An Architecture and Compiler for Scalable On-Chip Communication

Jian Liang, *Student Member, IEEE*, Andrew Laffely, Sriram Srinivasan, and Russell Tessier, *Member, IEEE*

***Abstract*—**
A dramatic increase in single chip capacity has led to a revolution in on-chip integration. Design reuse and ease-of-implementation have became important aspects of the design process. This paper describes a new scalable single-chip communication architecture for heterogeneous resources, adaptive System-On-a-Chip (aSOC), and supporting software for application mapping. This architecture exhibits hardware simplicity and optimized support for compile-time scheduled communication. To illustrate the benefits of the architecture, four high-bandwidth signal processing applications including an MPEG-2 video encoder and a Doppler radar processor have been mapped to a prototype aSOC device using our design mapping technology. Through experimentation it is shown that aSOC communication outperforms a hierarchical bus-based system-on-chip (SoC) approach by up to a factor of five. A VLSI implementation of the communication architecture indicates clock rates of 400 MHz in 0.18 micron technology for sustained on-chip communication. In comparison to previously-published results for an MPEG-2 decoder, our on-chip interconnect shows a run-time improvement of over a factor of four.

I. Introduction

Recent advances in VLSI transistor capacity have led to dramatic increases in the amount of computation that can be performed on a single chip. Current industry estimates [1] indicate mass production of silicon devices containing over one billion transistors by 2012. This proliferation of resources enables the integration of complex system-on-a-chip (SoC) designs containing a wide range of intellectual property cores. To provide high performance, SoC integrators must consider the design of individual intellectual property (IP) cores, their on-chip interconnection, and application mapping approaches. In this paper, we address the latter two design issues through the introduction of a new on-chip communications architecture and supporting application mapping software. Our communications architecture is scalable to tens of cores and can be customized on a per-application basis.

Recent studies [1] have indicated that on-chip communication has become the limiting factor in SoC performance. As die sizes increase, the performance effect of lengthy, cross-chip communication becomes prohibitive. Future SoCs will require a communication substrate that can support a variety of diverse IP cores. Many contemporary bus-based architectures are limited in terms of physical scope by the need for dynamic arbitration of communication resources. Significant amounts of arbitration across even a small number of components can quickly form a performance bottleneck, especially for data-intensive, stream-based computation. This issue is made more complex by the need to compile high-level representations of applications to SoC environments. The heterogeneous nature of cores in terms of clock speed, resources, and processing capability makes cost modeling difficult. Additionally, communication modeling for interconnection with long wires and variable arbitration protocols limits performance predictability required by computation scheduling.

Our platform for on-chip interconnect, **a**daptive **S**ystem-**On**-a-**C**hip (aSOC), is a modular communications architecture. As shown in Fig. 1, an aSOC device contains a two-dimensional mesh of computational *tiles*. Each tile consists of a core and an associated *communication interface*. The interface design can be customized based on core datawidths and operating frequencies to allow for efficient use of resources. Communication between nodes takes place via pipelined, point-to-point connections. By limiting inter-core communication to short wires with predictable performance, high-speed communication can be achieved. A novel aspect of the architecture is its support for both compile-time scheduled and run-time dynamic transfer of data. While scheduled data transfer has been directly optimized, a software-based mechanism for run-time dynamic routing has also been included.

To support the aSOC architecture, an application mapping tool, AppMapper, has been developed to translate high-level language application representations to aSOC devices. Mapping steps, including code optimization, code partitioning, communication scheduling, and core-dependent compilation are part of the AppMapper flow. Although each step has been fully automated, design interfaces for manual intervention are provided to improve mapping efficiency. Mapping algorithms have been developed for both partitioning and scheduling based on heuristic techniques. A system-level simulator allows for performance evaluation prior to design implementation.

Key components of the aSOC architecture, including the communication interface architecture, have been simulated and implemented in 0.18 micron technology. Experimentation shows a communication network speed of 400 MHz with an overhead added to on-chip IP cores that is similar to on-chip bus overhead. The AppMapper tool has been fully implemented and has been applied to four signal processing applications including MPEG-2 encoding. These applications have been mapped to aSOC devices containing up to 45 cores via the AppMapper tool. Performance comparisons between aSOC implementations and other more traditional on-chip communi-

This work was supported in by part by the National Science Foundation under grants CCR-0081405 and CCR-9988238. J. Liang and R. Tessier are with the Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA, 01003 USA. E-mail: tessier@ecs.umass.edu . A. Laffely is with the US Air Force, Hanscom AFB, MA, 01731 USA. S. Srinivasan is with Advanced Micro Devices, Austin, TX, 78741 USA.

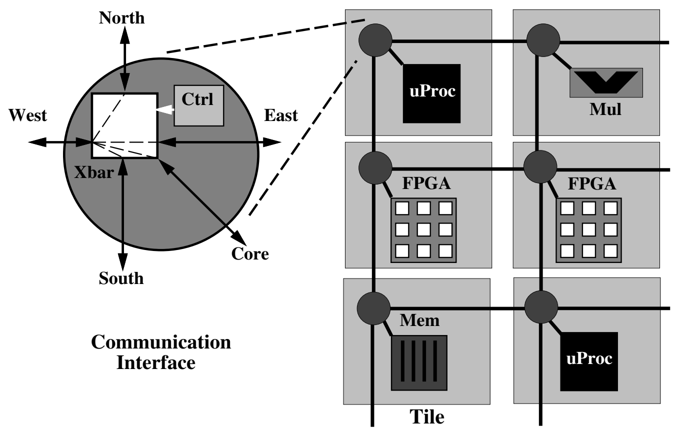


Fig. 1. Adaptive System-on-a-Chip (aSOC)

cation substrates, such as an on-chip bus, shows an aSOC performance improvement of up to a factor of five.

This paper presents a review of SoC-related communication architectures in Section II. Section III reveals the design philosophy of our communication approach and describes the technique at a high level. The details of our communication architecture are explained in Section IV. Section V demonstrates the application mapping methodology and describes component algorithms. Section VI describes the benchmarks used to evaluate our approach and our validation methodology. In Section VII, experimental results for aSOC devices with up to 49 cores are presented. These results are compared against the performance of alternative interconnect approaches and previously-published results. Section VIII summarizes our work and offers suggestions for future work.

II. Related Work

Numerous on-chip interconnect approaches have been proposed commercially as a means to connect intellectual property cores. These approaches include arbitrated buses [2], [3], [4] and hierarchical buses connected via bridges [5], [6], [7]. In general, all of these architectures have similar arbitration characteristics to master/slave off-chip buses with several new features including data pipelining [2], replacement of tri-state drivers with multiplexers [3], and separated address/data buses due to the elimination of off-chip pin constraints. These approaches, while flexible, have limited scalability due to the arbitrated and capacitive nature of their interconnection. Other notable, common threads through on-chip interconnect architectures include the simplicity of the logic needed on a per-node basis to support communication, their diverse support for numerous master/slave interconnection topologies [8], and their integrated support for on-chip testing. Several current on-chip interconnects [3], [7] support the connection of multiple buses in variable topologies (e.g. partial crossbar, tree). This support provides users flexibility in coordinating on-chip data paths amongst heterogeneous components.

Recently, several network-on-chip communication architectures have been suggested. Researchers at Stanford University propose an SoC interconnect using packet-switching [9]. The idea of performing on-chip dynamic routing is described although not yet implemented. MicroNetwork [10] provides on-chip communication via a pipelined interconnect. A rotating resource arbitration scheme is used to coordinate inter-node transfer for dynamic requests. This mechanism is limited by the need for extensive user interaction in design mapping.

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. For iWarp [11], inter-processor communication patterns were statically determined during program compilation and implemented with the aid of programmable, inter-processor buffers. This concept has been extended by the NuMesh project [12] to include collections of heterogeneous processing elements interconnected in a mesh topology. Although pre-scheduled routing is appropriate for static data flows with predictable communication paths, most applications rely on at least minimal run-time support for data-dependent data transfer. Often, this support takes the form of complicated per-node dynamic routing hardware embedded within a communication fabric. A recent example of this approach can be found in the Reconfigurable Architecture Workstation (RAW) project [13]. In our system, we minimize hardware support for run-time dynamic routing through the use of software.

III. aSOC Design Philosophy

Successful deployment of aSOC requires the architectural development of an inter-node communication interface, the creation of supporting design mapping software, and the successful translation of target applications. Before discussing these issues, the basic operating model of aSOC interconnect is presented.

A. Design Overview

As shown in Fig. 1, a standardized communication structure provides a convenient framework for the use of intellectual property cores. A simple core interface protocol, joining the core to the communication network, creates architectural modularity. By limiting inter-core communication to short wires exhibiting predictable performance, high-speed point-to-point transfer is achieved. Since heterogeneous cores can operate at a variety of clock frequencies, the communication interface provides both data transport and synchronization between processing and communication clock domains.

Inter-core communication using aSOC takes place in the form of data *streams* [12] which connect data sources to destinations. To achieve the highest possible bandwidth, our architecture is targeted towards applications, such as video, communications, and signal processing, that allow most inter-core communication patterns to be extracted at compile time. By using the mapping tools described in Section V, it is possible to determine how much bandwidth each inter-core data stream requires relative to available communication channel bandwidth. Since stream communication can generally be determined at compile time [12], our system can take advantage of minimized network congestion by scheduling data transfer in available data channels.

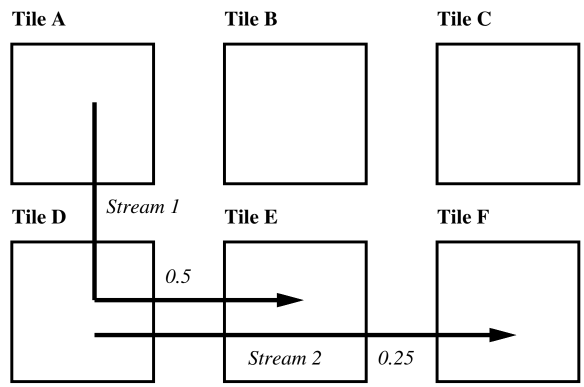


Fig. 2. Multi-core data streams 1 and 2. This example shows data streams from Tile A to Tile E and from Tile D to Tile F. Fractional bandwidth usage is indicated in italics.

