# aSOC: A Scalable, Single-Chip Communications Architecture

Jian Liang, Sriram Swaminathan, and Russell Tessier
Department of Electrical and Computer Engineering
University of Massachusetts
Amherst, MA. 01003.
{jliang, tessier}@ecs.umass.edu

## Abstract

*As on-chip integration matures, single-chip system designers must not only be concerned with component-level issues such as performance and power, but also with on-chip system-level issues such as adaptability and scalability. Recent trends indicate that next generation systems will require new architectures and compilation tools that effectively deal with these constraints. In this paper, a new single-chip interconnect architecture,* **a***daptive* **S***ystem-***O***n-a-***C***hip, is described that not only provides scalable data transfer, but also can be easily reconfigured as application-level communication patterns change. An important aspect of the architecture is its support for compile-time, scheduled communication. To illustrate the benefits of the architecture, three DSP benchmarks have been mapped to candidate SoC devices of assorted sizes which contain the new interconnect architecture. The described interconnect architecture is shown to be up to 5 times more efficient than bus-based SoC interconnect architectures via parallel simulation. Additionally, a preliminary layout of our architecture is shown and derived area and performance parameters are presented.*

## 1 Introduction

The steady increase of VLSI transistor capacity offers the promise of high-performance, single-chip computing in the near future. The most recent Semiconductor Industry Association (SIA) [2] roadmap predicts mass production of devices with over 1.4 billion transistors by 2012. Such dramatic shifts in available technology have already started to affect the way ASICs are designed and produced. System-on-a-chip integrators must now consider a large number of design issues, both architectural and physical, when assembling new architectures. Some of the most important issues facing SoC growth and acceptance include the design of intellectual property cores, the on-chip coordination of communication between cores, and the development of effective system-wide compilation environments. In this paper, we address the first two of these concerns through the development of a new SoC inter-core communication architecture and supporting architectural simulator. The communication architecture is scalable to large numbers of cores and can be reconfigured on a per-application basis to achieve adaptable performance.

Inter-chip bandwidth has long been recognized as a limiting factor in board-level system design. This bottleneck can be largely attributed to two factors: capacitive off-chip signal delays and a need for growing numbers of functional components to share common interconnect resources. While the former issue has been recently addressed through migration of individual components to SoC substrates, scalable interconnect continues to be an issue, even for SoC environments. As large single-chip systems are designed, on-chip global communication is likely to become the limiting factor to overall system performance. Dynamic arbitration for shared communication resources across even a small number of components can quickly form a performance bottleneck and long, heavily-loaded on-chip buses are likely to lead to long global communication cycles, limiting system throughput.

Our approach to on-chip communication is to supplement each SoC intellectual property core with a small, highly-optimized communication *interface*. As shown in Figure 1, each core and associated interface effectively forms a computational *node*. Communication between nodes takes place via pipelined, point-to-point connections. By limiting inter-core communication to short wires exhibiting predictable performance, high-speed communication can be achieved. For many SoC applications it is expected that communication between nodes can primarily be predicted at compile time and inter-node communication can be performed without the need for significant dynamic arbitration. This compile-time approach to scalable system-wide communication is drawn from previous work in coarse-grained parallel processing. In a number of previ-

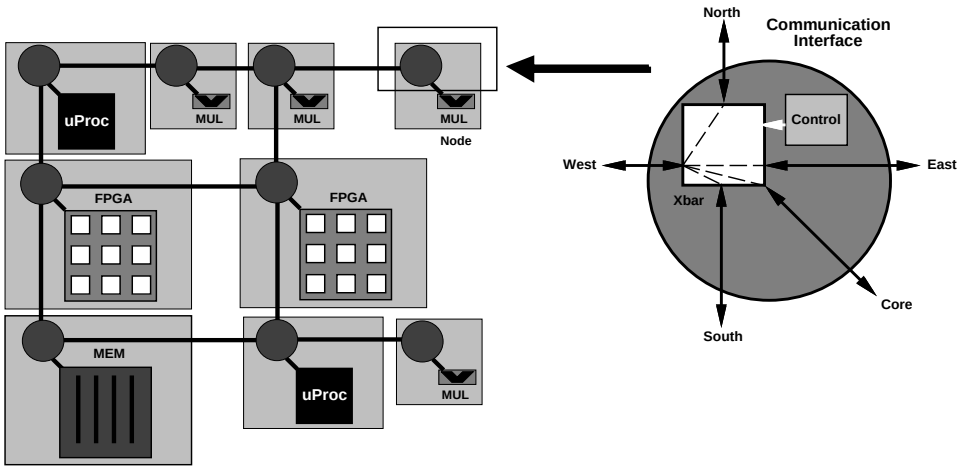



**Figure 1. Adaptive System-on-a-Chip (aSOC)**

ous projects [7] [18] [21], distributed, pipelined routing has been used to moderate global communication cycles across large parallel systems.

The availability of instruction-configurable processors and bitstream-configurable FPGA resources brings an important aspect of hardware adaptability to SoC environments. It is our belief that not only must the functionality of individual IP cores change from application-to-application, but also that *communication resources* between cores should be allocated on a per-application basis. Since communication in statically-scheduled systems is predictable and easily-verified, we believe that reconfiguration of intra- and inter-core functionality will be simplified, fulfilling our goal of an adaptive system-on-a-chip (aSOC).

