with the same 5-tuple: $<0,0,1,2,0>$. The former models contention while the latter does not. Fig. 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. 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 percent. This marks a departure of SONs from conventional networks: for the limited scale we have considered, contention and bisection bandwidth are not significant limiters for on-chip mesh implementations. This is presumably due to the inherent locality found within the instruction dependence graphs of computations.

8.4 Performance of Multiprocessor-Based SONs versus RawStatic

We measure the performance of SONs based on traditional multiprocessor communication mechanisms. These SONs include MsgFast, MsgSlow, and SMP—they have been described and modeled using our 5-tuple cost model in

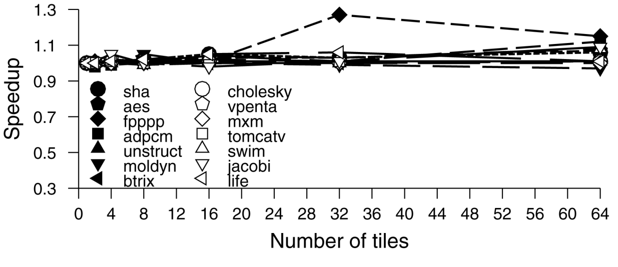


Fig. 14. Impact of removing contention.

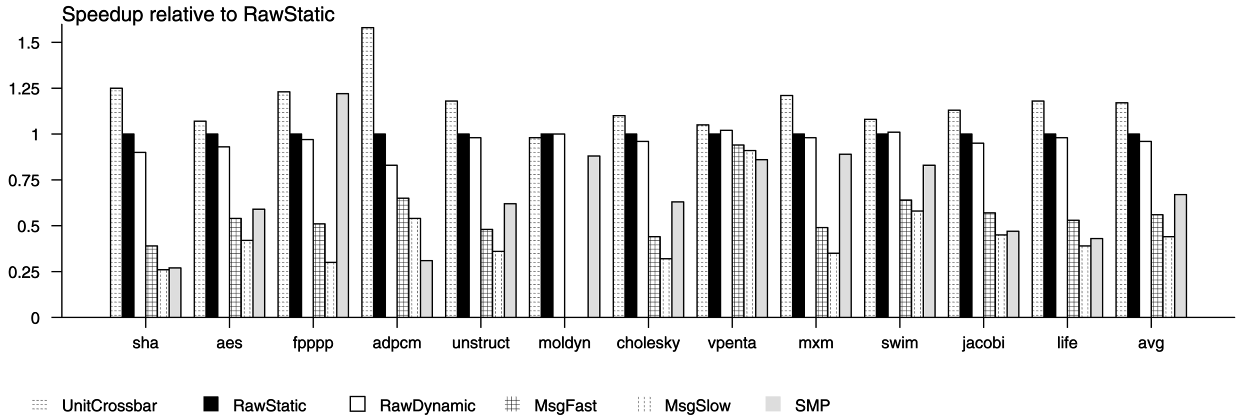


Fig. 15. Performance spectrum of SONs.

TABLE 4
Survey of SONs

	Superscalar	Distributed Shared Memory	Message Passing	Raw	RawDynamic	Grid	ILDP
Tuple (c:commit time)	<0,0,0,0,0>	<1,14+c,2,14,1>	<3,2+c,1,2,12>	<0,0,1,2,0>	<0,1,1,3,0>	<0,0,n/8,0,0> $0 \leq n \leq 8$	<0,n,0,1,0> n = 0, 2
AsTrO Class	DDD	SDD	SDS	SSS	SDS	SDD	DDS
# ALUs	4	Many	Many	4x4 to 32x32	4x4 to 32x32	8x8	8
# Fetch units for N Tiles	1	N	N	N	N	1	1
Frequency scalable?	No Monolithic	Yes Tiling	Yes Tiling	Yes Tiling	Yes Tiling	Yes Partial Tiling	Yes Partial Tiling
Bandwidth scalable?	No Broadcast	Yes Point-to-point	Yes Point-to-point	Yes Point-to-point	Yes Point-to-point	Yes Point-to-point	No Broadcast
Operand matching mechanism	Associative instruction window	Full-empty bits on table in cached RAM	Software demultiplexing	Compile-time scheduling	Compile-time scheduling	Distributed, associative instr. window	Full-empty bits on distributed register file
Free intra-node bypassing?	Yes	Yes	Yes	Yes	Yes	No	Yes

*AsTrO consists of **As**signment, operand **Tr**ansport, and operation **O**rdering.*

Section 3, with the optimistic assumption that the commit latency, c, is zero.