Cycle	Tile A	Tile D	Tile E	Tile F
0	core to South	core to East		
1	core to South	North to East	West to East	
2		North to East	West to core	West to core
3			West to core	

TABLE I
COMMUNICATION SCHEDULES FOR TILES IN FIG. 3

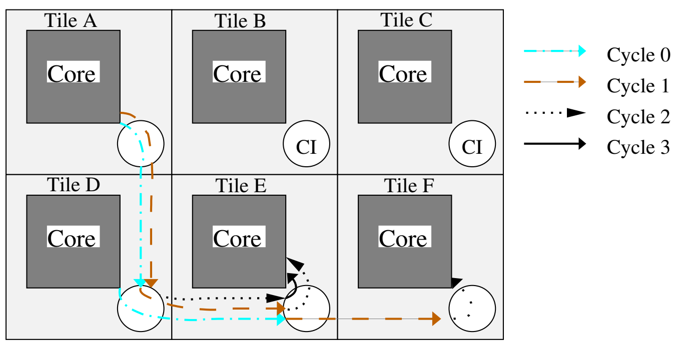


Fig. 3. Pipelined stream communication across multiple communication interfaces

As seen in Fig. 2, each stream requires a specific fraction of overall communication link bandwidth. For this example, *Stream 1* consumes $\frac{0.5}{1}$ of available bandwidth along links it uses and *Stream 2* requires $\frac{0.25}{1}$. This bandwidth is reserved for a stream even if it is not used at all times to transfer valid data. At specific times during the computation, data can be injected into the network at a lower rate than the reserved bandwidth, leaving some bandwidth unused. In general, the path taken by a stream may require data transfer on multiple consecutive clock cycles. On each clock cycle, a different stream can use the same communication resource. The assignment of streams to clock cycles is performed by a communication scheduler based on required stream bandwidth. Global communication is broken into a series of step-by-step hops that is coordinated by a distributed set of individual tile communication schedules. During communication scheduling, near-neighbor communication is coordinated between neighboring tiles. As a result of this bandwidth allocation, the dynamic timing of the core computation is decoupled from the scheduled timing of communications.

The cycle-by-cycle behavior of the two example data streams in Fig. 2 is shown in Fig. 3. For *Stream 2*, data from the core of *Tile D* is sent to the left (West) edge of *Tile E* during communication clock cycle 0 of a four-cycle schedule. During cycle 1, connectivity is enabled to transfer data from *Tile E* to the West edge of *Tile F*. Finally, in cycle 2 the data is moved to its destination, the core of *Tile F*. During four consecutive clock cycles, **two** data values are transmitted from *Tile A* to *Tile E* in a pipelined fashion forming *Stream 1*. Note that the data stream is pipelined and the physical link between *Tile D* and *Tile E* is shared between the two streams at different points in time. Stream transfer schedules are iterative. At the conclusion of the fourth cycle, the four-cycle sequence re-starts at cycle 0 for new pieces of data. The communication interface serves as a cycle-by-cycle switch for stream data. Switch settings for the four-cycle transfer in Fig. 3 are shown in Table I.

Stream-based routing differs from previous static routing networks [11]. Static networks demand that all communication patterns be known at compile time along with the *exact* time of all data transfers between cores and the communication network. Unlike static routing, stream-based routing requires that bandwidth be allocated but not necessarily used during a specific invocation of the transfer schedule. Communication is set up as a pipeline from source to destination cores. This approach does not require the *exact* timing of all transfers, but rather, data only needs to be inserted into the correct stream by the core interface at a communication cycle allocated to the stream. Computation can be overlapped with communication in this approach since the injection of stream data into the network is decoupled from the arrival of stream data.

B. Flow Control

Since cores may operate asynchronously to each other, individual stream data values must be tagged to indicate validity. When a valid stream data value is inserted into the network by a source core at the time slot allocated for the stream, it is tagged with a single *valid* bit. As a result of communication scheduling, the allocated communication cycle for stream data arrival at a destination is predefined. The data valid bit can be examined during the scheduled cycle to determine if valid data has been received by the destination. If data production for a stream source temporarily runs ahead of data consumption at a destination, data for a specific stream can temporarily back up in the communication network. To avoid deadlock, data buffer storage is required in each intermediate communication interface for each stream passing through the interface. With buffering, if a single stream is temporarily blocked, other streams which use the affected communication interfaces can continue to operate unimpeded. A data buffer location for each stream is also used at each core-communication interface boundary for intermediate storage and clock synchronization.

The use of flow control bits and local communication interface buffering ensures data transfer with the following characteristics:

- All data in a stream follows the same source-destination path.
- All stream data is guaranteed to be transfered in order.
- In the absence of congestion, all stream data requires the same amount of time to be transfered from source to destination
- Computation is overlapped with communication.

C. Run-time Stream Management

For a number of real-time applications, inter-core communication patterns may vary over time. This requirement necessitates the capability to invoke and terminate streams at various points during application execution and, in some cases, to dynamically vary stream source and destination cores at run-time. In developing our architecture, we consider support for the following two situations: (1) all necessary streams required for execution are known at compile-time but are not all active simultaneously at run-time and, (2) some stream source-destination pairs can only be determined at run-time.

1) Asynchronous Global Branching: For some applications, it is desirable to execute a specific schedule of stream communication for a period of time, and then in response to the arrival of a data value at the communication interface, switch to a communication schedule for a different set of streams. This type of communication behavior has the following characteristics:

- All stream schedules are known at compile time.
- The order of stream invocation and termination is known, but the time at which switches are made is determined in a data-dependent fashion.
- A data value traverses all affected communication interfaces (tiles) over a series of communication cycles to allow for a *global* change in communication patterns.

This *asynchronous global branching* technique [14] across predetermined stream schedules has been shown [15] to support many stream-based applications that exhibit time-varying communication patterns. The aSOC communication interface architecture supports these requirements by allowing local stream schedule changes based on the arrival of a specific data value at the communication interface. Depending on the value of the data, which is examined on a specific communication cycle, the previous schedule can be repeated or a new schedule, already stored in the communication interface, can be used. This technique does not require the loading of new schedules into the communication interface at run-time. Although our architecture supports run-time update of the schedule memory, our software does not currently exploit this capability. As a result, all required stream schedules must be loaded into the interface prior to run-time via an external interface and a shift chain.

The use of these branching mechanisms can be illustrated through the use of a data transfer example. Consider a transfer pattern in which *Tile D* in Fig. 4 is required to first send a fixed set of data to *Tile A* and then send a different fixed set of data to *Tile E*. To indicate the need for a change in data destination, the *Tile D* core iteratively sends a value to its communication interface. When this value is decremented by the core to a value of 0, control for the communication schedule is changed to reflect a change in data destination. The two communication interface schedules for *Tile D* which supports this behavior are shown in Table II. Each communication cycle is represented in the interface with a specific communication *instruction*. For cycles when data dependent schedule branching can take place, the target instruction for a taken branch is listed second under the *next instr.* heading. In these cycles, data is examined by the interface control to determine if branching should occur. The *Tile D - Tile A* stream schedule uses instructions 0 and 1. The *Tile D - Tile E* stream schedule uses instructions 2 and 3.

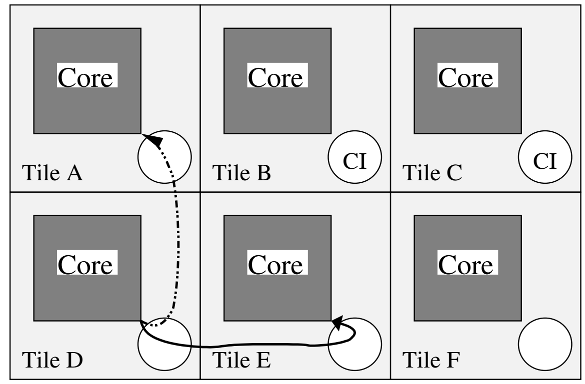


Fig. 4. Example of distinct stream paths for two communication schedules which send data from a source to different destinations.

Instr.	interface connection	next instr.	possible branch?	comment
0x0	core to North	0x1/-	N	*data to North*
0x1	core to interface	0x0/0x2	Y	*test count*
0x2	core to East	0x3/-	N	*data to East*
0x3	core to interface	0x2/0x0	Y	*test count*

TABLE II

DATA-DEPENDENT COMMUNICATION CONTROL BRANCHING FOR TILE D IN FIG. 4

2) Run-time Stream Creation: Given the simplicity of routing nodes and our goal to primarily support stream-based routing, communication hardware resources are not provided to route data from stream sources to destinations that have not been explicitly extracted at compile time (dynamic data). However, as we will show in Section VII, often streams extracted from the user program require only a fraction of the overall available stream bandwidth. As a result, a series of low-bandwidth streams between all nodes can be allocated at compile time via scheduling in a round robin fashion in otherwise unused bandwidth. Cores can take advantage of these out-of-band streams at run time by inserting dynamic data into a stream at the appropriate time so that data is transmitted to the desired destination core.

