

Adaptive Explicitly Parallel Instruction Computing

by

Surendranath Talla

A dissertation submitted in partial fulfillment
of the requirements for
the degree of Doctor of Philosophy
Department of Computer Science
New York University

December 16, 2000

Prof. Krishna Palem

Dissertation Advisor

© Surendranath Talla
All Rights Reserved 2000

To my parents

Acknowledgments

I thank my advisor Prof. Krishna Palem for his guidance, support and patience. This work would not have been possible without his advice and support. I am very grateful to the members of my thesis committee Profs. Robert Dewar, Ben Goldberg, Ed Schonberg, Jack Schwartz and Sudhakar Yalamanchili for their time and advice. Sincere thanks to Profs. P. S. Thiagarajan and Weng-Fai Wong for their detailed comments on the early versions of my dissertation and valuable advice. Thanks to Patrick Devaney and my ReaCT-ILP Laboratory colleagues Jin-woo Kim, Han-soo Kim, Allen Leung, Amit Nene, Igor Petchchanski, Rodric Rabbah for many useful discussions. Many thanks to Zhe Alex Ye from Chameleon Systems for taking the time to explain to me his work on Chimaera Reconfigurable Functional Unit which was very helpful for my experimental work. Many thanks to Anina Karmen for the numerous times she went out of the way to help me.

This research was supported in part by DARPA under Contract No. DABT63-96-C-0049 performed under the USC MAARC project. Part of this research could be conducted due to the availability of Trimaran, an open source ILP compiler infrastructure released by HP Labs, UIUC and NYU. I am grateful to New York University for the Graduate Teaching and Research Assistantships and the Dean's Dissertation Fellowship Award.

This work has been inspired in part by the extraordinary and pioneering work on EPIC architectures and compilation techniques done by the Compiler and Architecture Research group led by Dr. B. R. Rau and his colleagues at HP Labs in Palo Alto. I am very fortunate to have learnt my first lessons on ILP compilation from Dr. Scott Mahlke. I thank them for hosting me as a research intern for one beautiful summer.

I would also like to thank my friends and fellow students: Saugata Basu, Pravin Bhagwat, Raoul-Sam Daruwala, Charutosh Dixit, Deepak Goyal, Karp-joo Jeong, Jin-woo Kim, Han-soo Kim, Allen Leung, Amit Nene, Maria Papadopouli, Igor Petchchanski, Rodric Rabbah, Kanna Rajan, Ajay Rajkumar and Roman Yangarber for all that I learnt from them, and making college life a memorable experience. Special thanks to Ajay, Pravin, Maria and Roman for saving my life many times!

To Amma, Nanna, Andrew, Mallika, Maria Papadopouli, Maria Chelliah and Kavya, for their love and faith in me, words are inadequate to express my gratitude.

Preface

Given an application program, traditional approaches to improving performance involve one of two approaches—improve the performance either of the microprocessor or of the compiler generated code. Both these approaches are constrained by the fact that they have to conform to a *fixed* interface between the processor and the software that executes on it—the Instruction Set Architecture (ISA). The ISA is designed to be suitable for *all* applications that will be targeted to the microprocessor. A fixed ISA offers many advantages: compatibility, uniformity and simplicity. However, due to the very nature of its generality, the instruction mix offered by the ISA need not necessarily be the perfect match for a given application. In addition to the fixed ISA, poor scalability of dynamically scheduled architectures and limited available “traditional” instruction level parallelism has motivated us to look for alternative approaches to improving microprocessor performance on general purpose applications.

In this dissertation, we present a novel class of architectures that allow application programs to add and subtract functional units yielding a *dynamically varying instruction set interface* to the running application. At the core, processors belonging to this architectural class are composed of programmable logic as exemplified by a canonical Field Programmable Gate Array (FPGA). This programmable logic resource forms the basis for yielding such application specific customization. The availability of large amounts of fine-grained parallelism and explicit control over resource allocation in programmable logic has been shown to yield impressive performance gains over a large class of applications. However, programmable logic devices such as FPGAs demand a much lower level of control over micro-architectural elements compared to what a traditional RISC processor allows. This extra degree of control has made it enormously difficult to program these machines. To quote from the HICSS’97 position paper [117] written by my thesis advisor: “*a primary barrier in this regard is the absence of programming tools and software support to eventually compile algorithms implemented in standard and widely-used languages such as C onto the hardware platforms.*” In the same paper, a need for optimizing programming tools and compilation support and for real-time embedded system support is stressed. These observations form the basis for our research approach—to investigate architectures that allow rapid and efficient customization and present convenient abstractions so that effective compiler techniques may be developed to enable such customizations without compromising the performance potential of programmable logic.

In the past, several “reconfigurable” processor architectures based on programmable logic have been proposed [174, 178, 127, 52, 58, 65, 80]. These machines were also designed to be configured in an application specific manner to improve application performance. However, most of these efforts placed great emphasis at the micro-architectural level without any thought into how they would be programmed. We believe that this *bottom-up* approach is one of the main reasons why it has not been possible to develop compilers that can efficiently target these machines. This is the reason why these machines have largely remained as “exotic toys”, programmed manually, confined to research laboratories, instead of gaining acceptance as a viable processor technology for general purpose applications.

