2009

Application Specific Customization and Scalability of Soft Multiprocessors

Deepak C. Unnikrishnan
University of Massachusetts Amherst

APPLICATION-SPECIFIC CUSTOMIZATION AND SCALABILITY OF SOFT MULTIPROCESSORS

A Thesis Presented

by

DEEPAK C. UNNIKRISHNAN

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE

May 2009

ELECTRICAL AND COMPUTER ENGINEERING

APPLICATION-SPECIFIC CUSTOMIZATION AND SCALABILITY OF SOFT MULTIPROCESSORS

A Thesis Presented

by

DEEPAK C. UNNIKRISHNAN

Approved as to style and content by:

____

Russell G. Tessier, Chair

____

C. Mani Krishna, Member

____

Paul Siqueira, Member

____

C. V. Hollot, Department Head
Electrical and Computer Engineering

ACKNOWLEDGMENTS

I would first like to thank Altera Corporation and the National Science Foundation for funding this project and allowing me to work on it. I would like to thank Professor Russell Tessier for his motivation and guidance. I am also grateful to Professor Mani Krishna and Professor Paul Siqueira for their time and for being on my thesis committee. I would like to extend my gratitude to Jia Zhao and Ramakrishna Vadlamani of the Reconfigurable Computing Group, UMass, for their valuable suggestions on the project. Finally, I would like to thank my friends and family who have assisted me and who have supported me throughout my time here at UMass.

ABSTRACT

APPLICATION-SPECIFIC CUSTOMIZATION AND SCALABILITY OF SOFT MULTIPROCESSORS

MAY 2009

DEEPAK C. UNNIKRISHNAN

B.TECH E.C.E (Hons.), UNIVERSITY OF CALICUT, INDIA

M.S. E.C.E., UNIVERSITY OF MASSACHUSETTS, AMHERST

Directed by: Professor Russell G. Tessier

Soft multiprocessor systems exploit the plentiful computational resources available in field programmable devices. By virtue of their adaptability and ability to support coarse grained parallelism, they serve as excellent platforms for rapid prototyping and design space exploration of embedded multiprocessor applications. As complex applications emerge, careful mapping, processor and interconnect customization are critical to the overall performance of the multiprocessor system. In this thesis, we have developed an automated scalable framework to efficiently map applications written in a high-level programmer-friendly language to customizable soft-cores. The framework allows the user to specify the application in a high-level language called Streamit. After an initial analysis of the application, a soft multiprocessor system is generated automatically using a set of customizable SPREE processors which communicate with each other over point-to-point FIFO connections. Several micro-architectural features of the processors are then automatically customized on a per-application basis to improve system area, performance and power consumption. The efficiency and scalability of this approach has been validated using a diverse set of eight audio, video and signal processing benchmarks on soft multiprocessor systems consisting of one to sixteen processors. Results show that generated soft multiprocessor systems consisting of sixteen processors can offer up to 6x speedup over a conventional single processor system. Our experiments with soft multiprocessor interconnection networks show that point-to-point topologies perform approximately 2x better than mesh topologies. Finally, we demonstrate that application-specific customizations on the instruction set, memory size, and inter-processor buffer size can improve the area and performance of the generated soft multiprocessor systems. The developed framework facilitates rapid design space exploration of soft multiprocessors.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

With technology scaling, increased field-programmable gate array (FPGA) area and logic resources have enabled designers to integrate more hardware resources into the FPGA fabric. In particular, there has been considerable effort to integrate microprocessors and FPGAs. The first efforts in this direction began during the late 1990s when designers integrated microprocessors built using transistors called *hard cores* with the FPGA fabric. Leading vendors such as Altera and Xilinx have developed Excalibur [34] and Virtex II Pro [35] devices respectively incorporating hard cores and FPGA fabric on a single chip. Altera Excalibur devices integrate an ARM9 processor with a 1 million gate FPGA fabric while Xilinx Virtex II Pro devices incorporate two or more PowerPC processors with a 10 million gate FPGA fabric. However, in many cases, the fixed number of hard processors available on the chip does not match the application requirements. Hard processors impose severe routing constraints on the placement of custom logic on the FPGA fabric.

A soft processor is a microprocessor embedded into the FPGA fabric. Unlike hard processors, soft processors offer considerable flexibility to match the requirements of the application. For example, the number of soft processors in an FPGA can be varied to match the computational requirements of the application. Since soft processors are embedded into the FPGA fabric, placement and routing decisions are largely taken by automated computer-aided design (CAD) tools. The customizability of individual soft processors makes them attractive for resource-limited applications. Leading FPGA vendors, such as Altera and Xilinx, already offer 32-bit RISC soft processor IP blocks called Nios [36] and MicroBlaze [37], respectively. Soft processors are integral components of most system-on-a-programmable chip solutions available today.

The significant increase in FPGA resources has spurred interest in embedding multiple soft processors on the same FPGA substrate. Multiple soft processors integrated on a single FPGA device can serve as a flexible programming platform for fast application mapping without the need for intensive register transfer level (RTL) design. Soft multiprocessor systems also exhibit high degrees of task level parallelism which can be exploited to efficiently execute complex data processing applications. Typical applications involving these systems vary from initial hardware prototyping to final product designs for embedded multiprocessor systems.

It is projected that the amount of logic and memory resources in FPGAs is likely to grow substantially in the near future to support hundreds of soft processors. However, three major challenges constraining the widespread use of soft multiprocessors are their complex design, programmability and system-wide energy consumption. In this context, an automated and efficient mapping of applications written in a programmer-friendly high-level language to FPGAs is highly desirable. Unlike commercial off-the-shelf soft processor IP blocks which offer limited customizability, custom-generated soft processors can be better tailored to suit the requirements of the application. Hence, there is scope for large scale system-wide application specific optimizations to improve performance and minimize energy consumption.

The process of application parallelization across multiple processors is a well established research area. However, given the limited amount of logic, as compared to memory, available in an FPGA, application mapping to soft multiprocessors presents a number of interesting new challenges. These include the implementation of several critical processor features such as caches, large memories and routing tables, among others.

Previous work on soft multiprocessor systems has focused on the development of automated synthesis tools for smaller multiprocessor systems and the investigation of the performance of interconnection topologies. Although the potential of soft multiprocessor systems has been demonstrated in previous approaches, the primary focus has remained on relatively small multiprocessor systems targeting single or a small number of benchmarks. The primary contribution of this thesis is a comprehensive evaluation of the combined impact of soft multiprocessor synthesis, topology choices and scalability using a substantial collection of multiprocessor benchmarks on soft multiprocessor systems consisting of a large number of processors. Specific research objectives and challenges of the work include:

1. Development of a comprehensive evaluation platform for large soft multiprocessor systems by integrating high-level application compilers with synthesizable soft processor generators.
2. Modification of high-level application mapping tools to support FPGA aware task allocation and mapping.

3. Investigation of the impact of individual processor and interconnect optimizations on the overall performance of soft multiprocessor systems.
4. Evaluation of a large set of existing multiprocessor benchmarks available in the parallel computing community on soft multiprocessor systems.

The rest of the thesis is organized as follows: Chapter 2 provides insight into previous work. This includes a discussion of existing approaches that map applications onto soft multiprocessors and soft processor optimization techniques. Chapter 3 elaborates on the components of the proposed framework. Chapter 4 describes the design flow. Chapter 5 explains the evaluation strategies and results. Chapter 6 summarizes the thesis and gives directions for future work.

CHAPTER 2

BACKGROUND AND PREVIOUS WORK

This thesis work builds on previous research in single and multi soft processor design and implementation. Earlier work has primarily focused on area, power and performance evaluation of smaller soft multiprocessor systems in isolation. The following sections will survey some of the existing approaches to automatic synthesis, architectural optimizations and evaluation of soft multiprocessor systems.

2.1 Soft Multiprocessor Synthesis

A number of recent research papers have examined application mapping from high-level data flow graphs to multiple soft processors. Yujia et al. [1] and Ravindran et al. [2] have illustrated the feasibility of using soft multiprocessors for a high performance IPv4 packet forwarding application. In this study, a framework to determine the best multiprocessor configuration for the data plane implementation of an IPv4 packet forwarding application using integer linear programming techniques is considered. Initially, the IPv4 application is represented as a data flow graph. The data flow graph is partitioned into an array of Xilinx Microblaze [37] soft processors. The number of partitions may be reduced by manually clustering multiple application tasks together. Once all application partitions have been extracted from the data flow graph, integer linear programming (ILP) techniques are applied to derive the best architecture for each partitioning. The inputs to the ILP solver include hardware constraints and worst case task execution times. The objective is to maximize the overall throughput under the given system constraints. Figure 1 illustrates the final multiprocessor design after ILP based automated exploration.

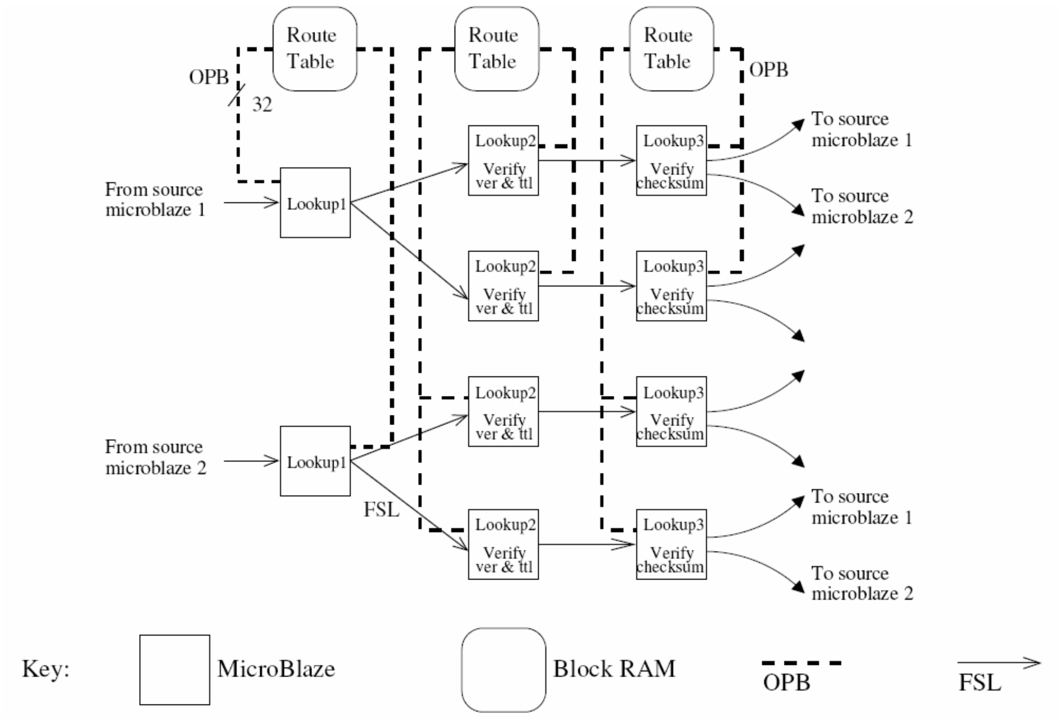


Figure 1: Multiprocessor design for IPv4 after automated exploration [1]

Although the described approach achieves better performance over hand-tuned designs, integer linear programming techniques are generally considered to be slow and may not scale well over larger problem sizes. Note that in this approach, an efficient partitioning requires careful manual clustering of tasks by the designer. The described methodology is also tuned for a single application.

A clustering and packing approach for soft multiprocessor synthesis targeted at an MJPEG application is described by Cong et al. [3]. The mapping consists of assignment of tasks to a number of soft processors interconnected by point-to-point FIFOs. The approach is targeted at throughput-constrained stream-oriented multimedia and network applications. The work is unique in that it takes latency, throughput and resources simultaneously into consideration during design space exploration. The application is initially represented as a synchronous data flow graph. The objective is to reduce the latency and improve throughput under constraints of communication costs and task execution times. To achieve this, a combination of labeling, clustering and packing algorithms are applied on the given task graph. Experiments using an MJPEG encoder application have produced multiprocessor configurations with high throughputs and significant reduction in design time compared to ILP approaches. However, the described approach only takes a single benchmark into consideration and can only be used for homogeneous processor systems consisting of a small number of processors. Further, the lack of processor optimizations after initial task mapping and resource allocation makes this scheme unattractive.

A methodology for automated multiprocessor system design, programming and implementation from a high-level system specification using static affine nested loop Pprograms (SANLP) is described in [4]. First, a Kahn process network (KPN) specification is derived from the application description. The derived KPN specification is given as input to the embedded system-level platform synthesis and application mapping (ESPAM) tool, as shown in Figure 2. The tool generates multiprocessor systems connected by point-to-point FIFO links from a predetermined set of IP blocks.

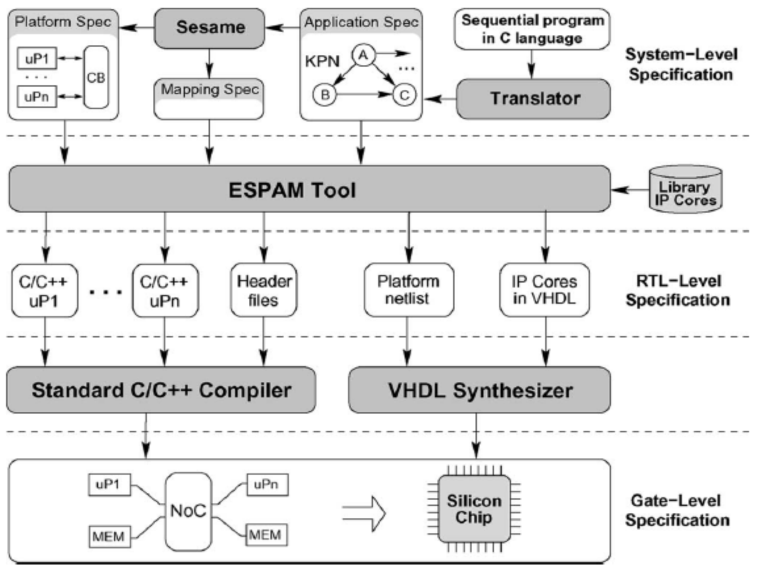


Figure 2: ESPAM Application Mapping Flow [4]

However, the proposed implementation is time consuming and selection of components from a standard IP library rules out any possibility of individual component optimization. Complex communication controllers are used as glue logic to interface standard components. Implementation of communication components using dual port memories is expensive in FPGAs. Also, the approach has been applied to relatively small multiprocessor systems with a restricted set of three applications.

2.2 FPGA-Based Networks-On-Chip

On-chip interconnects for integrating multiple soft processors have been examined in a series of recent studies. Saldana et al. [5] have examined the routability of several common network topologies as shown in Figure 3 to interconnect soft processors on FPGAs. This approach uses automated network topology generation from high-level specifications to generate multiprocessor systems consisting of up to 64 nodes. An important conclusion noted in the work is that modern FPGA fabrics are rich in resources and are capable of supporting highly-interconnected topologies such as direct point-to-point links. Like other previous approaches, this study is not comprehensive since automated approaches are applied only for interconnect topology generation.

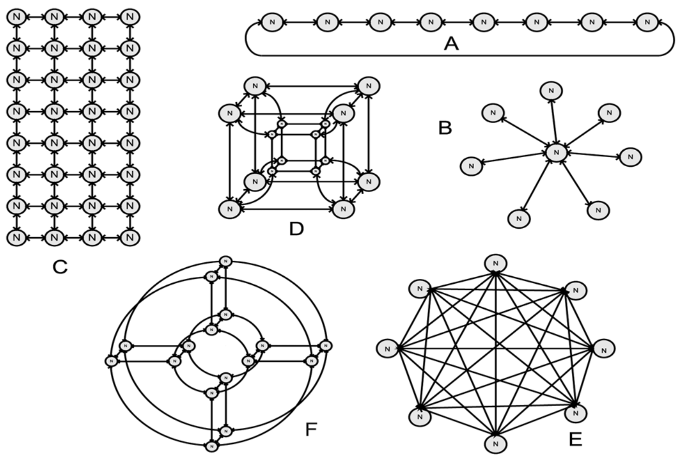


Figure 3: Topologies - A-Ring, B-Star, C-Mesh, D-Hypercube, E-Fully connected, F-Torus [5]

Studies [6] [7] have shown that NoCs can significantly outperform on-chip buses and thus provide system scalability. Kapre et al. [8] observed that time-switched and packet-switched butterfly fat trees can be efficiently mapped to FPGAs.

