Scalar Operand Networks:
Design, Implementation, and Analysis *

Michael Bedford Taylor, Walter Lee, Saman Amarasinghe, Anant Agarwal

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 latencies (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 detailed 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.*

1 Introduction

Today's wide-issue microprocessor designers are finding it increasingly difficult to convert burgeoning silicon resources into usable, general-purpose functional units. The problem is not so much that the area of microprocessor structures is growing out of control; after all, Moore's law's exponential growth is easily able to outpace a mere quadratic growth in area. Rather, it is the delay of the interconnect inside the processor blocks that has become unmanageable [1, 18, 33]. Thus, although we can build almost arbitrarily wide-issue processors, clocking them at high frequencies will become increasingly difficult. A case in point is the Itanium 2 processor, which sports a zero-cycle fully-bypassed 6-way issue integer execution core. Despite occupying less than two percent of the processor die, this unit spends half of its critical path in the bypass paths between the ALUs [16].

More generally, the pervasive use of global, centralized structures in these contemporary processor designs constrains not just the frequency-scalability of functional unit bypassing, but of many of the components of the processor that are involved in the task of naming, scheduling, orchestrating and routing operands between functional units [18].

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 gain-

* *MIT-LCS-TM-644. To appear.*

fully exploited by current architectures. Furthermore, other forms of parallelism, such as data parallelism, pipeline parallelism, and coarse-grained parallelism, can easily be converted into ILP.

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

Seeking to scale ILP processors, recent microprocessor designs in industry and academia reveal a trend towards distributed resources to varying degrees, such as partitioned register files, banked caches, multiple independent compute pipelines, and even multiple program counters. These designs include UT Austin's Grid[17], MIT's Raw[31] and SCALE, Stanford's Smart Memories[15], Wisconsin's ILDP[9] and Multiscalar[24], Washington's WaveScalar[27] 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 that coordinates these values into a coherent computation. SONs can be designed to have short wire lengths. Therefore, they can scale with increasing transistor counts. Furthermore, because they can be designed around generalized transport networks, scalar operand networks can potentially provide transport for other forms of data including I/O streams, cache misses, and synchronization signals.

Partitioned microprocessor architectures require scalar operand networks that combine the low-latency and low-occupancy operand transport of wide-issue superscalar processors with the frequency-scalability of multiprocessor designs. Several recent studies have shown that partitioned microprocessors based on point-to-point SONs can successfully exploit fine-grained ILP. Lee et al. [13] 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. [17] 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 networks, such as scalability and deadlock avoidance, they have many unique requirements including ultra-low latencies (a few cycles versus tens of cycles) and ultra-fast operation-operand matching (0 cycles versus tens of cycles). This paper identifies five important challenges in designing SONs, and develops the AsTrO taxonomy for describing their logical properties. The paper also defines a parameterized 5-tuple model that quantifies performance tradeoffs 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 occupancies have a first order impact on ILP performance. For our benchmarks running on a 64-tile microprocessor (i.e., 64 ALUs, 64-way

[1]pronounced ess-oh-en.

partitioned register file, 64 instruction and data caches, connected by an SON) we measure a performance drop of up to 20 percent when either the send or the receive occupancy is increased from zero to one cycle. The performance loss due to network transport contention, on the other hand, was discovered to average only 5 percent for a 64-tile Raw mesh. These results lead us to conclude that whether the network 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 introduces a taxonomy for the logical structure of SONs. Section 5 describes the design of two example SONs. Section 6 discusses the scalar operand network implementation in the Raw processor. Section 7 quantifies the sensitivity of ILP performance to network properties, and Section 8 examines application-specific operand usage statistics. Section 9 presents related work, and Section 10 concludes the paper.

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. An SON includes both the physical interconnection network (the transport network) and the associated operation-operand matching algorithms (hardware or software) that coordinate operands and operations into a coherent computation. Designing SONs was a simple task during the era of non-pipelined processors, but as our demands for parallelism (e.g., multiple ALUs, large register name spaces), clock rate (e.g., deep pipelines), and scalability (e.g., partitioned register files) have increased, this task has become much more complex. This section describes the evolution of SONs – from the early, monolithic register file interconnects to the more recent ones that incorporate routed point-to-point mesh interconnects.

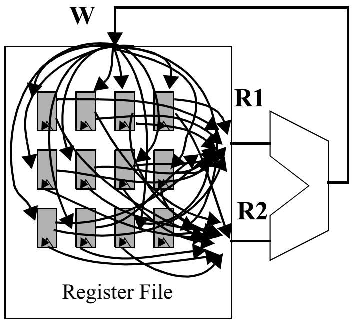

Figure 1. A simple SON.

A non-pipelined 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.

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

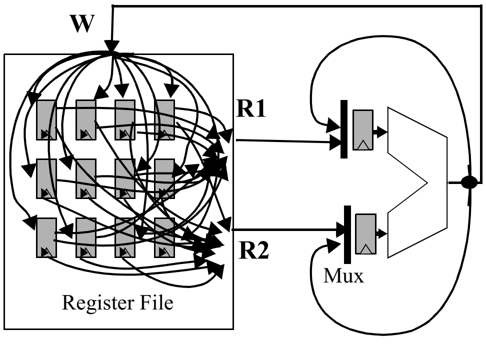

Figure 2. SON in a pipelined processor with bypassing links.

Figure 2 shows a pipelined, bypassed register-ALU pair. The SON now adds several new paths, multiplexers and pipeline registers, and partitions operand traffic into two classes: "live" operands routed directly from functional unit to functional unit, and "quiescent-but-live" operands routed "through time" (via self routes in the register file) and then eventually to the ALU. The partitioning improves the cycle time because the routing complexity of the live values is less than the routing complexity of the resident register set. This transformation also changes the naming system – the registers in the pipeline dynamically shadow the registers in the register file.

Figure 3 shows a pipelined 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 renaming also allows the quantity of simultaneous live values to be increased beyond the limited number of named live values fixed in the ISA. An even more scalable solution to this problem is to adopt an ISA that allows the number of named live values to increase with the number of ALUs.

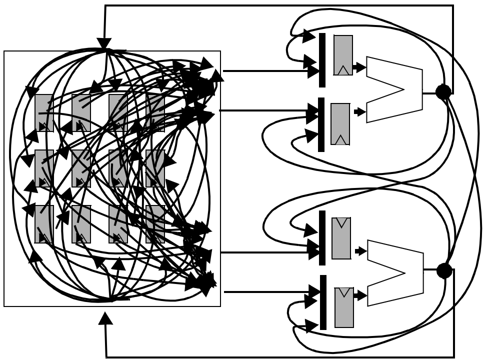

Figure 3. A pipelined processor with bypass links and multiple ALUs.

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.

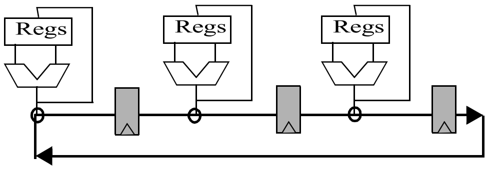

Figure 4. Multiscalar's SON

One solution is to partition and distribute the interconnect and the resources it connects. Figure 4 depicts the partitioned register file and distributed ALU design of the Multiscalar – one of the early distributed ILP processors. Notice that the Multiscalar pipelines results through individual ALUs with a one-dimensional multi-hop SON. Accordingly, the interconnect between the ALUs in the Multiscalar distinguishes it as an example of an early point-to-point SON.

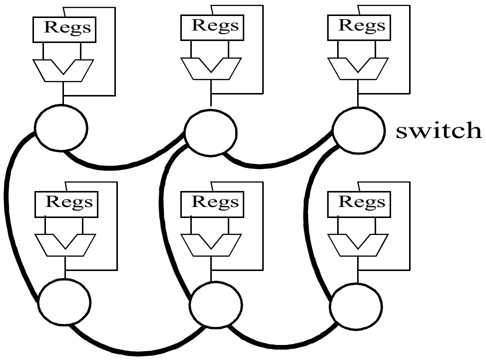

Figure 5. SON based on a 2-D point-to-point routed interconnect.