In contrast, this thesis takes a *top-down* approach that starts with an abstract model of a “reconfigurable processor”, whose computation can be related to a known computation model for which efficient compilation techniques are understood, and then successively refines it. *The guiding principle at each stage of the refinement being that of exposing the flexibility offered by programmable logic as much as possible without sacrificing the ability to compile efficiently to such a machine.* A road-map indicating this research approach was presented by us in [90]. The proposed abstract model called *Adaptive Instruction Level Parallel (AILP) Processing* describes a class of processor architectures whose data-paths allow multiple instructions to be processed on each cycle and allow application programs to dynamically alter the functional unit composition of the data-path of the processor using

the programmable logic resources provided by the base hardware.

In this dissertation, we present a detailed description of a subset of the AILP space called the *Adaptive Explicitly Parallel Instruction Computing (AEPIC)* architectures, whose definition represents a collection of ideas intended to enable **efficient reconfiguration of processor data-paths**. While AEPIC processor reconfiguration is affected by the executing program at runtime, the decisions of when and how to reconfigure are determined by the compiler and embedded in the application's executable. Our initial ideas and results from preliminary investigations in this direction were presented in [91].

AEPIC architecture is motivated by a desire to combine the advantages of the EPIC style of architectures (simpler architectures, known compilation technology) and those of programmable logic (fine-grained parallelism, explicit control over micro-architectural features). We believe that these architectures are at the right level of granularity for automatic compilation (unlike many of the purely FPGA based machines) and yet yield many of the performance benefits of programmable logic. This is evidenced by the similarity between the compilation techniques targeting conventional ILP architectures and the ones we have proposed for AEPIC architectures. Our preliminary results also indicate that these architectures are worthwhile direction to pursue.

Contents

Dedication	iii
Acknowledgment	iv
Abstract	xiv
1 Introduction	1
1.1 Generic Processors, Application Specific Architectures	1
1.1.1 Fixed Instruction Set Architectures	1
1.1.2 Compiler Specified, Dynamic Instruction Set Architectures	2
1.2 An Illustrative Example	3
1.2.1 A Speech Signal Codec	3
1.2.2 Performance On Traditional Architectures	6
1.2.3 Compiler Synthesized Instructions And Potential Benefits	7
1.3 Goals, Challenges, Approach	10
1.3.1 Goal	10
1.3.2 Challenges	11
1.3.3 Overview And Motivation For Our Approach	12
1.4 Summary Of Main Contributions	14
1.5 Organization Of This Dissertation	14
2 Background	17
2.1 Introduction	17
2.2 Definitions/Terminology	17
2.3 Programmable Logic Device: Issues	19
2.3.1 Dimensions of Reconfigurability	20
2.3.2 Programmable Logic Device Key Parameters	22
2.3.3 Programmable Logic Device Taxonomy	24
2.4 System Level Issues	25
2.5 Application Domain	26
2.5.1 Desirable application characteristics	26
3 Related Work	29
3.1 History	29
3.2 Application Studies	29
3.3 A Survey of the State of the Art	31
3.3.1 Programmable Reduced Instruction Set Computer (PRISC)	31
3.3.2 Gate Array Reconfigurable Processor (GARP)	33

3.3.3	Dynamic Instruction Set Computer (DISC)	35
3.3.4	Programmable Active Memories (PAM)	37
3.3.5	Reconfigurable Pipelined Data-path (RaPiD)	42
3.3.6	Cached Virtual Hardware (CVH), PipeRench	43
3.3.7	Chimaera	44
3.3.8	Reconfigurable Architecture Workstation (RAW)	46
3.3.9	Others	47
3.4	Summary	48
4	Adaptive Instruction Level Parallel Processing	51
4.1	Instruction Level Parallel Processing	51
4.1.1	Background	51
4.1.2	Limitations of current approaches to ILP	53
4.2	Dynamic Instruction Set Architectures	54
4.3	Adaptive Instruction Level Parallel Processing	55
4.3.1	AILP processing	55
4.3.2	AILP Machine Model	57
4.3.3	AILP Taxonomy	59
4.4	Adaptive Explicitly Parallel Instruction Computing	61
4.4.1	The C_{AEPIC} Class	62
4.5	Summary	62
5	Adaptive Explicitly Parallel Instruction Computing (AEPIC)	63
5.1	Dynamic Instruction Set Architectures: Issues	63
5.1.1	Long Reconfiguration Times	63
5.1.2	Large And Non-uniform Configuration Sizes	64
5.1.3	Context Switching Overheads	64
5.1.4	Modular Software Development	65
5.1.5	Large Variation In Operation Formats	65
5.1.6	Non-deterministic Latencies	65
5.1.7	Other Architecture Desirables	65
5.2	Adaptive Explicitly Parallel Instruction Computing	66
5.2.1	AEPIC Computation Model	66
5.2.2	Machine Model	67
5.2.3	The Adaptive Extension	67
5.2.4	The EPIC Core	68
5.2.5	Summary Of Key Features	69
5.3	Details Of The Architecture	71
5.3.1	Multi-context Reconfigurable Logic Arrays	71
5.3.2	Configuration Register File	73
5.3.3	Register Files	74
5.3.4	Instruction And Data Memory Hierarchy	75
5.3.5	Configuration Memory Hierarchy	76
5.4	Instruction Set	77
5.4.1	EPIC ISA	77
5.4.2	Adaptive Extension ISA	77
5.5	Architectural Parameters	79
5.5.1	MRLA Parameters	79