Several studies have examined the behavior of common parallel processing applications such as sorting networks on soft multiprocessor systems developed from commercial soft-core IP blocks. For example, Derutin et al. [9] evaluated the performance of a homogeneous soft multiprocessor architecture using a hypercube topology. A parallelized quicksort algorithm is used for the evaluation of multiprocessor systems consisting of 2, 4, 8 and 16 processors. However, like many other approaches, the processors used for the study consist of standard IP cores which are hardly customizable. The application parallelization was carried out manually which severely limits the scalability of this approach to larger multiprocessor systems and a wider set of benchmarks. A similar study described in [10] examines the performance of a parallelized merge sort application on a seven processor Xilinx Microblaze system. Each processing element is hooked to a router via the network interface. The routers are interconnected using a hypercube topology. A full adaptive minimal deadlock-free packet routing algorithm is used in the design.

In general, many of the approaches considered previously suffer from the following limitations. First, the applications are described in a non-user friendly fashion with constructs such as data flow graphs. The parallelization techniques considered previously use time-consuming and non-scalable approaches such as integer linear programming. Finally, the previous studies limit themselves to a restricted set of applications and soft multiprocessor system sizes.

2.3 Soft Processor Optimization

Soft processors have created a unique niche in the embedded design space with their ability to be customized to suit the requirements of the application. Recent studies on soft processor optimization have focused on area, performance and energy. It has been shown that application-specific customization has significant impact on the overall performance of the system. For example, Yiannacouras et al. [11] discuss the impact of microarchitectural customizations on automatically-generated synthesizable soft processors. In this work, a framework called Soft Processor Rapid Exploration Environment (SPREE) is developed. The framework can automatically generate customizable soft processor RTL descriptions from high-level textual descriptions of the ISA, data path and control path of the processor. The tool can be used to customize several aspects of a microarchitecture, such as the shifter implementation, pipeline depth, instruction set and forwarding logic. An overview of the SPREE infrastructure is shown in Figure 4.

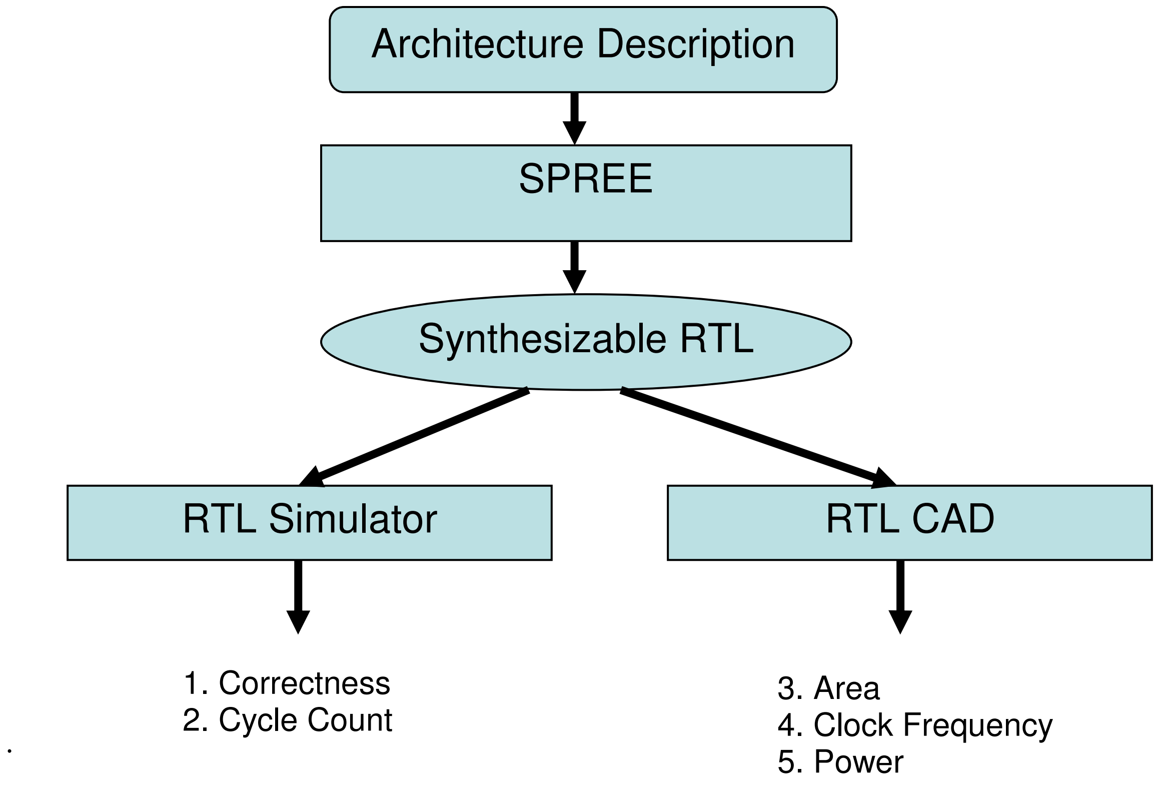


Figure 4: Overview of the SPREE System [11]

SPREE supports a library of basic components such as the register file, adder, sign-extender, fetch unit, etc. The user submits a high-level textual description of the data, control path, and the micro architectural features of the processor. The tool performs an integrity check on the submitted information to verify that the information can be used to generate a functional processor. Next, it instantiates the data path and control path of the processor according to the instruction set architecture description. It has been shown in this work that a tuned micro-architecture can offer up to a 30% improvement in performance and up to a 25% improvement in both area and energy.

A methodology to derive application specific embedded SIMD cores has been described in [12] by Hebert et al. In this work, a microcode analysis tool decodes the instructions in the same way as it is done in the processor into bit fields according to their encoding pattern. The decoded field values are fed into emulators which emulate the processor's controller. Results generated by the tool are used to optimize the original hardware model. Finally, the optimized model is given to the synthesis tool. The flow is summarized as shown in Figure 5.

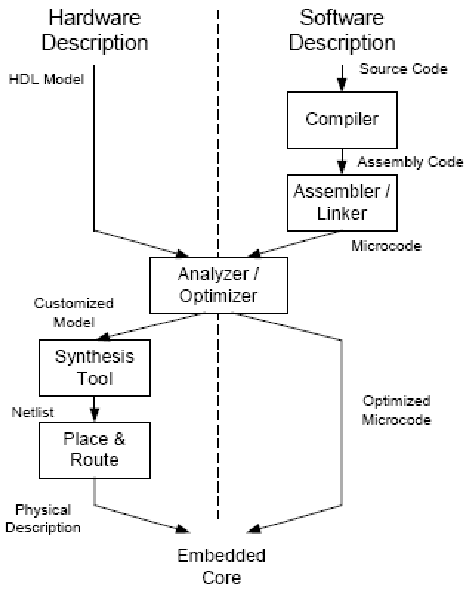


Figure 5: A methodology to derive application specific embedded cores[12]

Several application specific post-microcode analysis optimizations such as resource elimination, constant signal propagation, local constant tables, field recoding and data path width optimizations are applied on a template HDL model. This study has demonstrated large savings in lookup tables (ALUTs) for a single-instruction, multiple-data (SIMD) Pulse VI processor. However, the restricted focus on SIMD architectures and use of emulators to derive application-specific optimizations makes this scheme architectural specific and hence unattractive.

Researchers have considered multithreading to improve application performance and improve energy savings in soft processors. Dimond et al. [13] examines the use of multi-threading and custom instructions as techniques to maintain high throughput while minimizing processor area. In this approach, custom instructions are generated on a customizable multi-threaded processor (CUSTARD) by identifying frequently occurring segments of computation that can be evaluated using the same hardware datapath. The tool is illustrated in Figure 6. CUSTARD takes a set of inputs which include an application specified in a high-level language such as C, a template processor and a set of user defined processor parameters. Next, the compiler generates custom instructions to accelerate the application. The generated custom instructions are combined with designer parameters to instantiate a synthesizable netlist for the processor. The framework also supports hardware threads to improve performance since context switches in hardware threads take just a single cycle. The SRAM bits abundantly available in FPGAs can be used to implement hardware registers for each thread context.

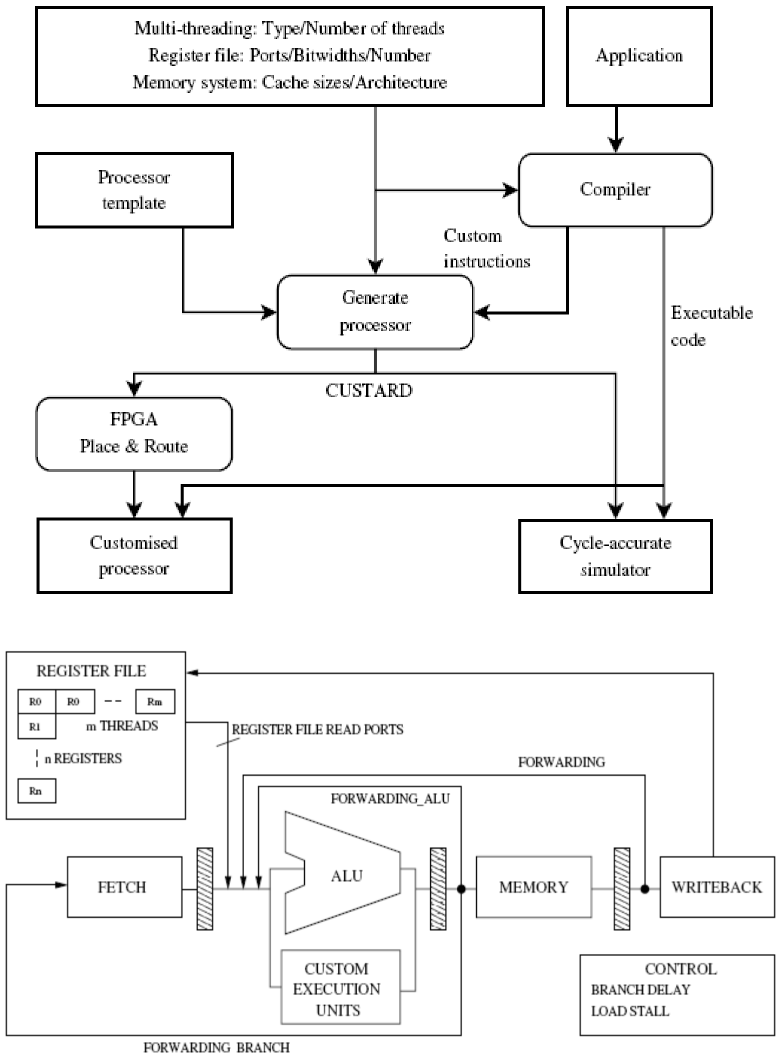


Figure 6: CUSTARD –Tool flow and microarchitecture [13]

Later work [14] involving soft processor synthesis has examined techniques such as instruction scheduling and recoding to improve energy savings. Instruction recoding is based on the principle that instructions with high frequency differ by only a few bits so that bit switching may be reduced. The switching frequencies of instructions are obtained from an execution profile of the application. Power-aware scheduling complements instruction scheduling. In this technique, tasks with low Hamming distances are scheduled closer to each other without affecting inter-task data dependencies. The work demonstrates a power saving of up to 74% obtained with six application benchmarks.

Fort et al. [15] use multithreading with custom functional units located outside the processor. The study shows that it is attractive to use a multithreaded processor in an FPGA environment because of significant area savings. Labrecque et al. [16] have extended the SPREE infrastructure [11] to support multithreaded soft processors. In this work, the authors show that that multithreaded soft processors are up to 106% more area efficient than non-multithreaded counterparts. Also, multithreaded processors are able to sustain high IPC when compared to their single threaded counterparts. It is noted that the key to improvement in the performance is a careful selection of ISA features, the number of registers, the number of threads and the intra-stage pipelining. A very important conclusion from this work is that off-chip memory latency is not a significant challenge for FPGA-based systems and a small on-chip memory is often sufficient to emulate an ideal cache.

2.4 Summary of Previous Approaches

In general, previous research on soft multiprocessors has focused on automatic synthesis systems, architectural optimizations and evaluation of interconnection topologies. However, many of these previous efforts primarily evaluated system area and performance and energy impacts in isolation without considering the underlying tradeoffs in system synthesis. Although the previous approaches provide initial analysis and emphasize the importance of automatic approaches towards soft multiprocessor design cycle times, conclusions regarding appropriate inter-processor topology and mapping effectiveness on a range of stream-based applications are not provided. The synthesis frameworks examined previously do not consider the impact of processor optimizations on large scale multiprocessor systems. None of the previous work on soft processor interconnect topologies considers a range of applications automatically mapped to a large number of soft processors.

Our work distinguishes itself from the previous approaches in the following ways:

1. Our work describes an automatic synthesis framework to assess the scalability of a large number of existing parallel computing applications on large soft multiprocessor systems.

2. The impact of a collection of architectural optimizations on soft multiprocessor systems are considered including:
    a. Interconnection network topology optimization such as tradeoffs between point-to-point and mesh-style interconnects.
    b. Unused instruction removal on individual soft processors based on the target application.
    c. Assessment of pipeline depth variation of individual soft processors on the performance of the multiprocessor system.
    d. Impact of tuning communication buffer sizes on the performance.
    e. Impact of tuning the memory size of individual processors.

3. Our work provides a system-level evaluation of stream applications on soft multiprocessor systems considering area, power and performance aspects.

CHAPTER 3

FRAMEWORK COMPONENTS

The proposed framework will be able to map applications written in a programmer-friendly high-level language to binaries that could be executed on customized soft processors. We target stream applications since they represent a large class of data-intensive applications most likely to dominate the embedded market in the near future.

The framework integrates compilers for high-level application mapping, profilers that extract application specific parameters and soft processor synthesis algorithms into a single automated design flow. This section describes the components of the proposed automatic synthesis framework.

3.1 Streamit – A compiler for stream-based applications

Streamit [18][20] is a high-level, architecture-independent language and compiler targeted at streaming applications. Streamit compiler maps stream programs to software-exposed architectures such as MIT RAW [19]. This section discusses the RAW microprocessor and explains how the Streamit compiler can be efficiently extended to support streaming applications on soft microprocessors.

The RAW computational fabric is a scalable, tiled architecture developed to exploit the copious logic resources in next generation CMOS processes. The RAW is a single chip multiprocessor consisting of sixteen identical programmable tiles. RAW has been fabricated using IBM's 180nm 1.8V 6-layer CMOS 7SF SA 27E copper process. The sixteen cores communicate with each other using 32-bit full duplex mesh networks. RAW supports a static and dynamic network. The entire communication is specified at compile time in the static network, while the dynamic network supports run time events. As illustrated in Figure 7 andFigure 8, RAW represents a regular multiprocessor architecture. Interconnect between the cores is pipelined to convert across-chip wire delays into network hops. The longest wire in the chip need not be more than the width of a tile. Hence, the propagation delay across a tile is just one cycle. The network and computational resources can be programmed using the RAW ISA. Thus, RAW exploits all forms of parallelism including instruction level, data level, thread level and stream parallelism.