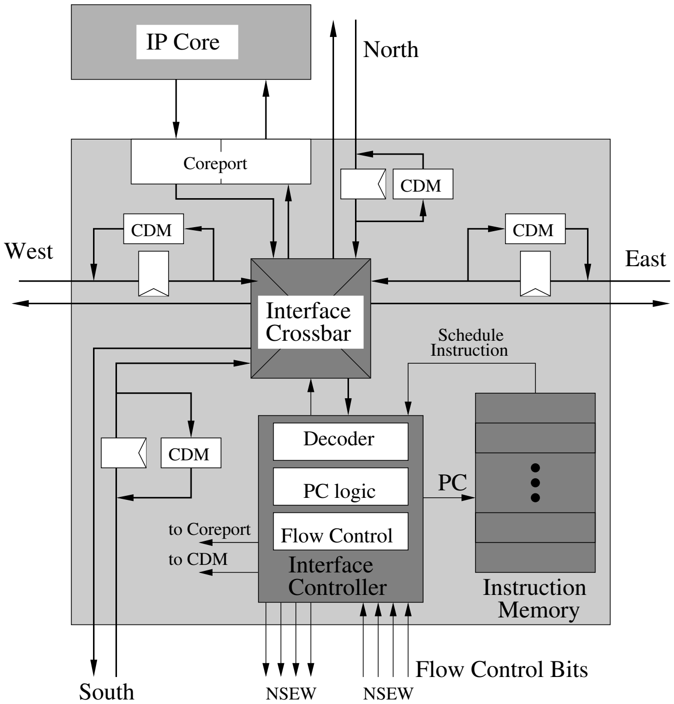


Fig. 5. Core and communication interface

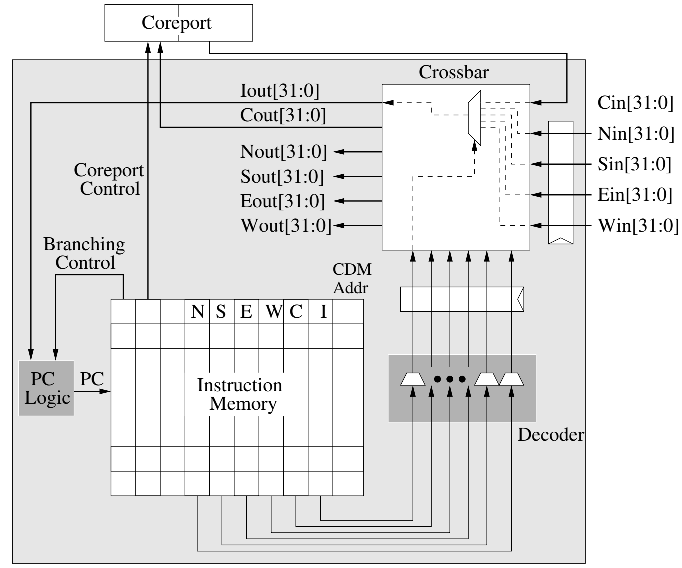


Fig. 6. Detailed communication interface

IV. aSOC Architecture

The aSOC architecture augments each IP core with communication hardware to form a computational tile. As seen in Fig. 5, tile resources are partitioned into an IP core and a communication interface (CI) to coordinate communication with neighboring tiles. The high level view of the communication interface reveals the five components responsible for aSOC communications functionality:

- **Interface Crossbar** - allows for inter-tile and tile-core transfer.
- **Instruction Memory** - contains schedule instructions to configure the interface crossbar on a cycle-by-cycle basis.
- **Interface Controller** - control circuitry to select an instruction from the instruction memory.
- **Coreport** - data interface and storage for transfers to/from the tile IP core.
- **Communication Data Memory (CDM)** - buffer storage for inter-tile data transfer.

The interface crossbar allows for data transfer from any input port ($North$, $South$, $East$, $West$, and $Coreport$) to any output port (five input directions and the port into the controller). The crossbar is configured to change connectivity every clock cycle under the control of the interface controller. The controller contains a program counter and operates as a microsequencer. If, due to flow control signals, it is not possible to transfer a data word on a specific clock cycle, data is stored in a communication data memory (CDM). For local transfers between the local IP core and its communication interface, the coreport provides data storage and clock synchronization.

A. Communication Interface

A detailed view of the communication interface appears in Fig. 6. The programmable component of the interface is a 32-word SRAM-based instruction memory that dynamically configures the connectivity of the local interface crossbar on a cycle-by-cycle basis based on a pre-compiled schedule. This programmable memory holds binary code that is created by application mapping tools. Instruction memory bits are used to select the *source* port for each of the six interface destination ports (N_{out}, S_{out}, E_{out}, W_{out}, C_{out} for the core, I_{out} for the interface control). *CDM Addr* indicates the buffer location in the communication data memory, which is used to store intermediate routed values as described in Section IV-C. A program counter PC is used to control the instruction sequence. Branch control signals from the instruction memory determine when data dependent schedule branching should occur. This control can include a comparison of the I_{out} crossbar output to a fixed value of 0 to initiate branching.

B. Coreports: Connecting Cores to the Network

The aSOC coreport architecture is designed to permit interfacing to a broad range of cores with a minimum amount of additional hardware, much like a bus interface. Both core-to-interface and interface-to-core transfer are performed using asynchronous handshaking to provide support for differing computation and communication clock rates. Both input and output coreports for a core contain dual-port memories (one input port, one output port). Each memory contains an addressable storage location for each individual stream, allowing multiple streams to be targeted to each core for both input and output.

The portion of the coreport closest to the core has been designed to be simple and flexible, like a traditional bus interface. This interface can easily be adapted to interact with a variety of IP cores. Since coreport reads and writes occur independently, the network can operate at a rate that is different than that of individual cores. Specific core interfacing depends on the core. For example, as described in Section VI-B, a microprocessor can be interfaced to the coreport via a microprocessor bus. For simpler cores, a state machine can control coreport/core interaction.

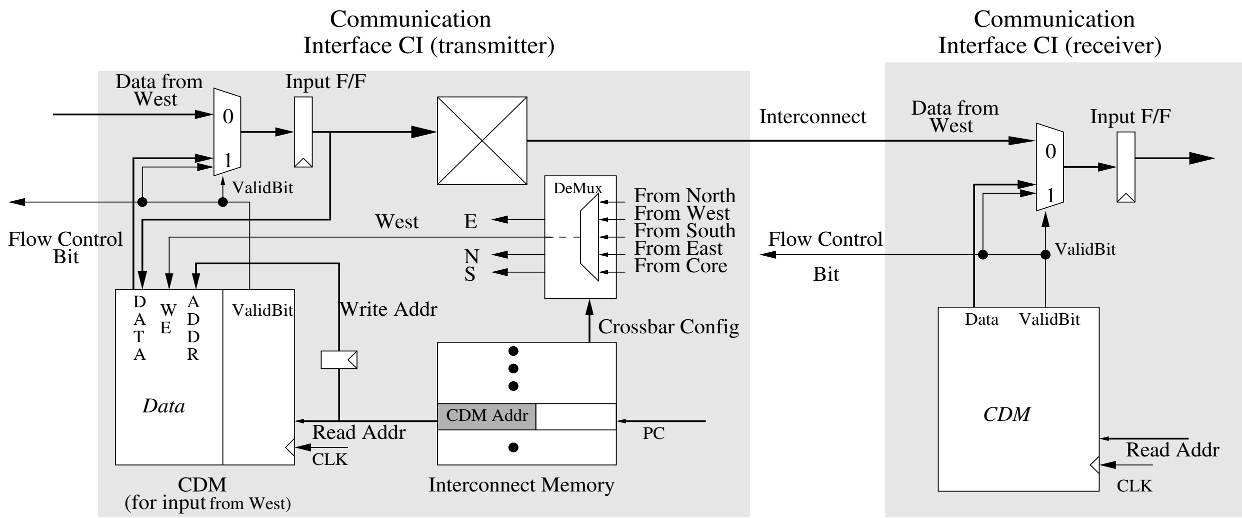


Fig. 7. Flow control between neighboring tiles

C. Communication Data Memory

As described in Section III-B, due to network congestion or uneven source and destination core data rates, it may be necessary to buffer data at intermediate communication interfaces. As shown in Fig. 7, the communication data memory (CDM) provides one storage location for each stream that passes through a port of the communication interface. To facilitate interface layout, the memory is physically distributed across the N, S, E, W ports. On a given communication clock cycle, if a data value cannot be transfered successfully, it is stored in the CDM. The flow control bits that are transfered with the data can be used to indicate valid data storage.

In aSOC devices, near-neighbor flow control and buffering is used. Fig. 7 indicates the location of the communication data memory in relation to inter-tile data paths. On a given communication clock cycle, the stream address for each port indicates the stream that is to be transfered in the next cycle. Concurrently, the crossbar is configured by instruction memory signals ($N..C$) to transfer the value stored in the crossbar register. This value is transfered to the receiver at the same time the receiver valid bit is sent to the transmitter. This bit indicates if the CDM at the receiver already has a buffered value for the transmitted stream. If a previous value is present at the receiver, the transmitted value is stored in the transmitter CDM using the write signals shown entering the CDM on the left in Fig. 7. A multiplexer at the input to the crossbar register determines if the transmitted or previously-stored value is loaded into the crossbar register on subsequent transfer cycles.

V. aSOC Application Mapping Tools

The aSOC application mapping environment, *AppMapper*, builds upon existing compiler infrastructure and takes advantage of user interaction and communication estimation during the compilation process. *AppMapper* tools and methodology follow the flow shown in Fig. 8. Individual steps include:

- **Preprocessing/conversion to intermediate format** - Following parsing, high-level C constructs are translated to a unified abstract syntax tree format (AST). After property annotation, AST representations are converted to the graph-based Stanford University Intermediate Format (SUIF) [16] that represents functions at both high and low levels.
- **Basic block partitioning and assignment** - An annealing-based partitioner operates on basic blocks based on core computation and communication cost models. The partitioner isolates intermediate-form structures to locate inter-core communication. The result of this phase is a refined task graph where the nodes are clustered branches of the syntax tree assigned to available aSOC cores and the inter-node arcs represent communication. The number and the type of nodes in this task graph match the number and type of cores found in the device. Following partitioning and assignment to core types, core tasks are allocated to individual cores located in the aSOC substrate so that computation load is balanced.
- **Inter-core synchronization** - Once computation is assigned to core resources, communication points are determined. The blocking points allow for synchronization of stream-based communication and predictable bandwidth.
- **Communication scheduling** - Inter-core communication streams are determined through a heuristic scheduler. This list-scheduling approach minimizes the overall critical path while avoiding communication congestion. Individual *instruction memory* binaries are generated following communication scheduling.
- **Core compilation** - Core compilation and communication scheduling are analyzed in tandem through the use of feedback. Core functionality is determined by native core compilation technology (e.g. field-programmable gate array (FPGA) synthesis, reduced instruction set computer (RISC) compiler). Communication calls between cores are provided through fine-grained send/receive operations.
- **Code generation** - As a final step, binary code for each core and communication interface is created.

These steps are presented in greater detail in subsequent subsections.