In this paper we evaluate the performance of 9 and 16 core aSOC devices with three common DSP benchmarks. Through the use of a parallel simulator, direct comparisons can be drawn between the new on-chip interconnect architecture and traditional on-chip buses. A preliminary layout of the interface yields area and performance estimates relative to the node cores. Available architectural simulators have been integrated into our simulation environment to provide for accurate evaluation of on-chip systems containing heterogeneous cores. An initial layout of the interface shows that it can be constructed efficiently and can allow for fast, pipelined performance.

## 2 Related Work

While there are numerous current proposals for on-chip interconnect, choices can generally be divided into two categories: arbitrated bus and packet-switched network. The most common communication model used for on-chip communication is the multi-component, arbitrated bus. This medium is characterized by a need for bus arbitration which results in a single bus master. Several commercial ventures have focused attention specifically towards on-chip interconnect. Sonics, Inc. [19] allows IP designers the capability to add a standard bus interface to cores, facilitating arbitrated on-chip bus transfer. Similarly, the Virtual Socket Interface Alliance (VSIA) has attempted to set the characteristics of this interface industry-wide through the creation of an on-chip bus standard. Three commercial on-chip interconnects, Wishbone [16], AMBA [10] and CoreConnect [11], support the connection of multiple buses in arbitrary topologies. All of the systems mentioned above, while flexible, appear to have limited scalability due to the arbitrated, non-pipelined nature of their interconnection buses. While it is likely that a small number of cores can be accommodated by a bus, new approaches will be necessary as system sizes scale. Additionally, for many applications communication patterns between modules are largely predictable at compile time limiting the need for constant run-time arbitration.

Packet-switched interconnect based on both compile-time static and run-time dynamic routing has been used effectively for multiprocessor communication for over 25 years. In general, a large majority of multiprocessor networks have used purely run-time dynamic route determination to simplify software development. While many applications require at least some dynamic route support, in many cases static routing can be used for most, if not all, communication. Compile-time static routing of communication has been used effectively in a number of parallel processing systems. In iWarp [7], inter-processor communication patterns were determined during program compilation and implemented with the aid of programmable, inter-processor buffers. This concept was extended by the NuMesh project [18] to include collections of heterogeneous processing elements interconnected in a mesh topol-

ogy. Both of these early systems incorporated a single processing element per chip and were primarily applied to signal processing applications which exhibited well-defined communication patterns. More recently, the Reconfigurable Architecture Workstation (RAW) project [21] has re-examined static communication as a mechanism for general-purpose computing. The compiler for this single-chip system automatically partitions program fragments across multiple homogeneous processing elements and then determines inter-processor communication using a space-time scheduler.

## 3 Scheduled Communication Architecture

Our experimental architecture has been developed with the belief that most, if not all, communication in data-intensive applications can be determined at compile-time. This approach emphasized hardware minimization and interconnect performance at the cost of some flexibility. While the architecture described in this paper currently supports some data-dependent routing capabilities in terms of the *number* of data values transferred between a source and destination, dynamic determination of data destinations at run-time is not supported. Extensions to the architecture to more fully support run-time destination determination are currently underway and a high-level view of the approach is described in Section 6.

Through the use of mapping approaches described in Section 3.2, it is possible to pre-determine communication in a statically-scheduled aSOC system. In general, for static routing, not all inter-core values need to be communicated on every cycle. As a result, inter-core values are multiplexed eliminating the overhead found in point-to-point circuit-switched approaches. A benefit of this approach is that inter-component communication can be pipelined, keeping communication throughput high and communication delays predictable.

As shown in Figure 1, since our conceptual system is regular and based on an optimized node, device families can be easily extended and device testability is enhanced. As a result of system scalability, numerous heterogenous cores can be linked to each other via communication interfaces. Since the communication interface is isolated from computing components, the architecture can make full use of resources such as processors, FPGAs, multiplier-accumulators, and other embedded components.

Before examining the detailed implementation of the intra-node communication interface, the architectural pieces of the interface are examined. A high-level view of the interface architecture, shown in Figure 2, reveals four distinct logical parts: FIFOs which buffer IP core data values, the *interface crossbar* where data values are physically exchanged between neighboring nodes and the intellectual