Fig. 15 shows the performance of the SONs for each benchmark on 64 tiles. Since all these SONs are modeled without contention, we use as baseline the network with the same 5-tuple as RawStatic ($< 0, 0, 1, 2, 0 >$), but without modeling contention. We normalize all performance bars to this baseline. For reference, we include UnitCrossbar, a single-cycle, contention-free crossbar SON with 5-tuple $< 0, 0, 0, 1, 0 >$.

Our results show that traditional communication mechanisms are inefficient for scalar operand delivery. Operand delivery via traditional message passing has high occupancy costs, while operand delivery via shared memory has high latency costs. The average performance is 33 percent to 56 percent worse than RawStatic.

8.5 Summary

These experiments indicate that the most performance critical components in the 5-tuple for 64 tiles are send and receive occupancy, followed closely by the per-hop latency, followed more distantly by send and receive latency. The 5-tuple framework gives structure to the task of reasoning about the tradeoffs in the design of SONs.[12]

9 Related Work

Table 4 contrasts some of the contemporary commercial and research SONs discussed in this paper. The first section of the table summarizes the 5-tuples, the AsTrO category, and

[12] Comparison between systems with and without multicast can be done with greater accuracy by ensuring that the average send occupancy appropriately includes occupancy costs due to operand replication.

the number of ALUs and fetch units supported in each of the SONs. 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.

The second, third, and fourth super-rows give the way in which the various designs address the frequency scalability, bandwidth scalability, and operation-operand matching challenges, respectively. All recent processors use tiling to achieve frequency scalability. Bandwidth scalable designs use point-to-point SONs to replace indiscriminant broadcasts with multicasts. So far, all of the bandwidth scalable architectures use static assignment of operations to nodes. The diversity of approaches is richest in the category of operand matching mechanisms, and in the selection of network transports. Specifically, the choice of static versus dynamic transport results in very different implementations. However, there is a common belief that the five tuples of both can be designed to be very small. Finally, a common theme is the use of intratile bypassing to cut down significantly on latency and required network bandwidth.

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 Grid and ILDP papers.

10 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 (SONs) that must somehow account for communication latency. Even though the microarchitectures are distributed, ILP can be exploited on these SONs because their latencies and occupancies are extremely low.

This paper makes several contributions: It introduces the notion of SONs and discusses the challenges in implementing scalable forms of these networks. These challenges include maintaining high frequencies, managing the bandwidth requirements of the underlying algorithms, implementing ultra-low cost operation-operand matching, avoiding deadlock and starvation, and handling exceptional events. This paper describes how two example SON designs deal with the five SON scaling challenges. These designs are based on an actual VLSI implementation called Raw.

The paper also surveys the SONs of recently proposed distributed architectures and distills their fundamental logical differences using a novel categorization: the AsTrO taxonomy.

The paper further identifies the key importance of operation-operand matching in SONs. The paper breaks down the latency of operation-operand matching into five components $< \mathrm{SO}, \mathrm{SL}, \mathrm{NHL}, \mathrm{RL}, \mathrm{RO} >$: send occupancy, send latency, network hop latency, receive latency, and receive occupancy. The paper evaluates several families of SONs based on this 5-tuple. Our early results show that send and receive occupancy have the biggest impact on performance. For our benchmarks, performance decreases by up to 20 percent even if the occupancy on the send or receive side increases by just one cycle. The per-hop latency follows closely behind in importance. Hardware support for multicast has a high impact on performance for large systems. Other parameters such as send/receive latency and network contention have smaller impact.

In the past, SON design was closely tied in with the design of other microprocessor mechanisms such as the register files and bypassing. In this paper, we identify the generalized SON as an independent architectural entity that merits its own research. We believe that research focused on the SON will yield significant simplifications in future scalable ILP processors.