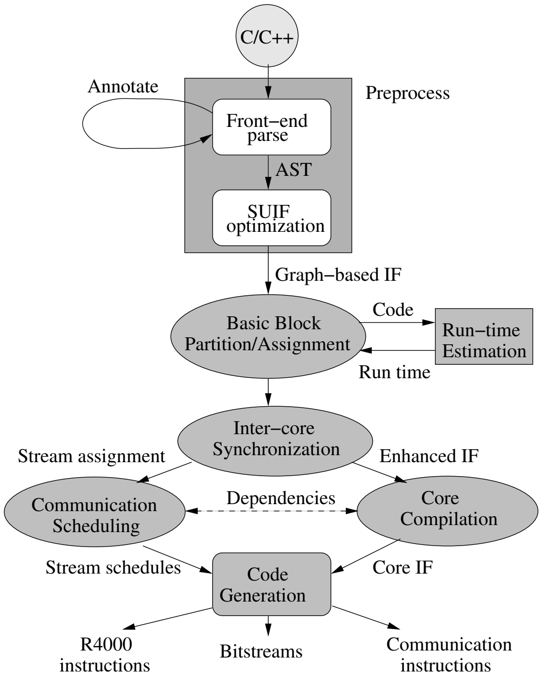


Fig. 8. aSOC application mapping flow

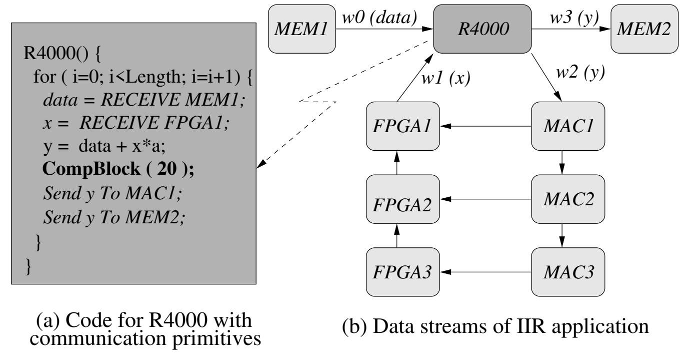


Fig. 9. Inter-core synchronization

A. SUIF preprocessing

The AppMapper front-end is built upon the SUIF compiler infrastructure [16]. SUIF provides a kernel of optimizations and intermediate representations for high-level C code structures. High-level representations are first translated into a language-independent abstract syntax tree format. This approach allows for object-oriented representation for loops, conditionals, and array accesses. Prior to partitioning, AppMapper takes advantage of several scalar SUIF optimization passes including constant propagation, forward propagation, constant folding, and scalar privatization [16]. The interprocedural representation supported in SUIF facilitates subsequent AppMapper partitioning, placement, and scheduling passes. SUIF supports interprocedural analysis rather than using procedural inlining. This representation allows for rapid evaluation of partitioning and communication cost and the invocation of dead-code elimination. Data references are tracked across procedures.

B. Basic Block Partitioning and Assignment

Following conversion to intermediate form, high-level code is presented as a series of basic blocks. These blocks represent sequential code, loop-level parallelism, and subroutine functions. Based on calling patterns, dataflow dependency between blocks is determined through both forward and reverse tracing of inter-block paths [17]. As a result of this dependence analysis, coarse-gained blocks can be scheduled to promote parallel computation. As shown in Fig. 9(b) for an infinite impulse response (IIR) filter, subfunction dependency forms a flowgraph of computation that can be scheduled. The most difficult part of determining this dependency is estimating the computation time of basic blocks across a range of cores to determine the core best suited for evaluation. The overall run time attributed to each basic block is determined by parameters of the computation. These include:

- β **- run time** - execution time of a single invocation of a basic block on a specific core. Value β is based on the number of clock cycles, the speed of the core clock, and the amount of available parallelism.
- λ **- invocation frequency** - the number of times each basic block is invoked.

The parameters lead to an overall core run time of $\beta \times \lambda$ for each function. Core run-time estimates, β, are determined through instruction counts or through simulation, prior to compilation using techniques described in Section VI-B. Clock rates, which vary from core to core, are taken into account during this determination. A goal of design mapping is to maximize the throughput of stream computation while minimizing c_{total}, the inter-core transport time for basic block data. For a specific core, this value measures the shortest distance to another core of a different type.

Assignment of basic blocks to specific cores requires a cost model which takes both computation and communication into account. For AppMapper, this cost is represented as:

$$cost = x \times T_{compute} + y \times \frac{1}{T_{overlap}} + z \times c_{total} \quad (1)$$

where $T_{compute}$ indicates combined computation time of all streams, $T_{overlap}$ indicates computational parallelism, c_{total} indicates combined stream communication time and x, y, and z are scaling constants. Minimization of this cost function forms the basis for basic block assignment. The value $T_{compute}$ is determined from β parameters for each core. Prior to basic block assignment, small code blocks are clustered using Equation 1 in an effort to minimize inter-core transfer. To support placement, a set of N bins are created, one per target core. During the clustering phase, communication time is estimated by the distance of the shortest path between the two types of target cores. At the end of clustering, a collection of N block-based clusters remains. Dataflow dependency is tracked through the creation of basic block data predecessor and successor lists.

For core assignment, clustered blocks are assigned to unoccupied cores so that the cost expressed in Equation 1 is minimized. *AppMapper* provides a file-based interface for users to manually assign basic blocks to specific cores, if desired. Following greedy basic block assignment to cores, a swapping step is used to exchange tasks across different types of cores subject to the cost function in Equation 1. This step attempts to minimize system cost and critical path length by load balancing parallel computation across cores. Load balancing is supported by the second term in Equation 1. Basic block assignment is complicated by the presence of multiple cores with the same functionality in an aSOC device. Following basic block assignment to a specific type of core, it is necessary to match the block to a specific core at a fixed location. Given the small number of each type of core available (typically less than 5), a full enumeration of all core assignments is possible. For later generation devices it may be possible to integrate this search with the basic block to core assignment phase.

C. Inter-core Synchronization

Synchronization between cores is required to ensure that data is consumed based on computational dependencies. Once basic blocks have been assigned to specific cores in the aSOC device, communication primitives are inserted into the intermediate form to indicate when communication should occur. These communications are blocking based on the transfer rate of the communication network. As shown in Fig. 9(a), the data transfer call to multiply-accumulate unit *MAC1* follows an assignment to y. As a result, *R4000* processing can be overlapped with *MAC1* processing. Each inter-core communication represents a data stream, as indicated by w-labeled arcs in Fig. 9(b).

D. Communication Scheduling

Following basic block assignment, the number of streams and their associated sources and destinations are known. Given a set of streams, communication scheduling assigns streams to communication links based on a fixed schedule. Inter-tile communication is broken into a series of time steps, each of which is represented by a specific instruction in a communication interface instruction memory. Schedule cycle assignment is made so that the schedule length does not exceed the instruction storage capacity of each communication interface instruction memory (32 instructions). Each unidirectional inter-tile channel can transmit one data value on each communication clock cycle. Only one stream can use a channel during a specific clock cycle. In general, the length of a schedule must be at least as long as the longest stream Manhattan path. During schedule execution, multiple source-destination data transfers may take place per stream. For example, two stream transfers take place per schedule in the example shown in Fig. 3. To allow for flow control, all transfers for a stream must follow the same source-destination path.

Our communication scheduling algorithm forms multi-tile connections for all source-destination pairs in *space* through the creation of multi-tile routing paths. Sequencing in *time* is made by the assignment of data transfer to specific communication clock cycles. This space-time scheduling problem has been analyzed previously [14] in terms of static, but not stream-based scheduling. For our scheduler, the schedule length L is set to the longest Manhattan source-destination path in terms of tiles. Streams are ordered by required stream bandwidth per tile. The following set of operations are performed to create a source-destination path for each stream prior to scheduling transfers along the paths:

- The shortest source-destination path for each stream is determined via maze routing using a per-tile cost function of $g_i = g_{i-1} + c_i$. In this equation, c_i is the cost of using a tile communication channel, g_{i-1} is the cost of the route from the path source to tile i, and g_i is the total cost of the path including tile i. The c_i cost value represents a combination of the amount of channel bandwidth required for the path in relation to the bandwidth available and the distance from the channel to the stream destination.
- For multi-fanout streams, a Steiner tree approximation is used to complete routing. After an initial destination is reached, maze routes to additional destinations are started from previously-determined connections.

Following the assignment of streams to specific paths, the assignment of stream data transfers to specific communication clock cycles is performed. Each transfer must be scheduled separately within the communication schedule. Scheduling is performed via the following algorithm:

1) Set the length of the schedule to the length of the longest Manhattan path distance, L. Specific schedule time slots range from 0 to $L - 1$.
2) Order streams by required channel bandwidth.
3) For each stream:
    a) Set start time slot s to 0.
    b) For each transfer:
        i) Determine if inter-tile channels along source-destination path are available during n consecutive communication clock cycles, where n is the stream path length.
        ii) If bandwidth available, schedule transfer communication, increment start time s, and go to step 3.b to schedule next transfer.
        iii) Else increment start time s and go to step 3.b.i.

If any stream cannot fit into the length of the stream schedule L, the schedule length is incremented by one and the scheduling process is restarted.

In Section III-C.1, a technique is described which allows run-time switching between multiple communication schedules. To support multiple schedules, the communication scheduling algorithm must be invoked multiple times, once per schedule, and the length of the *combined* schedules must fit within the communication interface instruction memory.

E. Core Compilation and Code Generation

Following assignment of basic blocks to cores and scheduling, basic block intermediate form code is converted to representations that can be compiled by tools for each core. Back-end formats include assembly-level code (R4000 processor) and Verilog (FPGA, multiplier). These tools also provide an interface to the simulation environment described in Section VI-B. The back-end step in AppMapper involves the generation of instructions for the R4000 and bitstreams for the FPGA. Each communication interface is configured through the generation of communication instructions.

F. Comparison to Previous Mapping Tools

To date, few integrated compilation environments have been created for heterogeneous systems-on-a-chip. The MESCAL system [18] provides a high-level programming interface for embedded SOCs. Though flexible, this system is based on a communication protocol stack which may not be appropriate for data stream-based communication. Several projects [19] [20] have adapted embedded system compilers to SOC environments. These compilers target bus-based interconnect rather than a point-to-point network. Cost-based tradeoffs between on-chip hardware, software, and communication were evaluated by Wan et al. [8]. In Dick and Jha [21], on-chip task partitioning was followed by a hill-climbing based task placement stage.

