AFRL-RY-WP-TR-2008-1229

STRUCTURED APPLICATION-SPECIFIC INTEGRATED CIRCUIT (ASIC) STUDY

William Dally, James Balfour, David Black-Schaffer, and Paul Hartke

Stanford University

JUNE 2008
Final Report

AIR FORCE RESEARCH LABORATORY
SENSORS DIRECTORATE
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7320
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE

NOTICE AND SIGNATURE PAGE

*//Signature//

ALFRED SCARPELLI
Project Engineer
Advanced Sensor Components Branch
Aerospace Components and Subsystems
Technology Division

//Signature//

BRADLEY J. PAUL
Chief, Advanced Sensor Components Branch
Aerospace Components and Subsystems
Technology Division
Sensors Directorate

//Signature//

TODD A. KASTLE
Chief, Aerospace Components and Subsystems
Technology Division
Sensors Directorate

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YY)* June 2008	**2. REPORT TYPE** Final	**3. DATES COVERED *(From - To)*** 10 August 2007 – 30 June 2008

4. TITLE AND SUBTITLE STRUCTURED APPLICATION-SPECIFIC INTEGRATED CIRCUIT (ASIC) STUDY	**5a. CONTRACT NUMBER** FA8650-07-C-7726
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER 69199F
6. AUTHOR(S) William Dally, James Balfour, David Black-Schaffer, and Paul Hartke	**5d. PROJECT NUMBER** ARPS
	5e. TASK NUMBER ND
	5f. WORK UNIT NUMBER ARPSNDBK
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Stanford University, Computer Systems Laboratory, Gates Building, Room 301, Stanford, CA 94305	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory, Sensors Directorate, Wright-Patterson Air Force Base, OH 45433-7320, Air Force Materiel Command, United States Air Force; DARPA/IPTO, 3701 Fairfax Drive, Arlington, VA 22203-1714	**10. SPONSORING/MONITORING AGENCY ACRONYM(S)** AFRL/RYDI
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S) AFRL-RY-WP-TR-2008-1229

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES
PAO case number 12309, 22 October 2008. Document contains color.

14. ABSTRACT
Many of the digital electronic subsystems in defense applications require the small-size and power efficiency of application-specific integrated circuits (ASICs). Unfortunately, the high price and long design time of ASICs make them prohibitively expensive for low-volume DoD applications or systems requiring a rapid response time. This study introduces the concept of a "structured ASIC" that is an array of building blocks (microprocessors, signal processors, logic blocks, and memories) connected by an interconnection network. The vast majority of demanding DoD applications can be realized by configuring and connecting these building blocks with efficiency comparable to an ASIC but with a fraction of the development time and expense. This study also proposes a programming system that maps a high-level description of an application to a structured ASIC component. While this study has demonstrated the feasibility of structured ASICs, much work remains to mature this technology. This report closes with a set of recommendations for a program to develop this technology further.

15. SUBJECT TERMS
integrated circuit, signal processor, multiprocessor, microprocessor, FPGA, ASIC, interconnection network

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT:**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** (Monitor)
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Alfred Scarpelli
Unclassified	Unclassified	Unclassified	SAR	42	**19b. TELEPHONE NUMBER** *(Include Area Code)* (937) 255-2911

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18

Table of Contents

Section **Page**

List of Figures iv

List of Tables v

1 Executive Summary 1

2 The case for Structured ASICs 2
  2.1 DoD Systems and Development Methods are Well Suited for Structured ASICs 2
  2.2 Fabricate a single configurable component and power down unused regions 2
  2.3 Configure from a high level description 3
  2.4 Define standard interfaces and libraries 3

3 Structured ASIC Component and System Architecture 4
  3.1 Design Flow 4
  3.2 Structured ASIC Architecture 5

4 On-Chip Interconnect 7
  4.1 Static network 7
  4.2 Dynamic Network 8

5 Signal Processing Blocks 10

6 Logic Block (L-Block) 13
  6.1 Further Research 15

7 Memory Block 16

8 Processor Block 17

9 Programming Systems 18
  9.1 Further Research 25

10 Next Steps 26
  10.1 Gap Analysis 26
  10.2 Application Studies 28
  10.3 Prototype Studies 28
  10.4 A Timeline 29

11 References 31

List of Figures

Figure **Page**

1. Proposed Structured ASIC Methodology. 4
2. Possible Structured ASIC Component. 5
3. Static Network Router. 8
4. Signal Processing Element. 10
5. Breakdown of Energy Expended in S-Block Processor and Comparison to Microprocessor. 11
6. A Signal Processing Module. 12
7. Kernel Code for a Convolution Filter. 18
8. Kernel Graph of a Sample Application with Data Sources and Sinks. 20
9. Sample Application after being Automatically Parallelized and Buffered. 22
10. Kernel Placement. 23
11. Evaluation of S-Block Prototype Compiler. 24

List of Tables

Table **Page**

1. L-Block Benchmark Resource Requirements for Different Technologies. 14
2. Project Gantt Chart.. 30

1 Executive Summary

Many of the digital electronic subsystems in defense applications require the small-size and power efficiency of application-specific integrated circuits (ASICs). Unfortunately, designing a typical ASIC currently costs over $20M and takes over 2 years, and costs roughly double with each technology generation. These high development costs make ASICs prohibitively expensive for many low-volume DoD applications, and the long design time prevents ASICs from being used in systems that must be fielded rapidly in response to a new threat.

Much of the time and effort spent developing an ASIC are incurred as part of the effort required to manually *lower* a high-level description of a system to an equivalent low-level description that is suitable for fabrication. For example, the initial system description may be a program implemented in a high-level programming language; the high-level specification must be first be converted to a suitable low-level description, such as a register-transfer level description in a hardware description language, before being converted to a detailed circuit-level implementation and layout for fabrication.

The digital components found in defense systems are typically composed from a small set of building blocks such as signal processors, microprocessors, logic blocks, memories, and external interfaces. A structured ASIC is a semiconductor component that contains an array of these building blocks that can be configured from a partitioned high-level description of a system or application. By eliminating the time-consuming and effort-intensive steps of *lowering* the high-level description, a structured ASIC can be designed in a fraction of the time of a conventional ASIC and for a fraction of the cost. Fabrication times for conventional ASICs are typically months; a programmable (soft) structured ASIC can be configured in seconds, enabling rapid prototyping and rapid response.

This initial study has established the feasibility of the structured ASIC approach. We have developed a straw-man design for a structured ASIC component, prototyped a simple programming system for this component, and demonstrated some simple kernels and applications on a simulation of the component. Our preliminary results are positive. The design of each structured ASIC component appears feasible. As importantly, the programming system makes it easy to capture a high-level design and map it to the structured ASIC component. Much work remains, however, to work out remaining details and to mature this technology to the point where it can be deployed.

This report describes research activities undertaken to identify the requirements of a structured ASIC, to establish the feasibility of a structured ASIC component and implementation methodology, and to identify limitations of current technologies and design methodologies which will need to be addressed in subsequent research programs. The report closes with recommendations for a program to develop this technology further so that structured ASICs can be made available to the DoD community.

2 The case for Structured ASICs

2.1 DoD Systems and Development Methods are Well Suited for Structured ASICs

The following observations about the characteristics of electronic systems in defense applications and the methods used to design them motivate our approach to structured ASICs.

1. Most digital electronic systems are composed of four types of building blocks: signal processors, microprocessors, memories, and logic. Signal processors perform the preponderance of the computationally-intensive processing in signal and image processing applications (e.g., the modem in a communication system). Microprocessors implement lower-speed control processes and user interfaces. Memories are used to stage data, store intermediate results, and hold lookup tables. Logic is used for bit-intensive applications (e.g., scramblers, CRC checkers, or arithmetic entropy coders) that are difficult to implement efficiently using the arithmetic and logic operations available on general-purpose signal processors and microprocessors. Logic blocks tend to be small. Interconnect, typically buses and interconnection networks, connect the various blocks in a system.

2. Modern integrated circuits are becoming power, not area limited. We are already to the point where due to power constraints we cannot afford to fill a chip with logic or arithmetic and run it at full speed.

3. Many defense systems are produced in low volume: fewer than 100,000 units of a particular component may be produced throughout the lifetime of the system. Low volume systems are dominated by non-recurring development costs, which typically exceed $20M, rather than per-unit costs, which are often $100 or less. Reducing the cost of low volume systems requires reducing the non-recurring costs associated with the engineering effort expended in the design, implementation, and verification phases of development. For high-volume (>1M units) systems a different approach can be taken.