5.5.2	Memory System Parameters	79
5.5.3	Fixed Core Parameters	79
5.6	Additional Notes	79
5.6.1	Multi-cluster Machines	79
5.6.2	Heterogeneous Machines	80
5.6.3	Interrupts And Exceptions	80
5.7	Summary	80
6	Compiling For AEPIC Targets	81
6.1	A Basic Compilation Framework For AEPICs	81
6.2	Partitioning	82
6.2.1	Partitioning Considerations	83
6.2.2	A General Framework For Code Partitioning	84
6.3	Instruction Synthesis	85
6.3.1	Instruction Synthesis Techniques	85
6.4	Configuration Selection	85
6.5	Configuration Allocation	86
6.5.1	Configuration Allocation Problem	86
6.5.2	Similarities With Register Allocation	86
6.5.3	Machine Model	87
6.5.4	Cost Model	88
6.5.5	Simplified AEPIC Allocation Model	88
6.5.6	Allocation Techniques Background	89
6.5.7	Interference	89
6.5.8	Spilling And Splitting Of Live Ranges	91
6.5.9	Pruning For Configuration Allocation	91
6.5.10	Graph Multi-coloring Configuration Allocator (GMCA)	93
6.5.11	Effect of procedure linkage conventions	95
6.5.12	Hardware support for allocation	96
6.6	Instruction Scheduling	97
6.6.1	AEPIC Features Relevant To Scheduling	99
6.6.2	Scheduling Model	99
6.6.3	The Resource Constrained AEPIC Scheduling Problem	100
6.6.4	Some Complexity Results	101
7	AEPIC Simulation	103
7.1	Introduction	103
7.1.1	Simulator Requirements	103
7.1.2	AEPIC Simulator Design Goals	103
7.2	Trimaran's EPIC Simulator	104
7.2.1	Overview	104
7.2.2	Simulator Code Generation Process	104
7.2.3	Structure Of The Simulator	105
7.2.4	The Interpreter	106
7.2.5	Handling Function Calls	107
7.2.6	Performance Monitoring Library	109
7.3	Design Of The AEPIC Simulator	109
7.3.1	Representing AEPIC Processor State	109

7.3.2	Simulation Of Instruction Execution	113
7.3.3	AEPIC Simulation Framework	117
7.4	Summary	119
8	Performance Evaluation	121
8.1	Introduction	121
8.2	Application Domain	122
8.3	Methodology	124
8.3.1	Compilation Environment	124
8.3.2	Machine Configurations	127
8.3.3	Experiments	128
8.4	Results And Discussion	129
8.4.1	Application Characteristics	129
8.4.2	Standard ILP Processor Performance	130
8.4.3	Hot-spot Distribution	131
8.4.4	Performance on $AEPIC_{\infty}$	133
8.4.5	Performance on $AEPIC_{2221-2C}$	134
8.5	Summary	135
9	Concluding Remarks	137
A	AEPIC Architectural Parameters	139
B	AEPIC Instruction Semantics	141
C	AEPIC ISA Summary	145
C.1	Notes On Instruction Format	145
C.2	Adaptive Extension Instructions	145

List of Figures

1.1	Fixed interface architectures	2
1.2	Compiler specified architectures	3
1.3	ADPCM decoder cycle counts	6
1.4	Basic block execution profile	6
1.5	ADPCM program hot-spots	7
1.6	IR and circuit for line 1	7
1.7	IR and circuit for lines 2-3	8
1.8	IR and circuit for lines 4-5	9
1.9	IR and circuit for lines 10-11	10
1.10	IR and circuit for lines 6-9	11
1.11	Reduction in cycle counts due to application specific instructions	12
1.12	Configuration/code size for IDCT	12
1.13	Control, parallelism and compilability	13
2.1	General purpose processor model	18
2.2	Generic programmable logic device	19
2.3	Customization Points of A Reconfigurable Device	20
2.4	A Classification of Programmable Logic Devices	21
2.5	reconfig-dimensions-figure	23
2.6	Processor-Reconfigurable Coupling Choices	25
3.1	Reconfigurable Computing Time-line	30
3.2	PRISC Data-path	31
3.3	PRISC PFU	32
3.4	Basic Organization of GARP Processor	33
3.5	GARP Array	36
3.6	The DISC System	37
3.7	DISC Execution Flowchart	38
3.8	PAM Array	39
3.9	A PAM System	41
3.10	RAPID Cell	42
3.11	RaPiD System	43
3.12	CVH Architecture	44
3.13	Chimaera Architecture	45
3.14	Chimaera Cell	46
3.15	RAW	47
3.16	RAW Tile	48
3.17	RAW Computing System	49

4.1	ILP computation	52
4.2	ILP machine model	52
4.3	ILP processing steps	52
4.4	ILP taxonomy	53
4.5	AILP Computation	55
4.6	AILP high-level interface	55
4.7	AILP processing steps	56
4.8	AILP machine state	56
4.9	AILP machine state and instructions	57
4.10	AILP taxonomy	60
4.11	Dynamic instruction set architecture space	62
5.1	Configuration/code size for IDCT	64
5.2	CFU context switching	64
5.3	Effect of function calls	65
5.4	AEPIC executable and abstract data-path	67
5.5	AEPIC machine model	68
5.6	Structure of MRLA	71
5.7	MRLA multiple contexts	71
5.8	CFUs on MRLA	72
5.9	Configuration register file and associated configurations	74
5.10	Configuration cache memory hierarchy	76
6.1	NP-hard problems from scheduling	101
6.2	Polynomially Solvable Instances	101
7.1	EPIC simulator generation in Trimaran	105
7.2	Read/write intervals of operands	113
7.3	AEPIC instruction execution	115
7.4	AEPIC simulator components	117
8.1	Components of compiler infrastructure	124
8.2	Compilation steps	126