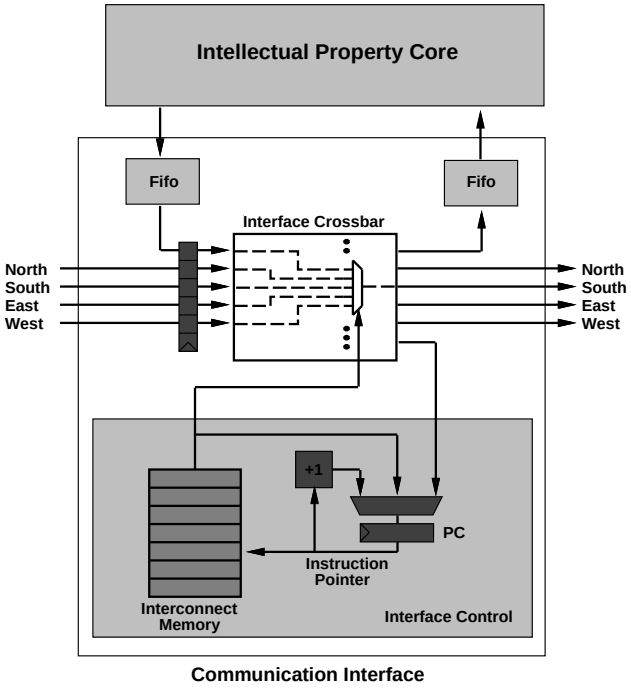



**Figure 2. Core and Communication Interface**

property core, an interconnect memory that holds per-cycle multiplexer select settings, and an instruction pointer (PC) that selects a single interface configuration stored in the configuration memory on a given clock cycle. Effectively, this type of high-level interface architecture can be thought of as a *reconfigurable communications processor* with instruction pointer (configuration pointer) and instructions (configuration memory). In a purely systolic mode, the instruction pointer repetitively selects the same sequence of communication instructions over and over again in a compiler-determined fashion. The multiplexer and associated logic in the instruction pointer section of the interface can compare data values taken from the crossbar to promote branches by changing the instruction pointer (PC) to point to a new sequence of crossbar settings. By using this approach, each inter-node wire may be assigned data from a different source on each communication cycle, effectively applying communication resources where they are needed.

The FIFO interface between a core and an interface crossbar allows for synchronization between potentially differing clock domains as well as some buffering capability. Synchronization between cores is aided by the use of a one-bit tag value that is transferred with each data value through the system to indicate data validity. If an attempt is made to read an empty FIFO, a data value with a tag labelled invalid will be obtained. If invalid data arrives at a core, its value is ignored. The interface FIFOs that were implemented and used in simulation were arbitrarily chosen

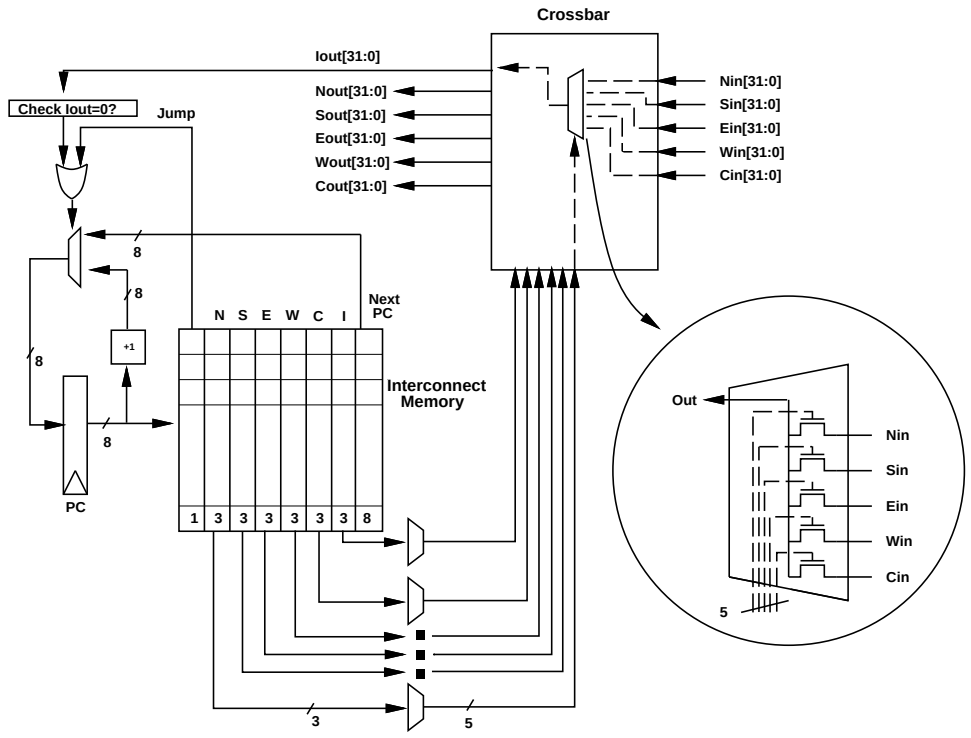



**Figure 3. Detailed Communication Interface**

to be of a depth of four to limit buffer area while providing multiple storage locations.

A similar, fixed-hardware implementation of the logical function of the communication interface has previously been described in [6]. In the earlier implementation, which was targeted to ASICs, the logical function of the communication interface controller was defined on a per-node basis as a large, fixed state machine. This interface circuitry, which was generated by the $DeepC$ compiler, was then synthesized and implemented in ASIC hardware and could not be changed from application-to-application. Our approach provides a more configurable solution at the cost of a programmable interconnect memory. The memory can be reprogrammed to accommodate a set of *different* applications using the same computing device. Additionally, the communication controller described in this paper interfaces to a heterogenous set of IP cores, each running at potentially different clock rates.

### 3.1 Communication Interface

A detailed view of the interconnect memory sequencer and interface crossbar (minus tags and FIFOs) appears in Figure 3. Each instruction accessed from the interconnect memory contains a number of data fields. Each of the six three-bit fields represents the source port for one of the interface crossbar output ports. Enabling pass transistors within the crossbar are driven by the output of three-to-five decoders. Additional fields within each communication instruction indicate the interconnect memory branch address, a comparison-select enable bit, and a bit to force a sequencer jump.