Figure 5 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 either an individual ALU in the Grid processor, or an individual pipeline in Multiscalar, Raw, or the ILDP processor.

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 operations and their corresponding operations are correctly matched at the destinations. See [13, 29] 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 Scalar Operand Networks

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

1. 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.

Intra-component frequency scalability As the issue width of a microprocessor increases, monolithic structures such as multi-ported register files, bypassing logic, selection logic, and wakeup logic grow linearly to quadratically in size. Although extremely efficient VLSI implementations of these components exist, their burgeoning size guarantees that intra-component interconnect delay will inevitably slow them down. Thus, these components have an asymptotically unfavorable growth function that is partially obscured by a favorable constant factor.

There are a number of solutions to the frequency scalability of these structures; the general themes typically include partitioning and pipelining. A number of recently proposed academic architectures [9, 15, 17, 24, 31] (and current-day multiprocessor architectures) compose their systems out of replicated tiles in order to simplify 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 inputs and outputs of the tiles are periodically registered so that the cycle time is not impacted. In effect, tiling ensures that the task of reasoning about frequency scalability need only be performed at the intercomponent level.

Inter-component 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. There are two clear examples of commercial architectures addressing inter-component delay: the Pentium IV, which introduced two pipeline stages that are dedicated to the crossing of long wires between remote components; and the Alpha 21264, which introduces a one cycle latency cost for results from one integer cluster to reach the other cluster. The Alpha 21264 marked the beginning of an architectural movement that recognizes that interconnect delay can no longer be ignored by the micro-architectural abstraction layer. Once interconnect delay becomes significant, high-frequency systems must be designed out of components that operate with only partial knowledge of what the rest of the system is doing. In other words, the architecture needs to be implemented as a distributed process. *If a component depends on information that is not generated by a neighboring component, the architecture needs to assign a time cost for the transfer of this information.* Non-local information includes the outputs of physically remote ALUs, stall signals, branch mispredicts, exceptions, and the existence of memory dependencies.

Managing latency As studies that compare small, short-latency caches with large, long-latency caches have shown, a large number of resources (e.g., cache lines) with long latency is not always preferable to a small number of resources with a short latency. This tradeoff between parallelism and locality is becoming increasingly important. On one hand, we want to spread virtual objects – such as cached values, operands, and instructions – as far out as possible in order to maximize the quantities of parallel resources that can

[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.

be leveraged. On the other hand, we want to minimize communication latency by placing communicating objects close together, especially if they are on the critical path. These conflicting desires motivate us to design architectures with non-uniform costs; so that rather than paying the maximum cost of accessing a object (e.g., the latency of the DRAM), we pay a cost that is proportional to the delay of accessing that particular object (e.g., a hit in the first-level cache). This optimization is further aided if we can exploit locality among virtual objects and place related objects (e.g. communicating instructions) close together.

2. Bandwidth Scalability Bandwidth scalability is the ability of a design to scale without inordinately increasing the relative percentage of resources dedicated to the SON. This includes not just the resources used in explicit transport networks but also the area of operand-management structures such as register files or instruction windows.

Bandwidth scalability is also a challenge that is making its way from multiprocessor designs to microprocessor designs. One key red flag of a non- bandwidth scalable architecture is the use of broadcasts in cases other than those directly mandated by the computation. For example, superscalars currently rely on global broadcasts to communicate the results of instructions. The output of every ALU is indiscriminately sent to every waiting instruction that could possibly depend on that value. Thus, if RB is the number of result buses of the processor, and WS is the window size of the processor, there are RB*WS individual routes and comparisons that are made on every cycle. As shown by the Alpha 21264, superscalars can handle the frequency scalability of broadcasting by pipelining these broadcast wires. This pipelining causes some dependent instructions to incur an extra cycle of delay but guarantees that broadcasting of results does not directly impact cycle time. Unfortunately, the usage of indiscriminate broadcast mechanisms carries substantial area, delay and energy penalties, and limits the scalability of the system.

The key to overcoming this problem is to 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. As a result, the broadcast network can be replaced with a point-to-point network of lesser bisection bandwidth that can perform unicast routes in order to exploit the bandwidth savings.

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 run-time system. The static mapping scheme is quite efficient because the lookup of directory entry does not incur lengthy communication delays. We call architectures [7, 8, 17, 31] 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]

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 intra-node bypassing for values that are only needed locally; however it still employs broadcast for values that may be needed by other nodes. It seems likely that a design like ILDP that uses centralized dispatch (admittedly a scalability problem in itself) can annotate the operations with their directory information dynamically as the operations are

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

dispatched to the physical operators for processing. In fact, this approach is explored and analyzed by [20]. 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. 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. If operation-operand matching can not be done efficiently, there is little point in scaling the issue-width of a processing system, because the benefits will rarely outweigh the overhead.

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

Send occupancy	average number of cycles that an ALU wastes in transmitting an operand to dependent instructions at other ALUs.
Send latency	average number of cycles of delay incurred by the message at the send side of the network without consuming ALU cycles.
Network hop latency	average number of cycles of delay, per hop, incurred travelling through the interconnect; more generally, this is the cost proportional to the physical distance between the sender and receiver.
Receive latency	average number of cycles of delay between when the final input to a consuming instruction arrives and when that instruction is issued.
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 [11] 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 subsections, we examine SONs implemented on a number of conventional systems and describe the components that contribute to the 5-tuple for that system. For these systems, we make estimates of 5-tuples for aggressive but conventional implementations. At one end of the spectrum, we consider superscalars, which have perfect 5-tuples, <0,0,0,0,0>, but limited scalability. On the other end of the spectrum, we examine message passing, shared memory and systolic systems, which have good scalability but poor 5-tuples. The Raw prototype, described in Sections 5 and 6, falls in between the two extremes, with multiprocessor-like scalability and a 5-tuple that comes closer to that of the superscalar: <0,1,1,1,0>.

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.

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

Multiprocessors employ a variety of communication mechanisms; two flavors are message passing and shared memory. In [29], we derived the 5-tuple of a message-passing implementation of an SON as ranging between $<3,2+c,1,1,7>$ and $<3,3+c,1,1,12>$ (referred to subsequently as *MsgFast* and *MsgSlow*) with c being the commit latency of the processor. An aggressive shared-memory SON implementation was determined to have a 5-tuple of $<1,14+c,2,14,1>$. For length purposes, we limit discussion in the next subsection to message-passing; more discussion of shared memory and systolic array SON implementations can be found in [29] and in part, in Supplemental Sections 11 and 12.

Message-passing operation-operand matching In discussing message-passing operation-operand matching, we assume that a dynamic transport network [5] is being employed 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 out-going operands. (Recall that the superscalar uses indiscriminate broadcasting to solve this problem.) There are a variety of methods for specifying this information. For instruction-mapped interfaces, the send instruction can leave encoding space (the log of the maximum number of nodes) or take a parameter to specify the destination node. For register-mapped interfaces, an additional word may have to be sent to specify the destination. Finally, dynamic transport networks typically do not support multicast, so multiple message sends may be required for operands that have non-unit fanout. These factors will impact the send and receive occupancies.

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 [32], or in a combination of hardware and software [14], causing a considerable receive occupancy.

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 [6, 25]. 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 a number of effective solutions for in-network deadlock have been proposed in the literature, endpoint deadlock is the main issue for concern in SONs.