List of Tables

2.1	Taxonomy of Reconfigurable Devices	21
2.1	Taxonomy of Reconfigurable Devices	22
2.2	Taxonomy Based on Key Parameters	24
3.1	GARP - Array Specific Instructions	34
4.1	AILP instruction set	58
4.1	AILP instruction set	59
4.2	A few AILP architectural subclasses	61
6.1	Instructions for configuration management	87
7.1	Adaptive extension instructions	116
8.1	Applications used for performance evaluation	122
8.2	Machine configurations	127
8.3	Application sizes	129
8.4	Dynamic opcode distribution	129
8.4	Dynamic opcode distribution	130
8.5	Performance on 9-issue EPIC processor	130
8.6	Hot-spot distribution	131
8.7	Examples of candidates for partitioning in various applications	132
8.8	AEPIC with FPGA array	134
8.9	AEPIC with Chimaera RFUs	134
A.1	Architecture parameters	139
B.1	Semantics Of Adaptive Extension Instructions	141
C.1	Adaptive extension ISA	146

Abstract

Current processors are programmed through a fixed interface called the Instruction Set Architecture (ISA). Consequently, a compiler targeting such a processor is forced to choose instructions from the provided instruction set while generating code for a given application. Often this instruction set is not a suitable match for the computational requirements of the application program. With in this context, we ask ourselves the following questions.

1. *Can application performance be improved if the compiler had the freedom to pick the instruction set on a per application basis?*
2. *Can we build cost-effective processors that provide the ability to efficiently emulate compiler determined instruction sets and yet are **not application specific**?*
3. *Given that the desired processor capabilities are feasible, can the compiler determine an optimal set of instructions for a given application and generate code that can effectively exploit the processor capabilities?*

In this thesis, we provide sufficient evidence to answer these questions in the affirmative. Through a combination of architectural innovations and novel compilation techniques, this dissertation demonstrates that it is possible to attain significant improvement in performance, up to an order of magnitude in some cases, on general purpose and multimedia applications over comparable fixed ISA processors.

We propose classes of microprocessors that allow application programs to add and subtract functional units yielding a dynamically varying instruction set interface to the running application without compromising current compatibility model.

First half of this dissertation describes this novel class of architectures, focusing on a specific subclass called *Adaptive Explicitly Parallel Instruction Computing (AEPIC)* architectures whose definition represents a collection of ideas intended to enable efficient reconfiguration of processor data-paths. While AEPIC processor reconfiguration is affected by the executing program at runtime, the decisions of when and how to reconfigure are determined by the compiler and embedded in the application's executable.

In the second half, a compilation framework targeting AEPIC processors is proposed. Several key compilation problems that need to be addressed in order to target AEPIC processors such as partitioning, instruction synthesis, configuration selection, resource allocation and scheduling are defined and efficient techniques for solving them are proposed for several of them.

Finally, we describe the design of a simulation and performance monitoring framework for AEPIC architectures. How such architectures can be used to improve application performance is demonstrated using a set of programs from the SPEC and MediaBench benchmarks. Experimental results indicate the significant role architectural features of AEPIC processors play in masking the overheads of micro-architectural reconfiguration.

Chapter 1

Introduction

“...I believe that to keep growing single-processor performance, we’ll have to add and subtract functional units, special memories, registers, and customization in a way that current compatibility model can’t live up to, and in a way that current compiler technology can’t yet handle.” –Joseph A. Fisher, 1997 [48]

1.1 Generic Processors, Application Specific Architectures

Through a combination of architectural innovations and novel compilation techniques, this dissertation demonstrates that it is possible to attain significant improvement in performance, up to an order of magnitude in some cases, on general purpose and multimedia applications over comparable processors using known compilation techniques.

The proposed architectures describe classes of simple microprocessors that allow application programs to add and subtract functional units yielding a dynamically varying instruction set interface to the running application without compromising current compatibility model.

First half of this dissertation describes this novel class of architectures, focusing on a specific subclass called *Adaptive Explicitly Parallel Instruction Computing (AEPIC)* architectures, whose definition represents a collection of ideas intended to enable efficient reconfiguration of processor data-paths. While AEPIC processor reconfiguration is affected by the executing program at runtime, the decisions of when and how to reconfigure are determined by the compiler and embedded in the application’s executable.

In the second half, a compilation framework targeting AEPIC processors is proposed. Several key compilation problems that need to be addressed in order to target AEPIC processors such as partitioning, instruction synthesis, configuration selection, resource allocation and scheduling are defined and efficient algorithms are proposed for several of them. Finally, the design of a simulation and performance monitoring framework is described. Experimental results using several applications from SPEC and MediaBench benchmarks establish our thesis.