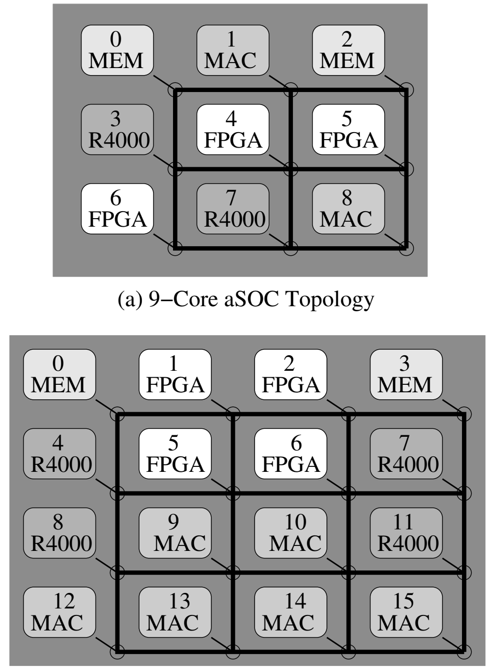
Our mapping system and these previous efforts have similarities to software systems which map applications to a small number of processors and custom devices (hardware/software co-design [22]), and parallel compilers which target a uniform collection of interconnected processors. Most codesign efforts [22] involve the migration of operational or basic block tasks from a single processor to custom hardware. The two primary operations performed in hardware/software codesign are the partitioning of operations and tasks between hardware and software and the scheduling of operations and communications [22]. The small number of devices involved (usually one or two processors and a small number of custom devices) allows for precise calculation of communication and timing requirements, facilitating partitioning and scheduling.

Our partitioning approach, which is based on task profiling and simulated annealing, extends earlier task-based codesign partitioning approaches [23], [24] to larger numbers of tasks and accurately models target processors and custom chips. Although all of these efforts require modeling of execution time, our approach addresses a larger number of target models and requires tradeoffs between numerous hardware/software partitions. This requires high-level modeling of both performance and partition size for a variety of different cores. Our approach to partition assignment of basic block tasks is slightly more complicated than typical codesign assignment. In general, the bus structure employed by codesign systems [23] limits the need for cost-based assignments. In contrast, our swap-based assignment approach for heterogeneous targets is simpler than the annealing based technique used to assign basic blocks to a large homogeneous array of processors [13]. Since blocks are assigned to specific target cores during partitioning, the assignment search is significantly more constrained and can be simplified.

Our stream-based scheduling differs from previous codesign processor/custom hardware communication scheduling [22]. These scheduling techniques attempt to identify exact communication latency between processors and custom devices to ensure worst-case performance across a variety of bus transfer modes (e.g. burst/non-burst) [19]. Often instruction scheduling is overlapped with communication scheduling to validate timing assumptions [22]. In contrast, our communication scheduling approach ensures stream throughput over a period of computation.

VI. Experimental Methodology

To validate the aSOC approach, target applications have been mapped to implemented aSOC devices and architectural models containing up to 49 cores. Parameters associated with the models are justified via a prototype aSOC device layout, described in Section VII. Examples of 9 and 16 core models are shown in Fig. 10. The models consist of R4000 microprocessors [25], FPGA blocks, 32Kx8 SRAM blocks (MEM), and multiply-accumulate (MAC) cores. The same core configurations were used for all benchmarks. The FPGA core contains 121 logic clusters, each of which consists of four 4-input lookup tables (LUTs) and flip flops [26]. The core population of all aSOC configurations are shown in Table III.



(a) 9−Core aSOC Topology

(b) 16−Core aSOC Topology

Fig. 10. aSOC topologies: 9 and 16 cores

Array Configuration	R4000	FPGA	Mem	MAC
3 × 3	2	3	2	2
4 × 4	4	4	2	6
5 × 5	9	2	4	10
6 × 6	13	2	6	15
7 × 7	18	3	8	20

TABLE III
aSOC device configurations

A. Target aSOC Applications

Four applications from communications, multimedia, and image processing domains have been mapped to the aSOC device models using the *AppMapper* flow described in Section V. Mapped applications include MPEG-2 encoding [27], orthogonal frequency division multiplexing (OFDM) [28], Doppler radar signal analysis [29], and image smoothing (IMG). An IIR filter kernel was used for initial analysis.

1) MPEG-2 Encoder: An MPEG-2 encoder was parallelized from sequential code [27] to take advantage of concurrent processing available in aSOC. Three 128×128 pixel

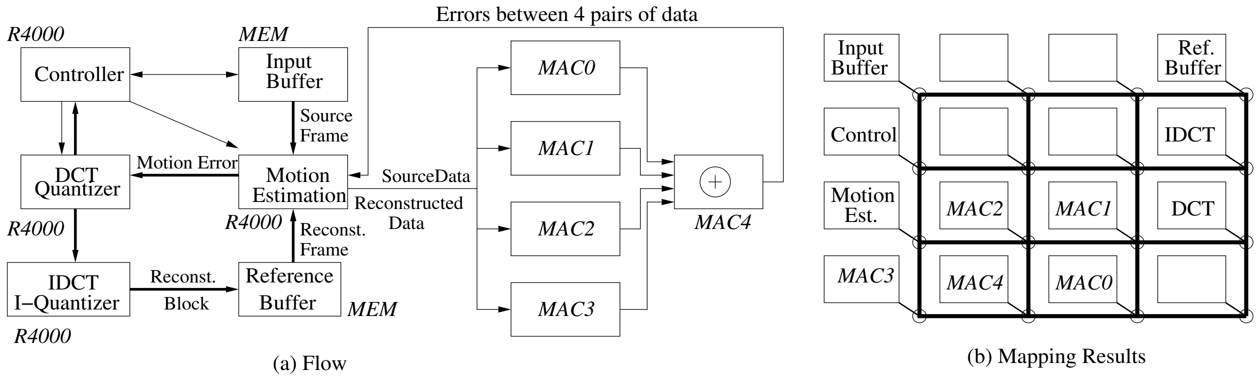


Fig. 11. Partitioning of MPEG-2 encoder to a 4×4 aSOC configuration

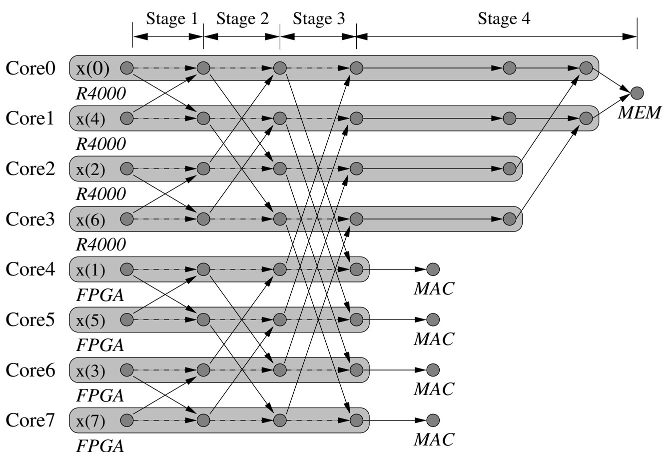


Fig. 12. OFDM mapped to 16 core aSOC model

frames, distributed with the benchmark, were used for aSOC evaluation. For the 4×4 aSOC configuration, MPEG-2 computation is partitioned as shown in Fig. 11. Thick arrows indicate video data flow and thin arrows illustrate control signal flow. Frame data blocks (16×16 pixels in size) in the Input Buffer core are compared against similarly-sized data blocks stored in the Reference Buffer and streamed into a series of multiply-accumulate cores via an R4000. These cores perform motion estimation by determining the accumulated difference across source and reconstructed block pixels, which is encoded by the discrete cosine transform (DCT) quantizer, implemented in an adjacent R4000. The data is then sent to the controller and Huffman coding is performed in preparation for transfer via a communication channel. Another copy of the DCT encoded data is transferred to the inverse discrete cosine transform (IDCT) circuit, implemented in an R4000 core. The reconstructed data from the IDCT core is then stored in the Reference Buffer core for later use. Data transfer is scheduled so that all computation and storage is pipelined.

2) Orthogonal Frequency Division Multiplexing: OFDM is a wireless communication protocol that allows data to be transmitted over a series of carrier frequencies [28]. OFDM provides high communication bandwidth and is resilient to RF interference. Multi-frequency transmission using OFDM requires multiple processing stages including inverse fast Fourier transform (IFFT), normalization, and noise-tolerance guard value insertion. As shown in Fig. 12, a 2048 complex-valued OFDM transmitter has been implemented on an aSOC model. The IFFT portion of the computation is performed using four R4000 and four FPGA cores. Resulting complex values are normalized with four MAC and four R4000 cores. A total of 512 guard values are determined by R4000 cores and stored along with normalized data in memory. The OFDM application exhibits communication patterns which change during application execution, as shown in the four stages of computation illustrated in Fig. 12. The run-time branching mechanism of the communication interface is used to coordinate communication branching for the four stages.

3) Doppler Radar Signal Analysis: A stream-based Doppler radar receiver [29] was implemented and tested using an aSOC device. In a typical Doppler radar system, a sinusoidal signal is transmitted by an antenna, reflects off a target object, and returns to the antenna. As a result of the reflection, the received signal exhibits a frequency shift. This shift can be used to determine the speed and distance of the target through the use of a Fourier analysis unit. The main components of the analysis include a fast Fourier transform (FFT) of complex input values, a magnitude calculation of FFT results and the selection of the largest frequency magnitude value. For the 16 core aSOC model, a 1024 point FFT, magnitude calculation, and frequency selection were performed by four R4000 and four FPGA cores. All calculation was performed on 64 bit complex values. Like OFDM, the Doppler receiver requires communication patterns which change during application execution. The run-time branching mechanism of the communication interface is used to coordinate communication branching for the four stages.