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 a good bet that cache misses, branch mispredictions, exceptions, interrupts and context switches will be among those challenges. For instance, if an SON is being implemented

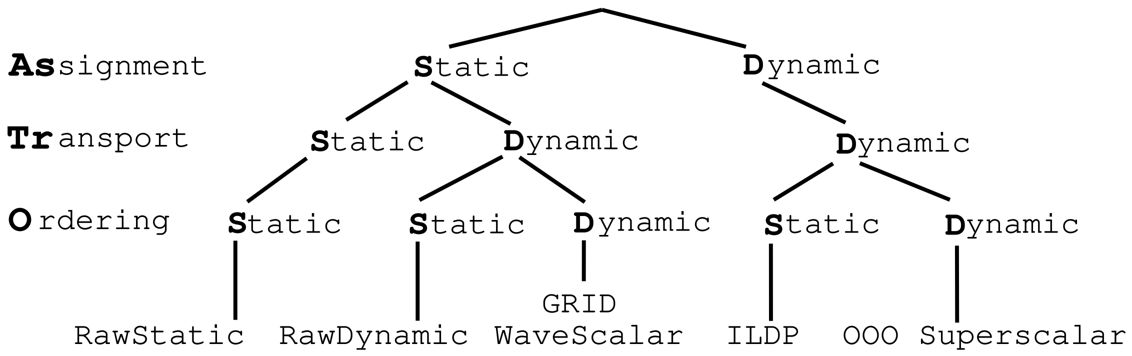

Figure 6. The AsTrO taxonomy of SONs.

using dynamic transport, how do context switches work? Is the state drained out and restored later? If so, how is the state drained out? Is there a freeze mechanism for the network? Or is there a roll back mechanism that allows a smaller representation of a process's context? Are the branch mispredicts and cache miss requests sent on the SON, or on a separate network?

In this section, we explored some of the challenges of implementing scalar operand networks. Furthermore, we examined the space of SONs implemented using existing legacy network interfaces. In the following three sections, we examine new SON designs which are optimized explicitly for operand transport. We start by describing the AsTrO taxonomy of SONs, which aids us in describing the high-level fundamental differences between the SONs. Then, the next section describes the operation of two example SON design from this taxonomy. Finally, the third section gives in-depth details of the Raw silicon implementation.

4 The AsTrO taxonomy for SONs

The AsTrO taxonomy describes the fundamental logical properties of an SON, namely: 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. In contrast to the 5-tuple, which compares the performance of different SONs, the AsTrO taxonomy captures the key differences in the way systems manage the flow of operands and operations.

Each of the three AsTrO components can be fulfilled using a static or dynamic method.

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. 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. The implications of these two properties are discussed in the next two sections.

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.

Figure 6 categorizes a number of systems into the AsTrO taxonomy. We remark in particular that *static assignment* is a key element of all of the current scalable designs. Like the MIMD taxonomy, the parameters are not wholly independent. In particular, dynamic assigment and static transport are an unlikely combination. This taxonomy extends that described in [30]. The taxonomy in [22] discusses only operand transport.

5 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.

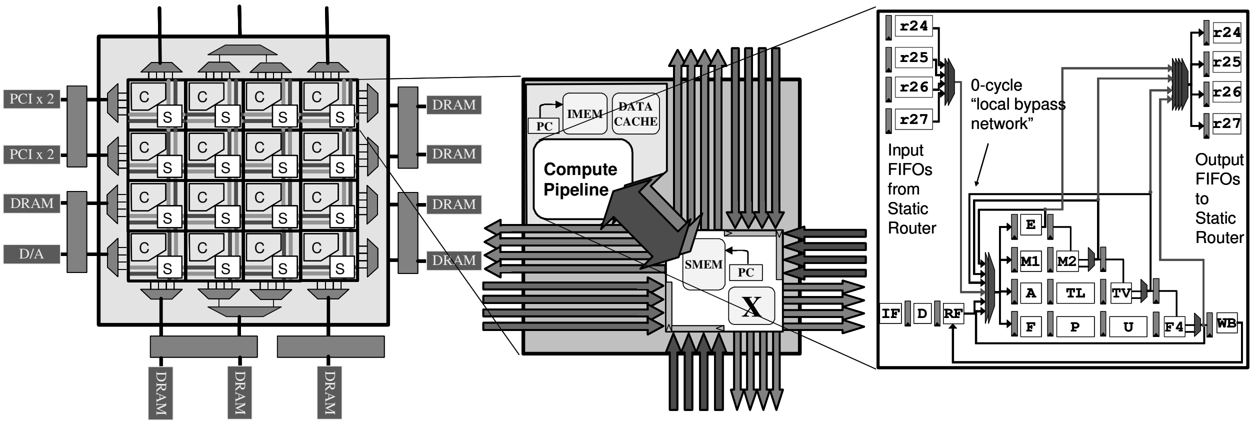

Figure 7. Architecture of the Raw Processor.

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 through tiling. A Raw processor is comprised of a 2-D 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 Figure 7 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, we do not have to worry about the frequency decreasing as we add more tiles. Each Raw tile contains a single-issue in-order compute processor, and a number of routers. In the case of the Raw implementation, the switching portions of the tile (demarkated by the "S"s 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 of the systems use the second dynamic router for cache-miss traffic.

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 to zero, 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 [7]. 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've deliberately designed an early commit point into our processor in order to eliminate the common multiprocessor communication-commit delay (i.e., $c = 0$) that was 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 implementation of a 0-cycle bypass for those operands that a compute processor both produces and uses locally further reduces the necessity to pay the full operation-operand matching cost.

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.

5.1 Operation of RawDynamic, an SDS SON

RawDynamic differs from RawStatic because it uses the Raw processor's dynamic dimension-ordered [26] worm-hole routed [5] 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 reasonable to conclude that dynamic transport SONs will have wider instruction words and wider transport network sizes[4] 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, the question of low-occupancy, low-latency broadcast and multicast in dynamic transport networks appears to be an area of active research[19].

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 implementation, we speculatively assume that incoming dynamic routes are straight-through-routes, which is the common case in dimension-ordered designs [23]. 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

[4] 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.

latency is paid for any turns in the network, and turns into and out of the compute processor. This, and the fact that a message traverses h+1 routers to go h hops, yields a transport latency of 2+h+1 cycles for h hops.

The receive end of the RawDynamic SON is where the major addition of new hardware occurs. Unlike with a static transport, the dynamic transport does not guarantee the ordering of words in the network. The addition of receive-side demultiplexing hardware is required, which examines the operand's numeric index, and places it into a known location in an incoming operand register file (IRF)[5].

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 space available in 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. Similarly, 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 can not 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.

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 over-committed. Reliance on only deadlock recovery for dynamic transport SONs carries two key challenges: First, there is a performance robustness problem due to the slow down 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 backwards 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[14]. Nonetheless, deadlock recovery is a promising solution 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[11] and in the Raw general network[31].

Deadlock Avoidance A deadlock avoidance strategy appears to have a more immediate method of satisfying the synchronization constraints 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

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

double (or more) buffering system for IRF operand slots.

We note also the desireability of a mechanism that changes the numeric IRF indices based on loop interation so that the operands of multiple iterations of the same loop could be live simultaneously (see parallels to register renaming). This is important for executing small loops without excessive overhead.

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

The recently proposed Flit-Reservation Flow Control System[21] suggests that dynamic transport routes can be accelerated by sending the header word ahead of time. As a result, the routing paths can be calculated ahead-of-time so that when the data word arrives, it can be routed without delay. This system has promise, since it is easy to imagine that a compute processor could send out the header signal as soon as it can determine that the instruction will issue, setting up the route as the result operand is being computed. Some recent proposals[17][27] assume that frequency and bandwidth scalable dynamic-transport SONs can be implemented with a 5-tuple of $<0,0,1,0,0>$ or better. Such an assumption can only be justified by a real implementation. Our experience implementing Raw suggests that the attainability of such a 5-tuple for a scalable dynamic-transport SON can not be taken for granted, even assuming the use of Flit-Reservation.