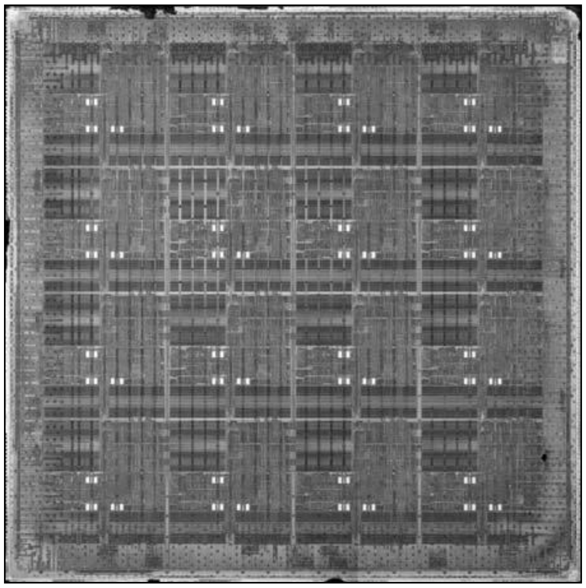


Figure 7: Die Photograph of the RAW Microprocessor [19]

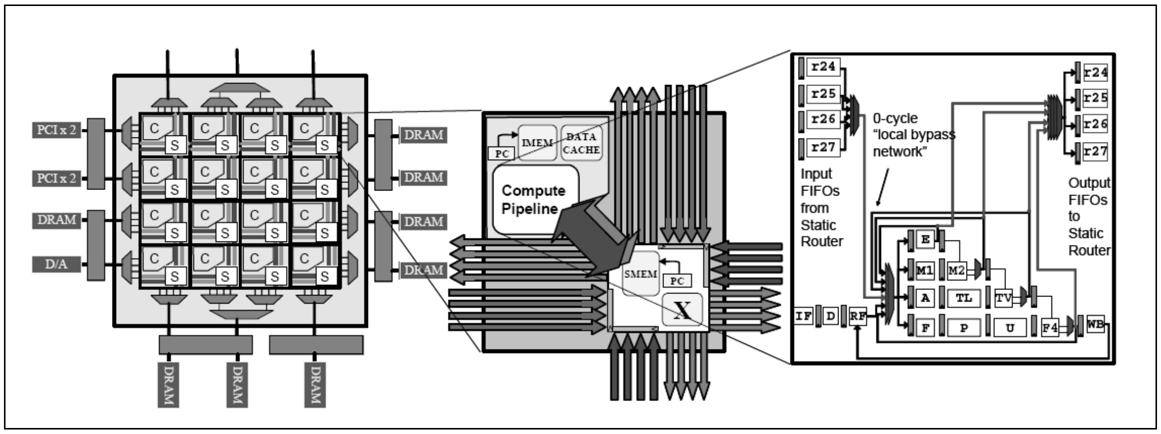


Figure 8: RAW fabric exposes on-chip interconnects to the software [19]

The RAW design philosophy favors regularity and simplicity. Each tile incorporates an 8-stage in-order MIPS style pipeline, 32KB instruction cache, 32KB data cache and a 4-stage single precision pipelined floating point unit. The on-chip networks are interfaced to tiles through bypassed, register-mapped static routers built into each tile. Each static router (switch processor) executes a basic instruction set that consists of routing instructions to forward the data between tiles. A neighboring inter-tile transfer takes 3 cycles while an inter-tile transfer involving N hops can be achieved in 2+N cycles [19]. The dynamic network support asynchronous events such as cache misses and interrupts.

3.1.1 Streamit Language Constructs

Streamit was initially developed as a language and compiler to exploit RAW's software exposed interconnects. The basic idea was to provide a portable programming model to communication-exposed architectures. The computation is modeled as a hierarchy of basic computational units called filters [18]. The filter can be imagined as a block of user-defined code which can process the streaming data. Each filter contains two parts – an init function and a work function. The init function is invoked during filter initialization while the work function models the steady state execution steps of the filter. Although each filter can be imagined to run on an individual tile, highly irregular filters can cause load balancing issues on the target architecture. To address this issue, Streamit supports fission/fusion operations to combine or split filters to match the granularity of the target architecture. A brief description of fission/fusion operations is given later in this section.

Each filter can communicate with other filters using push(), pop() and peek() methods. The push() method sends data into the output queue of the filter. A pop() method receives the data from an input queue of the filter. Peek() is a special operation that returns the value at an index in the input queue without removing the item.

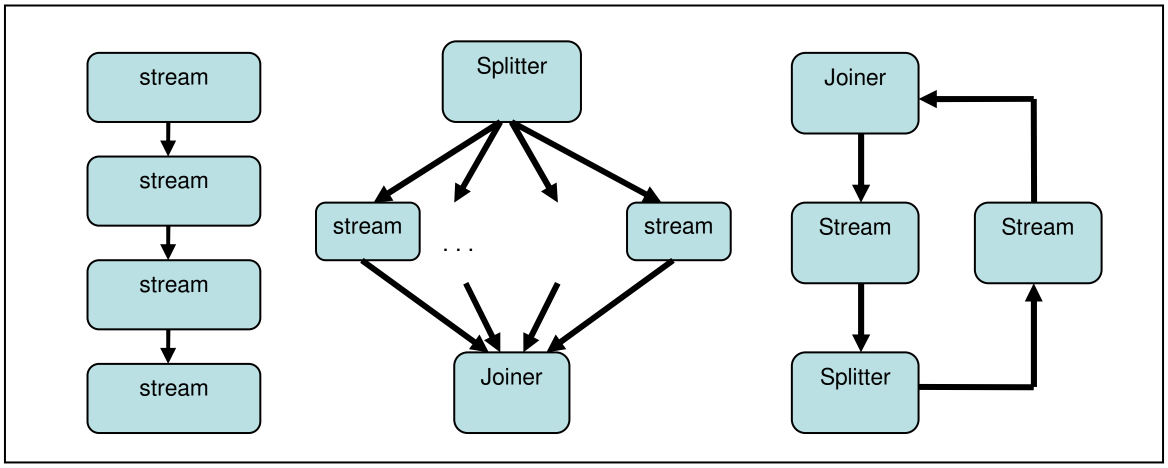
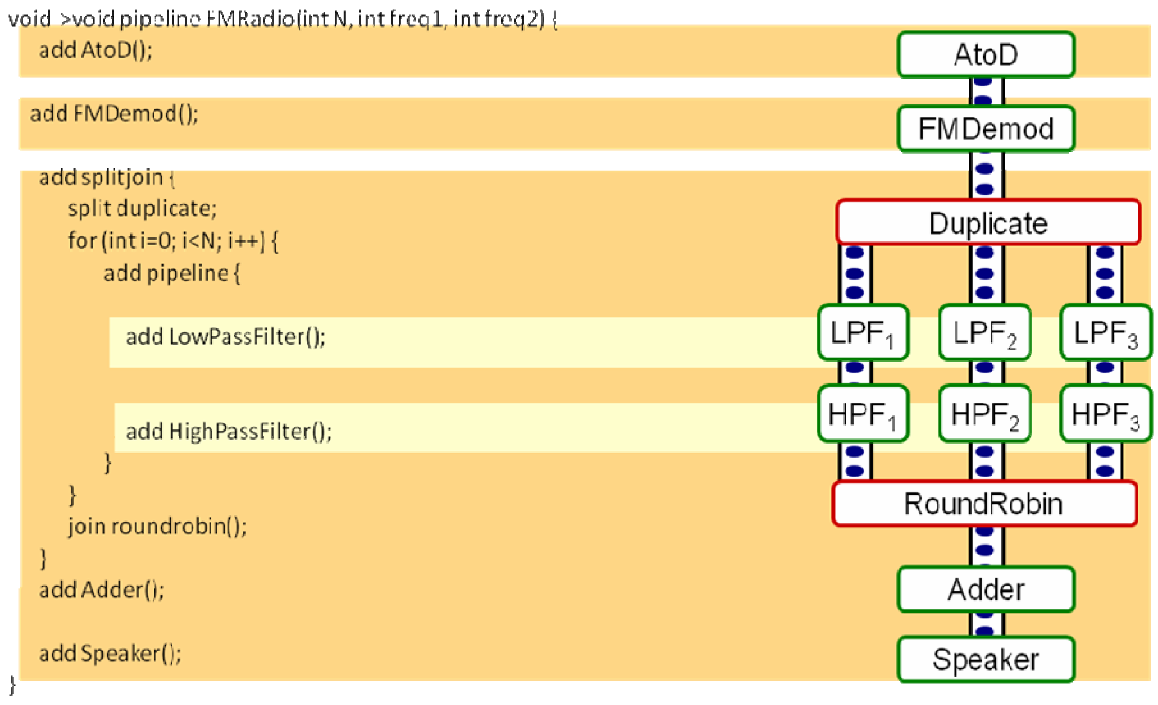


Figure 9: Stream structures supported by StreamIt [18]

A Streamit program is represented as a network of filters. The filters are interconnected by constructs such as pipelines, split-joins or feedback loops. The pipeline construct supports a sequential arrangement of the filters. The split-join specifies independent parallel data streams. Data is split into multiple streams at the splitter and later joined at the joiner. For example, a duplicate splitter sends a copy of each data item into each parallel stream. A round robin joiner roundrobin(n_1,n_2,…..n_m) sends the first n_1 items to the first stream, the next n_2 items to the next stream etc. The feedback loop construct supports cycles in a stream graph. Figure 9 illustrates the various hierarchical structures supported by the Streamit language. We illustrate Streamit with an example. Consider the representation of an FM Radio application as illustrated in Figure 10. The FM Radio consists of an analog to digital converter, FM demodulator, equalizer and a speaker. The equalizer can be thought of as a logical block composed of many low pass and high passes filters operating in parallel. The equivalent Streamit program for an FM Radio consists of a pipeline of filters that represent an A-D converter, FM demodulator, equalizer and speaker. The equalizer may be viewed as a component that consists of multiple band pass filters. Each band pass filter can be viewed as a hierarchical pipeline of low pass and high pass filters whose critical frequencies are set according to the characteristics of the filter. Since the components of equalizer can operate on the demodulated stream of data independently, the incoming stream is duplicated using a duplicate construct and later joined using a round robin joiner. Finally, the adder and speaker process the joined data stream to reconstruct the audio signal. It is interesting to note that the Streamit program imposes a well-defined structure on all the streams that exposes stream level parallelism in natural way. From a programmer perspective, this structure helps to incorporate the parallelism inherent in the application naturally into the program. From a compiler perspective, the well-defined structure of Streamit programs makes them easier to analyze than arbitrary graphs.



```
void >void pipeline FMRadio(int N, int freq1, int freq2) {
  add AtoD();

  add FMDemod();

  add splitjoin {
    split duplicate;
    for (int i=0; i<N; i++) {
        add pipeline {

          add LowPassFilter();

          add HighPassFilter();
        }
    }
    join roundrobin();
  }
  add Adder();

  add Speaker();
}
```

Figure 10: FM Radio–Streamit progam and the equivalent stream graph [18]

3.1.2 Streamit Compiler

The Streamit compiler consists of eight phases as shown in Figure 11. The front end is built on top of a Java based open source compiler infrastructure called KOPI. The front-end parses the Streamit syntax into a Java-like abstract syntax tree (AST). SIR conversion phase transforms the AST into a Streamit intermediate representation (SIR). Various structures in the stream graph are expanded during the graph expansion phase. Scheduling calculates the initial and steady state data transfer rates for each filter. The scheduler calculates two types of schedules – a non-repetitive initialization schedule and a repetitive steady state schedule.

The partitioning phase performs load balancing operations using fission/fusion transformations on the stream graph. The basic idea is that the compiler initially estimates the number of instructions executed by each filter in a single steady-state execution cycle of the program. Then, computationally intensive filters are split and less demanding filters are fused together. Vertical fusion algorithms combine multiple filters in a pipeline together to create a single filter while horizontal fusion combines parallel filters together. Vertical fission algorithms split a single filter into a series of parallel filters. Horizontal fission algorithms split a single filter into multiple pipelined components. Fission/fusion transformations are performed by simulating steady state execution schedules of the stream graph on the individual filters. More details on fission/fusion optimizations can be found in [18] and [20].

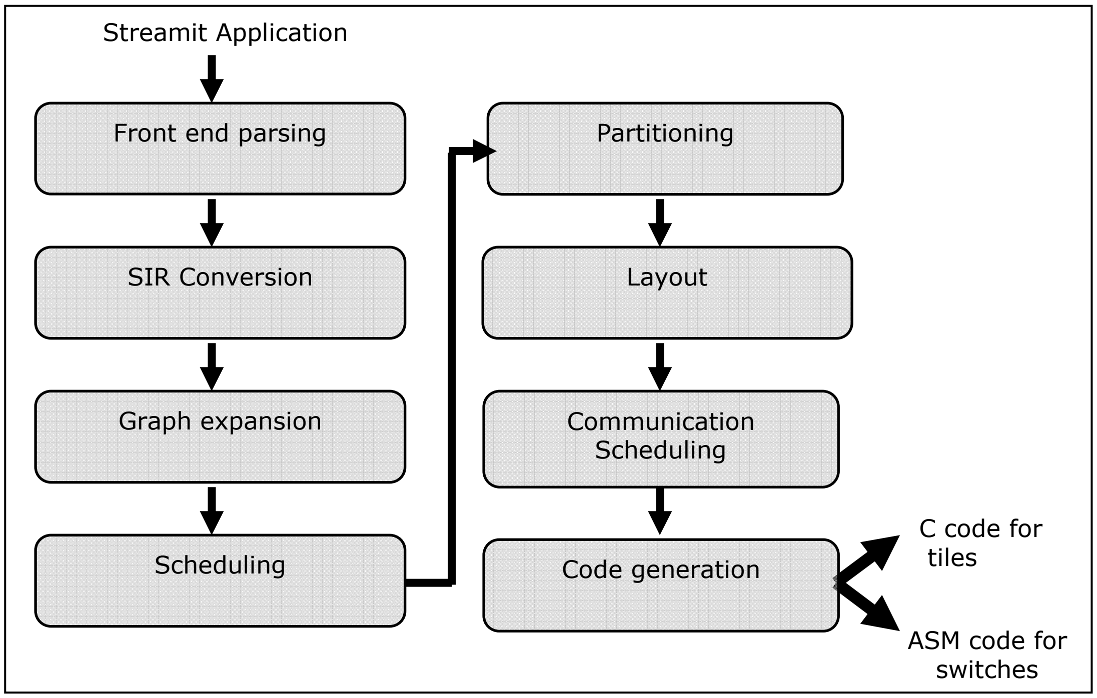


Figure 11: Streamit Compiler Phases [18]

The Streamit compiler currently supports three kinds of partitioning algorithms – based on a greedy algorithm, a greedier algorithm and dynamic programming. In the greedy approach, filters are first sorted in the descending order of computational requirements.

Then, a simple greedy algorithm is used to split heavy filters into smaller ones. The process is iterated until the heaviest filter can no longer be split or when the number of filters matches the granularity of the target architecture. If there are more filters than the number of processing elements, a similar greedy algorithm can be applied to combine multiple filters into coarser filters.

Layout refers to the assignment of partitioned filters into the processors in the target architecture so that the communication and synchronization costs are minimized. Streamit uses the simulated annealing algorithm for layout. Once nodes of the stream graph are assigned to the nodes of the target platform, the communication scheduler simulates the execution of nodes in the stream graph and records the communication patterns during simulation. These communication patterns are translated into routing instructions that are executed on each switch processor.

Finally, the code generation phase generates C code for each tile and switch instructions for each switch processor. The tile code contains translation of the filter functionality including statements to transfer data into or outside the filter. The communication schedule describes the static ordering of data to be sent or received. The schedule has an initialization part which runs exactly once and a steady state part that loops indefinitely.

3.2 Automatic Soft Processor Generation

The design space exploration of scalable soft processor systems requires automated approaches to generate the multiprocessor system. Automated approaches are required to customize each processor according to the application segment executing on it. Previous approaches to generate and customize single soft processor systems automatically have been analyzed in [11]. We extend the Soft Processor Rapid Exploration Environment (SPREE) described in [11] to support the automatic generation of a large number of customized multiprocessors. The SPREE framework generates synthesizable RTL descriptions of processors from high-level descriptions of the micro-architectural features such as the data path, control path and instruction set architecture. In this framework, the user specifies the processor as an interconnection of basic micro-architectural features, such as adders, instruction fetch units, and register files. Next, a set of scripts verify the validity of the specified description and generates a datapath description of the processor using a library of hand-coded basic components. Finally, the tool generates the control path logic necessary to coordinate the elements in the data path. An overview of the SPREE infrastructure is shown in Figure 12.

We generate our soft processors from the processor templates produced by SPREE. Although SPREE serves as a good tool to generate the basic components of our soft multiprocessor systems, the tool has some limitations. For example, SPREE considers only simple in-order issue processors with on-chip memories. This is not a serious limitation for our current evaluation since memory requirements for most of our benchmarks can be easily satisfied with the existing memory bits available in commercial FPGAs. In Chapter 5, we show that memory requirements of the application more or less remain the same or decrease slightly when the application is mapped over large multiprocessor systems.