4. Most of the cost and time spent developing a system is incurred lowering the high-level language description of the system to a low-level hardware description language, typically Verilog or VHDL, and then to physical design. The tooling costs are typically less than 10% of the total development cost.

5. The compute intensive portion of most systems is implemented in a high-level language (usually Matlab and/or "C" code) before system implementation is started.

6. Systems often share functions. For example, two systems may use the same compression standard or the same modem. This is particularly the case where there are standards – e.g., for communication, encryption, and compression. Much of the complexity of many systems can be captured in a library of a relatively small number of such common modules.

7. There is a need to rapidly change systems. For example to respond to a new threat or exploit a new opportunity.

The observations above about system characteristics and development processes lead to a number of concepts for realizing ASIC efficiency with economical development.

2.2 Fabricate a single configurable component and power down unused regions

Observations 1, 2, and 3 suggest that we may be able to build a chip with a collection of building blocks in a fixed ratio and then power down the building blocks that are not needed for a given application. This will result in a chip that is larger than a chip tailored for the application, but with comparable power efficiency. As long as we are in the low volume (< 100,000 units) realm, die area (which drives per-unit cost) is a secondary concern. We are more concerned with power and efficiency. Because chips are power, not area limited, a special purpose part, while smaller, would not be more capable. It would just hit the power limit at a lower cost.

2.3 Configure from a high level description

Observations 4 and 5 suggest that to save the time and cost of conventional ASIC development we need a low-effort path from the pre-existing high-level executable description of the system to hardware. We may impose on the designer to partition the system into modules (this step requires human intuition but is not particularly costly or time consuming). However, we cannot ask the designer to lower the design to the RTL level or to perform a gate-level physical design (even with automated CAD tools) as these two tasks (not tooling costs and fabrication delays) account for the bulk of the time and cost of modern ASIC design. Hence a structured ASIC must be a good compilation target for a high-level language.

2.4 Define standard interfaces and libraries

While key elements of a new system may need to be designed from scratch (hopefully only to the level of a high-level language), observation 6 suggests that many peripheral functions of a system can be composed from standard library modules. Moreover, a library of such modules would facilitate the rapid response required by observation 7 by allowing system designers to focus only on the new portions of the system, rather than reimplementing already existing modules.

3 Structured ASIC Component and System Architecture

This section provides a summary of a structured ASIC component and system architecture.

3.1 Design Flow

The system architecture allows an efficient implementation to be realized using the design flow shown in Figure . The designer starts with an existing high-level language description for their system and a set of pre-existing library modules. Designers construct a system by selecting a number of library modules and providing high-level descriptions of those modules that are not already available in the module library. Designers may manually decompose the high-level descriptions of user-defined modules to expose task-level and data-level parallelism and assist the programming tools in partitioning the system into functions and modules that will map to the four types of building blocks described below.

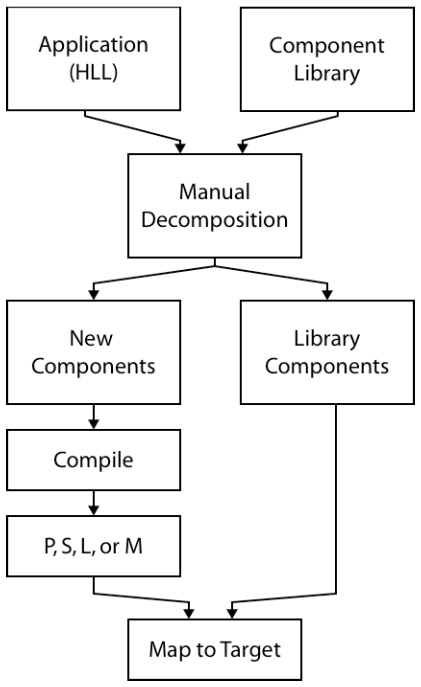

Figure 1: Proposed Structured ASIC Methodology.

An application written in a high-level language is manually decomposed drawing on reusable components from a library. Library components and any new components required are mapped to one of the four building blocks and assembled to complete the application.

Once the system is partitioned, the modules that were not derived from the library are each mapped to one of the four types of building blocks. Ideally these modules would be compiled from the high-level description. For the microprocessor, signal processor, and memory blocks this should be feasible. The details of this compilation process present many open problems that should be addressed by a continuing research program in this area.

The designer may need to provide more specific and detailed descriptions of the functions to be implemented in the logic blocks, possibly using a hardware description language that is supported by common logic synthesis tools. Usually those parts of a system that are more efficiently implemented in a logic module constitute a small portion of the system, and the required functionality is often more readily expressed in a hardware description language than a high-level programming language. Also, we expect that many of the logic blocks will be available in libraries as the functions that demands such blocks, such as scramblers, entropy coders, and encryption modules, tend to be defined by standards.

3.2 Structured ASIC Architecture

A possible structured ASIC component that supports this methodology is shown in Figure 2. The component consists of an array of building blocks in some predetermined mix of the four types. A family of such components may be available with different mixes. For example, a component for intensive signal processing applications could provide more signal processing blocks and fewer microprocessor or memory blocks. The individual building blocks are connected by a configurable interconnection network (the area between the blocks). A set of external interfaces connects the components together and to standard external memory and I/O devices.

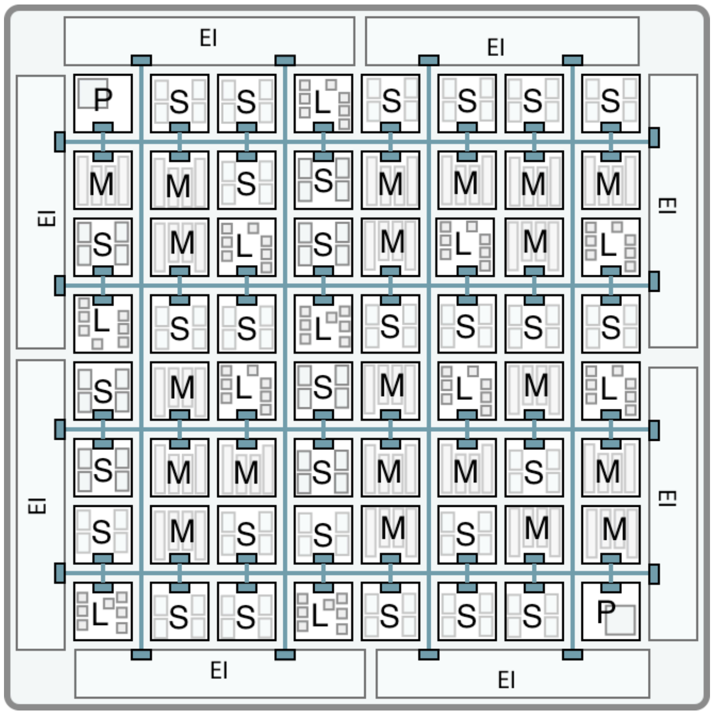

Figure 2: Possible Structured ASIC Component.

An array of microprocessors (P), signal processors (S), memory tiles (M), and configurable logic blocks (L) are connected by an interconnection network. External interfaces (EI) allow a component to be connected to other components and to standard external memory and /O devices.

Each microprocessor block (P) is simply a conventional embedded RISC processor. Its cache misses are carried out by messages over the interconnect, and it can read and write the state of other blocks via memory-mapped operations. Each signal processor block (S) may be either a highly efficient programmable signal processor or it may be a configurable array of arithmetic elements. The logic module (L) may be a small FPGA fabric (LUTs –lookup tables– connected by configurable interconnect) or it may be a sea of gates that is configured by an upper-layer metal mask. The memory (M) modules are conventional memory arrays along with a small amount of logic to allow them to be configured as queues or circular buffers. The memory modules may be configured as stand alone modules (e.g., to buffer a stream of data between two other modules), or can be configured as a part of an adjacent S or P module.

The interconnect provides both statically configured and dynamically routed connections between the modules. Configurable links provide programmable wiring from one module to a nearby module (perhaps a few tiles away). This allows the programming system to “wire up” modules in a flow-graph. As with the logic, this configuration may either be soft (FPGA-like) or hard (selected by a metal layer). Routed links allow any module to send a message to any other module. This provides complete connectivity for less intensive data flows, and a mechanism for distributing instructions to the microprocessor and signal processor blocks. The interface to the routed network also uses ready/valid flow control but the first byte(s) of each packet are stripped off by the network to be used for header functions. This header includes a byte count that identifies the end of the message.