In most cases the PC increments after each communication clock cycle to point to the next instruction in the interconnect memory. In the figure it can be seen that a path from the crossbar to the interface control does allow for some run-time routing decision-making regarding transfer lengths. The functionality of this circuitry is best illustrated through the use of a brief example. Consider the multi-step transfer of data from the local IP core (FIFO port $C_{in}$) to a set of destination ports. It is known at compile time that the multi-step transfer will be performed a number of times, but the exact number of sequences is not known. One iteration of the sequence is as follows:

```
1: Fifo out (Cin) -> South port (Sout)
2: Fifo out (Cin) -> West port (Wout)
3: Fifo out (Cin) -> East port (Eout)
4: Fifo out (Cin) -> Interface port (Iout)
```

The first three transfers can be accomplished by three consecutive state words stored in the interconnect memory which, after decoding, configure the local crossbar. For these three steps, the PC is incremented following each step completion. During the fourth step, a count value from the

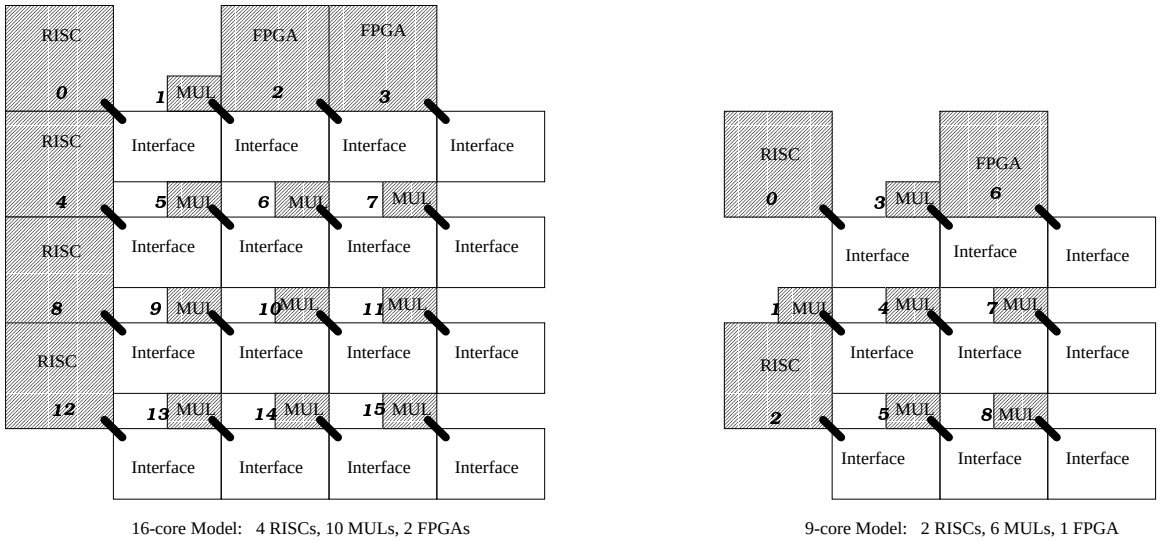



**Figure 4. 9-core and 16-core Model**

IP core is tested by the comparator to see if it has reached a value of zero. If this value has been reached, the PC is allowed to increment once again and the next series of communication patterns is started. Otherwise, the PC is reset to the address of the state word for Step 1 (stored in $NextPC$) and the sequence is started again. This type of repetitive data transfer over a fixed set of iterations is similar to the mechanism used for DMA transfer in current bus architectures. Note that the $Jump$ signal from the interconnect memory can also be used to force the PC to jump to the stored $NextPC$ location to create repetitive loops. In the absence of $Jump$ or a positive value from the comparator, the PC will increment to point to the next interconnect memory location.

### 3.2 Application Mapping

In this section the steps needed to map an application to an aSOC device are described. While for this paper these steps were performed using both software tools and manual intervention, an effort is currently underway to automate the entire compilation process.

1. The first stage of mapping is the assignment of computation to cores. An application is partitioned onto heterogeneous cores based on the nature of computation. For example, high-density multiplication is assigned to multiplier-accumulators, bit operations are targeted to the FPGAs, and sequential code is assigned to the RISCs.

2. Portions of the application are written in a format appropriate for the specific core. This includes C code for the MIPs R4000, RTL Verilog for the FPGA, and a simple RTL for multiply-accumulate.

3. Sources are compiled for target technologies. For example, GCC is used to target the MIPS R4000 and Quartus software is used to synthesize the FPGA logic design.

4. Data transfer between cores is broken into communication streams which follow shortest paths from source to destination cores. Where possible, communicating streams that pass through a communication interface are combined to allow multiple data transfers on the same communication clock cycle through the interface crossbar.

5. Stream data that passes through a communication interface is *scheduled* for a specific communication clock cycle based on data link availability. Note that since communication is coordinated throughout the entire system, the schedules of communication interfaces are coordinated. The result of scheduling for each interface is a set of instructions for its associated interconnect memory.

### 3.3 IP Core Models

For this paper three types of cores (Altera FPGA, MIPS R4000, and multiplier-accumulators) are used in 9 and 16 core configurations. As shown in Figure 4, the 9-core model contains 2 RISCs, 1 FPGA and 6 multiplier-accumulators. The 16-core model contains 4 RISCs, 2 FPGAs and 10 multiplier-accumulators. The diagram shows a rough layout of the various components and their interconnection.