Avenues for further research on SONs are plentiful, and relate to both reevaluating existing network concepts for scalar operands, and the discovery of new SON mechanisms. A partial list includes:

1. designing SONs for a high clock rate while minimizing their 5-tuples,
2. evaluating performance for much larger numbers of tiles and a wider set of programs,
3. generalizing SONs for other forms of parallelism such as streams, threads, and SMT,
4. exploration of both dynamic and static schemes for assignment, transport, and ordering,
5. evaluation of the impact of the AsTrO categories on cycle time and 5-tuple,
6. mechanisms for fast exception handling and context switching,
7. analysis of the trade offs between commit point, exception handling, and send latency,
8. low energy SONs, and
9. techniques for deadlock avoidance and recovery.

Acknowledgments

This research is funded by the US Defense Advanced Research Projects Agency (DARPA), the US National Science Foundation, ITRI, and the Oxygen Partnership.

References

[1] V. Agarwal, M.S. Hrishikesh, S.W. Keckler, and D. Burger, "Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures," *Proc. Int'l Symp. Computer Architecture,* pp. 248-259, 2000.

[2] Arvind and S. Brobst, "The Evolution of Dataflow Architectures from Static Dataflow to P-RISC," *Int'l J. High Speed Computing,* vol. 5, no. 2, pp. 125-153, June 1993.

[3] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal, "Maps: A Compiler-Managed Memory System for Raw Machines," *Proc. Int'l Symp. Computer Architecture,* pp. 4-15, 1999.

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

[5] J. Duato and T.M. Pinkston, "A General Theory for Deadlock-Free Adaptive Routing Using a Mixed Set of Resources," *IEEE Trans. Parallel and Distributed Systems,* vol. 12, no. 12, pp. 1-16, Dec. 2001.

[6] T. Gross and D.R. O'Halloron, *iWarp, Anatomy of a Parallel Computing System.* Cambridge, Mass.: MIT Press, 1998.

[7] J. Janssen and H. Corporaal, "Partitioned Register File for TTAs," *Proc. Int'l Symp. Microarchitecture,* pp. 303-312, 1996.

[8] H.-S. Kim and J.E. Smith, "An Instruction Set Architecture and Microarchitecture for Instruction Level Distributed Processing," *Proc. Int'l Symp. Computer Architecture,* pp. 71-81, 2002.

[9] J. Kim, M. Taylor, J. Miller, and D. Wentzlaff, "Energy Characterization of a Tiled Architecture Processor with On-Chip Networks," *Proc. Int'l Symp. Low Power Electronics and Design,* 2003.

[10] J. Kubiatowicz, A. Agarwal, "Anatomy of a Message in the Alewife Multiprocessor," *Proc. Int'l Supercomputing Conf.,* pp. 195-206, 1993.

[11] W Lee et al., "Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine," *Proc. Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 46-54, 1998.

[12] K. Mackenzie, J. Kubiatowicz, M. Frank, W. Lee, V. Lee, A. Agarwal, and M.F. Kaashoek, "Exploiting Two-Case Delivery for Fast Protected Messaging," *Proc. Int'l Symp. High Performance Computer Architecture,* July 1997.

[13] K. Mai, T. Paaske, N. Jayasena, R. Ho, W.J. Dally, and M. Horowitz, "Smart Memories: A Modular Reconfigurable Architecture," *Proc. Int'l Symp. Computer Architecture,* pp. 161-171, 2000.

[14] S.D. Naffziger and G. Hammond, "The Implementation of the Next-Generation 64b Itanium Microprocessor," *Proc. IEEE Int'l Solid-State Circuits Conf.,* pp. 344-345, 472, 2002.

[15] R. Nagarajan, K. Sankaralingam, D. Burger, and S.W. Keckler, "A Design Space Evaluation of Grid Processor Architectures," *Proc. Int'l Symp. Microarchitecture,* pp. 40-51, 2001.

[16] S. Palacharla, N.P. Jouppi, and J.E. Smith, "Complexity-Effective Superscalar Processors," *Proc. Int'l Symp. Computer Architecture,* pp. 206-218, 1997.

[17] D. Panda, S. Singal, and R. Kesavan, "Multidestination Message Passing in Wormhole k-Ary n-Cube Networks with Base Routing Conformed Paths," *IEEE Trans. Parallel and Distributed Systems,* 1999.

[18] L. Peh et al., "Flit-Reservation Flow Control," *Proc. Symp. High-Performance Computer Architecture,* 2000.

[19] K. Sankaralingam, V. Singh, S. Keckler, and D. Burger, "Routed Inter-ALU Networks for ILP Scalability and Performance," *Proc. Int'l Conf. Computer Design,* 2003.