5.2 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. Of interest relative to the previously discussed SDS SON implementation is that this SSS SON requires no special additional hardware structures in order to support sender synchronization, receiver synchronization and demultiplexing, deadlock avoidance and multicast.

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[6] of operands over individual network links using a *static router*. This router fetchs 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 allowed our simple implementation of a static transport network to achieve latencies lower than our aggressive dynamic transport network implementation: 1+h cycles for h hops.

[6] We distinguish this type of transport from a static-timing transport, in which both the ordering *and* timing of operand transport is 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, we note that static-timing transport eliminates much of the control and buffering inherent in routing and thus could afford extremely high throughput, low latency communication.

A static router executes an instruction as follows. In the first couple of cycles, 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, the route procedes. This system is perhaps more tightly synchronized than necessary. An alternative implementation could allow routes to proceed independently, subject to inter-route dependencies.

Receives Once the static transport network has routed an operand to the corresponding receiver, operation 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[7] 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, the Receive Latency of Raw is 1 cycle, thus the overall 5-tuple for RawStatic is <0,1,1,1,0>.

This brief summary of RawStatic is expanded further in [28], [13], and [31].

6 Implementation of the Raw Microprocessor

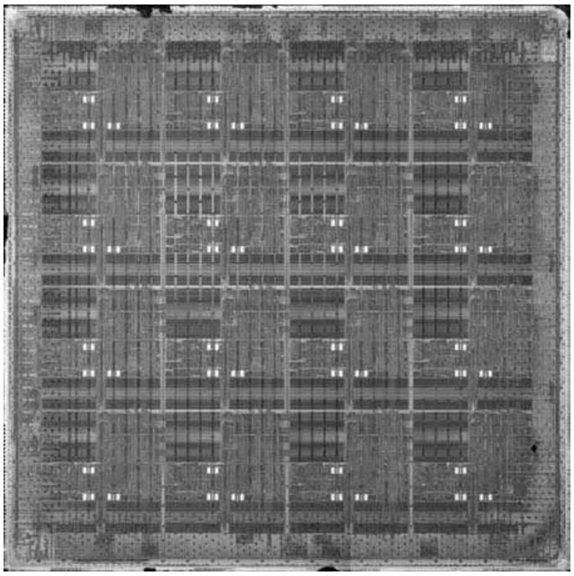

Figure 8. A die photo of the 16-tile Raw chip. The tiles and RAMs are easy to spot. A tile is 4 *mm* x 4 *mm*.

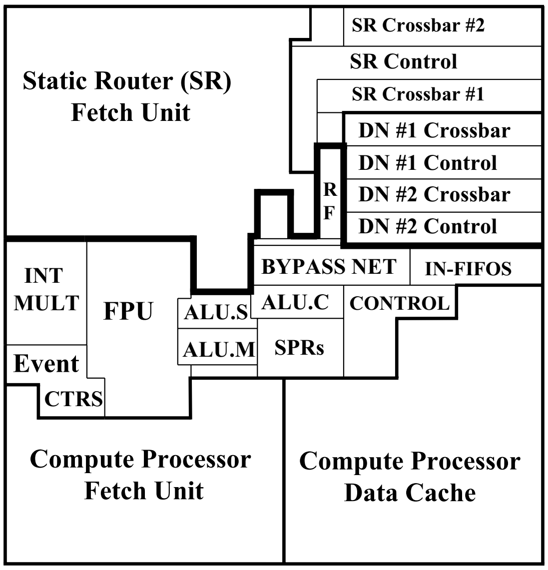

Figure 9. Floorplan of a Single Raw Tile. DN $=$ dynamic network. SR $=$ static router. Everything below the bold border is part of the compute processor. RF $=$ register file.

This section describes the implementation details of the Raw microprocessor. We focus on components used by RawStatic and RawDynamic. We received one hundred and twenty of the 180 nm, 6-layer Cu, 330 mm^2, 1657 pin, 16-tile Raw prototypes from IBM's CMOS7SF fabrication facility in October 2002. The chip core has been verified to operate at frequencies of 425 MHz at the nominal voltage, 1.8V, which is comparable to IBM PowerPC implementations in the same ASIC process. Figure 8 shows a die photo.

[7]Nonetheless, as the Results section of this paper will show, 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.

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 4 separate networks, two static and two dynamic. These networks consist of over 1024 wires per tile.

Physical Design The floorplan of a Raw tile is shown in Figure 9. Approximately 40% 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 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 almost all of the logic area) and thus have little flexibility beyond ensuring that the non-RAM logic is gravitated towards 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. However, the static router has a fetch unit, for which there is no equivalent in the dynamic transport. We note that a comparison of transport area is not sufficient to compare the areas of two SONs. Specifically, a full dynamic transport SON like RawDynamic requires the addition of hardware structures beyond the transport, which are not represented in the Raw floorplan. These structures must be weighed against the static router's fetch unit[8] in order to compare SON area. We estimate the area impact of these additional 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% wider paths in the crossbars in order to provide the same operand bandwidth (as opposed to raw bandwidth) and the same scalability (1024 tiles). Finally, a dynamic transport SON needs to encode one or more destinations in the compute processor instructions themselves, the multiple destinations being used for multicast. Thus, the required compute processor fetch unit RAM size for a given miss ratio may double or even triple, depending on the fanout that a given instruction can specify. The implementation of complete bandwidth-scalable dynamic transport SONs will shed more light on this subject.

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% 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% static transport, 15% dynamic transport, 39% compute processors, and 28% clock. Typical usage is 17W @ 425 MHz, with around 5-10% going to static transport, and 55% 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 [10]. A recent paper examines power tradeoffs in on-chip networks[34].

Examination of Tile-Tile operand path In this portion, we examine specifically the path an operand takes travelling in four clock cycles. First, it travels through the sender's rotate operator inside the ALU, and is latched into a FIFO attached to the sender's static router. The next cycle, the value is routed through the local static router across to the receiver tile's FIFO. A cycle later, it 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

[8] Raw mitigates this area impact by allowing unused static router RAM to be optionally used for compute processor data storage.

register of the receiver ALU. The total non-compute latency is thus three cycles.

Figure 10 super-imposes 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 Figure 9 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 8Kx32 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.[9] 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 the following tables, 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.

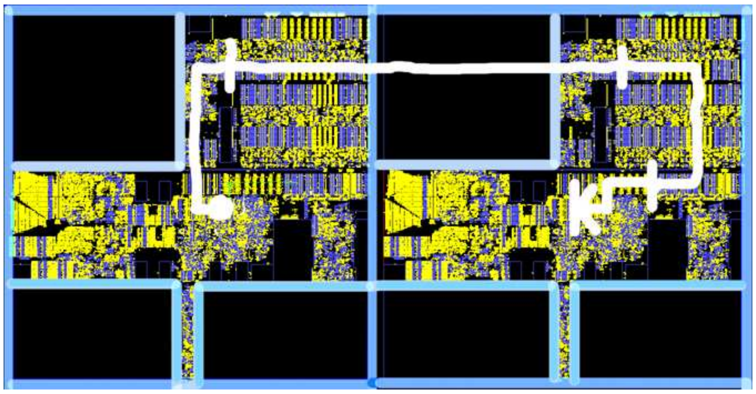

Figure 10. The path of an operand route between two tiles. Path is superimposed over an EDA screenshot of the placed cells. Darker cells are collections of single-bit registers.

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. These two numbers are not exclusive; one cannot simply state the amount of simultaneous output and input slack that could be exploited without examining the full path of gates between input and output. However, for our purposes, one useful statement can be made – that is, that the output slack of one stage and the input slack of the next stage can be combined in order to eliminate a register, if the sum of the slacks is greater than or equal to the cycle time. The elimination of such a register on this path would cause a reduction in the communication latency and thus in the 5-tuple.