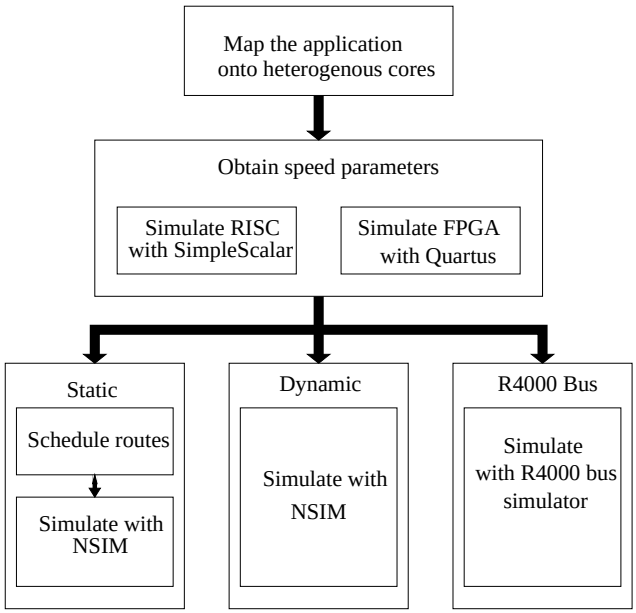



**Figure 5. Evaluation Methodology**

While each SoC implementation is roughly a mesh, cores are spaced to accommodate varying core sizes.

## 4 Evaluation Methodology

To evaluate the benefits of aSOC interconnect, architectural simulators for FPGA logic, a MIPs R4000, and a multiply-accumulate unit were used. These simulators were integrated with NSIM [15], an interconnect simulator that supports both static and dynamic routing protocols. The flow used to perform the simulations appears in Figure 5. For each core, an individual component simulator was used to determine core execution time until data transfers between cores were needed. The network simulator was used in response to communication requests to simulate data transfer in the network. Specific information about the simulators includes:

- *SimpleScalar simulator* [8] - This scalar microprocessor simulator provides a cycle-accurate simulation of the MIPS R4000 architecture. The SimpleScalar simulator was used to approximate the operation of a RISC processor.
- *Altera Quartus simulator* [5] - This simulator determines the behavior of RTL-level circuits targeted to FPGAs. FPGA logic block counts were determined by synthesizing designs to hardware using the embedded Quartus synthesis tools.
- *Nsim Parallel Interconnect Simulator* [15] - This simulator determines communication time for mesh-type networks based on both static and dynamic communication. The operation of multiple communication interfaces can be evaluated simultaneously through the invocation of multiple processing threads. For dynamic routing, packet headers are used to determine data movement.

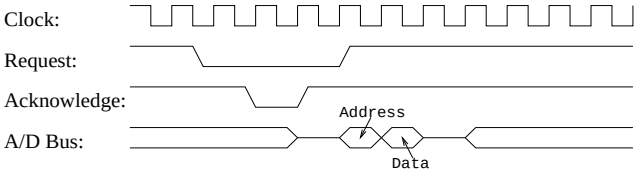



**Figure 6. MIPS R4000 Bus Arbitration**

To perform accurate simulation, execution-rate parameters were set for each component. The per-core clock rates used in our evaluation are discussed in Section 5. Execution-rate parameters were obtained by running the simulators independently and using polling at the cores to determine when requested data arrived. NSIM schedules data transfer so that accurate communication cycle times can be determined. Following application execution, post-run statistics can be gathered to determine the number of cycles needed for computation and communication.

To compare the benefits of a scheduled interconnect with an on-chip bus, a bus simulator was created based on the arbitrated processor bus of the MIPS R4000 architecture [3]. The bus was set to run at its highest architectural speed of 280MHz, which is the same as the clock speed of the R4000 processor. As shown in Figure 6, it takes 5 clock cycles to finish a write/read transaction. Needed cycles include **request**, **acknowledge**, **address**, **data**, and **release**. When the bus is granted to one master, all other requests remain pending until the bus becomes available once again. The master is chosen randomly by the arbitrator if more than one request arrives during the same cycle.

### 4.1 Benchmark Designs

The benchmarks used in our evaluation were selected from several important application domains: communications, multi-media, and image processing. Three benchmarks were selected: an image processing application, a convolutional encoder/Viterbi decoder and an IIR filter.

#### 4.1.1 Image Processing

The first benchmark evaluated was an image processing application based on median filtering [1]. In this application, a median filter is applied to a $60 \times 60$ pixel image. The median filter moves line by line across a picture. The scale value of each pixel is replaced by the average of the current

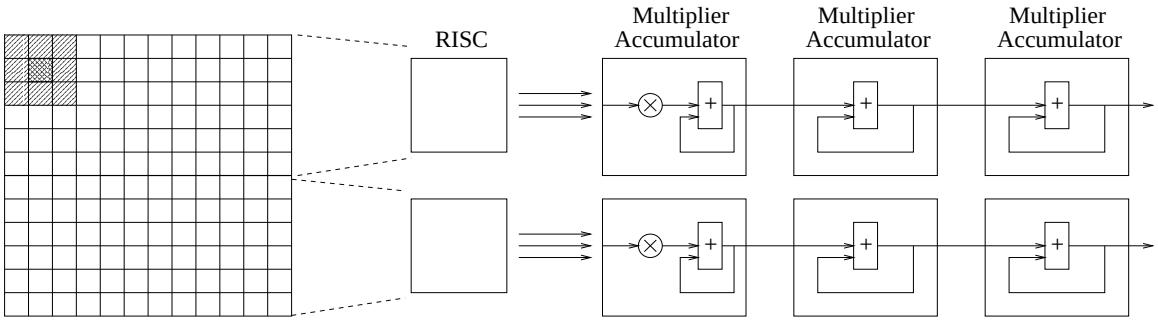



**Figure 7. Image Processing**

value and its eight neighbors. The result of this averaging is a local smoothing of color shades in an image. This technique is effective in increasing tolerance to noise.