1.1.1 Fixed Instruction Set Architectures

*A microprocessor’s Instruction Set Architecture (ISA) can be viewed as a contractual interface between the set of programs that are written for an architecture and the set of processor implementations of that architecture.*¹ The

¹B. R. Rau [126]

usual view is that this ISA is fixed and that these processors are manufactured with one particular implementation that realizes this ISA and the program codes are required to conform to this ISA (Figure 1.1). A consequence of these fixed ISA processors is that a compiler targeting such a processor is forced to choose instructions from the fixed instruction set while generating code for a given application. Often the instruction set is unsuitable for the computational requirements of the application program. For example, an instruction set which does not include an “efficient” Multiply-Accumulate (MAC) instruction is a poor match for digital signal processing applications which rely heavily on MAC operations. Instruction set requirements change with changing application domains. Current methods for accommodating the instruction set requirements of new application domains primarily follows the approach of extending the existing instruction sets. This is exemplified by the multi-media extensions to most commercial architectures (for e.g., the MMX extensions to IA-32 [120]). Although motivated by an important need to maintain compatibility with code generated for earlier instruction sets, this trend of extending instruction sets contributes to the already bloated instruction sets. The benefits are questionable [17] and processors are getting increasingly complex leading to added compilation difficulties and ever more expensive processors [22].

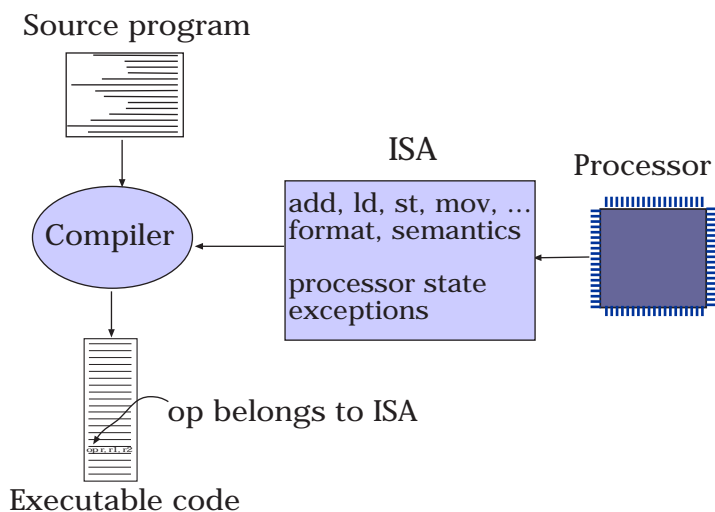


Figure 1.1: Fixed interface architectures

1.1.2 Compiler Specified, Dynamic Instruction Set Architectures

In contrast to fixed instruction set architectures, a *compiler specified, dynamic instruction set architecture* defines a class of processors whose instruction set interfaces are determined by the compiler and may be changed (from the point of view of the executing program) at runtime by the executing process. A compiler that generates code for such a processor takes a source program as input and generates (a) an executable version of the input program and, (b) a description of the architecture expected by the instructions in the executable (see Figure 1.2). This description of the architecture not only specifies the interface (names, types and formats of instructions, etc) but also contains information about how the processor should be reconfigured to efficiently emulate those instructions. It is likely that this style of processing simplifies processor designs since they are not hardwired with complex functional units, and, improves application performance since the eventual “configured” (by the compiler) micro-architecture is tuned to the needs of the application. Within this context, we ask ourselves the following questions.

1. Can we build cost-effective processors that provide the ability to efficiently emulate compiler synthesized instruction sets and yet are not application specific?

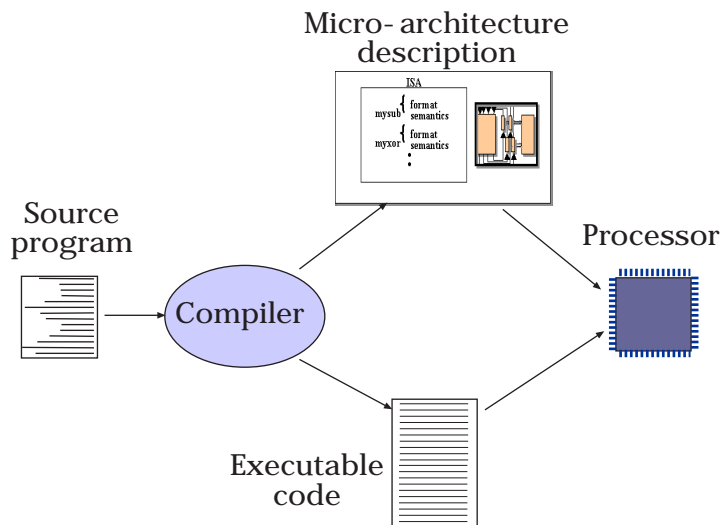


Figure 1.2: Compiler specified architectures

2. Given that processor capabilities are feasible, can the compiler determine an optimal set of instructions for a given application?
3. Once an instruction set is determined, can the program be translated into efficient code composed of instructions from the synthesized instruction set, possibly interspersed with instructions to *reconfigure* the processor to emulate the synthesized instructions?
4. Can the compilation task be performed in reasonable times?

These are some of the questions that form the basis for our research effort.

1.2 An Illustrative Example

1.2.1 A Speech Signal Codec

Here, we motivate our research effort through an application. This application is a speech codec (coder/decoder) based on Adaptive Differential Pulse Code Modulation (ADPCM) [123]. ADPCM is a form of Pulse Code Modulation (PCM) that produces a lower bit rate by recording only the difference between samples and adjusting the coding scale dynamically to accommodate large and small differences. A common implementation takes 16-bit linear PCM samples and converts them to 4-bit samples, yielding a compression rate of 4:1. Some applications such as telephony over ISDN or desktop video conferencing systems use ADPCM to digitize and compress a voice signal so that voice and data can be transmitted simultaneously over a digital facility normally used only for one or the other.