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

Of interest in the table to the right 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

[9]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 given here are worst-case numbers, and are extracted using CAD tools with parasitics from the actual wiring of the chip.

cycle of latency could be removed from the 5-tuple; reducing Raw's 5-tuple to $<0,1,1,0,0>$. 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 logic for a route) a half cycle earlier, taking advantage of the early arrival (2.3 ns) of the Valid signal into Cycle 2. However, because the static router can branch based on data values being routed on the processor, there may be a branch penalty for receiving, from the static router's perspective, "late" data.

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 towards 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 the next table, we summarize the timing properties of the operand route in the chip. Because the valid signal is not actually propagated through the operand path, but rather is consumed and regenerated at each step, many of the control paths are relatively complex to describe. We instead descibe the paths in terms of the arrival time of the data portion of the operand as it travels through the circuit. Although this is not as useful as a description of the control paths, it does given an intuition for the delays incurred, especially due to the transport of the data over long distances. Furthermore, the delays shown do include the time to calculate the control for the muxes controlling the flow of the operand, which is often the critical 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

Having examined the Raw SON implementation, we continue with an analysis of SON parameters.

7 Analytical Evaluation of SONs

This section presents results on the impact of SON properties on performance. We focus our evaluation on the 5-tuple cost model. First, we consider the impact each element of the 5-tuple has on performance. We find that occupancy costs have the highest impact. Then, we consider the impact of some factors not modeled directly in the 5-tuple. We find that the impact of contention is not significant, but the availability of multicast has a large effect on performance.

We also compare RawStatic with SONs based on traditional multiprocessor communication mechanisms. Though the multiprocessor mechanisms 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.

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

Simulator Our experiments were performed on Beetle, a validated cycle-accurate simulator of the Raw microprocessor [28]. 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 static SON. All instructions are assumed to hit in the instruction cache.

Beetle has two modes: one mode simulates the Raw prototype's actual static 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.

Memory model The Raw compiler maps each piece of program data to a specific home tile. This home tile becomes the tile that is responsible for caching the data on chip. The distribution of data to tiles is provided by Maps, Raw's compiler managed memory system [4, 12]. Using compiler analysis, Maps attempts to select a distribution that guarantees that any load or store refers to data assigned to exactly one home tile. Dense matrix arrays, for example, usually get distributed element-wise across the tiles. The predictability of the accesses allows memory values to be forwarded from the caches of the home tiles to other tiles via SON.

Compiler Code is generated by Rawcc, the Raw parallelizing compiler [13]. 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.

Before scheduling, Rawcc performs unrolling for two reasons. First, it unrolls to expose more parallelism. Second, unrolling is performed in conjunction with Maps to allow the compiler to distribute arrays, while at the same time keeping the accesses to those arrays predictable.

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

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.

Benchmarks Table 1 lists out benchmarks. Btrix, Cholesky, and Mxm, Vpenta are from Nasa7 of Spec92. Tomcatv and Swim are from Spec95. Fpppp-kernel consumes 50% of the run-time of Spec95's Fpppp. Sha and Aes implement the Secure Hash Algorithm and the Advanced Encryption Standard, respectively. Adpcm is from Mediabench. Jacobi and Life are from the Raw benchmark suite [3]. Fpppp-kernel, Sha and Aes are irregular codes, Moldyn and Unstructured are sparse matrix codes, while the rest are dense matrix codes. The problem sizes of the dense matrix applications have been reduced to cut down on simulation time. To improve parallelism via unrolled loop iterations, we manually applied an array reshape transformation to Cholesky, and a loop fusion transformation to Mxm. Both transformations can be automated.

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

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

	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	xxxx
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	xxxx
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

Table 1. Performance of Raw's static SON <0,1,1,1,0> for two to 64 tiles. Speedups are relative to that on one 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. Fpppp-kernel, Tomcatv, Cholesky, and Mxm contain modest amounts of parallelism and attain seven to 12-fold speedup, respectively. The remaining benchmarks (Btrix, Vpenta, Swim, Jacobi, and Life) have enough parallelism for the speedup to scale well up to 64 tiles, with speedups ranging from 20 up to 80. 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.

The parallelism scalability of the benchmarks can roughly divided into two classes. Most of the dense matrix codes, including Btrix, Vpenta, Mxm, Swim, Jacobi, and Life are able to continually gain performance with increasing number of tiles. For others, the speedups eventually tail off, with 16 tiles being the most frequent point where the performance peaks.

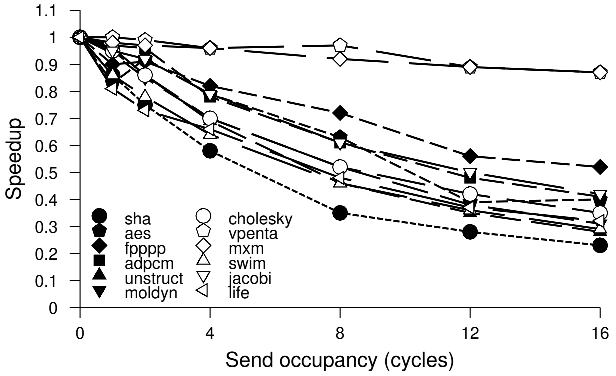

Figure 11. Effect of send occupancy on performance on 64 tiles, *i.e.,* <n,1,1,1,0>.

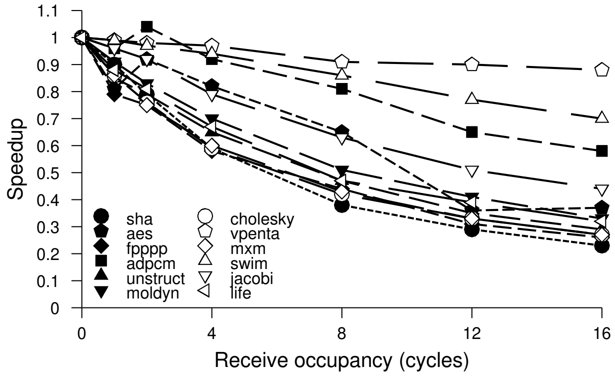

Figure 12. Effect of receive occupancy on performance on 64 tiles, *i.e.,* <0,1,1,1,n>.

Send and receive occupancies Next, we measured the effects of send and receive occupancies on performance. We emphasize that these occupancies are visible to the compiler so that it can account for them as best as it can when it schedules ILP. Figures 11 and 12 graph these effects. All data points are based on 64 tiles. Performance is normalized to that of the actual Raw static SON with 5-tuple <0,1,1,1,0>.

The trends of the two occupancy curves are similar. Overall, results indicate that occupancy impacts performance significantly. Performance drops by as much as 20% even when the send occupancy is increased from zero to one cycle. As expected, the incremental impact of additional cycles of occupancy is less severe.

The slope of each curve is primarily a function of the amount communication and the amount of slack in the schedule. Benchmarks that don't communicate a lot and have a large number of cache misses, such as Vpenta, have higher slacks and are able to better tolerate increase in send occupancies. Benchmarks with fine-grained parallelism such as Sha and Fpppp-kernel have much lower tolerance for high occupancies.

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

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

Figure 13 graphs these latency effects on performance on 64 tiles. A speedup of 1.0 represents

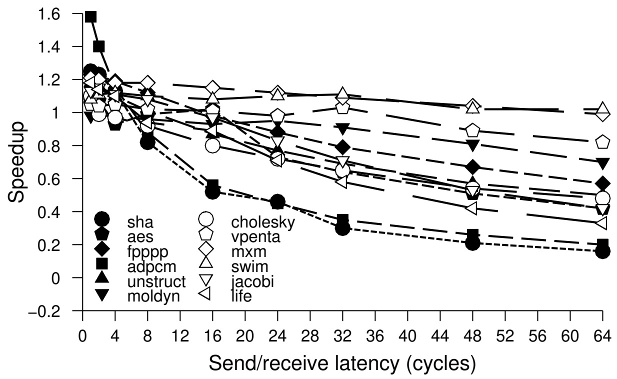

Figure 13. Effects of send or receive latencies on performance on 64 tiles, *i.e.,* <0,n,0,0,0> or <0,0,0,n,0>.

the performance of the Raw SSS SON with 5-tuple <0,1,1,1,0>.