4) Image Smoothing: A linear smoothing filter was implemented in multi-core aSOC devices for images of size 800×600 pixels. The linear filter is applied to the image pixel matrix in a row-by-row fashion. The scalar value of each pixel is replaced by the average of the current value and its neighbors, resulting in local smoothing of the image and reducing the effects of noise. To take advantage of parallelism, each image is partitioned into horizontal slices and processed in separate pipelines. Data streams are sent from memory (MEM) cores to multiple R4000s, each accepting a single data stream. Inside each MAC, each pixel value is averaged with its eight neighbor values resulting in nine intermediate values. Later in the

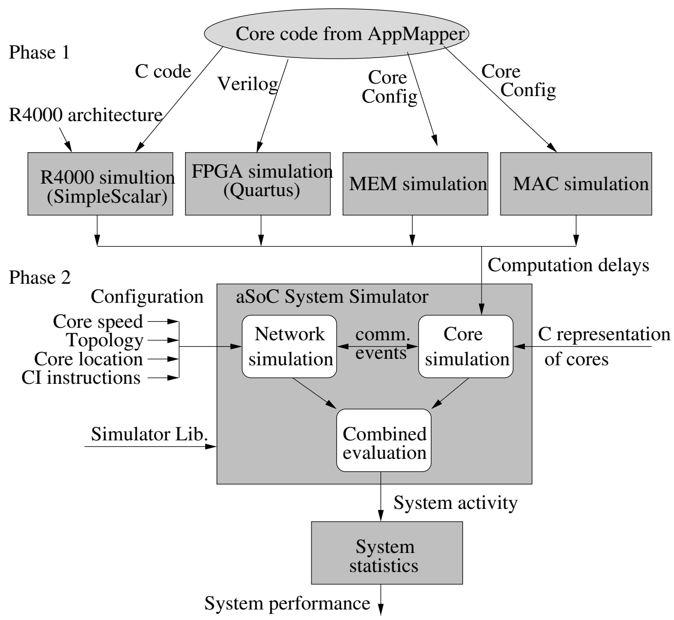


Fig. 13. aSOC system simulator

	Speed	Area (λ^2)
Comm. interface	2.5 ns	2500 × 3500
MIPs R4000 (w/o cache)	5 ns	4.3×10^7 [25]
MAC	5 ns	1500 × 1000
FPGA	10 ns	27500 ×26500
MEM	5 ns	7500 × 6500

TABLE IV
COMPONENT PARAMETERS

stream, an FPGA-based circuit sums the values to generate averaged results. These results are buffered in a memory core. This application was mapped to aSOC models ranging in size from 9 to 49 cores by varying the number of slices processed in parallel.

5) IIR Filter: A three-stage and a six-stage IIR filter were implemented using the 9 and 16 core aSOC models, respectively. The data distribution and collection stages of the filter use R4000s. MACs and FPGA cores are used to execute the middle stages of multiplication and accumulation. SRAM cores (MEM) buffer both source data and computed results. The overall application data rate is limited by aSOC communication speed.

B. Simulation Environment

To compare aSOC to a broad range of alternative on-chip interconnect approaches, including flat and hierarchical buses, a timing-accurate interconnect simulator was developed. This simulator is integrated with several IP core simulators to provide a complete simulation environment. The interaction between the computation and communication simulators provides a timing-accurate aSOC model that can be used to verify a spectrum of SoC architectures.

A flowchart of the simulator structure appears in Fig. 13. For our modeling environment, simulation takes place in two phases. In phase 1, simulation determines the exact number of core clock cycles between data exchanges with the communication network coreport interface. In phase 2, core computation time is determined between send and receive operations via core simulation which takes core cycle time into account. Data communication time is simultaneously calculated based on data availability and network congestion. Both computation and communication times are subsequently combined to determine overall run time.

During the first simulation phase, core computation is represented by C files created by AppMapper or user-created library files in C or Verilog. This compute information is used to determine the transfer times of core-network interaction. The execution times of core basic blocks are determined by invocation of individual core simulators. Cycle count results of these simulators are scaled based on the operating frequency of the cores. Specific simulators include:

- **Simplescalar** - This processor simulator [30] models instruction level execution for the R4000 architecture. The simulator takes C code as input and determines the access order and number of execution cycles between coreport accesses. Cycle counts are measured through the use of breakpoint status information.
- **FPGA block simulation** - Unlike other cores, FPGA core logic is first created by the designer at the register-transfer level. The Verilog-XL simulator is then used to determine cycle counts between coreport transfers. To verify timing accuracy, all cores have been synthesized to four-input LUTs and flip flops using Synplicity Synplify.
- **Multiply-accumulate** - The multiply-accumulate core is modeled using a C-based simulator. Given the frequency of an input stream, the simulator determines the number of cycles between coreport interactions.
- **SRAM memory cores (MEM)** - SRAM cores are modeled using a C-based cycle-accurate simulator.

Layouts, described in Section VII, were used to determine per-cycle performance parameters of the FPGA, multiply-accumulate, and memory cores.

The second stage of the simulator determines communication delay based on core compute times and instruction memory instructions. Following core timing determination, aSOC network communication ordering and delay is evaluated via the communication simulator. The instruction memory instructions are used to perform simulation of each tile's communication interface. This part of the simulator takes in multiple interconnect memory instruction files. High-level C code represents core and communication interfaces. As shown in Fig. 9(a), core compute delay is replaced with compute cycle (CompBlock) delays determined from the first simulation stage. The second input file to the simulator is a configuration file, previously generated by AppMapper. This file contains core location and speed information, the details of the inter-core topology and the interconnection memory instructions for each communication interface. These files are linked with a simulator library to generate an executable. When the simulator is run, multiple core and communication interface processes are invoked in an event-driven fashion based on data movement, production, and con-

Design	No. cores	No. Streams	**Max CI Instruct.**	**Max. Streams per CI**	Ave. Streams per CI	**Max. CPort Mem. Depth**	Ave. CPort Mem. Depth
IIR	9	11	2	5	3.5	2	2.0
IIR	16	20	2	5	3.5	4	2.7
IMG	9	8	2	3	2.0	2	1.5
IMG	16	15	4	4	3.5	4	1.9
IMG	25	20	4	7	2.0	4	2.1
IMG	36	28	4	7	1.8	4	1.5
IMG	49	36	4	7	2.6	4	1.5
Doppler	16	32	8	6	2.1	4	1.6
OFDM	16	39	9	6	2.2	4	1.5
MPEG	16	19	4	8	3.6	4	2.3
MPEG	25	37	5	8	3.2	4	3.0
MPEG	36	55	5	8	2.1	4	3.1
MPEG	49	73	5	8	5.3	4	3.0

TABLE V
BENCHMARK STATISTICS USED TO DETERMINE aSOC PARAMETERS

sumption. CDM and coreport storage is modeled to allow for accurate evaluation of inter-tile storage.

The simulator can model a variety of communications architectures based on the input parameter file. Architectures include the aSOC interconnect substrate, the IBM CoreConnect on-chip bus [7], a hierarchical CoreConnect bus, and a dynamic router. Parameters associated with aSOC, such as the core type, location, speed, and the communication interface configuration, can be set by the designer to explore aSOC performance on applications.

VII. RESULTS

To evaluate the benefits of aSOC versus other on-chip communication technologies, design mapping, simulation, and layout were performed. Benchmark simulation of aSOC models were used to determine architectural parameters. Core model assumptions were subsequently validated via layout. Our aSOC benchmark implementations were compared to implementations using alternative on-chip interconnect approaches and assessed versus previously-published work. As a final step, the communication scalability of aSOC was evaluated.

A. *aSOC Parameter Evaluation and Layout*

The benchmarks described in Section VI were evaluated using the aSOC simulator to determine aSOC parameters such as the required number of instructions per instruction memory. The cores listed in Table IV were used in configurations described in Section VI. R4000 performance and area were obtained from MIPs [25]. Multiply-accumulate, memory, and FPGA performance numbers were determined through core layout using TSMC 0.18um library parameters [31].

Benchmark run-time statistics determined via simulation are summarized in Table V. These statistics illustrate usage of various CI resources across a set of applications. The values were determined with parameters set to values which led to best-performance application mapping. Statistics which were used

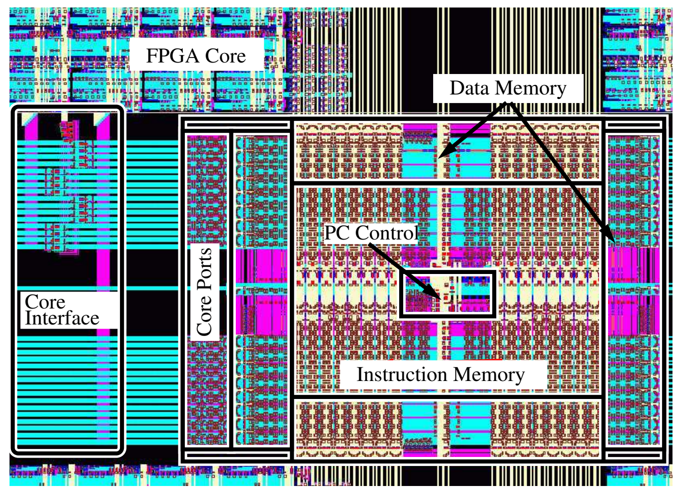


Fig. 14. Layout of FPGA core and communication interface

for CI architectural choices are highlighted in boldface. Although the maximum number of instructions per CI was relatively small for these designs (9), a depth of 32 was allocated in the aSOC prototype to accommodate expansion for future applications. Since the maximum total number of streams per CI is 8, each of the four CDM buffers per CI could be restricted to a depth of 2 in the prototype. The coreport memory depth was set to four, the maximum value in terms of streams across all benchmarks.