The external interfaces (EI) allow larger systems to be constructed by composing components. Each EI block supports a set of standard interfaces that allow each structured ASIC to communicate with (1) other structured ASICs, (2) external commodity DRAM memory, and (3) external I/O devices. The requirements could be met, for example, by using a PCI-Express physical layer to handle items (1) and (3), a DDR-3 physical layer to handle task (2), some LVCMOS I/Os to handle (3) for primitive external devices, and possibly some A/D and D/A modules to handle (3) for analog inputs. For high-performance data converters and specialized physical-layer interfaces, the interface may be more efficiently provided on a separate I/O chip.

4 On-Chip Interconnect

The on-chip interconnect provides data communication between the exernal interfaces and the four-types of tiles on a structured ASIC chip and consists of two networks, a static network for streaming communication and a dynamic network for general-purpose communiation. The static network is used for tightly coupled, streaming communication between application kernels running on nearby buildling blocks. The dynamic network is used for general-purpose communication between application kernels, the OS, and global memory. Additionally, the dynamic network supports scatter/gather, remote code execution, and other complex procedures.

4.1 Static network

The statically routed network (SRN) is a mesh network with one router per building block. Routes are configured at application load time and rarely changed during the course of the application. Because of the static configuration, message sending is extremely low overhead. No header information (source, destination, priority, etc.) is required allowing very small (1 word) packet sizes. Additionally, there is no contention for physical channels or switches, giving a fixed latency and bandwidth. This makes the static network ideal for real time applications. The static network uses elastic-buffer flow control, so there is minimal flow control overhead. The static network supports message sending and background block transfers, or DMAs.

The static network uses elastic-buffer flow control within the channels. This flow control scheme uses the retiming elements (flip-flops or latches) in the pipelined channel as storage elements, with flow control between retiming elements implemented using a ready/valid handshake. A datum advances to the next retiming element when the downstream element is ready (has space available to receive the datum) and the driving latch valid (holds a valid datum to transmit). The paths through the configured network behave like distributed FIFO queues. This behavior provides a simple abstraction at the endpoints that hides the latency of the channel from the communicating entities, which should simplify the process of mapping applications to the structured ASIC.

A static network router is shown in Figure 3. Each router has 5 input ports (NSEW and injection) and 5 output ports (NSEW and ejection). Each port has four (4) 8-bit physical channels with a single elastic buffer register each. The routers are configured at application load time to connect channels from the input ports to channels of the output ports. Not all connections are possible. For example, you cannot connect any channel from input port 0 to any channel of output port 0 (a loopback connection). Additionally, to minimize hardware, channel *n* on any input port, can only be connected to channel *n* on any output port. Each router contains several control registers controlling the output muxes which control which inputs connect to which outputs. Programming these registers in each router effectively sets up routes between nodes.

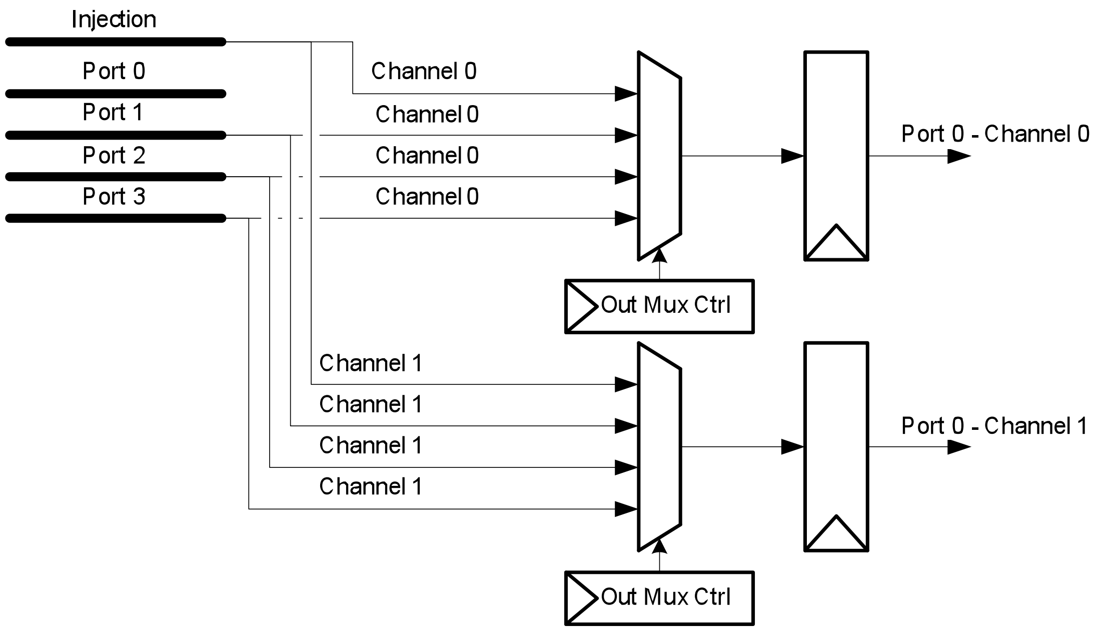

Figure 3: Static Network Router.

The static network router has five ports each of which is four channels wide. Each channel of each port selects its input using a four-to-one multiplexer controlled by a configuration register. One port each of two channels are shown here.

The static network is configured via messages sent over the global network. When the network interface in a block receives a static network configuration packet, it decodes the packet and programs the appropriate control registers. Each SRN router needs to be programmed separately. If a static route takes N hops, then N-1 network configuration packets must be sent. Routes are typically programmed by the OS or hardcoded into the executable. If the OS determines the route, it can take into account routes being used by other applications and provision resources accordingly. Other than the latency, the route chosen will not affect the application.

We have developed a design for low-powered channel circuits that reduces transport energy by up to two orders of magnitude (from 300pJ/bit-mm to 3pJ/bit-mm in a 65nm technology) compared to conventional on-chip channels with full-swing repeaters. These circuits would be used to reduce the cost of transferring data on both the configured interconnect and the routed interconnect. We have also developed a low-power flit buffer circuit that reduces write power by 65% and read power by 17% compared to a conventional SRAM circuit. The design techniques can be used to improve the energy efficiency of the memory modules in the structured ASIC component.

4.2 Dynamic Network

The dynamically routed network (DRN) is a general purpose global interconnection network. Every building block is a node in the network. Messages in the dynamic network have a format of: *{header, id, destination, length, data}*. The header specifies the type of packet and the location of the data. The ID is a unique identifier attached to the message. This is used to identify a response with a particular message. The ID is generated by the network interface. Data is of variable size, specified by the length.

Dynamic network packets can be sent from any building block to any other building block. On arrival, the packet is interpreted based on the header of the packet and the type of the building block. For example, configuration packets cause configuration registers to be written, get and put packets transfer block of data to and from memories – to load code or transfer data, etc…

We have not yet determined the topology of the dynamic network. However, a flattened butterfly topology [2] appears to be well-suited for this module.

5 Signal Processing Blocks

The signal processing elements (SPElements) within the signal processing blocks use a distributed and hierarchical data register organization. Most data references are satisfied by the distributed collection of small (4-word) operand register files (ORFs) located at the inputs of the functional units. These small register files capture short-term data locality to keep a significant fraction of the data bandwidth local to each functional unit. The ORFs are preceded by a switch which allows data produced in one functional unit to be passed to the ORFs located in another. Instructions are issued from a collection of software-controlled Instruction Registers (IRs). The IRs are implemented as shallow (64-entry) register files that are distributed among the functional units. Storing instructions in shallow register files reduces the cost of accessing the stored instructions, while placing the register files within the functional units they control reduces the cost of transferring instructions to the datapath control points.

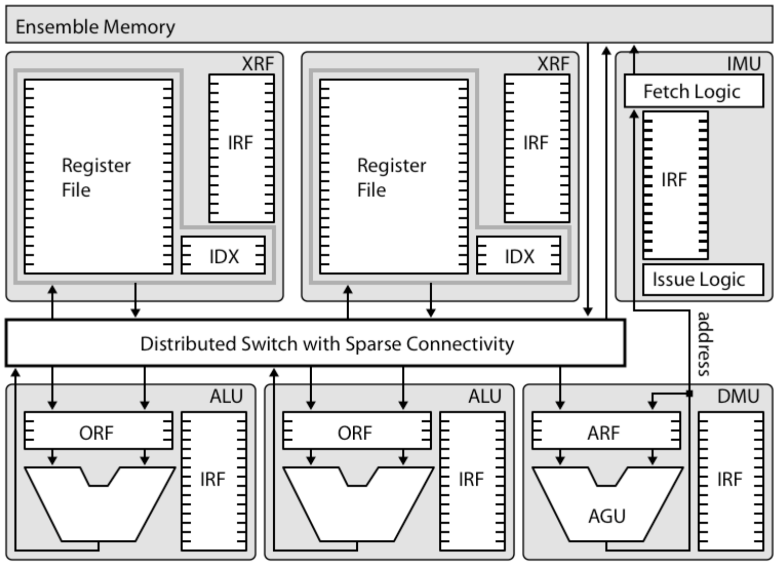

Figure 4: Signal Processing Element.