In our implementation, two RISCs and six multiplier-accumulators are used to perform computation. As shown in Figure 7, the luminance of the dark pixel becomes the average of the shadowed area. The image is divided into two slices and the multiplier-accumulators perform the averaging computations. For the 16-core aSOC model, four RISCs and ten multiply-accumulate cores are used. Computational partitioning for this aSOC organization takes place along four slices.

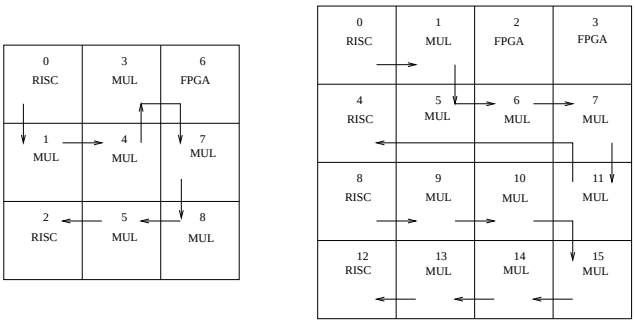
#### 4.1.2 Viterbi Decoding

The second benchmark implemented was a communication channel simulator. This design implements the operations needed to simulate a communication channel encoder and decoder. The Viterbi algorithm [20] was applied in the construction of the decoder. Four stages were needed to complete the design. First, data to be transmitted is fetched from local memory followed by input encoding using convolutional codes [20]. After the encoded signal was subjected to random white Gaussian noise. the received signals were decoded with a Viterbi decoder.

Due to their sequentiality, the data generator and encoder were assigned to the RISCs. The multiplier-accumulators were used to implement the noise generator and to add the noise to the encoded information bit sequence. Finally, the decoder was modeled on an FPGA. The decoder portion is based on single-bit operations, which is well suited to exploit the fine-grained parallelism found in FPGAs. Application partitioning and data flow is shown in the Figure 9.

#### 4.1.3 IIR Filter

Infinite impulse response (IIR) filters are widely used in signal processing. For this benchmark, a five stage IIR filter is realized. Figure 8 shows the partitioning of computations to aSOC components. After initial pre-processing by a RISC,




a) 9-core Model b) 16-core Model

**Figure 8. Partitioning of the IIR Benchmark**

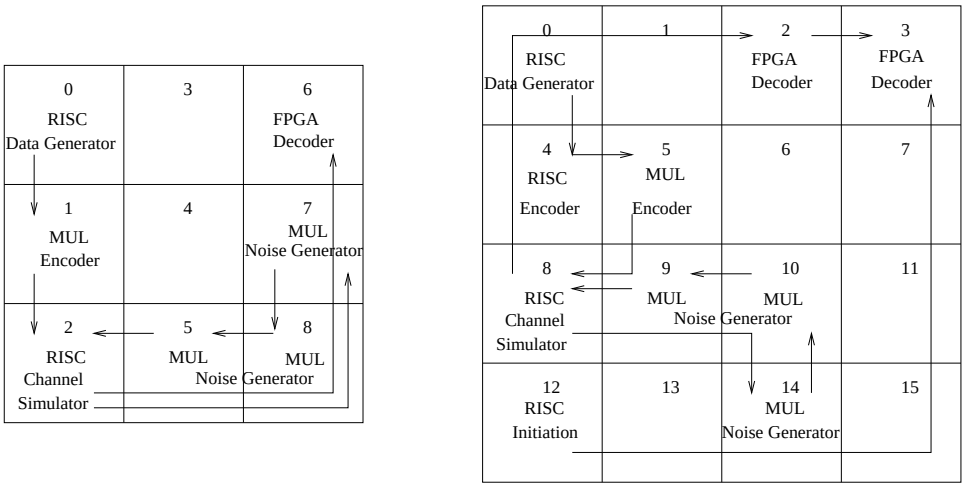
each multiplier-accumulator executes one stage of multiplication and accumulation. The results are gathered by a second RISC. In the 9-core model, one filter is implemented. In the 16-core model, since there are 4 RISCs and 10 multipliers, two IIR filters can be executed at the same time.

## 5 Results

### 5.1 Core and Interface Component Parameters

To better calibrate architectural simulations, an initial layout of both the communication interface and several cores (FPGA and multiplier-accumulator) were created. All layout was performed in 1.2 micron technology to serve as a preliminary proof-of-concept. To afford comparisons to other technologies (e.g. the R4000 core in 0.3 micron technology), some parameter scaling was needed, as discussed below. The performance of the key structure of this system, the communication interface, including interface crossbar, interconnect memory, sequencer, and FIFO port to the local core, was evaluated using SPICE.

The entire communication interface, designed in 1.2 micron technology, is about $6,000\lambda * 5,000\lambda$ in size and contains about 50,000 transistors. As shown in the Figure 10, the interface crossbar is about $4,000\lambda * 2,800\lambda$ in size and




a) 9-core Model b) 16-core Model

**Figure 9. Partitioning of the Viterbi Benchmark**

contains about 2,300 transistors. In the SPICE simulation, the critical path through the interface crossbar in 1.2 micron technology is about 8.5 ns. Based on the fixed-voltage scaling relationship for short channel devices [17], the propagation delay is determined to be $\frac{1}{s}$ when the technology scales down by a factor of $s$. Thus, the interface crossbar can be expected to run in 1-2 nS clock in 0.3 micron technology. To allow for conservative estimation, we assume that the interface crossbar operates under a 3 nS clock period at this feature size. The layout of the sequencer did not limit the clock frequency of the interface. The size of the memory in this layout was 256x32 bits.