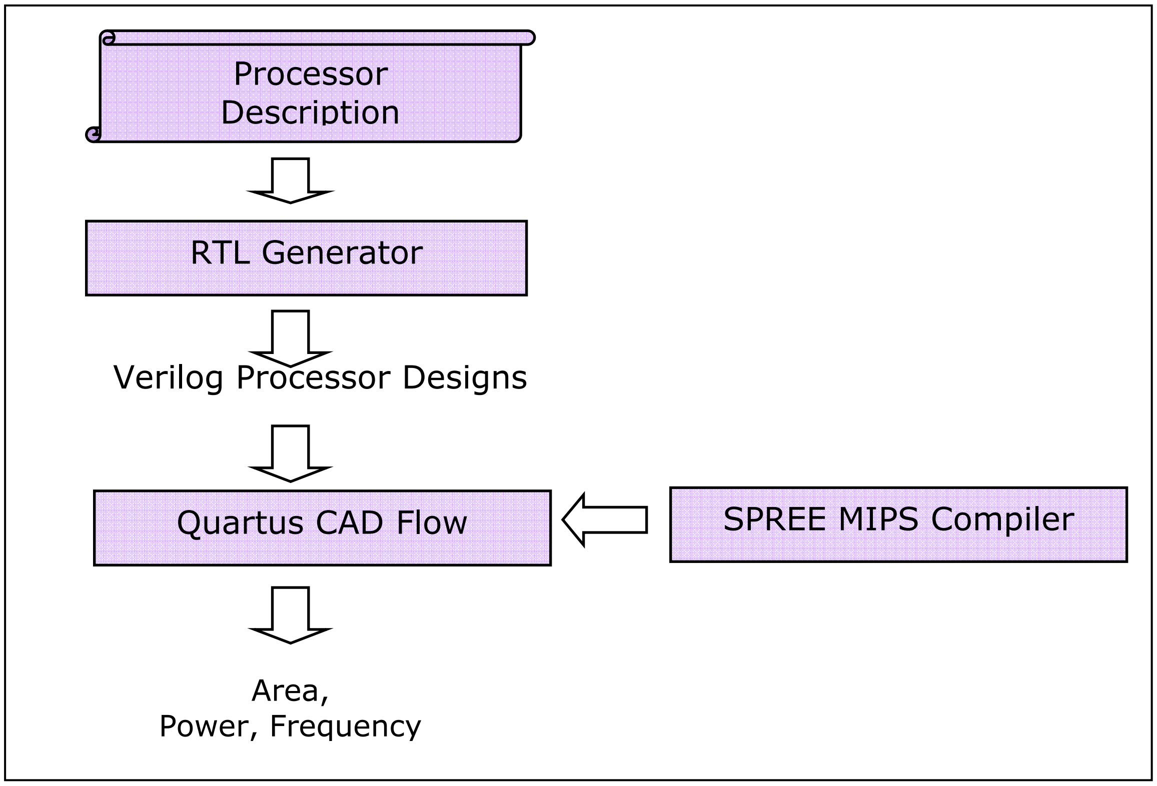


Figure 12: Soft Processor Rapid Exploration Environment (SPREE) [11]

Although caches are not supported in the present architecture, we do not consider their absence as a serious limitation since the proximity of memory and logic in FPGAs enables abundant on-chip memory bits to be used as good alternative to complicated on-chip caches. In future work, we plan to extend our approach to support off-chip memories. Some of the other limitations include a lack of support for dynamic branch predictions, exceptions and operating systems. In general, the simplicity of processors helps us to fit large multiprocessor systems on standard FPGAs and study the impact of several micro-architectural parameters on the overall area, power and performance of our soft multiprocessor systems. A simple 3-stage SPREE processor is shown in Figure 13.

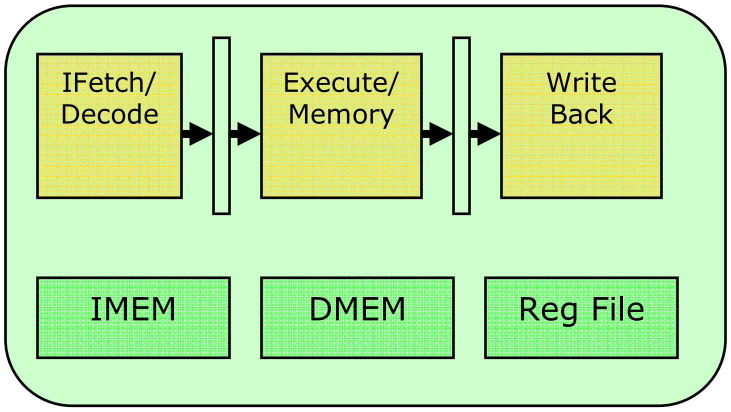


Figure 13: Architecture of a simple SPREE Processor

3.2.1 Soft Multiprocessor Interfaces

The choice of interconnection topology plays an important role in the performance of communication-intensive multiprocessor systems. The soft processors in our multiprocessor systems are interconnected using simple point-to-point FIFO links. We justify this architectural choice with the following reasons:

a. Studies regarding interconnection topologies [5] show that the rich modern FPGA fabric is capable of supporting highly-interconnected topologies such as direct point-to-point links.

b. Point-to-point links are more scalable than bus-based networks since the number of links increases proportionately with the number of processors in the system. The number of links increases linearly in a mesh type topology whereas it increases quadratically in a fully interconnected topology.

c. The RAW-style architecture with dedicated switch router per processor is likely to consume more logic and memory resources in FPGAs since each switch processor has a dedicated processor pipeline and instruction memory. Since the routing and placement tasks are handled by automated CAD tools in FPGAs, the mesh topology need not result in a strict grid-layout within the FPGA.

Synchronization is implemented in a very simple and efficient way by blocking read/write operations on FIFO empty/full conditions. Although this approach is very similar to [4], the latter uses expensive and complicated dual-port BRAM based communication controllers for synchronization. In contrast, our approach uses inexpensive logic registers available in FPGAs. The approach proposed in [4] has the benefit that any processor can interface with any other processor in the system through a communication controller. This flexibility is made possible through a complicated addressing scheme where an interface unit attached to the communication controller decodes each FIFO address and generates write control signals for FIFOs. Although the scheme is attractive due to its flexibility, the inherent complexity of the communication controller makes it a poor interconnect solution in terms of area. Instead, we implement FIFO blocking mechanisms in software that check FIFO empty/full conditions and execute empty loop instructions during blocked transfers. The software approach has the benefit that it simplifies the integration of processors and minimizes hardware resources required for synchronization. Figure 14 shows an example of an interconnection where processors are connected together by FIFOs. Each FIFO in the illustrated example has a capacity of ‘n’ words where each word represents 32 bits or 4 bytes of data. 32 bits were selected to match the size of the processor register. The FIFOs can be read or written simultaneously. The FIFO implements a half-duplex communication between a pair of processors. Two FIFOs can implement a full-duplex communication between each pair of processors. Each processor has memory mapped input/output ports which can be interfaced directly to the FIFOs. Memory mapped ports facilitate reads/writes to the FIFOs through conventional load/store instructions. To minimize the inter-tile data transfer latency, which is critical to instruction level parallelism, the memory mapped I/O ports are integrated into the bypass paths of the processor pipeline. A typical inter-tile transfer is described as follows: During the first cycle, the execution result from the producer is written to a FIFO location through a store instruction. During the next cycle, the consumer loads the value into its register through a load instruction. Thus, each inter-tile transfer consumes only 2 cycles.

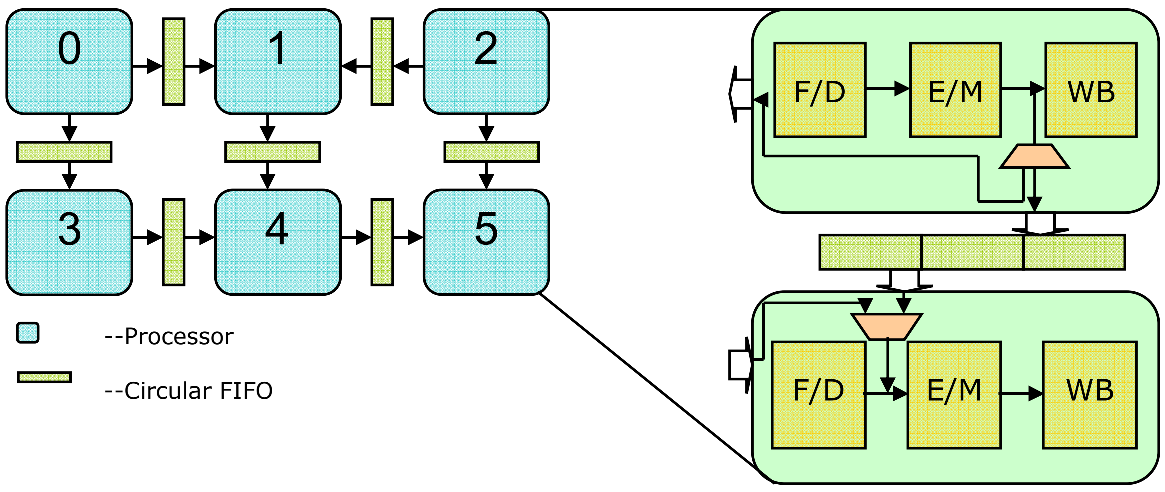


Figure 14: Set of processors interconnected by FIFOs

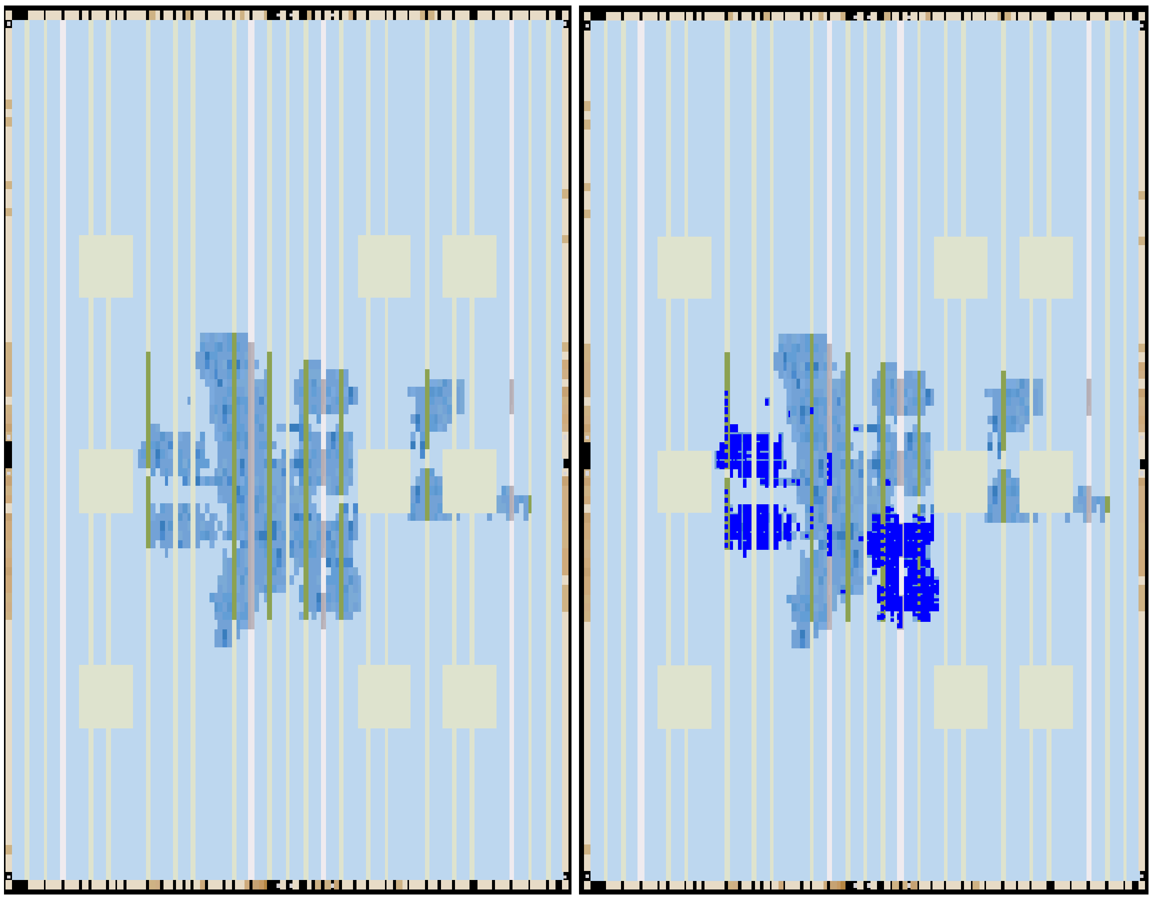
ALUTs	11
Registers	72
Memory bits	128

Table 1: Resource usage of a simple FIFO

The FIFO can be implemented using a minimal set of logic resources as shown in Table 1. Hence our approach guarantees that the logic resources required for interconnect do not seriously constrain the scalability of the soft multiprocessor system.

3.2.2 Interconnection Topologies

Topology refers to the arrangement of processors and links in the multiprocessor system. Topology has a direct impact on the performance of the multiprocessor system since it dictates the way processors exchange data among themselves. Smaller multiprocessor systems use bus-based approaches. Previous research has been shown that bus-based topologies do not scale well with larger multiprocessor systems since the constraint on resources steadily increases with the number or processors. Pipelined channels could overcome the limitations of buses since the number of links can grow linearly with the number of computational nodes. RAW uses pipelined channels interconnected in a mesh-type topology to interconnect its sixteen processing nodes. The mesh topology guarantees that the maximum distance the clock has to travel is across a pipelined channel. By increasing the pipelined channels, the clock frequency can be improved significantly. The mesh topology adapts well to the growing wire delay architecture models since propagation delays can be translated to network hops. To obtain high performance, tiles which communicate with each other often need to be placed closer to one another. Non-neighboring tiles must forward the data via intermediate tiles through network hops. Thus, judicious placement-routing policies when combined with different architectural techniques can combat the increasing wire delays. In contrast, placement and routing for FPGAs is largely a responsibility of automated design compilers. Manual placement and routing in FPGAs can often result in sub-optimal performance due to the regularity in logic arrangement and inflexibility in the location of resources such as memories and I/O pins. Thus, a multiprocessor system with a mesh topology may not result in a strict mesh-like placement in FPGAs. For example, Figure 15 shows the layout of a six processor system mapped onto a Stratix II EP2S180F1508C3 device.



(A) Complete 16 processor system **(B) Highlighted Processors 1,2,3,4**

Figure 15: Layout of a 16 processor system on a Stratix II device

The performance of mesh based architectures is dependent on how well the communication patterns of the application are mapped onto the underlying hardware. In RAW, the compiler statically schedules the data transfer orderings at each switch router. Although the data-hop based mesh-topologies can be directly mapped onto soft multiprocessors, they do not take full advantage of the non-mesh layouts in FPGAs as shown in Figure 15. For example, mesh topologies incur significant synchronization overhead for inter-processor data transfers since each transfer requires status check and read/write operations on the FIFOs. However, the synchronization cost may be reduced by using direct point-to-point links between each pair of communicating processors. Although the point-to-point topology can transform into a fully-connected network in the worst-case, we will demonstrate in Chapter 5 that the point-to-point links do not increase latency for many applications.

We illustrate three types of interconnection soft multiprocessor interconnection topologies – mesh, point-to-point and hypercube. Consider two kinds of interconnection topologies, as illustrated in Figure 16 and Figure 17. The labels on each processor in the illustration indicate the steady state communication patterns of the corresponding processors. In the mesh type topology, each soft processor has at most four ports – North, South, East and West. The communication between non-neighboring processing nodes must hop through intermediate nodes.

In the example illustrated in Figure 16, processor 3 produces two values under steady state conditions which are sent to its North and East ports. Processors 0 and 4 receive the data, compute the results and route the results to processor 1. Processor 1 assembles the data in their respective arriving sequence and forwards them to processor 5 via processor 2. Note that in this example, the data produced by processor 0 and 4 could have been forwarded to 5 through direct links. An alternate topology that uses direct point-to-point links between processor 0, 4 and 5 is shown in Figure 17. Clearly, the point-to-point topology incurs fewer synchronization and transfer hops when compared to the mesh-topology. We will analyze the area and performance benefits of using a point-to-point topology against a mesh topology in Chapter 5.