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 necessary correspond to amount of available parallelism: it is possible to have a large amount of fine-grained parallelism, or a small amount of coarse grained parallelism.

Sensitivity to latency varies greatly between the benchmarks. Mxm, Swim, and Vpenta exhibit "perfect" locality and thus coarse grained parallelism, so they are not sensitive to latency at all. Note that in our compilation framework, perfect locality not only requires that the computation exhibit good locality, but that memory accesses also exhibit good locality. Thus a traditionally coarse-grained application like Jacobi has non-perfect memory access locality and is thus sensitive to network latency. The remaining benchmarks have finer grained parallelism and incur slowdown to some degree. Sha and Adpcm, two irregular apps with the least parallelism, also have the finest grained parallelism and suffer the worst slowdowns.

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

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

Figure 14 shows the result of this experiment. In general, the same benchmarks that are insensitive to send/receive latencies (Mxm, Swim, and Vpenta to a lesser degree) are also insensitive to mesh latencies. Of the remaining latency-sensitive benchmarks, a cycle of network hop latency has a higher impact than a cycle of send/receive latency, which means all those applications have at least some communication between non-neighboring tiles.

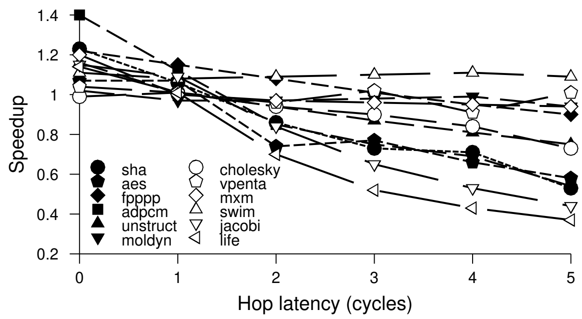

Figure 14. Effects of network hop latency on performance on 64 tiles, *i.e.,* <0,0,n,1,0>.

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

Impact of other factors on performance We now consider some factors not directly modeled in our 5-tuple: multicast and network contention.

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.

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

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: <0,1,1,1,0>. The former models contention while the latter does not. Figure 15 plots this

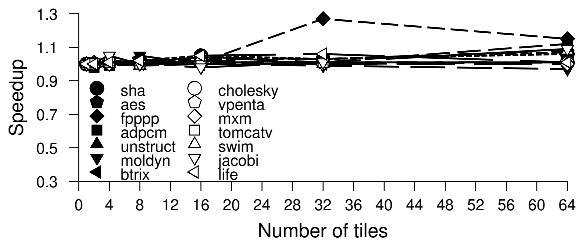

Figure 15. Impact of removing contention.

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%.

Benefit of locality driven placement We conduct the following experiment to study the benefit of locality driven placement on performance. Normally, Rawcc assigns instructions to virtual tiles, then it assigns virtual tiles to physical tiles by placing in close proximity virtual tiles that communicate a lot with each other. In this experiment, we replace locality driven placement with a random one, and we measure its effects on both execution time and route count, which is a count on the number of routes performed on the static network. The change in execution time measures the end-to-end benefit of locality-driven placement. The change in route count measures the increase in network traffic due to the loss of locality.

Benchmark	Slowdown	Change in Route Count	ILP	Routes /Cycle
Sha	51%	89%	2.4	2.2
Aes	12%	68%	3.8	1.3
Fpppp-kernel	10%	52%	7.6	5.2
Adpcm	22%	75%	0.9	1.3
Unstructured	11%	187%	1.7	0.2
Cholesky	5%	9%	9.1	2.1
Vpenta	0%	0%	81.4	0.7
Mxm	0%	-23%	11.9	1.8
Swim	0%	79%	24.3	2.7
Jacobi	17%	122%	62.1	9.0
Life	54%	151%	48.7	13.9

Table 2. Impact of locality on performance.

Table 2 presents the results of this experiment on 64 tiles. To help understand the results, the table includes the following data for the 64-tile base case: the level of exploited ILP, defined to be the speedup of the application on 64 tiles over a single tile; and the average route count per cycle.

For about one third of the applications, their performance is not adversely effected by randomized placement. These applications all have high ILP and low route count per cycle. Interestingly, an application may be sensitive to the method of placement either because it has low ILP, or because it has high route count per cycle. Applications with low ILP (Sha, Aes, Fpppp-kernel, Adpcm, Unstructured) only profitably uses a subset of those tiles. Randomizing tile placement thus destroys much of the application locality and leads to significant slowdown and higher network utilization. On the other hand, having a high route count indicates that an application communication a lot. Unless each tile communicates with every other tile with the same frequency, the application would benefit from locality driven placement.

The results demonstrate that fast access to a few tiles is more important than uniformly medium-speed access to all the tiles. Thus, a mesh SON will likely have better performance than a crossbar SON.

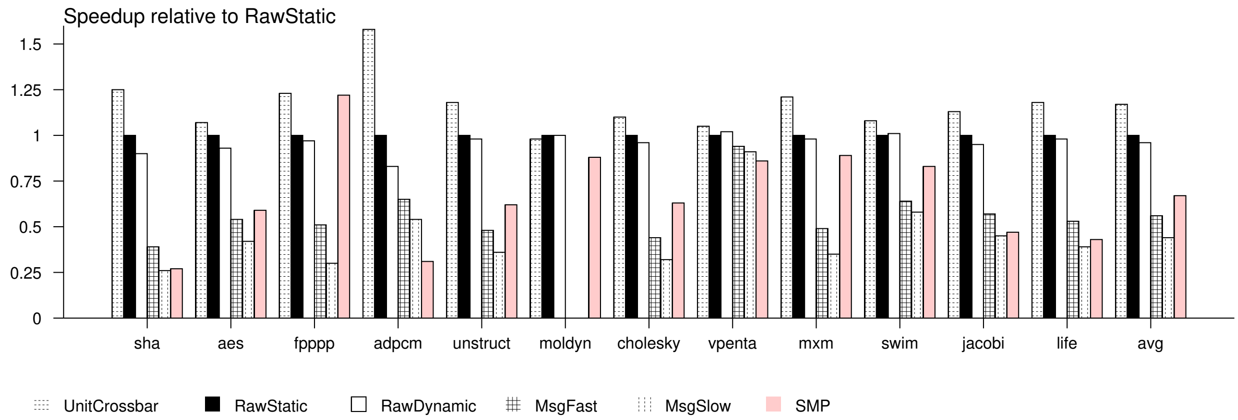

Figure 16. Performance spectrum of SONs.

Performance of multiprocessor-based SONs vs 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 Section 3, with the optimistic assumption that the commit latency, c, is zero.

Figure 16 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 (<0,1,1,1,0>, 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% to 56% worse than RawStatic.

Summary The results in this section provide some validation that our 5-tuple cost model is useful. It suggests that the model captures to first order the performance of operand networks. [10]

8 Operand Analyses

To better understand the flow of operands inside an SON, we analyse the operand traffic at receiver nodes.

Analysis of Operand Origins Figure 17a analyses the origin of operands used by the receiver nodes. *Fresh* operands are computed locally, making use of 0-cycle local bypassing or the local register file; *remote* operands arrive over the inter-tile transport network; and *reload* operands originate from a previous spill to the local data cache. The figure present these results for each benchmark, varying the number of tiles from 2 to 64. An operand is counted only once per generating event (calculation by a functional unit, arrival via transport network, or reload from cache) regardless of how many times it is used. Although these numbers are generated using Raw's SSS SON, the numbers should be identical for an SDS SON with the same 5-tuple. Not surprisingly, remote operands are least frequent in dense matrix benchmarks, followed by sparse matrix benchmarks, followed by the irregular benchmarks.