The Signal Processing Blocks are composed of 4 signal processing elements. The signal processing elements use a hierarchical and distributed data register organization, which consists of the indexed register files (XRFs) and operand register files (ORFs), and distributed instruction registers, implemented in instruction register files (IRFs), to supply data and instructions to the ALUs.

We have completed a prototype of a compiler backend for scheduling signal processing kernels for the distributed and hierarchical register organization used in the signal processing blocks. The principle challenge when scheduling instructions for the distributed and hierarchical register organization is scheduling paths through the registers to the operations that consume them. The prototype scheduler adopts a unified approach to register allocation and instruction scheduling based. The scheduler models the architecture as a resource graph in which the graph nodes are storage and execution resources and the graph edges describe the connectivity. The compiler schedules operations at the granularity of register transfers and arithmetic operations by finding paths through the resource graph. To avoid backtracking during scheduling, the compiler ensures that a set of disjoint paths are available for routing operands to unscheduled instructions. This invariant is preserved by formulating the open-path problem as a maximum flow problem on the resource graph.

The prototype scheduler has been evaluated on six embedded kernels. The prototype compiler produces schedules that are within 1.7× of the performance of manually assembled kernels. The prototype scheduler lacks sophisticated algorithms for allocating registers and scheduling operations at basic block boundaries. Consequently, the scheduler performs better when scheduling code with large basic blocks. Existing solutions to similar problems, such as trace scheduling, generally address these problems by restructuring the code to increase the length of basic blocks. These approaches are unattractive to us because they will tend to produce kernels that cannot be captured in the instruction registers. Further research is needed to extend the compiler algorithms we have developed to improve the allocation of registers across basic blocks and the scheduling of operations across basic blocks.

Even when issued from instruction registers, fetching and issuing instructions constitutes a significant fraction of the energy expended in the prototype signal processing block, as is illustrated in Figure 5. The efficiency with which instructions are delivered deteriorates when the instruction registers cannot capture those instructions that are most frequently executed in a kernel and the instructions must be loaded from local memories.

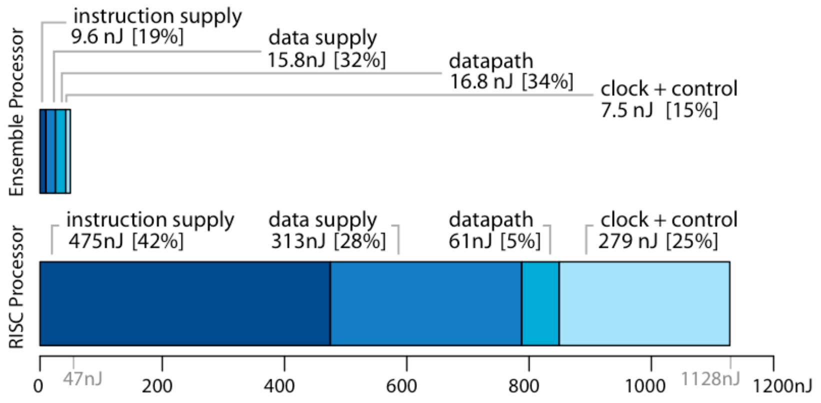

Figure 5: Breakdown of Energy Expended in S-Block Processor and Comparison to Microprocessor. *The breakdowns represent the geometric mean over a collection of common signal processing kernels, including a two-dimension convolution, a discrete cosine transform, an FIR filter, and a Viterbi decoder.*

To further improve the efficiency with which instructions are delivered, the prototype signal processor has been extended to allow the neighboring processors to pool their instructions registers. A group of 4 neighboring processors form an Ensemble, as illustrated in Figure 6. The processors in an Ensemble can join together to operate as a single SIMD processor. This allows the processors to allocate some of their instruction registers to a pool of shared instruction registers. Instructions from the pool of shared are issued to all 4 processors at once. Each processor can reserve some of its instruction registers for its own use. This allows the processors to independently execute instructions in those portions of a kernel that lack SIMD parallelism. The overhead of entering and exiting SIMD mode is modest, comparable to executing a branch instruction.

One promising approach to the design of the signal processing block is described in [1].

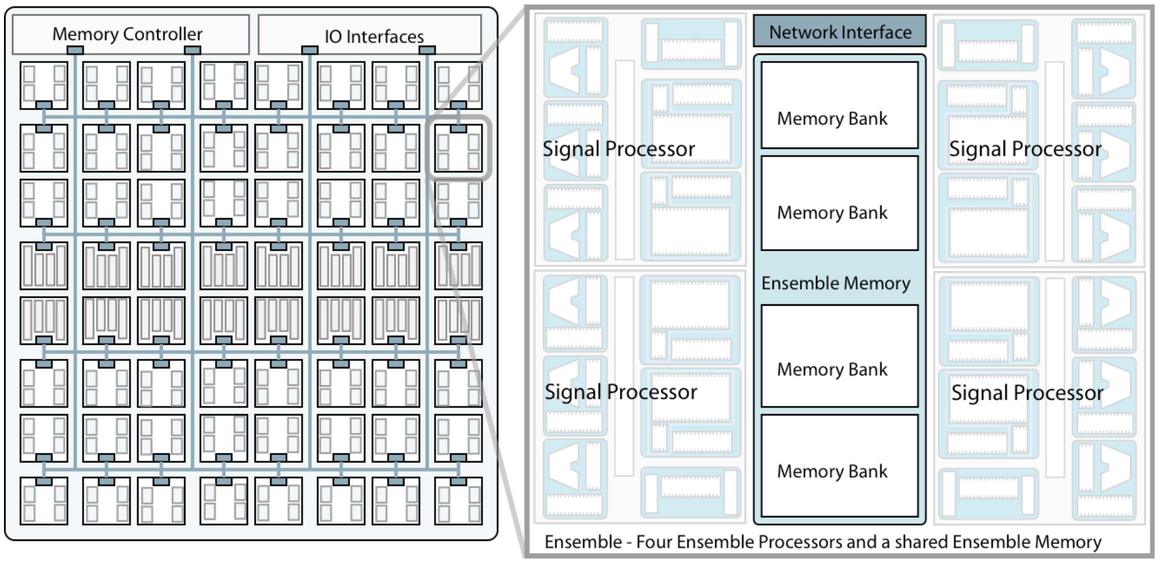

Figure 6: A Signal Processing Module.

A signal processing module is an ensemble of 4 signal processing elements, the processors within the Ensemble. To capture the instructions of longer kernels and to reduce the cost of issuing instructions, the instruction registers of the 4 processors within the Ensemble can be chained together to provide a single instruction register file with 4 times the capacity.

6 Logic Block (L-Block)

The Logic Blocks (L-Blocks) are optimized to implement modules that are dominated by bit-level operations. Examples of such modules include scramblers, ciphers, entropy coders and decoders, and error correction coders and decoders. In a conventional digital signal processor (DSP), most of these functions are implemented using sequences of application specific instructions, such as those used to perform Viterbi decoding in a communications DSP. The efficiency of a structured ASIC is improved by streamlining the S-Blocks for arithmetic-intensive processing and L-Blocks for bit-intensive processing. The modules that would be mapped to L-Blocks typically comprise a small portion of the system. Consequently the kernels that are mapped to L-Blocks are expected to be small and limited in number.

Three kernels were selected for the L-Block benchmark study:

1. Huffman coding is an entropy encoding algorithm used for lossless data compression that encodes each symbol of an alphabet with a bit string. The benchmark Huffman encoder accepts the letters of the alphabet, A-Z, using 5-bit inputs. The output is a serial bit stream of the encoded characters.

2. Cyclic Redundancy Checks (CRC) are used for error detection of data in communication networks. This benchmark CRC uses the Ethernet polynomial and accepts one 32-bit inputs per cycle.

3. The NIST Advanced Encryption Standard (AES) is a block cipher used in symmetric key cryptography. The benchmark AES encryption uses a block size of 128 bits and a key size of 128 bits and implements one round per clock cycle amenable to Cipher Block Chaining (CBC) configurations.