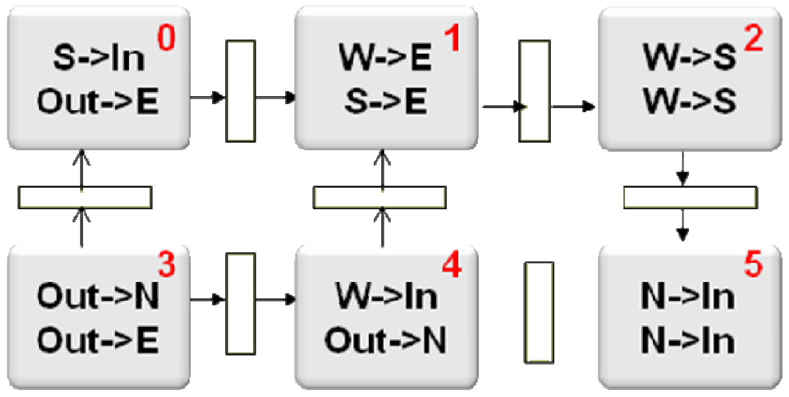


Figure 16: Mesh topology

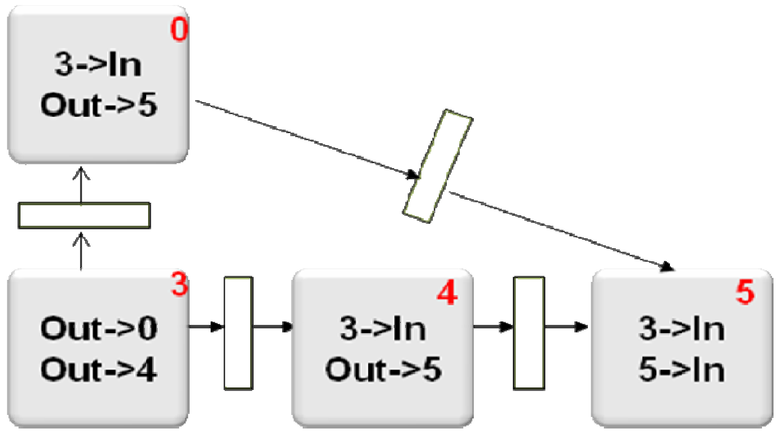


Figure 17: Point-to-point topology

A hypercube topology is illustrated in Figure 18. In this figure, each node represents a processor and each edge represents a FIFO channel between a pair of processors. The hypercube topology offers more flexibility than the mesh topology through additional communication links. However, the hypercube is less flexible when compared with a fully-connected network. We examine the impact of using a hypercube topology for interconnecting the soft processors in Chapter 5.

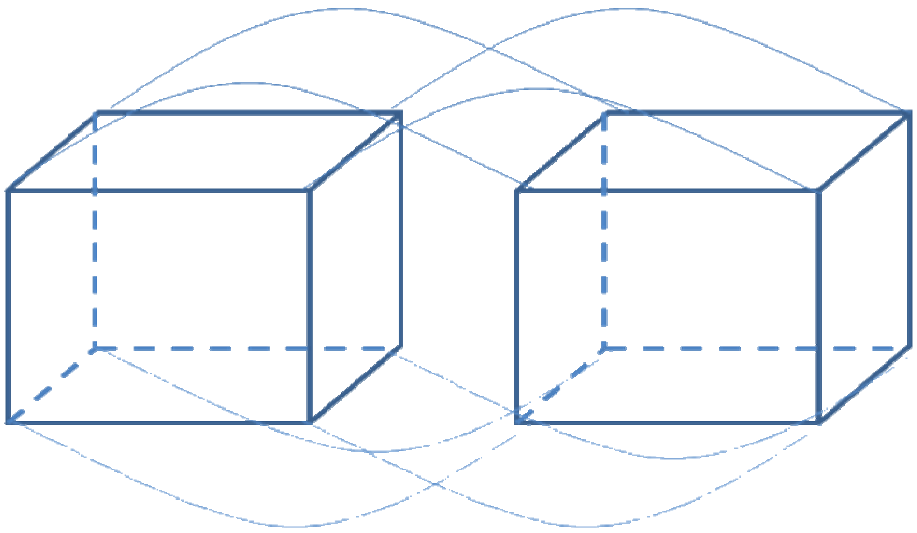


Figure 18: Hypercube topology for 16 processors

3.2.3 Application Specific Soft Multiprocessor Optimizations

Previous research has shown that application-specific micro-architectural customizations on individual soft processors can save significant area and power. In some cases, logic reduction has been shown to improve performance. We investigate the impact of individual processor optimizations on performance, area usage and power consumption for overall multiprocessor systems. Some of the optimizations under consideration are described below:

Application-specific instruction subsetting and memory sizing

Applications typically require far fewer instructions than are supported by the instruction set architecture. Figure 19 shows the average instruction usage of 7 Streamit benchmarks mapped onto 16 core soft multiprocessors. Figure 20 shows the average percentage of used instructions in each processor for a software FM radio application. As observed in

Figure 19, all applications, except DES, use less than 50% of the supported ISA. Smaller kernels such as Lattice filter use only about 26% of the available ISA. This motivates us to study the impact of using reduced decode logic and control circuitry for individual processors on the basis of application-specific instruction usage patterns.

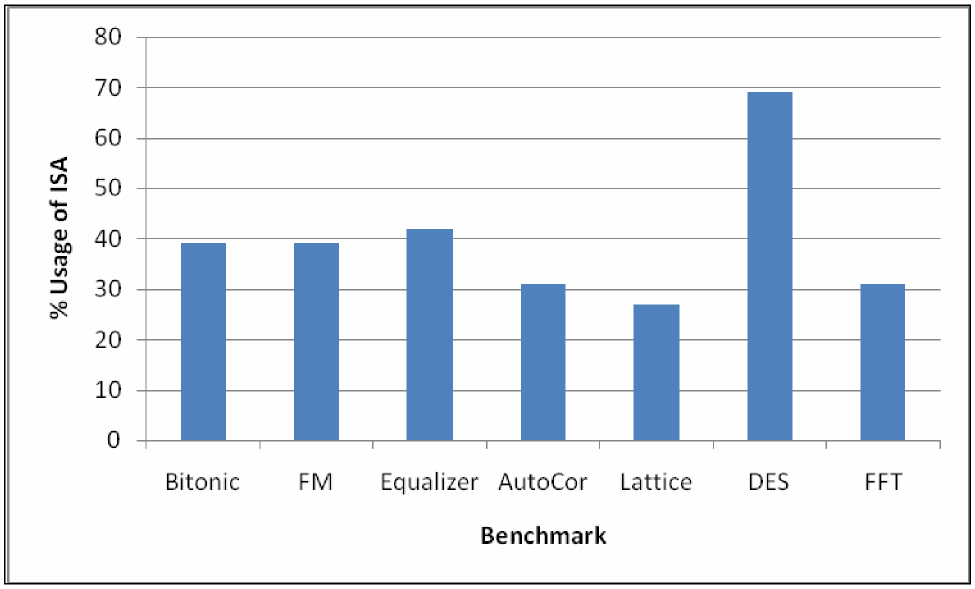


Figure 19: Instruction usage for 7 Streamit benchmarks on 16 soft processors

As applications are mapped over a large number of processors, they become finer grained. Each processor requires less on-chip memory to store instructions and data for its application segment. We evaluate the impact of application granularity on on-chip memory later in Chapter 5.

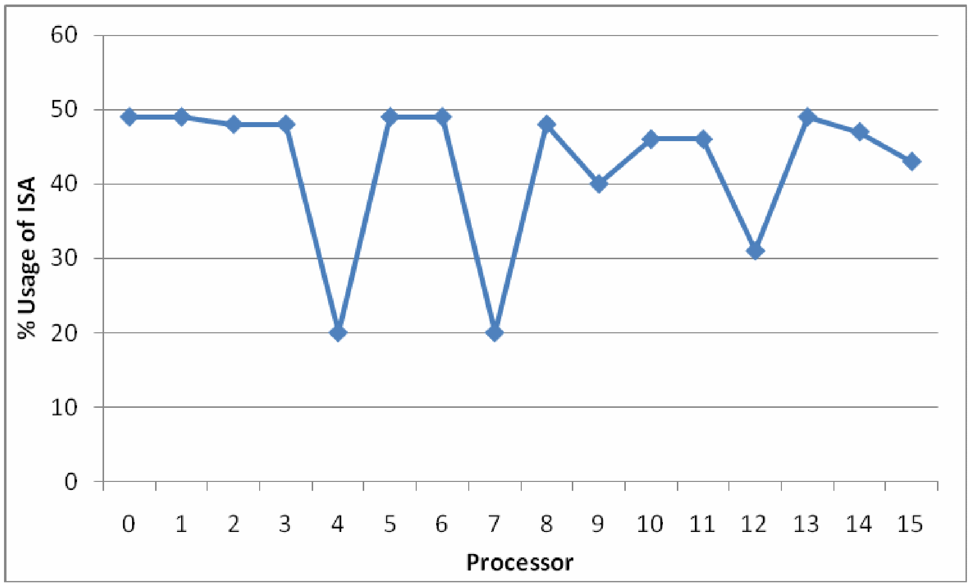


Figure 20: Percentage ISA usage for a Software FM Radio over 16 processors

Pipeline stage optimization

The number of pipeline stages influences the complexity, size and performance of any processor. Deepening the pipeline is likely to increase the area of the processor, as observed in [11]. Although the addition of pipeline registers can improve the clock frequency, the CPI may be adversely affected due to a significant increase in branch penalties. We analyze the impact of tuning the pipeline depth of individual processors on the performance of multiprocessor systems.

FIFO buffer sizing

Stream applications are typically communication intensive. Most stream applications consist of kernels that interact with each other in real time to exchange data. In this context, the architecture of the inter-processor interconnect plays a vital role in the performance of communication intensive architectures. In many cases communication overhead must be significantly reduced to achieve high speedups. It is worthwhile to take a look at how variations in buffer sizes can affect application performance.

CHAPTER 4

DESIGN FLOW

Our soft multiprocessor design framework extends an existing stream compiler and integrates a processor generator to create a scalable flow for soft multiprocessor systems. Figure 21 shows an overview of the proposed design flow for the soft multiprocessor synthesis framework. The tool allows the designer to specify different parameters of the multiprocessor system such as the topology, the number of processors and the custom features such as the pipeline depth and interconnect buffer size.

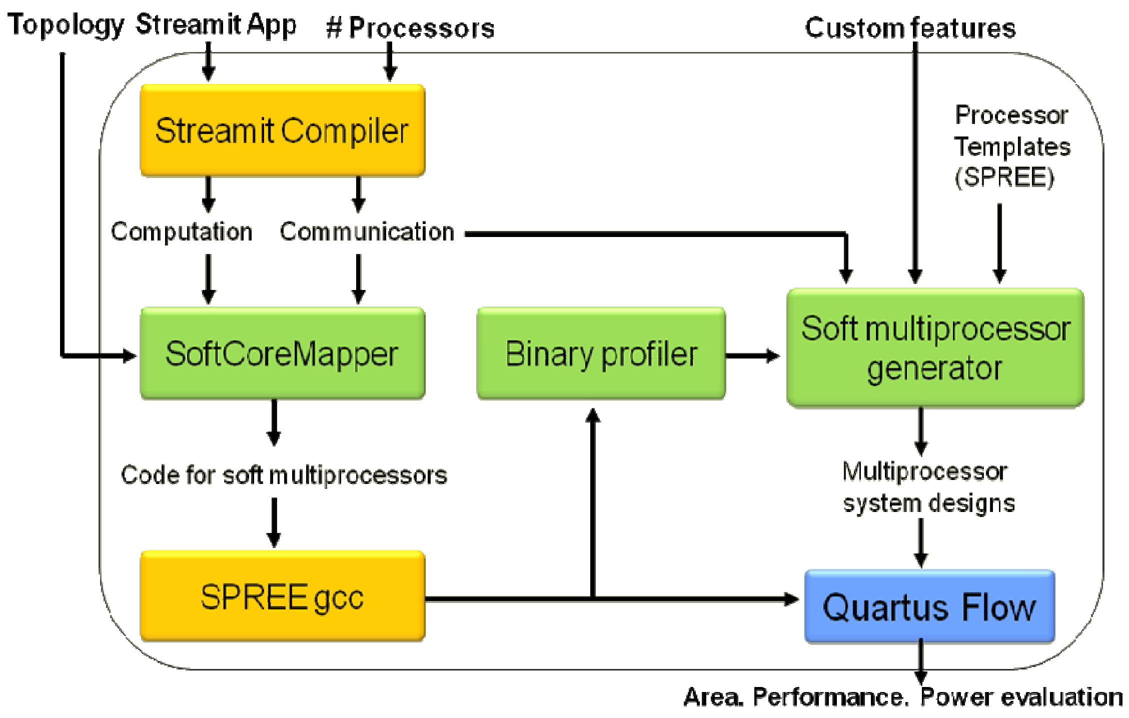

Topology
StreamIt App
# Processors
Custom features
StreamIt Compiler
Processor Templates (SPREE)
Computation
Communication
SoftCoreMapper
Binary profiler
Soft multiprocessor generator
Code for soft multiprocessors
Multiprocessor system designs
SPREE gcc
Quartus Flow
Area, Performance, Power evaluation

Figure 21: Design flow for soft multiprocessor synthesis framework

The application is specified in Streamit. The Streamit compiler maps the application to a subset of the processors in the RAW architecture based on the number of processors specified by the user. The mapping involves phases such as graph expansion, partitioning, layout and scheduling. Streamit generates code for the RAW architecture that has processors and communication controllers that coordinate the communication between the individual cores. However, the code generated by Streamit cannot be executed directly on our soft multiprocessor designs for two reasons – First, soft multiprocessor systems generated using our flow do not support dedicated communication controllers. Hence, there is a need to map the communication schedule produced by Streamit onto the computation code. Second, the generated multiprocessor systems support point-to-point and hypercube topologies in addition to the mesh topology supported by the Streamit compiler. We developed a tool called SoftCoreMapper that extends the Streamit compiler passes to support the above requirements. Specifically, SoftCoreMapper performs the following operations on the Streamit output:

Dead Code Elimination – In this phase, RAW-specific routines and segments of the application are removed to reduce the code size and remove irrelevant operations. Specifically, this phase removes RAW initialization routines and replaces floating point operations with their equivalent integer operations.

Communication Rescheduling – Communication rescheduling analyzes the communication patterns produced by the Streamit compiler to derive a suitable schedule for the target topology of the soft multiprocessor system. At present, the rescheduler supports a point-to-point and hypercube topology. However, this phase can be extended to support other topologies as well. We illustrate the communication rescheduling algorithm for a point-to-point topology in Figure 22.

```
1. Comm schedule -  directed graph
2. For each generated data in graph
3. {
4.    Traverse the graph to discover hop edges
5.    Eliminate hop edges
6.    Insert point-to-point edges
7. }
8. Reschedule communication
```

Figure 22: Rescheduling algorithm for point-to-point topology

For a point-to-point topology, the communication schedule generated by Streamit is represented as a data flow graph where the nodes represent the individual processors and the edges are represented by the instructions that transfer the data between the processors. Next, for each processor and each data value produced by that processor, we traverse the data flow graph for the generated data from source to destination(s). The traversal may produce multiple paths depending on whether the data is consumed by a single or multiple processors. Next we define a hop edge as an instruction that transfers data between two processors without performing any operation on the data. For each data path in step 4 of Figure 22, we discover all the hop edges and eliminate them. Next, a direct edge is inserted between the producer processor and all the consumers of the data. Finally, the resulting sub graph is used to reschedule the communication for the point-to-point topology.

Communication Mapping – Since the generated soft multiprocessor designs do not support dedicated communication controllers for managing the communication between the processing cores, there is a need to integrate the communication, which is explicitly specified in the schedule generated by Streamit, into the application code for each processor. This phase analyses the computation and communication patterns to find a one-to-one mapping between the application code and the communication schedule. Next, register-mapped data transfer statements in the application code are replaced with memory-mapped communication statements.

Synchronization and Code Generation- In the final phase, the SoftCoreMapper identifies portions of the application code where data communication occurs and inserts synchronization primitives. Examples of synchronization primitives include register comparison operations to check the empty or full conditions of FIFOs.

Once the SofCoreMapper generates code for each soft microprocessor, the code is compiled through a modified MIPS gcc compiler supported by the SPREE package. The compiled binaries are analyzed by an application binary profiler to determine the application-specific instruction usage patterns of each processor.

A significant challenge in the design space exploration of large-scale soft multiprocessor systems is the generation of the systems itself. To address this issue, we designed an Automatic Soft Multiprocessor Generator (ASMG). This tool accepts various parameters of the multiprocessor system from the user such as the pipeline depth of each processor and the interconnect buffer size. It also allows the user to customize the instruction set logic according to the profiling information generated by the application binary profiler. Next, ASMG generates the Verilog descriptions for the multiprocessor systems and customizes the data path and control path logic to suit the requirements of the application. The switch schedules produced by Streamit are analyzed to derive communication ports for each processor. Finally, interconnection networks are generated according to the communication patterns generated by the rescheduler. In a mesh topology, the number of I/O interfaces is at most four. In case of a direct point-to-point topology, each processor can directly interface with all its data sources and sinks.