A prototype SoC device, including aSOC interconnect, was designed and implemented based on experimentally-determined parameters. The 9-tile device layout in a 3×3 core configuration contains lookup-table based FPGA cores with 121 clusters of 4 four-input LUTs, a complete communication interface, and clock and power distribution. Each tile fits a size of $30,000 \times 30,000\lambda^2$ with $2,500 \times 3,500\lambda^2$ assigned to the communication interface and associated control and clock circuitry (about 6% of device area). An *H-tree* clock distribution

	9-Core Model		16-Core Model				
Execution Time (mS)	IIR	IMG	IIR	IMG	MPEG	Doppler	OFDM
R4000	0.049	327.0	0.350	327	152	0.80	4.40
CoreConnect	0.012	22.0	0.016	30.5	173	0.13	0.21
CoreConnect (burst)	0.012	18.9	0.015	24.3	172	0.13	0.21
aSOC	0.006	9.6	0.006	7.3	83	0.11	0.18
aSOC Speed-up vs. burst	2.0	2.0	2.5	3.3	2.1	1.2	1.2
Used aSOC links	8	8	33	27	41	26	45
aSOC max. link usage	10%	8%	37%	28%	25%	2%	4%
aSOC ave. link usage	7%	7%	22%	25%	5%	2%	3%
CoreConnect busy (burst)	91%	100%	100%	99%	67%	32%	37%

TABLE VI

COMPARISON OF aSOC AND CORECONNECT PERFORMANCE

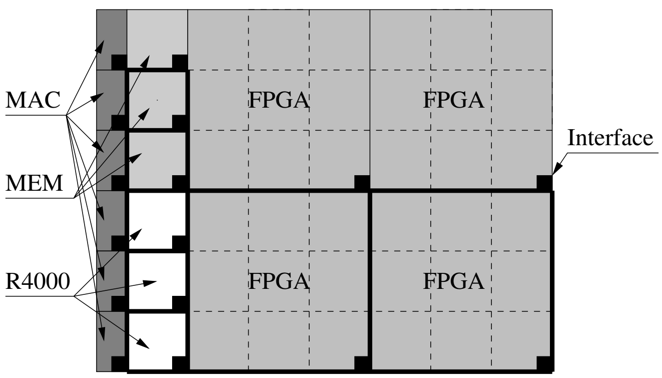


Fig. 15. Non-uniform aSOC core configuration

network is used to reduce clock skew between tiles. Layout was implemented using TSMC 0.18um library parameters resulting in a communication clock speed of 400 MHz. The critical path of 2.5 ns in the communication interface involves the transfer of a flow control bit from a CDM buffer to the read control circuitry of a neighboring CDM buffer, as shown in the right-to-left path in Fig. 7. A layout snapshot of a communication interface, coreport, and a single FPGA cluster appears in Fig. 14.

The layouts of the communication interface and associated cores support the creation of a non-uniform mesh structure which is populated to optimize space consumption. As shown in Fig. 15, tile sizes range from $10,000 \times 10,000\lambda^2$ to $30,000 \times 30,000\lambda^2$. From data in Table IV it can be determined that the communication interface incurs about a 20% area overhead for the R4000 processor. For comparison, an embedded Nios processor core [32] and its associated AMBA bus interface [6] were synthesized. A total of 206 out of 2904 total logic cells (7%) were required for the AMBA interface, with additional area required for bus wiring. This result indicates that the aSOC communication interface is competitive with on-chip bus architectures in terms of core overhead.

B. Performance Comparison with Alternative On-Chip Interconnects

A series of experiments were performed to compare aSOC performance against three alternative on-chip communication architectures: a standard CoreConnect on-chip bus, a hierarchical CoreConnect bus, and a hypothetical network based on run-time dynamic routing [33]. Performance was evaluated using the aSOC simulator described in Section VI-B. In these experiments, the IP cores with parameters shown in Table IV were aligned in the 9 and 16 configurations shown in Fig. 10. For each interconnect approach, the relative placement of cores and application partitioning was kept intact. Only the communication architecture which connects them together was changed for comparative results.

To evaluate aSOC bandwidth capabilities, a benchmark-based comparison is made for aSOC versus the IBM CoreConnect processor local bus (PLB) [7]. The PLB bus architecture allows for simultaneous 32-bit read and write operations at 133 MHz. When necessary, bus arbitration is overlapped with data transfer. The architecture requires two cycles to complete data transfer: one cycle to submit the address and a second cycle to transport the data. CoreConnect PLB supports burst transfers up to 16 words. The maximum possible speedup for a burst transfer versus multiple single-word transfers is about $2\times$.

It can be seen in Table VI that aSOC performance improvement over CoreConnect increases with a larger number of cores. Run times on a single 200 MHz R4000 are provided for reference. Relative aSOC improvement over CoreConnect burst transfer is indicated in the row labelled **aSOC speedup**. For most designs the CoreConnect implementation leads to saturated or nearly-saturated bus usage (as indicated by the row labelled **CoreConnect busy**).

A limiting factor for shared on-chip buses is scalability. To provide a fairer comparison to aSOC, a set of experiments was performed using a hierarchical version of the CoreConnect bus. Three separate CoreConnect PLBs connect **rows** of cores shown in Fig. 10. A CoreConnect OPB bridge [7] joins three subbuses (for 9 cores) or four subbuses (for 16 cores). When a cross-subbus transfer request is made, the OPB bridge serves as a bus slave on the source subbus and a master for the

	9-Core Model		16-Core Model				
Execution Time (mS)	IIR	IMG	IIR	IMG	MPEG	Doppler	OFDM
Hier. CoreConnect	0.013	26.0	15.7	37.4	178	0.15	0.22
aSOC	0.006	9.6	7.0	7.3	83	0.11	0.18
aSOC Speed-up	2.1	2.7	2.2	5.1	2.2	1.4	1.2
subbus 0 busy	85%	97%	99%	100%	94%	30%	30%
subbus 1 busy	72%	83%	99%	61%	94%	12%	27%
OPB bridge busy	40%	65%	81%	60%	93%	16%	36%

TABLE VII

COMPARISON OF aSOC AND HIERARCHICAL CORECONNECT PERFORMANCE

	9-Core Model		16-Core Model			
Time (mS)	IIR	IMG	IIR	IMG	MPEG	OFDM
Dynamic	0.008	14.4	8.7	9.7	162.0	0.19
aSOC	0.006	9.6	7.0	7.3	82.5	0.18
Speedup	1.3	1.5	1.3	1.3	2.0	1.1

TABLE VIII

COMPARISON OF aSOC AND DYNAMIC NETWORK PERFORMANCE

MPEG-2 Decoder	Throughput (Mbps)
CoreConnect [34]	0.68
aSOC	2.88
OFDM	Throughput (Mbps)
CoreConnect [35]	2.19
aSOC	5.67

TABLE IX

COMPARISON TO PUBLISHED WORK

destination subbus. As shown in Tables VI and VII, for all but one design, aSOC speedup versus the hierarchical CoreConnect bus is larger than speedup versus the standard CoreConnect bus. This effect is due to the overhead of setting up cross-bus data transfer.

In a third set of experiments, the aSOC interconnect approach was compared to a hypothetical on-chip dynamic routing approach. This dynamic routing model applies oblivious dynamic routing [33] with one 400 MHz router allocated per IP core. Tile topology for the near-neighbor dynamic network is the same as shown in Fig. 10. For each transmitted piece of data, a header indicating the coordinates of the target node is injected into the network, followed by up to 20 data packets. To allow for a fair comparison to aSOC flow control, the routing buffer in each dynamic router is set to be the maximum size required by an application. The results in Table VIII indicate that the aSOC is up to 2 times faster than the dynamic model. Performance improvements are based on the removal of header processing and compile-time congestion avoidance through scheduling.

C. Comparison to Published Results

Several experiments were performed to compare the results of aSOC interconnect versus previously-published on-chip interconnect results. An MPEG-2 decoder, developed from four Motorola PowerPC 750 cores interconnected with a CoreConnect bus, was reported in [34]. The four 83 MHz *compute nodes* require communication arbitration and contain an associated on-chip data and instruction cache. During decoding, frames of 16×16 pixels are distributed to all processors and results are collected by a single processor. To provide a fair performance comparison to this MPEG-2 decoder, our aSOC simulator was supplemented with SimpleScalar 3.0 for PowerPC [30] and applied to four PowerPC 750 core tiles interconnected with communication interfaces. The partitioning of computation was derived from [34], following consultation with the authors. In the experiment, the PowerPC cores run at 83 MHz and the aSOC communication network runs at 400 MHz. Table IX compares our results to previously published work. Unlike the 64-bit, 133 MHz CoreConnect model, aSOC avoids communication congestion by avoiding arbitration and providing a faster transfer rate (32 bits at 400 MHz) due to point-to-point transfers.

In previously published work [35], OFDM was also implemented using four 83 MHz PowerPC 750 cores interconnected with a 64-bit, 133 MHz CoreConnect bus. Each packet of OFDM data contains a 2048-complex valued sample and a 512-complex valued guard signal. This application was partitioned into four stages: initiation, inverse FFT, normalization and guard signal insertion. Each stage was mapped onto a separate processor core. Like the MPEG-2 decoder described above, the same mapping of computation to 83 MHz PowerPC 750 cores was applied to aSOC and modeled using SimpleScalar and aSOC interconnect simulators. Results are shown in Table IX. The aSOC implementation achieves improved performance for this application by providing high bandwidth and pipelined transfer.

Unlike the results for MPEG-2 and OFDM shown in Tables V through VIII, communication is not overlapped with computation during execution of the applications. This approach is consistent with the method used to obtain the previously-published results [34], [35].

D. Architectural Scalability

An important aspect of a communication architecture is scalability. For interconnect architectures, a scalable interconnect

Threads	Core Config.	Used Cores	Comm. Cycles	Throughput pixel/uS	Mbps
1	4×4	12	33,002,480	0.60	7.15
2	5×5	23	34,906,796	1.13	13.51
3	6×6	34	34,916,246	1.69	20.27
4	7×7	45	35,311,402	2.23	26.72

TABLE X

SCALABILITY OF THE MPEG2 ENCODER ON ASOC

Slices	Core Configuration	Used Cores	Execution Time (mS)
2	3×3	8	9.61
4	4×4	14	7.27
6	5×5	20	4.84
9	7×7	38	4.75

TABLE XI

SCALABILITY OF IMAGE SMOOTHING FOR 800X600 PIXEL IMAGE

can be defined as one that provides scalable bandwidth with reasonable (e.g. linear) latency increase as the number of processing nodes increase and as the computing problem size increases [36]. Under this definition, aSOC provides scalable bandwidth for many applications, including MPEG-2 encoding and image smoothing.

The MPEG-2 encoder is Fig. 11 can be scaled by replicating core functionality, allowing for multiple frames to be simultaneously processed in separate threads. A bottleneck of this approach is a common *Input Buffer* and data collection buffer at the input and output of the encoder. Since the communication delay of distributing the data to threads can be overlapped with computation, communication congestion and data buffer contention can lead to performance degradation as design size scales. Table X illustrates scalable performance improvement for multiple MPEG-2 threads implemented on aSOC. Device sizes ranging between 16 and 49 cores were considered. Total communication cycles increased marginally to accommodate routing and *Input Buffer* contention.