Through simulation, the multiplier was determined to perform multiplication and accumulation with a clock cycle of 5 nS. The performance of the FPGA for each design was taken from Quartus estimates and varied from design to design. All FPGA designs required fewer than 500 four-input look-up tables. Area and performance information regarding the MIPS R4000 core was taken from [4]. This core is approximately 1.4 $mm^2$ in size in 0.18 micron technology and runs at a clock speed of 280 MHz. Note that these figures do not consider the associated R4000 cache. Overall layout results are summarized in Table 1.

| | Speed | Area ($\lambda^2$) |
|---|---|---|
| interface | 3 ns | $6000 \times 5000$ |
| MIPs R4000 (w/o cache) | 3.5ns | $4.3 \times 10^7$ [4] |
| multiplier | 4ns | $1500 \times 1000$ |
| FPGA | 20ns | $10000 \times 10000$ |
| FIFO | Async | $1500 \times 2200$ |

**Table 1. Component Parameters**

## 5.2 Simulation Results

Once the performance of the aSOC structures were determined, it was possible to compare the new static routing architecture with existing bus-based models. Since the NSIM tool used for simulation also supports adaptive dynamic routing [9], numbers using this packet-switched approach were also generated. In general, these numbers should be considered for comparative purposes only as it is unclear whether this type of communication protocol would be appropriate for an SoC environment.

The simulation results of the three benchmarks using the 9 and 16 core models are presented in Table 2. The times noted in the table indicate the simulated execution times for each benchmark considering the component speeds mentioned previously. The performance of each application executed on a single RISC core with attached memory is provided for reference. The specified link utilization is the average of the individual link utilizations across all links. Each per-link utilization is calculated by dividing the total number of communication interface cycles by the number of cycles the link is occupied by valid data. The bus has only one link, the 9-core model has 12 links, and the 16-core model has 24 links.

From the results in Table 2, it can be seen that static communication yields substantially better performance than the embedded bus model. Link utilizations for aSOC are substantially smaller than the bus since communication is distributed and pipelined throughout the system. In the image processing and IIR benchmarks, the bus is fully utilized while utilization for static communication is about 10 to 20

| | | Image Processing | | | | IIR | | | | Viterbi | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Risc | Bus | Dyn. | aSOC | Risc | Bus | Dyn. | aSOC | Risc | Bus | Dyn. | aSOC |
| 9 cores | Time (mS) | 1072 | 332 | 151 | 133 | 114 | 25.0 | 22.4 | 21.6 | 29.7 | 6.4 | 6.0 | 5.5 |
| | Link Util. | - | 100% | 11.1% | 6.3% | - | 100% | 12.2% | 5.5% | - | 40% | 2.4% | 1.5% |
| | Used Links | - | 1 | 9 | 9 | - | 1 | 7 | 8 | - | 1 | 7 | 6 |
| 16 cores | Time (mS) | 1072 | 333 | 81.1 | 71.5 | 114 | 44.6 | 11.2 | 10.9 | 29.7 | 8.2 | 6.4 | 6.2 |
| | Link Util. | - | 100% | 17.0% | 10.6% | - | 100% | 24.4% | 13.5% | - | 84% | 2.8% | 1.2% |
| | Used Links | - | 1 | 11 | 10 | - | 1 | 14 | 13 | - | 1 | 13 | 16 |

**Table 2. Performance of the Benchmarks**

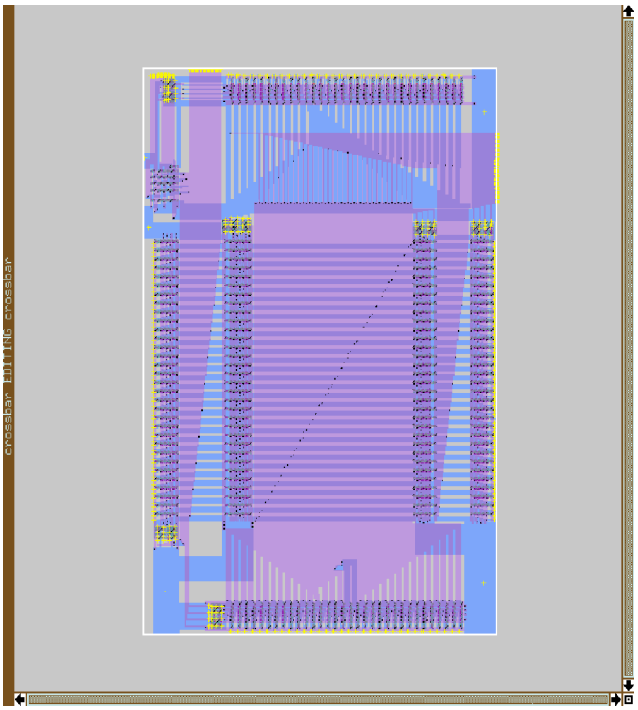


**Figure 10. Layout of the Interface Crossbar**

percent. This implies that performance will likely be improved as the number of SoC cores is increased. In effect, through static scheduling it was possible to eliminate the congestion caused by the bus and header overhead present in dynamic routing.

Observation of the results of static and dynamic communication indicates that the former achieves a run time improvement for the benchmarks versus dynamic communication. Note that these results are based on the assumption that both static and dynamic communication interfaces operate at the same speed, which generally would not be the case. Because of the need for run-time header processing and virtual circuit evaluation, the interface would likely require additional complexity, leading to slower performance.