The multiprocessor Verilog HDL files are synthesized with the Altera Quartus synthesis framework and simulated using the ModelSim [33] simulator to derive area, power and performance results.

CHAPTER 5

EXPERIMENTAL RESULTS

Soft multiprocessor systems consisting of 1, 4, 9 and 16 processors were generated using our framework. We synthesized our designs to Altera DE2 and DE3 development boards consisting of 90nm Cyclone II EP2S180 and 65nm Stratix III EP3SL150 FPGAs, respectively. The performance was measured in terms of absolute wall clock type per output, a measure of throughput for streaming applications. The wall clock time was obtained by multiplying the cycles required to produce an output under steady state conditions by the inverse of the maximum operating frequency of the design reported by the Quartus compiler. To assess the maximum frequency of each design, we synthesized each design with a timing constraint of 150MHz.

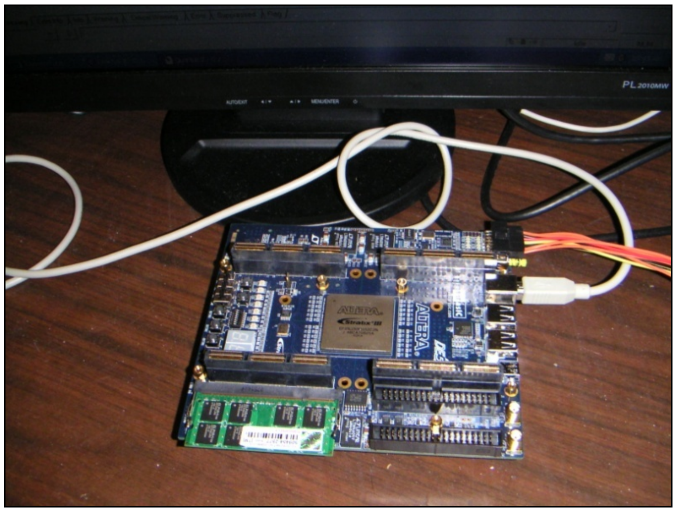


Figure 23: Altera DE3 board with Stratix III device EP3SL150

In the following sections, we evaluate the performance, area and power consumption of our designs and assess their scalability for all the benchmarks. Finally, we investigate the impact of application specific microarchitectural customizations on the generated designs.

5.1 Benchmarks

The proposed framework was evaluated using a set of benchmarks available with the Streamit compiler. This set consists of signal processing kernels and security, sorting and multimedia applications. Table 2 describes some benchmarks used to evaluate our framework.

Benchmark	**Description**
Bitonic	High performance bitonic sorting network
DES	Implementation of DES Encryption Algorithm
FFT	Fast Fourier Transform kernel.
Filterbank	Filterbank for multirate signal processing application
FM	Software FM Radio with multiband equalizer
Autocor	Filter which generates autocorrelation series for input
Lattice	Ten stage lattice filter
Equalizer	An equalizer program for audio applications

Table 2: Framework Evaluation Benchmarks

Most streaming applications fall in the category of signal processing, audio, video, multimedia, encryption and networking. In the benchmark set under consideration, applications such as FFT and Filterbank represent small signal processing kernels. Larger applications such as an audio beamformer, FM Radio and Equalizer reuse the kernels to create complex real-world applications. Many signal processing and audio/video benchmarks require floating point computations which are not currently supported by the basic SPREE processor [11]. As a workaround, we replace floating point computations by their equivalent fixed point operations in software.

5.2 Interconnection topology variation

In this experiment, we measure the run time performance of four applications for mesh and point-to-point topologies. Figure 24 shows the normalized application speedup of a point-to-point topology against a mesh topology. All the processors in the designs consist of three stage pipelines. The cycles for output and maximum design frequency for all the benchmarks are given in Table 3. Overall, point-to-point interconnect outperforms a mesh-style network for all applications by a factor of between 1.1x and 2x. Point-to-point topologies gain significant cycle speedups due to reduced synchronization overhead from the elimination of network hops. Point-to-point topologies consumed 28.6% less cycles when compared to mesh-style topologies on average. Interestingly, point-to-point topologies also gave slightly better performance in terms of design frequency.

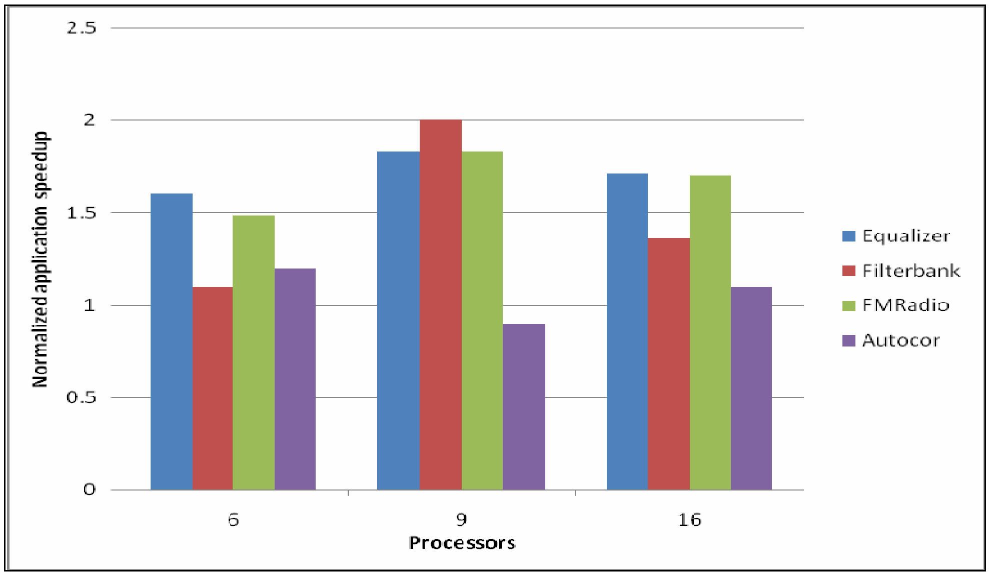


Figure 24: Performance of point-to-point topology vs. mesh topology

Application	**Mesh Clock Cycles**			**Point-to-point Clock Cycles**		
	6	**9**	**16**	**6**	**9**	**16**
Equalizer	15144	8625	4138	9812	4765	2475
Filterbank	3353	3625	1954	3021	1339	1503
FMRadio	14637	8923	4006	9816	4930	2392
Autocor	250	189	224	211	214	208

Application	**Mesh Design Freq**			**Point-to-point Design Freq**		
	6	**9**	**16**	**6**	**9**	**16**
Equalizer	127.6	122.0	118.8	127.2	122.5	121.0
Filterbank	124.0	123.0	118.0	122.5	121.8	121.4
FMRadio	128.7	121.9	119.0	126.5	121.7	121.3
Autocor	124.2	122.5	118.5	122.6	121.7	120.5

Table 3: Comparison of clock cycles and frequencies

For a sixteen processor system, the point-to-point topology shows an average 2% improvement in design frequency. This frequency improvement results from the removal of unnecessary input/output FIFO ports. In a mesh-style topology, many processors need close to four ports as these nodes perform data forwarding in addition to computation. The improvement is observed even though processors with large data fan-outs (sources) and fan-ins (sinks) in point-to-point topologies typically require more than four ports. For example, in a mesh-style topology for a 16 processor FM Radio application, the average port usage per processor is approximately 3, while for a point-to-point topology, the average port usage per processor is approximately 2. The processors executing splitter and joiner filters in the point-to-point topology for this application requires 11 and 9 ports, respectively. For smaller designs, like AutoCor, cycles per output increases or remains unchanged when parallelized over larger multiprocessor systems since increased communication costs dominate over the reduced computation costs. A comparison of the area costs of mesh and point-to-point topologies show that in larger multiprocessor systems, the point-to-point topologies consume about 2 to 5% less area than the mesh topologies.

In all designs, the critical path is located within the three-stage processor logic. Thus, the addition of point-to-point links does not degrade the maximum design frequency significantly, although the addition of more point-to-point links may make the FPGA more difficult to route. The number of point-to-point links scales linearly with processor count in most designs.

In another experiment, we compare the hypercube topology against a mesh and a point-to-point topology. The hypercube has more flexibility in terms of connections when compared to a mesh topology. However, the hypercube does not offer an unlimited connectivity as in the point-to-point case. The results are plotted in Figure 25.

In general, the hypercube topology gives a modest 2 to 8% improvement over the mesh topology. The performance gain results from the reduced number of cycles due to the increased connection flexibility. However, the direct point-to-point topology still outperforms both mesh and hypercube by around 60% in the applications under consideration. Our results also indicate that the performance of the topology is an application specific variable and point-to-point topologies can give better performance for coarse-grained applications, such as FMRadio and Equalizer, rather than fine-grained kernels.

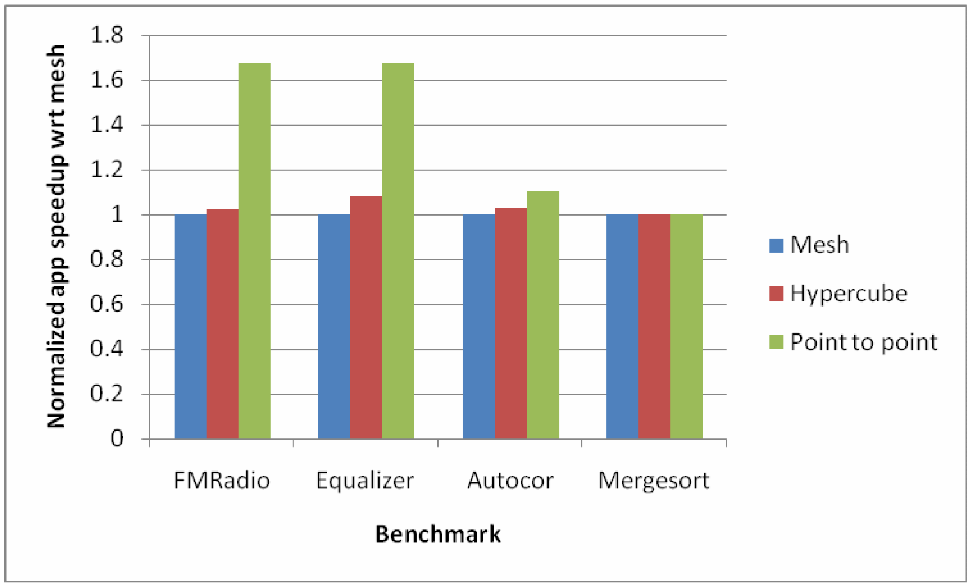


Figure 25: Performance of point-to-point and hypercube topologies

5.3 Customization of pipeline depth

The choice of microarchitectural pipeline depth of each processor influences the overall throughput of the application. The impact of three, four and five stage pipelining on application performance is studied in this discussion. The three stage pipeline consists of the fetch/decode, execute/memory and the write back stages. Four stage pipelines extend three stage pipelines by splitting the execute/memory stages into two separate stages. Finally, the five stage pipelines extend the four stage pipelines by adding an additional execution stage. We found that deepening individual processor pipelines from three to four stages can give substantial performance improvements of 22% on average at a 9.6% increase in area. Figure 26 shows the relative execution time per output for six stream benchmarks mapped over 16 processors.

The four-stage pipeline multiprocessor systems generally give better performance than their three-stage and five-stage counterparts. The critical paths of the multiprocessor systems for all designs are within the individual processors. In three-stage pipelines, the critical path is located between the register file and memory write-back logic through the branch predictor. For four- and five-stage pipelines, the critical path is between the register file and memory write-back logic through the integer multiplier.

The relative performance improvement of the four-stage pipelines results from improved per-processor performance. On average, the maximum design frequency improves by 26% from 118 MHz to 149 MHz as a transition from three to four-stage pipelines is made. However, the maximum design frequency remains largely unchanged when the pipeline depth is increased to five since the critical path remains between register file and memory write-back logic through the integer multiplier.

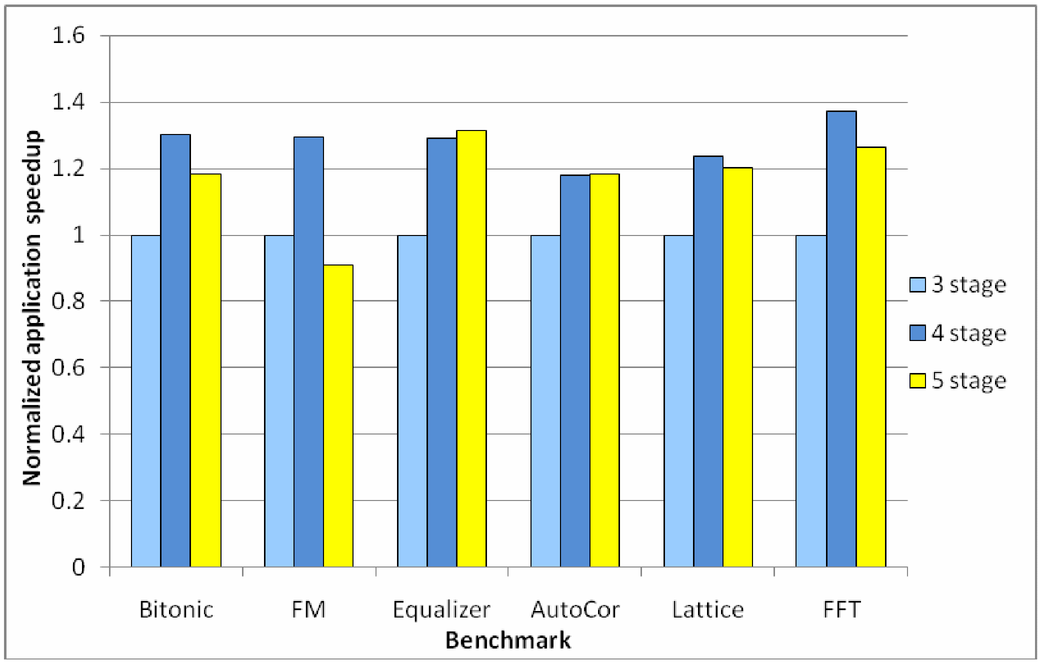


Figure 26: Performance of 4 and 5 stage pipelines against a 3 stage pipeline

As more stages are added to the pipeline, an increase in the number of cycles per output is observed for all the applications. When compared to three-stage pipeline multiprocessor systems, the cycles per output increases by 5% for four-stage systems and by 14% for five stage systems. The trends are consistent for 6 and 9 processor design cases. The increase in cycles can be attributed to two factors. First, the processors generated by the SPREE framework use interlocking to resolve data hazards. As pipeline depth increases, it becomes increasingly difficult for the compiler to support independent instructions within the interlocking window, which introduces more stalls. SPREE uses a simple static *branch not taken* prediction scheme [11]. In general, branch mispredictions can be costly in deeper pipelines. Also, it can be difficult to support branch delay slot instructions in deeper pipelines, causing more stalls. Stalls due to branch mispredictions and data hazards in individual processor pipelines can ripple across multiple processors in communication-intensive stream applications.

5.4 Customization of communication buffer depth

Stream applications are often communication-intensive since they consist of a pipeline of tasks. In many cases, communication overhead must be amortized to achieve effective performance. Figure 27 shows the variation of normalized application speedups with varying FIFO sizes for five benchmarks mapped to nine processors using previously-discussed topology and processor pipeline preferences. For large applications, we observe that the cycle reduction (e.g. throughput) increased once a critical FIFO size is reached. For example, for Bitonic sort, the application speedup improved by over 20% when FIFO size was increased from 8 to 16 words.

Smaller applications, such as AutoCor and Lattice, benefit little from an increase in buffer sizes due to limited inter-processor communication. In general, well-matched communication buffers prevent communication stalls without wasting system resources. Each soft multiprocessor system consists of customizable processors which communicate using simple FIFO buffers. In previous work [4], communication controllers (CC) were used to interconnect processors. Each CC requires 468 four-input LUTs and about 128 flip flops for four word storage. In contrast, our synthesis results indicate that each FIFO requires only 11 LUTs, 72 registers and 128 memory bits, a small fraction of available FPGA resources.