Four technologies have been targeted for L-Block study:

1. FPGA 4-LUTs: Xilinx ISE 8.2 implementation tools with Xilinx Synthesis Technology (XST) used to target a Virtex-4 FPGA.
2. FPGA 6-LUTs: Xilinx ISE 8.2 implementation tools with Xilinx Synthesis Technology (XST) used to target a Virtex-5 FPGA.
3. TSMC 130nm ASIC. Synopsys Design Compiler used to synthesize the design, with the TSMC provided TCB013GHP standard cell library as the target technology.
4. TSMC 130nm Mask-Programmable Sea of Gates. Synopsys Design Compiler used to synthesize the design, with the TSMC provided TCB013GHP standard cell library as the target technology. The choice of cells is restricted to 2-input NAND gates and basic D flip-flops during the technology mapping phase to simulate the resource constraints of a mask-programmable sea of gates.

Table 3 lists the resource requirements for the targets described above. The standard cell library uses a cell height of 3.69um (nine vertical metal tracks). For reference, a minimum drive inverter is 0.92um wide (two horizontal metal tracks), and a minimum drive NAND2 gate is 1.38um wide (three horizontal metal tracks). For comparison, an S-block implemented in the TCB013GHP library has an area of 0.48mm^2, which is 5.7x larger than the ASIC implementation of the AES cipher. From this analysis we can see that an L-block with 1K to 2K 4-input LUTs would satistfy the demands of even the most complex logic modules while taking a fraction of the area of an S-block or P-block.

	Huffman Encoder	Ethernet CRC	AES Encryption
FPGA			
LUT4 Flip-Flops	20	32	403
LUT4 Instances	30	151	899
BlockRams	1	0	17
LUT6 Flip-Flops	20	33	402
LUT6 Instances	31	99	577
BlockRams	0	0	17
ASIC			
Combinational [um^2]	840	2,875	84,182
Sequental [um^2]	509	1,419	14,853
Restricted ASIC			
Combinational [um^2]	1,571	5,189	147,607
NAND2 Gates	316	1,044	29,712
Sequential [um^2]	509	815	13,494
DFF Gates	20	33	402

Table 1: L-Block Benchmark Resource Requirements for Different Technologies.

The number of sequential elements (D flip-flops) required by each benchmark can be determined by the LUT4 or LUT6 Flip-Flops entries in Table 3. As is typical of FPGA implementations, more LUTs are required than flip-flops, which leaves a number of the flip-flops unused. With one flip-flop provided for each LUT, the benchmarks above exhibit flip-flop utilizations in the range of 21% – 67%. To approach the energy efficiency of an ASIC, the clock signal routed to the unused flip-flops would need to be gated. The tables used in the AES benchmark, which are stored in block RAMs under the FPGA implementations, would be implemented in M-Blocks.

The area of the sequential elements decreases when the ASIC gate selection is restricted because larger, more complex flip-flops, such as those with input multiplexers, are not available. Instead, the functionality is implemented in combinational logic. This accounts for the greater increase in combinational area observed for the Ethernet CRC benchmark. The increase in combinational area that results when the gate selection is restricted could be reduced by incorporating some complex gates (such as AOI, XOR, and multiplexers) and some simple arithmetic circuits (such as small counters and comparators which are used in the state machines). For example, the combinational logic in the CRC benchmark is dominated by the XOR gates that fold the incoming 32-bit word of data into the checksum. The large (10x) difference in the resource requirements of the benchmarks suggests it may be desirable to partition L-Blocks internally or to distribute a kernel over a number of smaller L-Blocks to achieve high utilization within an L-Block.

The input and output interfaces of the benchmarks differ substantially. The Huffman encoder accepts 5-bit character codes, the CRC module accepts its input data as 32-bit words, while the AES cipher accepts its input as 128-bit words. The Huffman encoder produces a serial bit-stream output, the CRC module produces a 32-bit checksum after processing an entire frame, while the AES cipher produces encoded 128-bit words after some number of cycles. However, all of these interfaces are easily adapted to the byte-wide channels of the static network. The elastic-buffer flow control of this network can handle both the periodic traffic of blocks such as the AES module as well as the data dependent traffic of the Huffman module.

6.1 Further Research

Our work has identified the following areas that require further research.

1. **Composition of the logic fabric:** A broader range of kernels should be analyzed to determine a preferred combination of combinational and sequential logic elements. Related problems are how to map multiple small modules to a single L-Block, and how to partition a module that is too large for a single L-Block across multiple L-Blocks.

2. **Interface to other blocks:** The correct mechanism for interfacing with other blocks remains to be determined. We have identified two possible interfaces: a streaming interface and a bulk transfer interface. A streaming interface would stream data in to and out of an L-Block using the static network. Modules implemented in the L-Block may require local buffering to adjust input and output rates. A bulk transfer interface would use Memory Blocks to stage bulk data transfers in to and out of an L-Block – via either the static or dynamic network. The memories could be operated as large input and output queues to decouple consumption and production rates. It should be useful to allow an L-Block to use both streaming and bulk transfer interfaces; for example, an entropy coder might receive a frame of data to encode over a block transfer interface and then output the encoded bit-stream over a streaming interface.

7 Memory Block

Each memory block consists of a conventional SRAM memory array and an interface to the network. The network interface parses incoming messages to transfer individual memory words, blocks of memory, and to implement common data structures such as queues. Each memory block can be configured to be accessed via either the static or the dynamic network.

8 Processor Block

Each processor block consists of a conventional microprocessor (e.g., a MIPS 24K or ARM 11) along with an instruction cache and data cache (nominally 16K Bytes each). The processor is connected to both the static and dynamic networks via memory mapped interfaces and cache misses are automatically converted into memory access messages in the network. The processor can be configured to boot from an external serial ROM connected to the I/O interface via the network.

9 Programming Systems

The programming system for a structured ASIC must be designed to provide an easy to use, yet flexible, environment for describing applications that can then be efficiently and automatically parallelized and mapped by the compilation system. The applications that will be implemented on the structured ASIC platform exhibit demanding computation requirements. Consequently, the applications will need to be distributed over many compute resources, such as those provided by the S-Blocks and L-Blocks. Programmers writing applications will need the flexibility to express arbitrary computations to allow new and emerging algorithms to be described. To reduce effort involved in developing a new system, programmers must be able to reuse portions of existing designs. Furthermore, the programming system must automate aspects of the development process to simplify the effort required for programmers to develop reusable components for use in applications that target hundreds of processors blocks.

The goals of programmer flexibility, application reuse, and compiler automation can be achieved by building applications from discrete computation kernels. This approach provides a level of isolation between the unit of computation and the overall data movement within the application. The programmers retains the ability to write arbitrary code in the language of their choice within each kernel, but by providing a parameterized interface to the kernel, the compilation system can analyze and manipulate it, and other programmers can reuse it. Similar approaches have been successfully used in programming systems such as Mathwork Simulink. However, while Simulink primarily provides a rapid prototyping environment, the programming system for a structured ASIC must provide tools that first parallelize the application to meet computation requirements, and then map the application to the target architecture in a manner that achieves efficiency.

The kernels serve to encapsulate a unit of computation and storage that can be analyzed and manipulated at the system level. Figure 7 presents the source code for a kernel that implements a convolution filter. Kernels have unrestricted access to local computation and storage resources. Kernels communicate through explicitly parameterized input and output interfaces. Using constraints provided by the programmer and program analysis of the kernels, the programming system infers what resources a kernel uses, such as computation cycles and memory storage, to permit later analysis. This explicit partitioning of an application into computation kernels is natural for data-intensive signal processing applications, and closely follows the way in which designers reason about applications.

Making kernels parameterized facilitates re-use. For example, the kernel of Figure 7 is parameterized by the height and width of the convolution kernel and by the required data rate and hence can be used in almost any situation requiring a convolution.

```
public void runConvolve() {
   double[][] in = readInput("dataIn");
   double[][] result = new double[1][1];
   for (int x=0; x<width; x++)
         for (int y=0; y<height; y++)
               result[0][0] += in[x][y]*coeffcients[width-x-1][width-y-1];
   writeOutput("out", result);
}

public void loadCoeff() {
   coeffcients = readInput("coeffcients");

}
```

Figure 7: Kernel Code for a Convolution Filter.

Note that there is one method called for loading the coefficients and one for running the convolution. The coefficients are stored in memory locations that are local to the kernel.

Applications are built by instantiating kernels, connecting their the inputs and outputs, and inserting data sources and sinks. Figure 8 illustrates a simple system constructed from four kernels. This approach is flexible enough to handle any application (whether they be regular or not) and allows for seamless reuse of kernels. The data sources specify the data rates processed by the application. The programmer can also cleanly specify the level of parallelism possible for each kernel by adding data-dependency edges between kernels whose degree of parallelism is limited (e.g., entropy encoders or other kernels with internal state). The resulting application graph specifies the behavior of the application, but must be further manipulated to map it to the targeted platform.