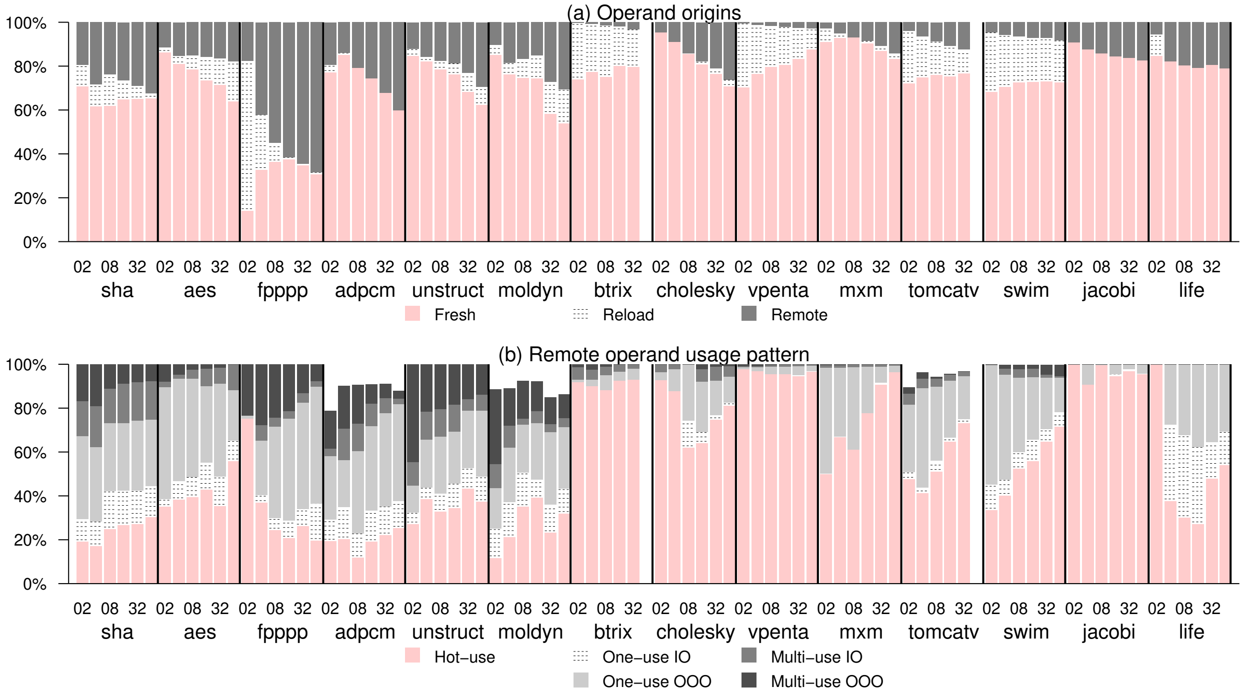

Figure 17. a) Analysis of operand origins. b) Analysis of remote operand usage pattern. For each benchmark there are six columns, one for each power of two configuration from two to 64.

Analysis of Remote Operand Usage Pattern We further analyze the remote operands from Figure 17a by their usage pattern. The data measures the ability of the compiler to schedule sender and receiver instruction

[10]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.

	Superscalar	Distributed Shared Memory	Message Passing	Raw	RawDynamic	Grid	ILDP
Tuple (c:commit time)	$<$0,0,0,0,0$>$	$<$1,14+c,2,14,1$>$	$<$3,3+c,1,1,12$>$	$<$0,1,1,1,0$>$	$<$0,2,1,2,0$>$	$<$0,0,n/8,0,0$>$ $0 \leq n \leq 8$	$<$0,n,0,1,0$>$ 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

Table 3. Survey of SONs. AsTrO consists of **As**signment, operand **Tr**ansport, and operation **O**rdering.

sequences so as to allow values to be delivered just-in-time. Such scheduling impacts the optimal size of the IRF in the case of the SDS SON. In the case of the SSS SON, it reflects the importance of a facility to time-delay or re-cast operands, as mentioned in Section 5. This facility spares the compute processor from having to insert move instructions (in effect, a hidden receive occupancy) in order to preserve operands that have not arrived exactly when needed.

We classify the remote operands at the receiver node as follows. *Hot-uses* are directly consumed off the network and incur no occupancy. *One-uses* and *Multi-uses* are first moved into the register file and then used – *One-uses* are subsequently used only once, while *Multi-uses* are used multiple times. We also keep track of whether the operand arrives in the same order in which a tile uses them. An SON operand is *in-order (IO)* if all its uses precede the arrival of the next SON operand; otherwise it is *out-of-order (OOO)*.

Figure 17b shows the relative percentages of each operand class.[11] By far the most significant use-type are Hot-Use operands, followed by One-use OOO operands. We can compute the effective receive occupancy by summing the fraction of operands that are not Hot-Uses. This value varies greatly between the benchmarks, ranging from around 0 (100% Hot-uses) to 0.8 (20% Hot-uses). The average effective occupancy over all the benchmarks for 64 tiles is 0.44. Recall that although this occupancy is less than one, Figure 11 shows that even a single cycle of occupancy can have a large effect on performance.

We intend to extend the Raw compiler so that it utilizes the static routers' existing register files and reduce this occupancy to nearly zero. These register files can be used to time-delay OOO operands, and to re-cast multi-use operands into the compute processor. Future work will determine the optimal of this structure.

9 Related Work

Table 3 contrasts a number of commercial and research systems and the approaches that they use to overcome the challenges of implementing SONs. The first section of the table summarizes the 5-tuples, the AsTrO category, and the number of ALUs and fetch units supported in each of the SONs. Note that

[11] Due to how the compiler handles predicated code, a few of the benchmarks have SON operands that are not consumed at all, which explains why a few of the frequency bars sum up to less than 100%.

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 intra-tile 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.

This paper extends an earlier framework for operand networks that was given in [30], and expands upon the work described in [29].

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 latencies. Even though the microarchitectures are distributed, ILP can be exploited on these SONs because their latencies 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. The challenges include maintaining high frequencies, dealing with limited bandwidth related to scalable networks, implementing ultra-low cost operation-operand matching, avoiding deadlock and starvation, and achieving correct operation in the face of exceptional events. The paper looks at several recently proposed distributed architectures that exploit ILP and discusses how each addresses the challenges, using a new AsTrO taxonomy to distinguish between their logical properties. This paper also describes two SON designs in the context of the Raw microprocessor and discusses how the implementations deal with each of the challenges in scaling SONs.

This paper breaks down the latency of operand transport into five components $<$SO, SL, NHL, RL, RO$>$: send occupancy, send latency, network hop latency, receive latency, and receive occupancy. The paper evaluates several families of SONs based on this 5-tuple. Our early results show that send and receive occupancies have the biggest impact on performance. For our benchmarks, performance decreases by up to 20 percent even if the occupancy on the send or receive side increases by just 1 cycle. The per-hop latency follows closely behind in importance. Hardware support for multicast has a high impact on performance for large systems. Other parameters such as send/receive latencies and network contention have smaller impact.

In the past, the design of SONs was closely tied in with the design of other mechanisms in a microprocessor – for example, register files and bypassing. In this paper, we attempt to carve out the generalized SON as an independent architectural entity that merits its own research. We believe that research focused on the scalar operand network will yield significant simplifications in future scalable ILP processors.

Avenues for further research on scalar operand networks are plentiful, and relate to both a re-evaluation of existing network concepts in the scalar operand space, and the discovery of new SON mechanisms. A partial list includes: (1) Designing SONs for a high clock rate while minimizing the 5-tuple parameters, (2) evaluating the performance for much larger numbers of tiles and a wider set of programs, (3) generalizing SONs so that they support other forms of parallelism such as stream and thread parallelism in SMT-style processing, (4) exploration of both dynamic and static schemes for assignment, transport and ordering, (5) mechanisms for fast exception handling and context switching, (6) analysis of the tradeoffs between commit point, exception handling, and send latency, (7) low energy SONs, and (8) techniques for deadlock avoidance and recovery in SONs.