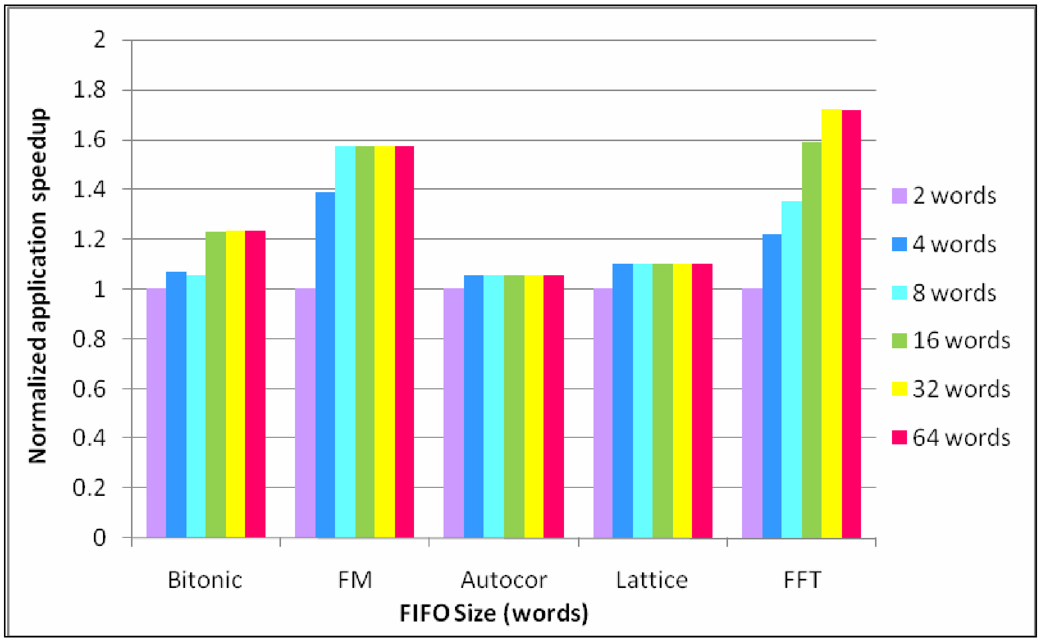


Figure 27: Impact of the interconnect buffer size on application performance

5.5 Soft multiprocessor ISA subsetting and memory size optimization

In general, soft microprocessors use only a portion of their ISA for filter implementation. As discussed in the previous sections, the average instruction set usage for majority of the benchmarks mapped over to sixteen processors was typically less than 50% of the available instructions. In fact, smaller applications such as Lattice consumed only about

26% of the available instructions. We showed in Chapter 3 that for a given application, the usage of instructions per processor in the multiprocessor system is highly variable. For example, the instruction usage of each processor in a sixteen processor system for software FM Radio application varied between 20% and 50%.

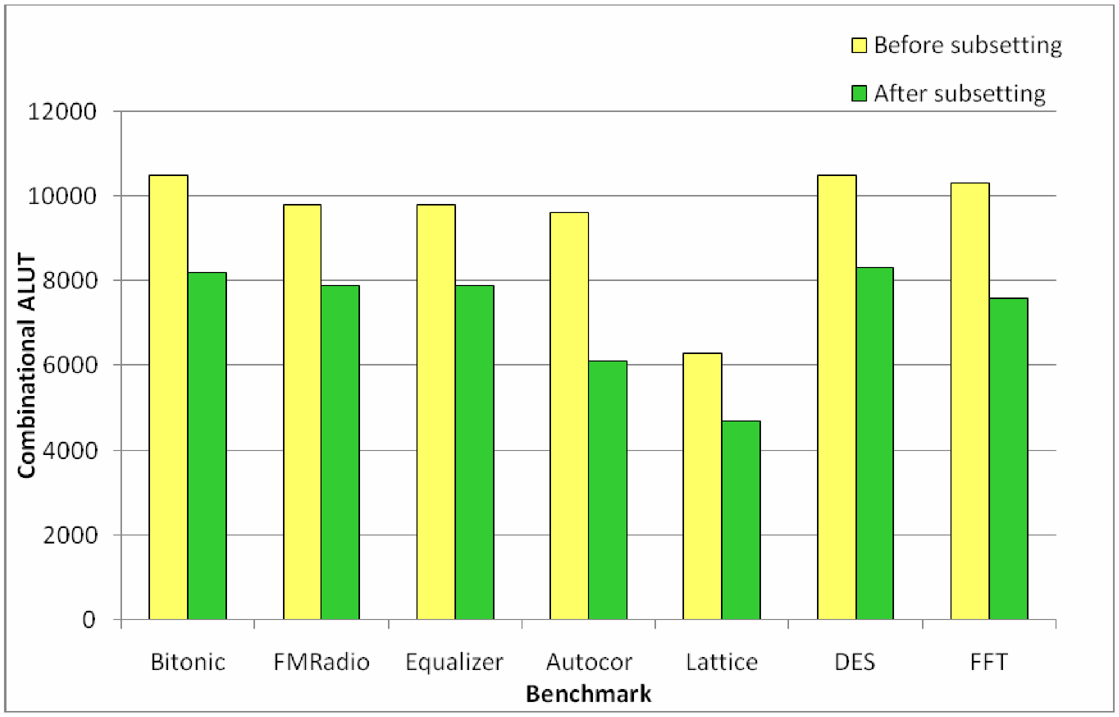


Figure 28: Area savings by instruction set customization for 16 processors

All these observations lead to the possible area savings that one could derive by customizing the instruction set in each processor according to the segment of the application running on it. We used the results from the binary profiler to customize the processors for each application. The results are plotted in Figure 28. On average, instruction set customization yielded a 27% percent improvement in area for the seven multiprocessor designs. The majority of the area savings were obtained in the decode logic and control circuitry in each processor. On average, the power consumption of subsetted designs consistently decreased by about 30% for 6, 9 and 16 processor designs.

A modest 4.2% improvement in maximum design frequency was also observed for the customized designs. The detailed frequency results for sixteen processor designs are illustrated in Table 4.

Application	Frequency Before Instruction Removal	Frequency after Instruction removal
FMRadio	119.0	123.0
Beamformer	118.8	121.2
Autocor	118.5	123.5

Table 4: Design frequency improvement by instruction subsetting

As our soft multiprocessor systems use on-chip memory bits in the FPGA for storing program code and data, memory is a critical resource that limits the number of soft multiprocessors that can be embedded into each FPGA. The memory requirements are further constrained by the fact that some of the components like the register file and the data memory needs dual port RAMs for simultaneous access of two operands. We use M4K and BRAM memory bits to implement instruction and data memories for the processors. In the following paragraphs, we present the results of scalability of soft multiprocessor systems from a memory point of view. The results and the following analysis reveal some interesting conclusions.

The average memory usage per processor in soft multiprocessor systems is plotted in Figure 29 as processors are scaled up from one to sixteen. Note that the memory requirement of each processor decreases significantly as the application is spread across more processors. For example, the memory required by each processor in larger benchmarks such as DES and Filterbank decreases nearly by an order of magnitude when processors are scaled from one to six. These results illustrate that by customizing each processor according to reduced memory size, it is possible to scale streaming applications across larger soft multiprocessor systems. Figure 30 plots the total memory usage of the entire soft multiprocessor system as processor count is scaled up from one to sixteen. Surprisingly, the memory requirements do not significantly increase as more processors are added to the multiprocessor system. The total memory usage of some of the larger benchmarks is lower than the memory requirements of the single processor system. The reduction is attributed to the lower memory requirement of each processor for smaller kernels. This result further corroborates our earlier observations that it is possible to scale soft multiprocessor systems for streaming applications if the memory size of each processor is customized on an application-specific basis.

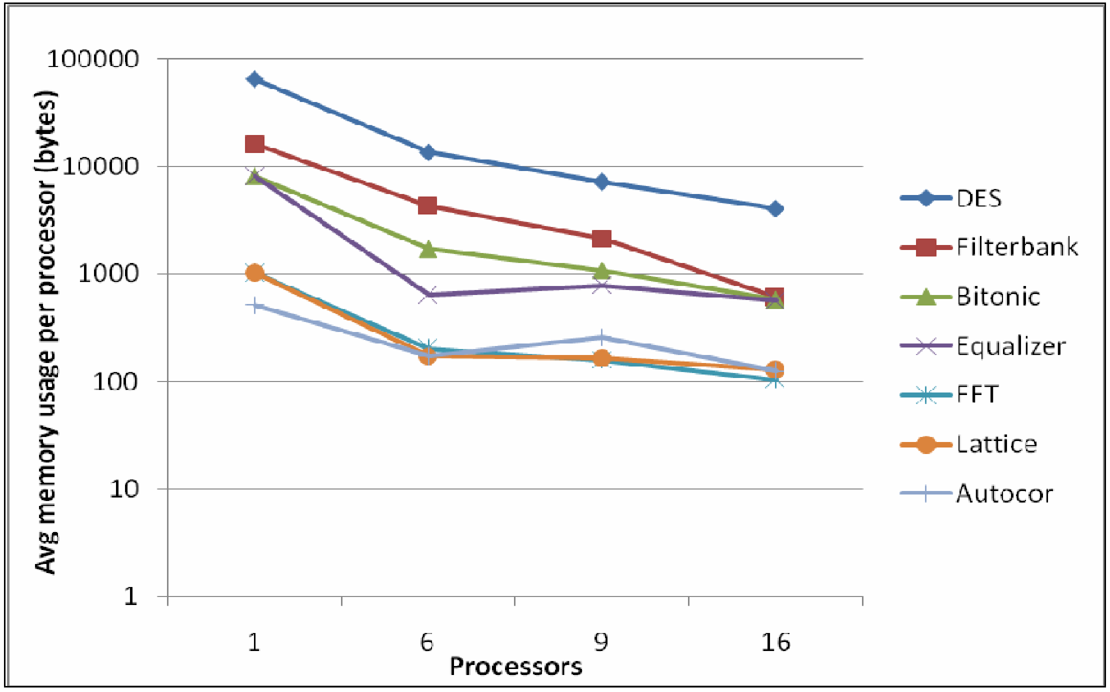


Figure 29: Average memory usage per processor for eight benchmarks

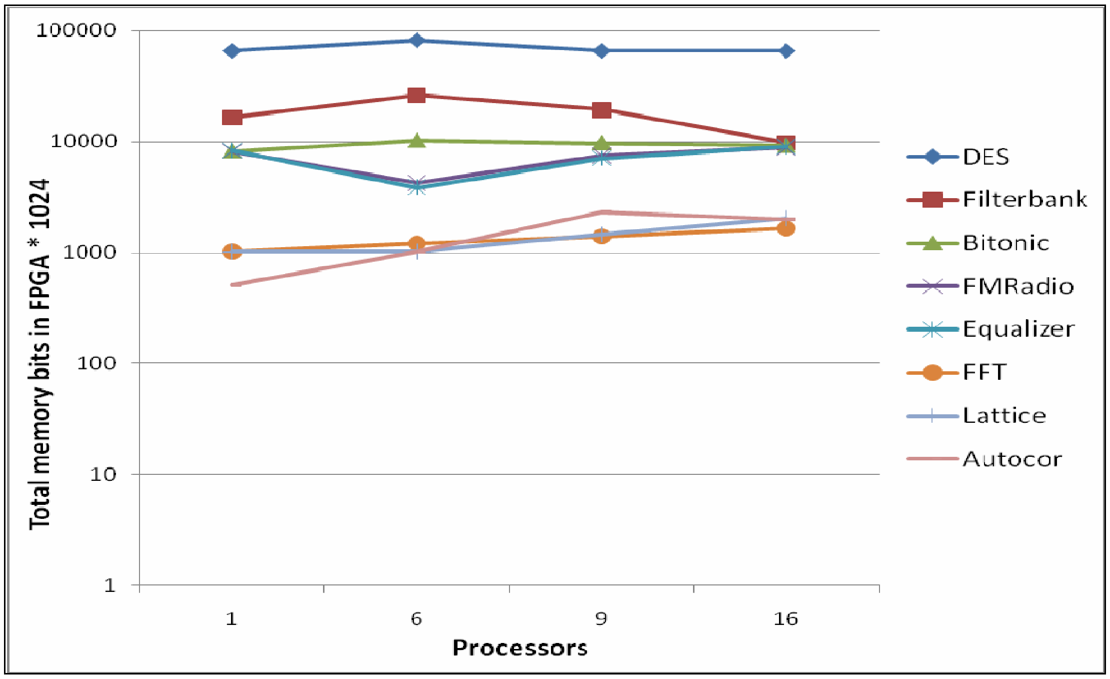


Figure 30: Total memory usage of scaling soft multiprocessor systems

5.6 Application scalability

Figure 31 shows the application speedup for the set of eight benchmarks normalized to a single soft core system for the parameters described in previous subsections. Each processor in the soft multiprocessor system consists of a three stage pipeline. The processors are interconnected using a point-to-point topology with all the interconnect buffers having a width of four words. The cycles per output and maximum design frequency in MHz are given in Table 5. The performance of larger applications such as DES, Bitonic and Filterbank improves by about a factor of 5x when parallelized over sixteen processors. The speedup improvement is primarily attributed to the significant amount of coarse-grained task-level parallelism present in these applications. However, the performance of smaller benchmarks such as Autocor and Lattice, degrades when parallelized over multiple processors. The performance degradation is due to the increased communication overhead which is present when the application is parallelized over larger multiprocessors. A similar trend is seen for the Filterbank benchmark as processor counts are scaled up from nine to 16 processors.

As seen in Table 5, the maximum frequency of all the designs degrades when more soft processors are embedded on the FPGA substrate. On average, a 11% frequency degradation is observed when all applications are mapped to 16 processors. The critical paths in these designs are within the processors, between the register file and memory through the branch predictor.

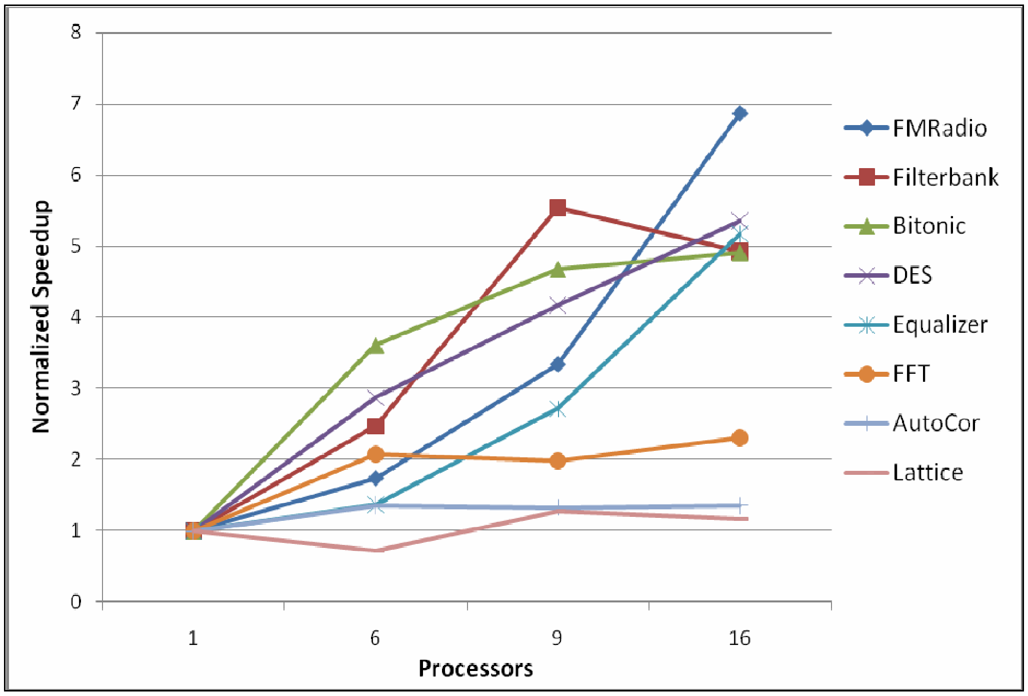


Figure 31: Application speedup of 8 benchmarks over 1 to 16 processors

	Clock Cycles				**Frequency (MHz)**			
Benchmark\Processors	**1**	**6**	**9**	**16**	**1**	**6**	**9**	**16**
DES	69094	23338	16452	11527	131	127	122	121
Bitonic	13511	3628	2883	2470	131	123	122	121
Filterbank	7986	3021	1339	1503	131	127	131	118
FMRadio	17728	9816	4930	2392	131	127	130	117
Equalizer	13862	9812	4765	2475	131	127	123	121
FFT	137	64	63	54	131	127	121	119
Autocor	306	211	214	208	131	123	122	121
Lattice	55	75	40	43	131	130	121	122

Table 5: Clock cycles and Frequency for 8 applications

Figure 32 shows the dynamic core power consumption at 50 MHz for 1, 4, 9 and 16 processor designs for seven benchmarks. A single processor design consumes about 60 to 100 mW of dynamic power at 50MHz. The dynamic power consumption scales up linearly when the number of processors is increased from one to four. The power consumption for 9 and 16 processor designs for Bitonic sort show mostly linear growth. In larger designs, each processor switches fewer times on average to produce the same number of outputs. However, increased communication and synchronization power costs increase the overall dynamic power. Note that the power consumption of smaller benchmarks such as Autocor, Lattice and FFT are considerably lower than those of the larger benchmarks. We observed that these applications were not large enough to distribute enough work to approximately 25% of the available processors in sixteen processor systems. Also, each processor in such benchmarks performed less computation due to the fine granularity of the application.