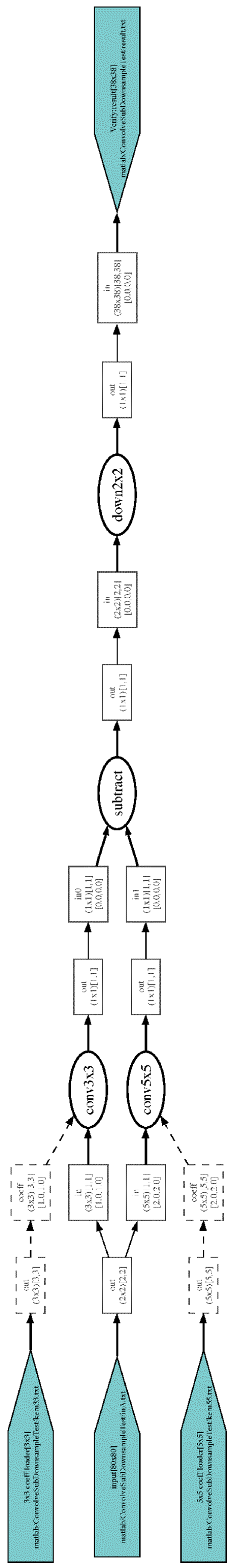

Figure 8: Kernel Graph of a Sample Application with Data Sources and Sinks.

This application passes an image through two filters, computes the difference, and downsamples the result. The kernels are represented by ovals, and data connections by arrows. The boxes annotating the arrows indicate intermediate storage that has been allocated to buffer data between kernels.

The kernel graph structure of the application and the parameterized resource requirements for each kernel allow the compilation system to manipulate the application at a high-level. By propagating the input rates from the data sources through the application, the compilation system can determine the required data processing rate for each kernel. Combined with a description of the capabilities and resources of the targeted hardware (i.e., the number of operations per second or words of memory), the degree of required parallelization can then be determined for each kernel. For data-parallel kernels, the compilation system can automatically replicate the kernel as required to meet its throughput requirements, and insert appropriate buffering and data distribution. The compilation system can also explore merging kernels before parallelizing them to explore whether time multiplexing performs better than space multiplexing. This automatic parallelization relieves the programmer of the need to partition, stage, and synchronize data for many separate processors. Figure 9 illustrates the kernel graph representation of the implementation that results after transformations are applied to the system of Figure 8. As illustrated, kernels are duplicated when necessary to achieve throughput objectives. Additional buffers are introduced to distribute and reassemble data streams when a kernel is replicated. For non-data-parallel kernels, the programmer can either parallelize the kernel by hand or provide a library that partitions the kernel as required. This allows non-regular partitioning (such as that required to parallelize an FFT) to be included seamlessly through libraries or manual kernel instantiation.

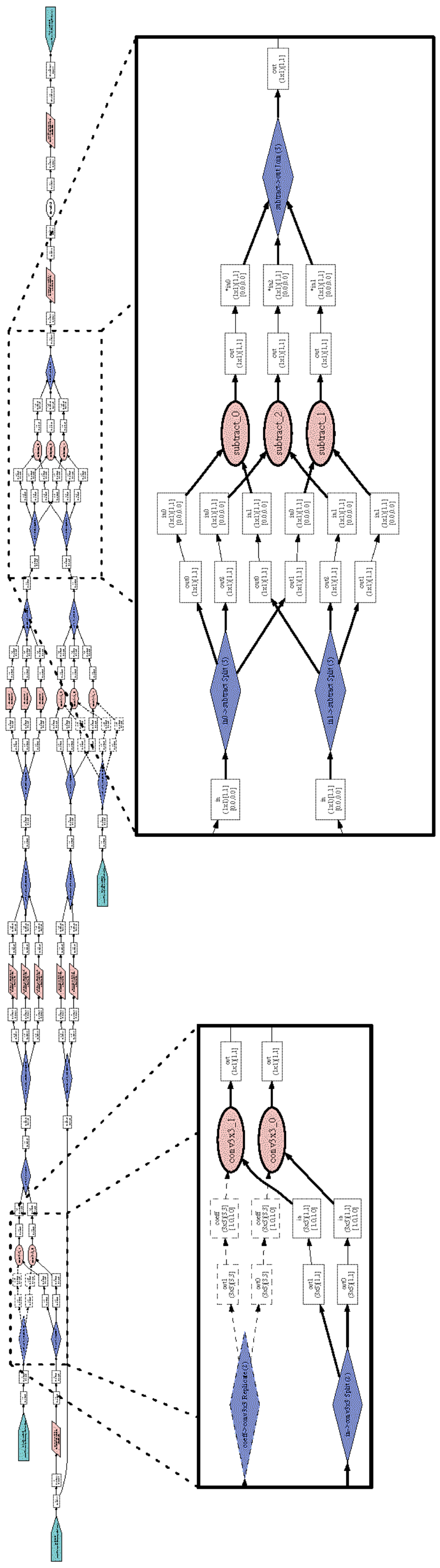

Figure 9: Sample Application after being Automatically Parallelized and Buffered.

This is a complete and correct description of the application.

Once the application has been parallelized and buffered appropriately, the compilation system can map the application to the hardware. The data sources' rates and input sizes are once again propagated through the now parallelized application, and all edges are annotated with their data communication rates. This rate information allows placement of the kernels to minimize the energy spent on communication by placing kernels on logically close processors or logic blocks. Figure 10 illustrates how the programming system automatically improves an initial, random assignment of kernels to blocks to reduce communication costs. The resulting placement represents the complete application with sufficient automatically-inserted parallelism to meet the specified rate requirements on the targeted hardware platform.

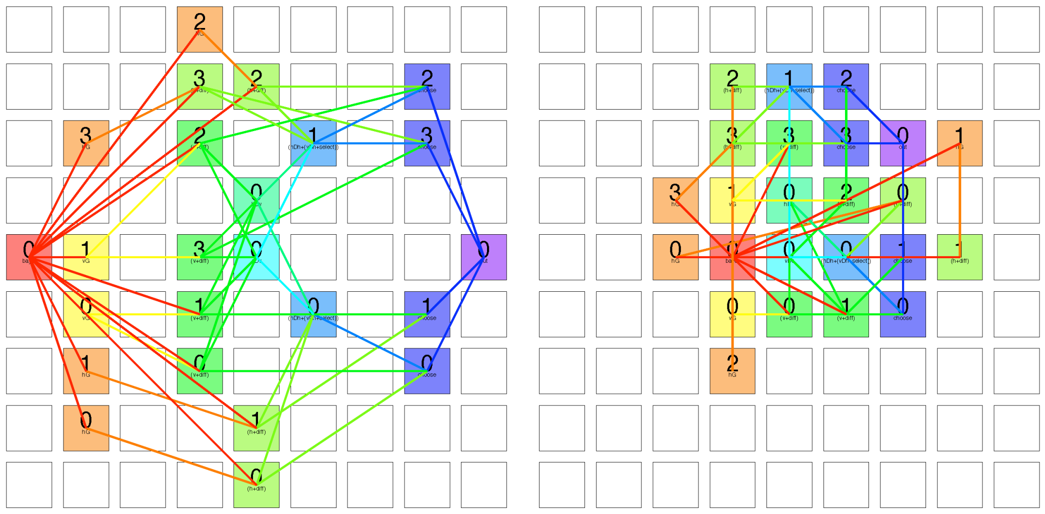

Figure 10: Kernel Placement.

Each block is colored to indicate the kernel assigned to it. The numbers indicate an instance of a kernel that has been duplicated. Arcs indicate communication. The initial placement appears on the left, while the improved placement after simulated annealing to minimize communication cost appears on the right.

The application placement information is then used to appropriately compile the kernels or merged kernels placed on each processing block. Kernels scheduled for the P-blocks can be readily compiled with a standard compiler such as gcc, using libraries to interface with the inputs and outputs to each kernel. Code for the L-Blocks can either consist of kernels written in HDL or a subset of C which is then compiled down to gates by any of a variety of commercially available tools. The S-Block code is compiled by an optimizing compiler that efficiently targets the signal-processing specific features of the DSP cores.