The “C” program for the decoder part of the speech codec is shown in Program 1. ADPCM code is the Intel/DVI ADPCM code, an implementation of the algorithm from the IMA Compatibility Project proceedings, Vol 2, Number 2; May 1992, obtained from the MediaBench benchmark suite [93]. The `adpcm_decoder` function takes *len* bytes of compressed speech data passed via the *indata* parameter and writes the uncompressed speech data to the *outdata* parameter.

Program 1 ADPCM Decoder

```
/*step variation table */
static int indexTable[16] = {-1,-1,-1,-1,2,4,6,8,-1,-1,-1,-1,2,4,6,8};
static int stepsizeTable[89] =
    {7,8,9,10,11,12,13,14,16,17,19,21,23,25,28,31,
    34,37,41,45,50,55,60,66,73,80,88,97,107,118,130,143,
    157,173,190,209,230,253,279,307,337,371,408,449,494,
    544,598,658,724,796,876,963,1060,1166,1282,1411,1552,
    1707,1878,2066,2272,2499,2749,3024,3327,3660,4026,
    4428,4871,5358,5894,6484,7132,7845,8630,9493,10442,
    11487,12635,13899,15289,16818,18500,20350,22385,24623,
    27086,29794,32767 };
```

void adpcm_decoder

```
(char* indata, short* outdata, int len,struct adpcm_state state)
{
    int sign, delta, vpdiff, index, inputbuffer, bufferstep = 0;
    signed char *inp = (signed char *)indata; short *outp = outdata;
    int valpred = state->valprev;
    int index = state->index;
    int step = stepsizeTable[index];
    ...
}
```

Program 2 ADPCM Decoder (contd.)

```
for ( ; len > 0 ; len- )
{
    if ( bufferstep ) delta = inputbuffer & 0xf;
    else { inputbuffer = *inp++; delta = (inputbuffer >> 4) & 0xf;}
    bufferstep = !bufferstep;
1   index += indexTable[delta];
2   if ( index < 0 ) index = 0;
3   if ( index > 88 ) index = 88;
4   sign = delta & 8;
5   delta = delta & 7;
6   vpdiff = step >> 3;
7   if ( delta & 4 ) vpdiff += step;
8   if ( delta & 2 ) vpdiff += step>>1;
9   if ( delta & 1 ) vpdiff += step>>2;
10  if ( sign ) valpred -= vpdiff;
11  else valpred += vpdiff;
12  if ( valpred > 32767 ) valpred = 32767;
13  else if ( valpred < -32768 ) valpred = -32768;
14  step = stepsizeTable[index];
    *outp++ = valpred;
}
state->valprev = valpred;
state->index = index;
}
```

1.2.2 Performance On Traditional Architectures

First we look at the performance of the ADPCM codec on a wide-issue EPIC processor using state-of-the-art ILP compilation techniques. The EPIC processor used for the experiment is a 9-wide (9 functional units comprised of 4 integer, 2 floating point, 2 memory and 1 branch units) HPL-PD EPIC processor [84]. The Trimaran ILP compiler [70] was used to target this machine. A summary of the performance is shown in Table 1.3. The first three columns show the total number of processor cycles consumed to compute the two functions of the program, using three different region formation methods: basic blocks [5], superblocks [105] and hyperblocks [103]. The final column shows the total cycle counts when there are no processor resource constraints (meaning, an infinite number of functional units of each type are assumed to exist on the processor) under the basic block region formation method.

Function	B_{4221}	S_{4221}	H_{4221}	B_{∞}
main	2274	3829	1975	1975
adpcm_decoder	5706136	3857210	5706136	3821956
Total cycles	5708410	3861039	5708111	3823931

Figure 1.3: ADPCM decoder cycle counts

Let us take a closer look at how the total cycles are distributed across the code. The basic block level execution profile is shown in Table 1.4. The **BB#** column in the table gives the basic block number, the **DynCyc** column gives the total number of cycles consumed by that basic block and the **SL** column gives the length of the static schedule for that basic block.

BB#	DynCyc	SL	BB#	DynCyc	SL	BB#	DynCyc	SL
1	444	3	10	590080	4	19	295040	2
2	1184	8	11	57148	2	20	0	2
3	295040	2	12	442560	3	21	295040	2
4	147520	2	13	117902	2	22	0	2
5	368800	5	14	442560	3	23	0	0
6	1032640	7	15	130422	2	24	590080	4
7	13372	2	16	295040	2	25	444	3
8	295040	2	17	140874	2	26	740	5
9	0	2	18	154166	2			

Figure 1.4: Basic block execution profile

Let us now consider the most compute intensive regions of the source code. Mapping the basic block level execution profile (Table 1.4) back to the source code level, we can identify the “hot-spots” (compute intensive regions) at the source code level of the program. Some of these hot-spots are numbered in the source code (lines numbered 1 through 14 in the `adpcm_decoder` function shown above). The total cycles consumed by these lines of the code and their individual contribution towards the total computation time (in percentage) are shown in Table 1.5. In all these 14 lines contribute to about 84% of the total execution time of the program. Note: In the second column of Table 1.5, the numbers in parentheses denote that the source code in that row (from the first column) contributes to only that many cycles in that basic block per execution of the basic block.