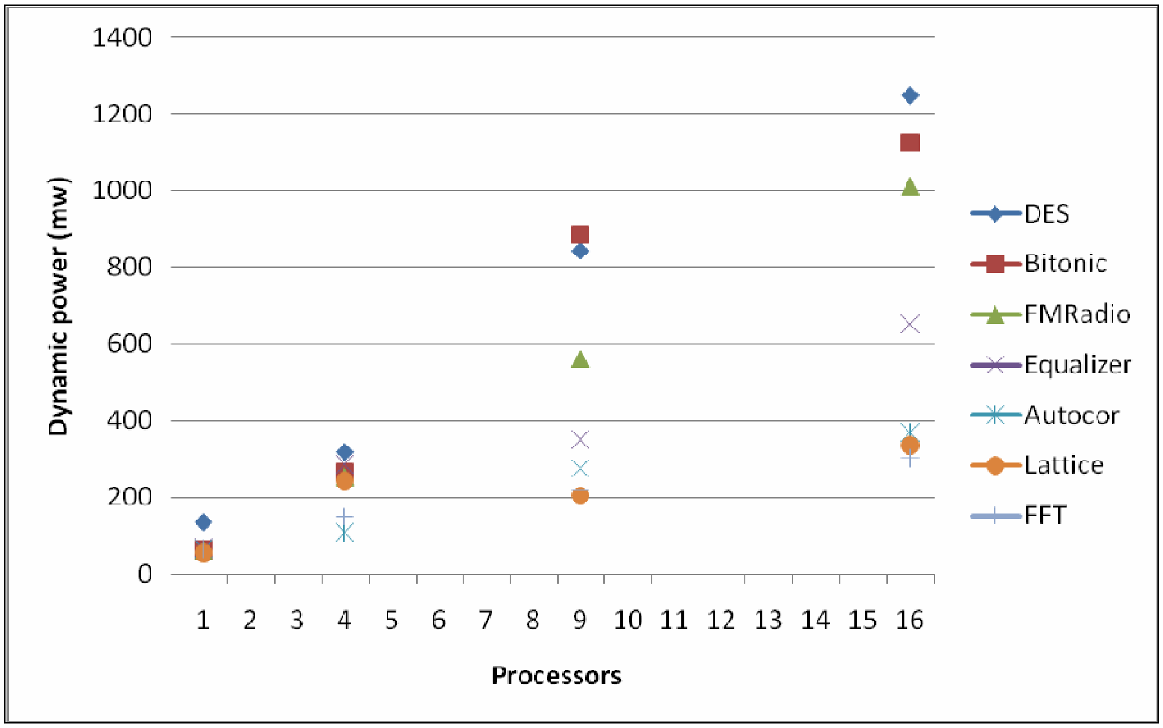


Figure 32: Dynamic power consumption of 1, 4, 9 and 16 systems

5.7 Combined impact of customizations

In this section, the combined impact of all the optimizations is considered. The application speedup of four benchmarks under their best case and worst case configurations are considered for 16 processors. The best case configuration is the choice of micro-architectural pipeline depth, interconnection topology and instruction set that gives the best application performance in absolute execution time. The worst case configuration uses the multiprocessor parameters that give the worst case application performance. In the given example, the best case is represented by a multiprocessor system where each processor has a four stage pipeline with all the instructions subsetted according to the requirements of the application segment. The best case uses ideal interconnect buffer sizes and a direct point-to-point topology. In contrast, the worst case design uses processors with five stage pipelines with a full instruction set. The processors are interconnected using a mesh topology with each FIFO configured for the worst case word size. Figure 33 shows the normalized application speedup of the best case configurations of four benchmarks against their worst case configurations for each optimization and in total.

On average, the performance of applications improves by a factor of 2.1x when all the customizations are applied on the soft multiprocessor system. The primary factors contributing to the overall application speedup are the choice of the pipeline stage depth and the choice of the interconnection topology. Although instruction subsetting saves considerable area, it contributes only 4% improvement to the overall application speedup. Our results indicate that a judicious choice of interconnection topologies and microarchitectural features can give significant performance and area benefits in soft multiprocessor systems.

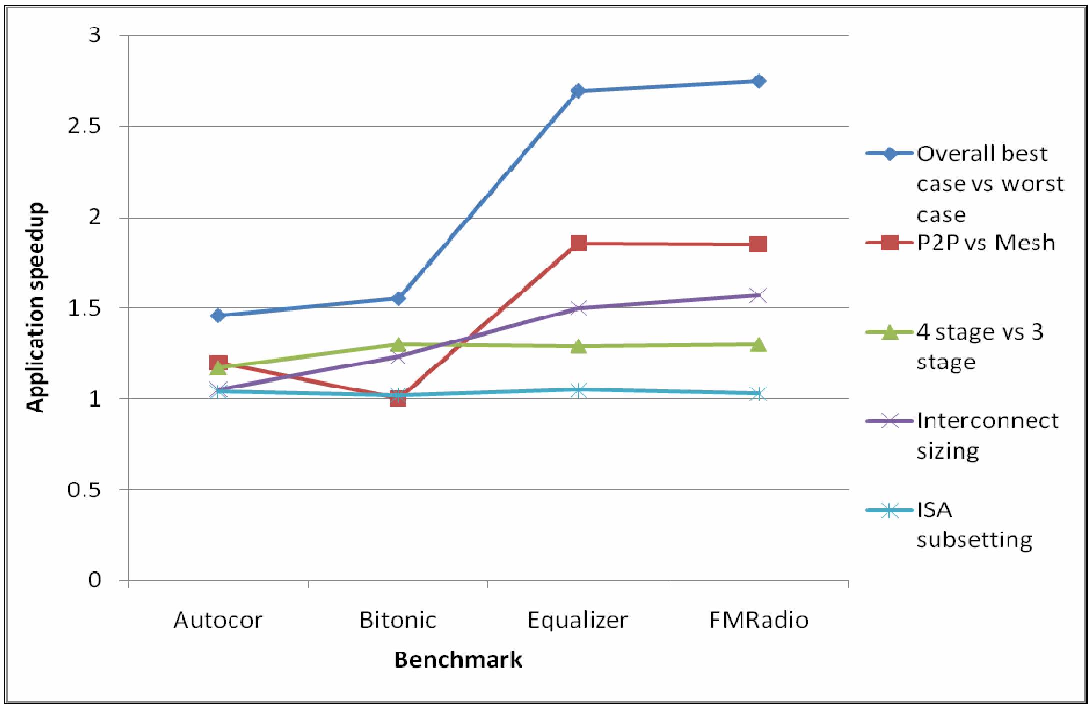


Figure 33: Impact of combined optimizations

Previously in [11], it was determined that a single SPREE soft processor demonstrates an 11% speedup over an Altera NIOS II/s processor. Our results add to this improvement.

CHAPTER 6

SUMMARY AND FUTURE WORK

The thesis has outlined an automatic soft multiprocessor generation and synthesis framework to facilitate the rapid design space exploration of soft multiprocessors. Our framework is capable of generating scalable soft multiprocessor systems by integrating efficient communication structures with customizable processors. The tool supports a high-level application compilation infrastructure that integrates state of the art streaming compilers with our own tools. The developed compilation infrastructure can be used to synthesize applications written in Streamit language to binaries that are executable on individual processors. Our approach has been verified with a diverse set of existing parallel computing benchmarks that represent the signal processing, multimedia and security application domains.

Results show that soft multiprocessor systems consisting of sixteen processors generated using our framework can offer 5x to 6x speedup over their uniprocessor counterparts synthesized in modern FPGAs. We illustrated that a judicious selection of various micro-architectural features such as interconnection topology, pipeline depth, inter-processor buffer size, memory size and customized instruction set can improve area by around 26% and performance by a factor of 2.1X in many applications. Our evaluation of soft multiprocessor interconnection topologies shows that highly interconnected topologies such as point-to-point can offer better performance than regular mesh topologies.

In the future, we plan to improve our soft multiprocessor systems by supporting advanced features such as off-chip memory accesses and better branch prediction schemes. We also plan to look into aggressive high-level compiler optimization techniques to improve application performance. We hope that the developed framework will facilitate rapid design space exploration of soft multiprocessors in the FPGA community.

BIBLIOGRAPHY

[1] Y. Jin, N. Satish, K. Ravindran, K. Keutzer, "An automated exploration framework for FPGA-based soft multiprocessor systems," *In International Conference on Hardware/Software Co design and System Synthesis (CODES)*, September 2005, pp. 273-278.

[2] K. Ravindran, N. Satish, Y. Jin, K. Keutzer, "An FPGA-based soft multiprocessor system for IPv4 packet forwarding," *In International Conference on Field Programmable Logic and Applications (FPL)*,August 2005, pp. 487-492.

[3] J. Cong, G. Han, W. Jiang, "Synthesis of an application-specific soft multiprocessor system," *In International Conference on Field Programmable Logic and Applications*, 2007, pp. 99-107.

[4] H. Nikolov, T. Stefanov, E. Deprettere, "Efficient automated synthesis, programming, and implementation of mult-processor platforms on FPGA chips," *In International Conference on Field Programmable Logic and Applications (FPL)*, August 2006, pp. 1-6.

[5] M. Saldana, L. Shannon, J.S. Yue, S. Bian, J. Graig, P. Chow, "Routability of Network Topologies in FPGAs," *In IEEE Transactions on Very Large Scale Integration Systems*, March 2007, pp. 948-951.

[6] H.C. Freitas, D.M. Colombo, F.L. Kastensmidt, P.O.A. Navaux, "Evaluating Network-on-Chip for Homogeneous Multiprocessors in FPGAs," *In IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2007, pp. 3776-3779.

[7] H. Nikolov, T. Stefanov, and E. Deprettere, "Systematic and automated multiprocessor system design, programming, and implementation," *In IEEE Trans. On Computer-Aided Design of Integrated Circuits and Systems (TCAD),* March 2008, pp. 542-555.

[8] N. Kapre, N. Mehta, M. DeLorimier, R. Rubin, H. Barnor, M. Wilson, M. Wrighton, A. DeHon, "Packet switched vs. time multiplexed FPGA overlay networks," *In IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM),* April. 2006, pp. 205-216.

[9] J.P. Derutin, L.Damez, A. Desportes, J.L.L. Galilea, "Design of a scalable network of communicating soft processors on FPGA," *In International Workshop on Computer Architecture for Machine Perception and Sensing (CAMPS)*, September 2006, pp. 184-189.

[10] L. Sun, E. Aboulhamid, and J.-P. David, “Network on chip using a reconfigurable platform,” *In IEEE Midwest Symposium on Circuits and Systems*, Dec. 2003, pp. 819-822.

[11] P. Yiannacouras, J.G. Steffan, J. Rose. “Application-specific customization of soft processor microarchitecture,” *In International Symposium on Field-Programmable Gate Arrays (FPGA)*, February 2006, pp. 201-210.

[12] O. Hebert, I.C. Kraljic, Y. Savaria. “A method to derive application-specific embedded processing cores,” *In International Conference on Hardware Software Codesign (CODES)*, September 2000, pp. 88-92.

[13] R. Dimond, O. Mencer, W. Luk, “CUSTARD- A customizable threaded FPGA soft processor and tools,” *In International Conference on Field Programmable Logic and Applications (FPL)*, August 2007, pp. 1-6.

[14] R.G. Dimond, O. Mencer, W. Luk, “Combining Instruction Coding and Scheduling to Optimize Energy in System-on-FPGA,” *In IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2006, pp. 175-184.

[15] B. Fort, D. Capalija, Z. Vranesic, and S. Brown, “A multithreaded soft processor for SoPC area reduction,” *In IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM),* Apr. 2006, pp. 131-142.

[16] M. Labrecque, J.G. Steffan, “Improving pipelined soft processors with multithreading,” *In International Conference on Field-Programmable Logic and Applications (FPL)*, August 2007, pp. 210-215.

[17] M. Labrecque, P. Yiannacouras, J. G. Steffan, “Scaling Soft Processor Systems,” *In IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2008, pp. 195-205.

[18] M. I. Gordon, W. Thies, S. Amarasinghe, “Exploiting coarse-grained task, data, and pipeline parallelism in stream programs,” *In International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2006, pp. 151-162.

[19] M.B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, A. Agarwal, “Evaluation of the RAW Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams,” *In International Symposium on Computer Architecture (ISCA)*, June 2004, pp. 2

[20] M.I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A.A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe, "A Stream Compiler for Communication Exposed Architectures," *In International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2002, pp. 291-303.

[21] R. Hoare, S. Tung, K. Werger. "An 88-Way multiprocessor within an FPGA with customizable instructions," *In Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS)*, April 2004, pp. 258.

[22] S. Cravan, C. Patterson, P. Athanas, "A Methodology for generating application-specific heterogeneous processor arrays," *In Proceedings of the Hawaii International Conference on System Sciences (HICSS)*, January 2006, pp. 251a-251a.

[23] J. Yu, G. Lemieux, "A case for soft vector processors in FPGAs," *In International Conference on Field-Programmable Technology (FPT)*, December 2007, pp. 341-344.

[24] R. Lysecky, F. Vahid, "A Study of the speedups and competitiveness of FPGA soft process cores using dynamic hardware/software partitioning," *In Proceedings of the Design, Automation and Test in Europe (DATE)*, March 2005, pp. 18-23.

[25] F. Sun, S. Ravi, A. Raghunathan, N. K. Jha, "Synthesis of application-specific heterogeneous multiprocessor architectures using extensible processors," *In Proceedings of the International Conference on VLSI Design (VLSID)*, January 2005, pp. 551-556.

[26] J. Yu, G. Lemieux, C. Eagleston, "Vector processing as a soft-core CPU accelerator," *In International Conference on Field Programmable Logic and Applications (FPL)*, September 2008, pp. 222-232.

[27] S. Craven, C. Patterson, P. Athanas, "Configurable Soft Processor Arrays using the OpenFire Processor," *In Military and Aerospace Programmable Logic Devices (MAPLD)*, September 2005

[28] M.A.R. Saghir, M. El-Majzoub, P. Akl, "Datapath and ISA customization for soft VLIW processors," *In IEEE International Conference on Reconfiguurable Computing and FPGAs (ReConFig)*, September 2006, pp. 1-10.

[29] D. Sheldon, R. Kumar, F. Vahid, D. Tullsen, R. Lysecky, "Conjoining soft-core FPGA Processors," *In IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, November 2006, pp. 694-701.

[30] S. Kirkpatrick, J. C.D. Gelatt, and M. Vecchi. "Optimization by Simulated Annealing," *In Science,* May 1983

[31] P. Yiannacouras, J. Rose, J. Gregory Steffan, “The Microarchitecture of FPGA-Based Soft Processors,” *In International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, September 2005, pp. 202-212.

[32] F. Sun, S. Ravi, A. Raghunathan, N.K. Jha, “Custom-Instruction synthesis for extensible-processor platforms,” *In IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol 23, Issue 2, December 2002, pp. 216-228.

[33] http://www.model.com/

[34] “Altera Excalibur devices,” http://www.altera.com/products/devices/arm/arm-index.html.

[35] “Xilinx Virtex II Pro,” www.xilinx.com/xlnx/xil_prodcat_landingpage.jsp?title=Virtex-II+Pro+FPGAs

[36] “Nios,” http://www.altera.com/products/ip/processors/nios2/ni2-index.html

[37] “Microblaze,” http://www.xilinx.com/products/design_resources/proc_central/microblaze.htm