In addition to implementing the programming system tools described above, we have implemented a prototype compiler that compiles kernels written in a high-level language and generates code for an SPElement. The prototype compiler performs basic code optimizations such as constant folding, copy propagation, and dead code elimination. The prototype compiler implements a unified instruction scheduling and register allocation algorithm based on the concept of scheduling operand paths through the distributed and hierarchical data registers. To contrast the quality of compiled code, Figure 11 presents the normalized throughputs of the compiled and assembled kernels. Despite our compiler performing only limited intra-basic block optimizations and no inter-basic block optimizations, compilation incurs a similar penalty for both processors: the average throughput using the SPElement compiler is 1.75x worse, while the average RISC throughput using gcc is 1.72x worse.

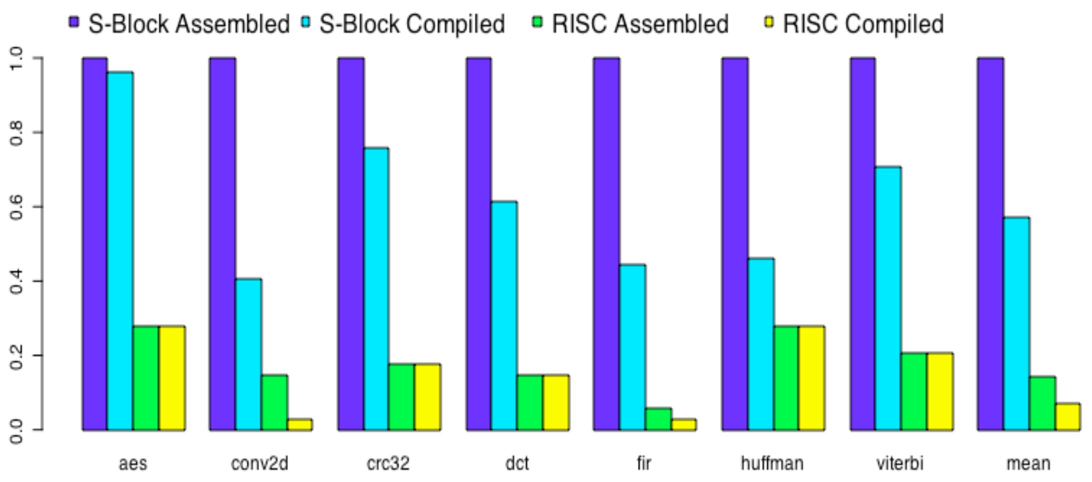

Figure 11: Evaluation of S-Block Prototype Compiler.

The throughput of each implementation is normalized to the assembled S-Block implementation. The average reduction in S-Block throughput observed when the prototype SPElement compiler is used is 1.75x, while the average reduction in RISC throughput when gcc is used is 1.72x.

While the approach described in this section deals with many of the time-consuming and error-prone aspects of programming the many cores of the structured ASIC architecture, it does not address all of them. In particular global resource access, variable rate computation, and variable modes computation need to be addressed. For applications which require random global resource access, such as motion vector searches, the current approach requires that all memory accesses go through one centralized kernel. This could negatively impact performance and scalability. Incorporating global access as part of the application graph would allow this to be incorporated into the higher-level parallelization and mapping. In addition, the proposed data analysis can only determine the degree of parallelization required for a kernel if given static data rates. For applications where the rate varies, this will result in an over-provisioning of resources compared to the common case. Similarly, applications with different modes of computation, such as an compression engine that switches between inter- and intra-frame compression based on the motion vector residuals, will need to fully-provision resources for both modes simultaneously. To reduce this over-provisioning, a more flexible rate definition incorporating both the distribution of rates and a higher-level approach to error handling needs to be incorporated into the application model.

Our research has developed a method for automatically analyzing and mapping real-time streaming applications to systems with many processors. This category of applications includes most modern signal processing, image processing, and communications applications. The key insight is that to automate this process, the applications need to be described in such a way that the structure and communications patterns are exposed to the compiler. This allows the compiler to understand data reuse and data dependencies to automatically parallelize the applications to meet the real-time performance requirements.

The compiler system we have developed takes a real-time streaming application consisting of streams of data and computation kernels. The inputs to the application are annotated to describe the size of the data and how fast it is received. Using this information, the compiler system analyzes the application to determine how much computational power and buffering space are required. Each part of the application is then parallelized to insure it receives its required resources, and appropriate buffers are inserted. The result is a new description of the application that is guaranteed to process the data at the real-time rates specified by the inputs, and can then be mapped to the target many-core processor.

In order to make this work, however, the compiler system must also analyze the data buffering required between the kernels, and insert appropriate buffer kernels to hold the intermediate data. The system we have developed uses two-dimensional data as its basic input type, to allow natural description of image and video processing. The compiler analyzes the data streams between kernels and inserts buffers of appropriate sizes at the input and output edges of the kernels.

9.1 Further Research

Our work has identified the following areas that require further research.

1. **Algorithms for combining kernels and collapsing buffers:** These algorithms are needed to improve throughput and locality, to reduce memory requirements, and to improve utilization.

2. **Techniques for automatically mapping kernels to resources**: These techniques are needed to select the modules that will implement different parts of a kernel. This would involve selecting the blocks to implement the computation and allocating memory blocks for storing code and intermediate data. This could involve identifying kernels that should be partitioned into smaller kernels with more homogenous operations; for example, the tools might recommend splitting a kernel into an arithmetic component and a bit-manipulation component so that the former could be implemented in an S-Block and the latter in an L-Block.

3. **Methods for integrating the mapping phase into the programming system.** This is needed to integrate the mapping phase, in which blocks are selected and detailed performance measures can be estimated, into a larger programming system in which various (software) components are described in other ways.

4. **Extending the programming system to support a component library.** Significant productivity gains will be achieved if a library of common kernels, subsystems composed of multiple kernels and streams, and so forth, are available to system implementers. The programming system needs a way of describing the modules that are available in the library. The modules in the library would need to be parameterized; for example, the library would need to provide FIR filters of different lengths. The library could provide information instructing the programming system on how to effectively partition modules after parameter values are bound and real-time performance requirements are specified. This would allow the user of a library module to benefit from the efforts and detailed knowledge of the module designer.

10 Next Steps

This study has clearly established the feasibility of building a structured ASIC component and its programming system. However much work remains before such a component is available for use in defense systems. This section identifies the technology gaps and sketches a research program to fill them.

The largest gap exists with the programming system and libraries. Considerable work is required to mature the prototype programming system to the point where it can be used by average programmers. Also, a library of common modules is needed to realize many of the advantages of structured ASICs.

10.1 Gap Analysis

In this section, we assess the technology state of each component of a structured ASIC and identify gaps that need to be filled to make a structured ASIC component a reality. The components we consider are the modules (P, S, L, M, and EI) of a structured ASIC component, the on-chip interconnect of the component, the programming system for each module and for the component, and libraries.

- Processor Module: Capable microprocessors along with compilers, debuggers, and ports of popular operating systems (e.g., Linux, Nucleus, etc…) are available from vendors such as MIPS and ARM. Such components can directly meet the needs of the Structured ASIC processor module. There is no gap in this area.

- Signal Processing Module: While our signal processing block study has sketched one possible architecture for the block and established the feasibility of building a programmable block that has near ASIC efficiency, this technology is far from being ready for deployment. Specific areas that require further research include:
    - Programming tools for efficiently mapping signal processing algorithms to one or more S modules. This includes tools for partitioning applications across multiple modules, tools for compiling kernels to individual modules, and debugging tools. Applications that have dynamic load characteristics while demanding real-time performance are expected to be particularly challenging.
    - Architecture studies to identify the optimal mix of instructions, function units, and machine organization for an S module. This work should further improve the efficiency of an S module.
    - Circuit and layout studies to determine if optimized circuits and layout can close the remaining gap with ASIC efficiency.
    - Development of a prototype S module incorporating the findings of the above studies.
    - Demonstration of a number of kernels on a prototype S module.
    - Demonstration of a full-scale DoD application on a system including S modules.
    - Development of libraries for common signal processing functions to facilitate rapid development of new applications on a system including S modules.

- Logic module: The logic module builds on mature FPGA and ASIC technology and can leverage the standard logic synthesis tools developed for these targets. Hence little additional work is needed to make this module ready for deployment. The only issues that remain to be resolved are :
    - Integration of the valid/ready interfaces with the logic.
    - Developing the means to configure (and dynamically reconfigure) the logic of a configurable L module.
    - Developing a library of common logic functions (e.g., convolutional coders and decoders, scramblers and descramblers, encryption functions, etc…) to facilitate rapid development of new applications using L modules.
    - Determining the appropriate size and degree of configurability for an L-module.

  - Determining how many flip-flops and what (if any) embedded storage to include in an L-module.