A limitation of the static architecture is its instruction memory, which may require the interface control circuitry to be large to accommodate a spectrum of communication patterns. For the benchmarks chosen in this study, the number of required memory locations was small (generally 5 - 20) due to the regularity of the applications.

## 6 Summary and Future Work

In this paper, we have analyzed a new communication substrate for on-chip communication. The performance of this architecture has been compared directly to traditional on-chip SoC buses with dynamic routing used for comparative reference. The use of scheduled communication allows for predictable data transfer at a fast clock rate. Its distributed nature allows for scalable bandwidth across a range of heterogenous cores. It has been shown that this architecture can be developed compactly so that it can be integrated into a variety of possible cores. Additionally, this structure can accommodate cores of varying sizes, aligned in a mesh-like structure.

Several important tasks remain in the development of the SoC environment. We are currently in the process of adapting a heterogeneous compiler to our computational substrate. This compiler divides applications into parts, each of which fit into a specific core. The compiler determines data communications between the cores in a space-time fashion along the lines of [12] and has the capability to generate interconnect memory contents for each individual interface. Recent approaches to heterogeneous software partitioning and software-hardware co-design have also been proposed in [13].

A significant limitation of the current architecture is its lack of support for data routing to destinations that are determined at run-time. One approach that is under evaluation is to designate some scheduled routing cycles as dynamic. On a number of these cycles, headers may be examined by interface circuitry and destination information may be stored in a small memory. Subsequent dynamic data transfers can then use this stored header information to forward data toward the correct destination. A high-level view of this type

of approach was previously outlined in [14], but not subsequently implemented.

There is still work to be done to improve hardware implementation. In our experiments, the communication interface is assumed to run at about 333 MHz, which is about the same speed as a low-end RISC processor. It is believed that with additional interface crossbar and interface control pipelining, higher run-time speeds can be achieved, perhaps approaching 1GHz.

## 7 Acknowledgments

This project was significantly enhanced by the contributions of the following people: Wayne Burleson, Vibhor Garg, Prashant Jain, Ravinder Rachala, Ramaswamy Ramaswamy, Mandeep Singh, and Keerthy Thodima. The people listed implemented the layout for the communication interface control circuitry and the crossbar. We also thank Chris Metcalf for donating the NSIM software for our project and Altera Corporation for donating the Quartus software. Jian Liang was supported by a University of Massachusetts Healey Endowment Grant.

## References

[1] *Image Processing*. Division of Computer Research and Technology, National Institutes of Health, 1985.

[2] *The National Technology Roadmap for Semiconductors*. Semiconductor Industry Association, 1997. http://notes.sematech.org/ntrs/PubINTRS.nsf.

[3] *MIPS R4000 Microprocessor User's Manual*. MIPS Corporation, 2000.

[4] *MIPS R4000 web page*. MIPS Corporation, 2000. www.mips.com/publications.

[5] *www.altera.com*. Altera Corporation, 2000.

[6] J. Babb, M. Rinard, C. Moritz, W. Lee, M. Frank, R. Barua, and S. Amarasinghe. Parallelizing Applications to Silicon. In *Proceedings, IEEE Workshop on FPGA-based Custom Computing Machines*, Napa, Ca, Apr. 1999.

[7] S. Borkar. Supporting Systolic and Memory Communication in iWarp. In *Proceedings 17th International Symposium on Computer Architecture*, 1990.

[8] D. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. *University of Wisconsin, Madison Computer Science Department*, June 1997. Technical Report 1342.

[9] W. Dally and H. Aoki. Deadlock-free Adaptive Routing in Multicomputer Networks using Virtual Channels. *IEEE Transactions on Parallel and Distributed Systems*, 4(4), April 1993.

[10] D. Flynn. AMBA: Enabling Reusable On-Chip Design. *IEEE Micro*, pages 20–27, July 1997.

[11] International Business Machines, Inc. IBM CoreConnect Information Web Site. In *http://www.chips.ibm.com/products/powerpc/cores*, 2000.

[12] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe. Space-time scheduling of instruction-level parallelism on a raw machine. In *Proceedings: Eighth ACM Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[13] K. McKinley, S. K. Singhai, G. E. Weaver, and C. C. Weems. Compiler Architectures for Heterogeneous Processing. *Languages and Compilers for Parallel Processing, Lecture Note s in Computer Science*, (1033):434–449, Aug. 1995.

[14] C. Metcalf. Dynamic Routing with NuMesh. *MIT Lab for Computer Science, NuMesh Memo 7*, Apr. 1991.

[15] C. Metcalf. The NuMesh Simulator. *MIT Lab for Computer Science, NuMesh Memo 5*, Jan. 1992.

[16] W. Peterson. Design Philosophy of the Wishbone SoC Architecture. In *Silicore Corporation*, 1999. http://www.silicore.net/wishbone.htm.

[17] J. M. Rabaey. *Digital Integrated Circuits*. Prentice-Hall, Boston, Ma, 1996.

[18] D. Shoemaker, C. Metcalf, and S. Ward. NuMesh: An Architecture Optimized for Scheduled Communication. *Journal of Supercomputing*, 10:285–302, 1996.

[19] Sonics, Incorporated. Sonics, Inc. Corporate Web Site. In *http://www.sonicsinc.com*, 1999.

[20] A. J. Viterbi. Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm. *IEEE Transactions on Information Theory*, Apr. 1967.

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