Source lines	Basic blocks	Cycles per iteration	Total cycles	Percent of total time
1	6(4)	4	590080	10.3
2-3	6(2),7,8,9	5.1	750972	13.2
4-5	10(2)	2	295040	5.2
6-9	10(2),11,12,13,14,15,16(2)	11.1	1633152	28.6
10-11	16(1),18(1),17	2.5	365677	6.4
12-13	19,20,21,22	4	590080	10.3
14	24	4	590080	10.3

Figure 1.5: ADPCM program hot-spots

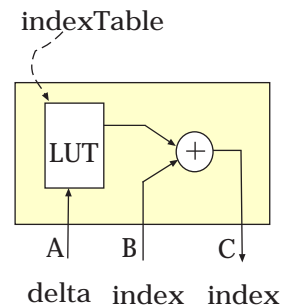
1.2.3 Compiler Synthesized Instructions And Potential Benefits

From the program “hot-spots” Table 1.5, it seems that the average number of cycles consumed to compute the source lines marked 1-14 per iteration is higher than what one would expect. Let us examine the marked code sections individually to see what we mean.

`index += indexTable[delta]`

`;; r1: delta; r5: index`

```
shl r2, r1, 2
add r3, r2, <indexTable>
ld r4, r3
add r5, r5, r4
```



(a)

(b)

Figure 1.6: IR and circuit for line 1

Line 1 The EPIC assembly code for this source line is shown in Figure 1.6(a). For each execution of line (1), 4 processor cycles are consumed assuming each assembly instruction can be computed in a single cycle. However, note that the *indexTable* is a static table of 16 constant values (just 5 unique values). Now, consider the circuit in Figure 1.6(b). The look-up table (LUT) stores the 16 values of the *indexTable*. Given the inputs *delta* and *index*, this circuit computes the new value of *index* through a simple look-up and addition operation. This combined operation can be performed in a single cycle in current semiconductor (0.18μ BiCMOS, 5 metal layer) technology. Note that this circuit also reduces the memory traffic avoiding potential cache misses since the memory reads (*ld* operation in the assembly code) are eliminated.

Lines 2-3 The EPIC assembly code for this source line is shown in Figure 1.7(a). For each execution of lines

```

if (index < 0) index = 0;
if (index > 88) index = 88;

```

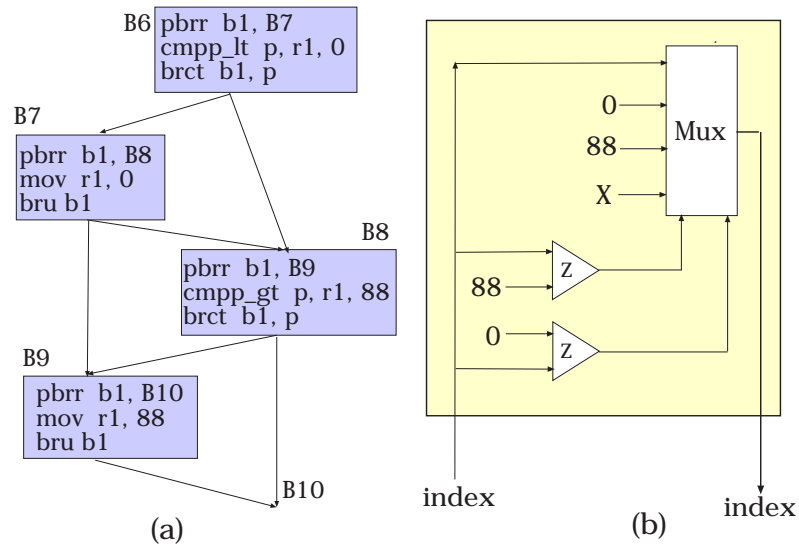


Figure 1.7: IR and circuit for lines 2-3

marked 2-3, on an average 5.1 processor cycles are consumed. However, these two *if* statements basically saturate the value of *index* variable, an operation that can be performed by a simple circuit composed of a 4x1 multiplexer and two comparators as shown in Figure 1.7(b). This circuit too can compute the new value of *index* in a single cycle.

Lines 4-5 Lines marked 4-5 extract the sign and magnitude of *delta* and assigns the values to variables *sign* and *delta*. Unless an extra register is used to keep a duplicate copy of *delta*, these two lines require a minimum of 2 cycles on any processor. Assembly for HPL-PD EPIC processor is shown in Figure 1.8(a). It is easy to see that the simple “bit-extract” circuit in Figure 1.8(b) can perform the task in a single cycle.

Lines 6-9 The assembly code shown in the form of an intermediate representation (IR) for these lines is shown in Figure 1.10(a). On an average this section of the code consumes 11.1 cycles for each iteration of the loop. Even in the best case (code generated manually), when (a) all the if conditions are computed in parallel, (b) the shifts are performed in parallel, and (c) the updates to *vpdiff* are performed in an optimal manner, the number of cycles required would still be 6. However, the circuit in Figure 1.10(b) can perform this computation in 3 cycles in the worst case, although it is quite possible that the number of cycles can be further reduced. Note that the *enable* signal if “on” would allow the addition to be performed. If *enable* is “off”, only the right input operand of the adder is propagated to its output.

Lines 10-11 The assembly code and the circuit for this *if* statement is shown in Figure 1.9. Again, the circuit can perform the computation in a single cycle while the EPIC code consumes 2.5 cycles on average per iteration. Note that even with predication, this statement cannot be computed in less than 2 cycles per execution.