- Memory module: Like the L-module, the memory module leverages mature memory technology. The only remaining issues here are:
  - Defining the details of the address generation portion of an M module to support common data structures. It is likely that this will need to be programmable with a small amount of simple microcode – which implies the need for a programming system.
  - Integrating the valid/ready interfaces with the M module.

- Configurable interconnect: The configurable interconnect needs further study to refine both its architecture and its implementation. The interconnect provides dynamically configurable FIFO channels between the Structured ASIC modules.
  - Analysis of whole applications is needed to determine the number of channels, the topology of interconnection, and the degree of configurability required.
  - Alternative implementation techniques for implementing channels, buffers, and configuration need to be investigated.

- Routed interconnect: The routed interconnect is an instance of an on-chip interconnection network with FIFO channel inputs and outputs. This is a nascent technology and requires exploration of:
  - Topologies that give the best performance per unit cost and energy on workloads of typical structured ASICs.
  - Routing algorithms
  - Flow control methods
  - Circuits for implementing channels, buffers, switches, and allocators.

- External interfaces: A number of defense applications need to be surveyed to determine the requirements for the external interface modules. Specific questions that need to be answered include:
  - What set of I/O types are required for typical defense applications (LVCMOS2.5, LVCMOS3.3, LVDS, PCIe, USB, etc…).
  - What external converters are readily available to support these I/O types.
  - Are analog input and outputs required? If so, what precision and sampling rates are required of the A/D D/A modules? Are external A/D and D/A modules a viable alternative.

- Overall programming system: **The programming system represents the largest gap**. No comparable system is available today that permits the functions of an ASIC to be defined at a high-level and directly implemented without first lowering the defintion to RTL. Specific areas that need work include:
  - Investigation of alternative high-level notations for describing the overall application – as a connection of modules. Is a simple text-based language adequate or is a graphical description, or some hybrid, more effective.
  - Compilation tools to map the high-level description to the structured ASIC component. Note that this may include combining or splitting modules if needed to meet real-time constraints. This task becomes particularly challenging for dynamically changing applications where the mapping may need to change over time to handle a changing workload.
  - Investigation of programming models and languages for the S-module. Can Matlab m-code or Simulink models be used directly. Is C-code more appropriate?
  - Compilation tools to generate code for the S-modules. There are novel problems here in code generation for architectures with exposed communication.

- Libraries: The utility of the Structured ASIC concept depends to a large extent on the availability of libraries that cover the needs of key users developing radars, sonars, communications, electo-optical systems, etc… While some library development can be done on demand, without a critical mass of library modules, it will be difficult for developers to be productive. The key issues here are (1) identifying required library modules (and reference applications) for key application areas, and (2) developing these libraries. It is critical that these libraries be developed in a parameterized way, so they can be reused in different situations rather than being over-specialized for a single application.

10.2 Application Studies

Due to limited time and resources, the present study has focused largely on kernels. While these kernel studies have largely established the feasibility of the concept, there are issues that can only be explored in the context of a full application. Module interconnection issues, for example, are an application-level, not a kernel-level, issue.

To address these application-level issues and to provide benchmarks and data to drive other studies, one or more full-scale applications should be mapped to a straw-man structured ASIC architecture. Candidates for target applications include a JTRS modem, a SAR system, and an MTI radar.

As a part of the application studies, libraries for each of the selected application areas should be constructed. The library construction can in part be *on demand* – driven by the specific needs of the application at hand. However, the libraries should also be *filled out* to include common functions required by the application area even if they are not needed by the specific application being implemented. It is important that a fairly complete library be developed early on to provide key benchmarks for prototype development.

10.3 Prototype Studies

As the gaps described in above are filled by conducting concept studies, a structured ASIC prototype should be constructed to identify any remaining issues and to mature the technology to the level where it can be deployed.

We recommend that this prototyping take place in three steps: functional simulation, FPGA emulation, and silicon implementation. First, a detailed architecture and design for the prototype structured ASIC should be developed and captured in a functional simulator. This functional simulator is a cycle-accurate executable specification of the structured ASIC implemented in a high-level programming language (e.g., C or Java). Application mapping should be conducted at the functional simulation level to identify issues with the architecture and design, and the design updated as a result of the mapping before proceeding to the emulation step.

At the emulation step, the structured ASIC component is implemented on a conventional FPGA (e.g., a Xilinx Virtex-5). Due to the inefficiency of emulation, the size of the Structured ASIC component that can be implemented in this manner is limited and it will not operate at full frequency. However, the emulation will identify any issues involved in lowering the design from the functional simulation level to the RTL level and will permit applications to be run at near real time. The design should be updated based on the findings from emulation before proceeding to the silicon prototype.

The silicon prototype is an actual structured ASIC component. To minimize development time and cost it may be implemented entirely using a conventional *standard-cell* methodology. This will sacrifice some performance and efficiency. The silicon prototype will serve as a technology demonstrator for Structured ASICs. It could even be deployed in certain applications. The intent is that commercial vendors would develop their own Structured ASIC components modeled after the silicon prototype but using more efficient implementation styles.

10.4 A Timeline

The Gantt chart below shows a 5-year program to conduct the research necessary to close the gaps described above, develop a prototype structured ASIC component, and evaluate this component as a technology demonstrator. Activities related to the programming system and libraries, the S-module, and the interconnect all start in Q1 of Year 1 as these are the critical-path activities. Work on the L-module and M-module starts as needed to intersect with a functional simulator operational in Y3Q2 and a demonstration on an FPGA in Y4Q2. Application and library development is ongoing but with releases timed to correspond with the evaluation on the functional simulator, FPGA prototype, and silicon prototype.

Task		Year 1				Year 2				Year 3				Year 4				Year 5			
		Q 1	Q 2	Q 3	Q 4	Q 1	Q 2	Q 3	Q 4	Q 1	Q 2	Q 3	Q 4	Q 1	Q 2	Q 3	Q 4	Q 1	Q 2	Q 3	Q 4
Programming System																					
	System-level concept	X	X	X	X																
	System-level prototype					X	X	X	X	X	X										
	System-level eval											X	X	X	X						
	S-module concept	X	X	X	X																
	S-module prototype					X	X	X	X	X	X										
	S-module eval											X	X	X	X						
Libraries																					
	Identify key modules	X	X	X	X																
	Implementation			X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
S-module																					
	Architecture studies	X	X	X	X	X	X														
	Circuit/layout studies				X	X	X	X	X	X											
	Prototype S-module							X	X	X	X	X	X								
L-module																					
	Architecture studies			X	X	X	X														
	Circuit/layout studies						X	X	X	X											
	Prototype L-module									X	X	X	X								
M-module																					
	Architecture studies			X	X	X	X														
	Circuit/layout studies						X	X	X	X											
	Prototype M-module									X	X	X	X								
Interconnect																					
	Architecture studies	X	X	X	X	X	X														
	Circuit/layout studies				X	X	X	X	X	X											
	Prototype interconnect								X	X	X	X	X								
External Interfaces																					
	Study requirements							X	X	X	X										
	Prototype interfaces											X	X								
Applications																					
	Select and Specify Apps		X	X	X	X															
	Code and Debug						X	X	X	X	X	X	X	X	X	X	X	X	X	X	
	Evaluate on Prototypes										X				X	X					X
Prototypes																					
	Functional simulation							X	X	X	X										
	RTL Implementation									X	X	X	X								
	FPGA prototype												X	X							
	FPGA evaluation														X	X					
	RTL for Si Prototype												X	X	X	X					
	Layout for Si Prototype														X	X	X	X			
	Fabricate Si Prototype																		X	X	
	Evaluate Si Prototype																				X

Table 2: Project Gantt Chart

11 References

[1] Balfour, Dally, Black-Schaffer, Parikh, and Park, “An Energy-Efficient Processor Architecture for Embedded Systems,” *IEEE Computer Architecture Letters,* Vol. 7, No. 1, Jan 2008, pp. 29-32.

[2] Kim, Balfour, and Dally, “Flattened Butterfly Topology for On-Chip Networks,” *IEEE/ACM International Symposium on Microarchitecture*, 2007, pp. 172-182.

List of Acronyms, Abbreviations, and Symbols

ALU	arithmetic and logic unit
CAD	computer-aided design
ASIC	application-specific integrated circuit
DDR	double data rate
DMA	direct memory access
DRAM	dynamic random access memory
EI	external interface
FGPA	filed programmable gate array
IR	instruction register
LUT	lookup table
ORF	operand register file
OS	operating system
ROM	read-only memory
RTL	register transfer language
SIMD	single instruction multiple data
SRAM	static random access memory
SRN	statically routed network
VHDL	VHSIC hardware description language