Using a similar multiprocessing technique, the image smoothing application was parallelized across a scaled number of cores using multiple threads applied to a fixed image size. Each 3-pixel high slice is handled by an R4000, a MAC and an FPGA. Table XI illustrates the scalability of the application across multiple simultaneously-processed slices. The image source and destination storage buffers are shared across slices. Application execution time scales down with increased core count until contention inside the storage buffers eliminates further improvement.

In a final demonstration of architectural scalability, a number of multi-point Doppler evaluations were implemented on a 16 core aSOC model. Execution time results of the Doppler application using CoreConnect and aSOC interconnect approaches are shown in Table XII. The benefits of aSOC over CoreConnect are due to the elimination of bus arbitration.

N	32	64	128	256	512	1024
R4000	50.0	89.0	176.0	263	792	1900
CoreConnect	3.7	7.4	28.2	60	130	340
aSOC	2.7	6.6	18.3	46	110	260

TABLE XII

DOPPLER RUN TIME FOR N POINTS (TIMES IN US)

VIII. CONCLUSIONS AND FUTURE WORK

A new communication substrate for on-chip communication (aSOC) has been designed and implemented. The distributed nature of the aSOC interconnect allows for scalable bandwidth. Supporting mapping tools have been developed to aid in design translation to aSOC devices. The compiler accepts high-level design representations, isolates code basic blocks, and assigns blocks to specific cores. Data transfer times between cores are determined through heuristic scheduling. An integrated core and interconnect simulation environment allows for accurate system modeling prior to device fabrication. To validate aSOC, experimentation was performed with four benchmark circuits. It was found that the aSOC interconnect approach outperforms the standard IBM CoreConnect on-chip bus protocol by up to a factor of five and compares favorably to previously-published work. A nine-core prototype aSOC chip including both FPGA cores and associated communication interfaces was designed and constructed.

We plan to extend this work by considering the addition of some dynamic routing hardware to the communication interface. The use of stream-based programming languages for aSOC also provides an opportunity for further investigation.

REFERENCES

[1] *The International Technology Roadmap for Semiconductors*, Semiconductor Industry Association, 2001.

[2] *IDT Peripheral Bus: Intermodule Connection Technology Enables Broad Range of System-Level Integration*, IDT, Inc., 2000.

[3] *Wishbone: System-on-Chip (SoC) Interconnect Architecture for Portable IP Cores, Revision B.3*, Silicore, Inc., Sept. 2002.

[4] *Silicon Micronetworks Technical Overview*, Sonics, Inc., Jan. 2002.

[5] P. J. Aldworth, "System-on-a-Chip bus architecture for embedded applications," in *Proc. IEEE Int. Conf. on Computer Design*, Austin, TX, Oct. 1999, pp. 297–298.

[6] D. Flynn, "AMBA: Enabling reusable on-chip design," *IEEE Micro*, vol. 17, no. 1, pp. 20–27, July 1997.

[7] *The CoreConnect Bus Architecture*, International Business Machines, Inc., Sept. 1999.

[8] M. Wan, Y. Ichikawa, D. Lidsky, and J. Rabaey, "An energy-conscious exploration methodology for heterogeneous DSPs," in *Proc. IEEE Custom Integrated Circuits Conf.*, Santa Clara, CA., May 1998, pp. 111–117.

[9] W. J. Dally and B. Towles, "Route packets, not wires: On-chip interconnection networks," in *Proc. ACM/IEEE Design Automation Conference*, Las Vegas, NV., June 2001, pp. 684–689.

[10] D. Wingard, "MicroNetwork-based integration for SOCs," in *Proc. ACM/IEEE Design Automation Conference*, Las Vegas, NV., June 2001, pp. 673–677.

[11] S. Borkar, R. Cohn, G. Cox, T. Gross, H. T. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb, "Supporting systolic and memory communication in iWarp," in *Proc. 17th Int. Symp. Computer Architecture*, June 1990, pp. 70–81.

[12] D. Shoemaker, C. Metcalf, and S. Ward, "NuMesh: An architecture optimized for scheduled communication," *Journal of Supercomputing*, vol. 10, no. 3, pp. 285–302, Aug. 1996.

[13] 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*, vol. 30, no. 9, pp. 86–93, Sept. 1997.

[14] 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 *Proc. Eighth ACM Conf. on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 1998, pp. 46–57.

[15] J. Babb, M. Rinard, C. A. Moritz, W. Lee, M. Frank, R. Barua, and S. Amarasinghe, "Parallelizing applications to silicon," in *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, Napa, CA, Apr. 1999, pp. 70–80.

[16] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjing, S. Liao, C. W. Tseng, M. Hall, M. Lam, and J. Hennessy, "SUIF: An infrastructure for research on parallelizing and optimizing compilers," *ACM SIGPLAN Notices*, vol. 29, no. 12, pp. 31–37, Dec. 1994.

[17] J. Babb, R. Tessier, M. Dahl, S. Hanono, and A. Agarwal, "Logic emulation with Virtual Wires," *IEEE Trans. Computer-Aided Design*, vol. 16, no. 6, pp. 609–626, June 1997.

[18] K. Keutzer, S. Malik, R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli, "System-level design: Orthogonalization of concerns and platform-based design," *IEEE Trans. Computer-Aided Design*, vol. 19, no. 12, pp. 1523–1543, Dec. 2000.

[19] P. Knudsen and J. Madsen, "Integrating communication protocol selection with hardware/software codesign," *IEEE Trans. Computer-Aided Design*, vol. 18, no. 8, pp. 1077–1095, Aug. 1999.

[20] K. Lahiri, A. Raghunathan, and S. Dey, "Performance analysis of systems with multi-channel communication," in *Proc. Int. Conf. on VLSI Design*, Calcutta, India, Jan. 2000, pp. 530–537.

[21] R. Dick and N. K. Jha, "MOCSYN: Multiobjective core-based single-chip system synthesis," in *Proc. European Conf. Design, Automation and Test*, Munich, Germany, Mar. 1999, pp. 263–270.

[22] G. DeMicheli and R. Gupta, "Hardware/Software Codesign," *Proc. IEEE*, vol. 85, no. 3, pp. 349–365, Mar. 1997.

[23] P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli, "System level hardware/software partitioning based on simulated annealing and tabu search," *Design Automation for Embedded Systems*, vol. 2, no. 1, pp. 5–32, Jan. 1997.

[24] R. Ernst, J. Henkel, and T. Benner, "Hardware software cosynthesis for microcontrollers," *IEEE Design and Test of Computers*, vol. 10, no. 4, pp. 64–75, Dec. 1993.

[25] *MIPS R4000 Product Specification*, MIPS Corporation, 2000.

[26] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*, Kluwer Academic Publishers, Norwell, MA, 1999.

[27] M. Ghanbari, *Video Coding: An Introduction to Standard Codecs*, The Institution of Electrical Engineers, London, England, 1999.

[28] D. Kim and G. Stuber, "Performance of multiresolution OFDM on frequency-selective fading channels," *IEEE Trans. Vehicular Technology*, vol. 48, no. 5, pp. 1740–1746, Sept. 1999.

[29] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, Inc., Upper Saddle River, NJ, 1999.

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

[31] T. Schaffer, A. Stanaski, A. Glaser, and P. Franzon, "The NCSU design kit for IC fabrication through MOSIS," in *Int. Cadence User Group Conf.*, Austin, TX, Sept. 1998, pp. 71–80.

[32] *Altera NIOS Processor Handbook*, Altera Corporation, 2003.

[33] W. Dally and H. Aoki, "Deadlock-free adaptive routing in multicomputer networks using virtual channels," *IEEE Trans. Parallel and Distributed Systems*, vol. 4, no. 4, pp. 466–475, April 1993.

[34] K. Ryu, E. Shin, and V. Mooney, "A comparison of five different multiprocessor SoC bus architectures," in *Proc. EUROMICRO Symp. on Digital Systems Design*, Warsaw, Poland, Sept. 2001, pp. 202–209.

[35] K. Ryu and V. Mooney, "Automated bus generation for multiprocessor SoC design," in *Proc. European Conf. Design, Automation and Test*, Munich, Germany, Mar. 2003, pp. 282–287.

[36] D. Culler, J. Pal Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufman, San Francisco, CA, 1999.

PLACE
PHOTO
HERE

Jian Liang received the B.S. and M.S. degree in electrical engineering from Tsinghua University, Beijing, China, in 1996 and 1999, respectively. He is currently pursuing the Ph.D. degree at the University of Massachusetts, Amherst. His research interests include reconfigurable computing, system-on-a-chip design, digital signal processing and communication theory.

PLACE
PHOTO
HERE

Andrew Laffely received the M.S. degree in electrical engineering from the University of Maine and the Ph.D. in electrical and computer engineering from the University of Massachusetts, Amherst. Dr. Laffely currently is involved in the operational test of various aircraft platforms for the United States Air Force. He has previously taught at the United States Air Force Academy.

PLACE
PHOTO
HERE

Sriram Srinivasan received the B. Tech. degree in electrical engineering from the Indian Institute of Technology, Madras, India, in 2000 and the M.S. degree from the University of Massachusetts, Amherst in 2002. He is currently working as a design engineer for Advanced Micro Devices (AMD), involved in the design of next generation microprocessors. His current research interests include VLSI circuit design for low-power and high-speed applications.

PLACE
PHOTO
HERE

Russell Tessier is an assistant professor of electrical and computer engineering at the University of Massachusetts, Amherst. He received the B.S. degree in computer engineering from Rensselaer Polytechnic Institute, Troy, N.Y. in 1989 and S.M. and Ph.D. degrees in electrical engineering from the Massachusetts Institute of Technology in 1992 and 1999, respectively. Dr. Tessier was a founder of Virtual Machine Works, a logic emulation company currently owned by Mentor Graphics. Prof. Tessier leads the Reconfigurable Computing Group at UMass. His research interests include computer architecture, field-programmable gate arrays, and system verification.