Acknowledgments Raw is funded by Darpa, NSF, ITRI and the Oxygen Alliance.

References

[1] AGARWAL, HRISHIKESH, KECKLER, AND BURGER. Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. In *International Symposium on Computer Architecture* (2000).

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

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

[4] BARUA, LEE, AMARASINGHE, AND AGARWAL. Maps: A Compiler-Managed Memory System for Raw Machines. In *International Symposium on Computer Architecture* (1999).

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

[6] DUATO, AND PINKSTON. A General Theory for Deadlock-Free Adaptive Routing using a Mixed Set of Resources. *IEEE Transactions on Parallel and Distributed Systems 12*, 12 (December 2001).

[7] GROSS, AND O'HALLORON. *iWarp, Anatomy of a Parallel Computing System*. The MIT Press, Cambridge, MA, 1998.

[8] JANSSEN, AND CORPORAAL. Partitioned Register File for TTAs. In *International Symposium on Microarchitecture* (1996).

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

[10] KIM, TAYLOR, MILLER, AND WENTZLAFF. Energy Characterization of a Tiled Architecture Processor with On-Chip Networks. In *International Symposium on Low Power Electronics and Design* (2003).

[11] KUBIATOWICZ, AND AGARWAL. Anatomy of a Message in the Alewife Multiprocessor. In *International Supercomputing Conference* (1993).

[12] LARSEN, AND AMARASINGHE. Increasing and Detecting Memory Address Congruence. In *International Conference on Parallel Architectures and Compilation Techniques* (2002).

[13] LEE ET AL. Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine. In *Conference on Architectural Support for Programming Languages and Operating Systems* (1998).

[14] MACKENZIE, KUBIATOWICZ, FRANK, LEE, LEE, AGARWAL, AND KAASHOEK. Exploiting Two-Case Delivery for Fast Protected Messaging. In *HPCA* (July 1997).

[15] MAI, PAASKE, JAYASENA, HO, DALLY, AND HOROWITZ. Smart Memories: A Modular Reconfigurable Architecture. In *International Symposium on Computer Architecture* (2000).

[16] NAFFZIGER, AND HAMMOND. The Implementation of the Next-Generation 64b Itanium Microprocessor. In *IEEE International Solid-State Circuits Conference* (2002).

[17] NAGARAJAN, SANKARALINGAM, BURGER, AND KECKLER. A Design Space Evaluation of Grid Processor Architectures. In *International Symposium on Microarchitecture* (2001).

[18] PALACHARLA, JOUPPI, AND SMITH. Complexity-Effective Superscalar Processors. In *International Symposium on Computer Architecture* (1997).

[19] PANDA, SINGAL AND KESAVAN. Multidestination message passing in Wormhole k-ary n-cube Networks with Base Routing Conformed Paths. In *IEEE Transactions on Parallel and Distributed Systems* (1999).

[20] PARCERISA, SAHUQUILLO, GONZALEZ, AND DUATO. Efficient Interconnects for Clustered Microarchitectures. In *International Conference on Parallel Architectures and Compilation Techniques* (2002).

[21] PEH, AND DALLY. Flit-Reservation Flow Control. In *International Symposium on High-Performance Computer Architecture* (2000).

[22] SANKARALINGAM, SINGH, KECKLER, AND BURGER. Routed Inter-ALU Networks for ILP Scalability and Performance. In *International Conference on Computer Design* (2003).

[23] SEITZ, BODEN, SEIZOVIC, AND SU. The Design of the Caltech Mosaic C Multicomputer. In *Research on Integrated Systems Symposium Proceedings* (Cambridge, MA, 1993), MIT Press.

[24] SOHI, BREACH, AND VIJAYKUMAR. Multiscalar Processors. In *International Symposium on Computer Architecture* (1995).

[25] SONG, AND PINKSTON. A Progressive Approach to Handling Message Dependent Deadlocks in Parallel Computer Systems. *IEEE Transactions on Parallel and Distributed Systems 14*, 3 (March 2003).

[26] SULLIVAN, AND BASHKOW. A Large Scale, Homogeneous, Fully Distributed Parallel Machine. In *4th Annual Symposium on Computer Architecture* (1977).

[27] SWANSON, MICHELSON, SCHWERIN, AND OSKIN. WaveScalar. In *International Symposium on Microarchitecture* (2003).

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

[29] TAYLOR, LEE, AMARASINGHE, AND AGARWAL. Scalar Operand Networks: On-Chip Interconnect for ILP in Partitioned Architectures. In *International Symposium on High Performance Computer Architecture* (2003).

[30] TAYLOR ET AL. How to Build Scalable On-Chip ILP Networks for a Decentralized Architecture. Tech. Rep. 628, MIT, April 2000.

[31] TAYLOR ET AL. The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Prgrams. *IEEE Micro* (March 2002).

[32] VON EICKEN, CULLER, GOLDSTEIN, AND SCHAUSER. Active Messages: A Mechanism for Integrated Communication and Computation. In *19th International Symposium on Computer Architecture* (May 1992).

[33] WAINGOLD ET AL. Baring It All to Software: Raw Machines. *IEEE Computer 30*, 9 (Sept. 1997).

[34] WANG, PEH AND MALIK. Power-Driven Design of Router Microarchitectures in On-Chip Networks. In *International Symposium on Microarchitecture* (2003).

11 Supplemental Section: Shared Memory operation-operand matching

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

The send and receive occupancies of this approach are difficult to evaluate. The sender's store instruction and receiver's load instruction only occupy a single instruction slot; however, the processors may still incur an occupancy cost due to limitations on the number of outstanding loads and stores. The send latency 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 non-network time required for the cache protocols to process the receiver's request for the line from the sender's cache.

This approach has number of benefits: First, it supports multicast (although not in a way that saves bandwidth over multiple unicasts). Second, it allows a very large number of live operands due to the fact that the physical register file is being implemented in the cache. Finally, because the memory address is effectively a tag for the value, no software instructions are required for demultiplexing. In [29], we estimate the 5-tuple of this relatively aggressive shared-memory SON implementation to be $<$1,14+c,2,14,1$>$.

12 Supplemental Section: Systolic array operation-operand matching

Systolic machines like iWarp [7] were some of the first systems to achieve low-overhead operation-operand matching in large-scale systems. iWarp sported register-mapped communication, although it is optimized for transmitting streams of data rather than individual scalar values. The programming model supported a small number of pre-compiled communication patterns (no more than 20 communications streams could pass through a single node). For the purposes of operation-operand matching, each stream corresponded to a logical connection between two nodes. Because values from different sources would arrive via different logical streams and values sent from one source would be implicitly ordered, iWarp had efficient operation-operand matching. It needed only execute an instruction to change the current input stream if necessary, and then use the appropriate register designator. Similarly, for sending, iWarp would optionally have to change the output stream and then write the value using the appropriate register designator. Unfortunately, the iWarp system is limited in its ability to facilitate ILP communication by the hardware limit on the number of communication patterns, and by the relatively large cost of establishing new communication channels. Thus, the iWarp model works well for stream-type bulk data transfers between senders and receivers, but it is less suited to ILP communication. With ILP, large numbers of scalar data values must be communicated with very low latency in irregular communication patterns. iWarp's five-tuple can modeled as $<1,6,5,0,1>$ - one cycle of occupancy for sender stream change, six cycles to exit the node, four or six cycles per hop, approximately 0 cycles receive latency, and 1 cycle of receive occupancy. An on-chip version of iWarp would probably incur a smaller per-hop latency but a larger send latency because, like a multiprocessor, it must incur the commit latency cost before it releases data into the network.