[20] G. Sohi, S. Breach, and T. Vijaykumar, "Multiscalar Processors," *Proc. Int'l Symp. Computer Architecture,* pp. 414-425, 1995.

[21] Y.H. Song and T.M. Pinkston, "A Progressive Approach to Handling Message Dependent Deadlocks in Parallel Computer Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 14, no. 3, pp. 259-275, Mar. 2003.

[22] H. Sullivan and T.R. Bashkow, "A Large Scale, Homogeneous, Fully Distributed Parallel Machine," *Proc. Fourth Ann. Symp. Computer Architecture,* pp. 105-117, 1977.

[23] S. Swanson et al., "WaveScalar," *Proc. Int'l Symp. Microarchitecture,* 2003.

[24] M. Taylor, "The Raw Processor Specification," ftp://ftp.cag.lcs. mit.edu/pub/raw/documents/RawSpec99.pdf, 2004.

[25] M. Taylor et al., "How to Build Scalable On-Chip ILP Networks for a Decentralized Architecture," Technical Report 628, Massachusetts Inst. of Technology, Apr. 2000.

[26] M. Taylor et al., "The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs," *IEEE Micro,* pp. 25-35, Mar. 2002.

[27] M. Taylor et al., "Scalar Operand Networks: On-Chip Interconnect for ILP in Partitioned Architectures," *Proc. Int'l Symp. High Performance Computer Architecture,* 2003.

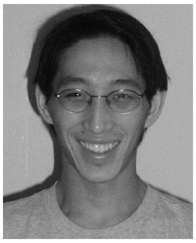
[28] M. Taylor et al., "Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams," *Proc. Int'l Symp. Computer Architecture,* 2004.

[29] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation," *Proc. 19th Int'l Symp. Computer Architecture,* May 1992.

[30] E. Waingold et al., "Baring It All to Software: Raw Machines," *Computer,* vol. 30, no. 9, pp. 86-93, Sept. 1997.

[31] H. Wang, L. Peh, and S. Malik, "Power-Driven Design of Router Microarchitectures in On-Chip Networks," *Proc. Int'l Symp. Microarchitecture,* 2003.

Michael Bedford Taylor received the AB degree from Dartmouth College in 1996 and the SM degree from the Massachusetts Institute of Technology (MIT) in 1999. He coauthored the first version of the x86-to-PowerPC emulator, Virtual PC. More recently, he was the lead architect of the MIT Raw microprocessor. His research interests include microprocessor scalability, interconnection networks, physical limitations on computation, and relativistic complexity theory. He is a PhD candidate in computer science and electrical engineering at MIT and the recipient of an Intel PhD fellowship.



Walter Lee received the SB and MEng degrees from the Massachusetts Institute of Technology (MIT) in 1995. He is the lead architect of Rawcc, the Raw parallelizing compiler. His research interests include parallelizing compilers, scheduling algorithms, and compilation for tiled architectures. He is a PhD candidate in computer science and electrical engineering at MIT and the recipient of an IBM PhD Research Fellowship Award.

Saman P. Amarasinghe received the BS degree in electrical engineering and computer science from Cornell University in 1988, and the MSEE and PhD degrees from Stanford University in 1990 and 1997, respectively. He is an associate professor in the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology (MIT) and a member of the MIT Computer Science and Artificial Intelligence Laboratory. He leads the Commit Compiler Group and is the coleader of the MIT Raw project. His research interests are in discovering novel approaches to improve the performance of modern computer systems without unduly increasing the complexity faced by either application developers, compiler writers, or computer architects. He is a member of the IEEE.

Anant Agarwal received the BTech degree in electrical engineering from IIT Madras, India, in 1982, and the MS and PhD degrees in electrical engineering from Stanford University in 1984 and 1987, respectively. He is currently with the Computer Science and Artificial Intelligence Laboratory (CSAIL) at the Massachusetts Institute of Technology (MIT) as a professor of electrical engineering and computer science. At Stanford, he participated in the MIPS and MIPS-X projects. He led the Alewife and Sparcle projects at MIT, which designed and implemented a scalable distributed shared-memory machine and an early multithreaded microprocessor. He currently codirects the Raw and Oxygen projects. Raw is developing a wire-exposed microprocessor architecture that supports both stream parallelism and ILP. Oxygen is building a pervasive computing environment in which humans interact with computation using speech and vision. He is a member of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.