Lines 12-13 This *if* statement is identical in structure to the code in lines 2-3. So the expected benefit should be identical to that obtained in the case of lines 2-3.

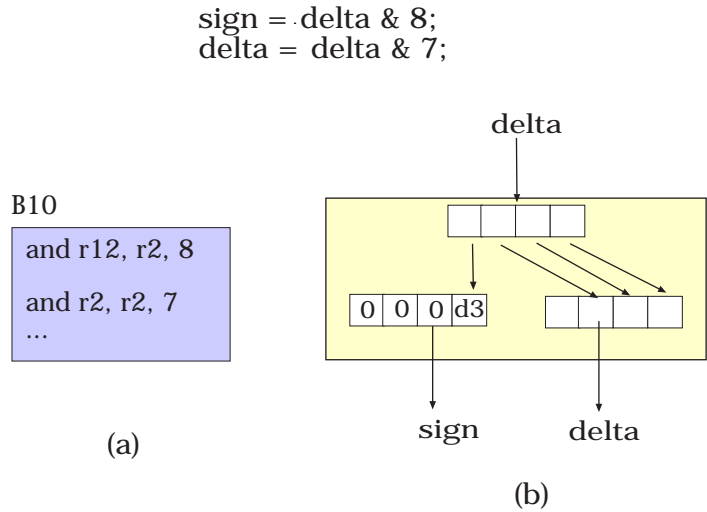


Figure 1.8: IR and circuit for lines 4-5

Line 14 Again, just as in the case of line 1, here too the *stepSizeTable* is a table of constant values. Though the table is somewhat larger, it can still be accommodated inside a functional unit. Here, the look-up table (LUT) stores the 89 values of the *stepSizeTable*. Again, the operation can be performed in a single cycle and there is the added benefit of avoiding memory accesses (after the table is initialized).

Let us for the moment consider that there exists a processor that, in addition to the EPIC instructions also provides a mechanism that allows the compiler to communicate the 7 special instructions that compute the circuits identified in the above description for each of the seven (numbered) segments of the source code. If a compiler were to correctly identify these code sections and map them to the appropriate special instruction, then the total cycle count of the application can be reduced by 3487144 (See Table 1.11). This yields a speedup of 2.57 compared to execution on the B_{4221} 9-issue EPIC processor using basic-block region formation. Even when compared to any another EPIC machine configuration (see Table 1.3), the performance benefits can be substantial. For example, even against an EPIC machine with infinite resources, the speedup is 1.72.

Note that we did not account for the overhead of communicating the “description” of these 7 special instructions to the processor. First, we do not (yet) have an idea of how much information needs to be communicated to the processor to “implement” these instructions intended to be used by this application. Second, it is not clear if the overhead is in addition to the cost of the actual instruction processing (perhaps it could be overlapped with useful processing). However, the gains are quite substantial and clearly warrants further investigation.

Great performance improvements for several other applications using similar methods such as the one used above further motivated us to determine if one can achieve such results *in practice*. Hence we set ourselves the goal of designing processors such that they can be customized for each application in a way most suitable for that particular application but at the same time keeping the basic architecture and processor implementation simple, devoid of any application specific features.

The essence of this dissertation is to show that, indeed, such processors can be built and effectively targeted to yield gains as estimated in the above example.

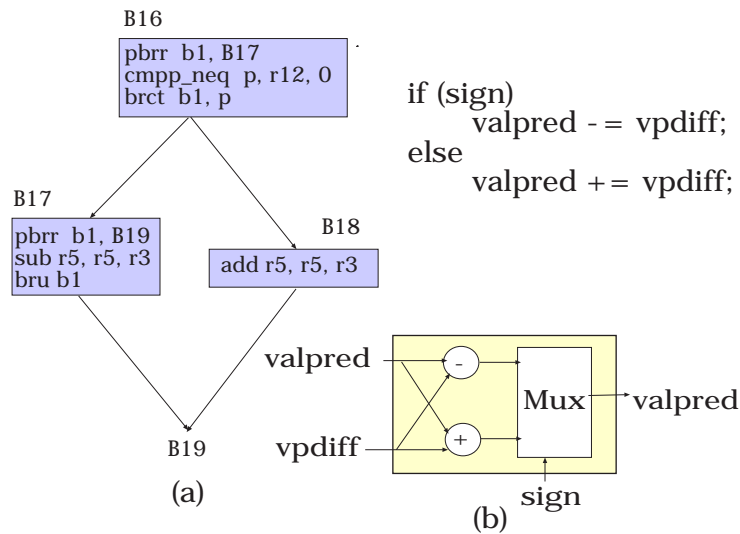


Figure 1.9: IR and circuit for lines 10-11

1.3 Goals, Challenges, Approach

1.3.1 Goal

Our goals are twofold:

1. *Design processors that can be efficiently reconfigured at runtime to emulate compiler synthesized instructions and yet are simple and generic enough to be able to be used as suitable targets for a large class of applications.*
2. *Develop compilation techniques that can synthesize application specific instructions and generate optimal code, code that not only performs the intended computation but also efficiently reconfigures the processor to support the instructions synthesized by the compiler.*

In order for the processor to emulate a compiler synthesized instruction I , the processor needs to provide some sort of programmable hardware resource to the compiler so that such resource can be configured to emulate I . It is likely that this additional flexibility provided by the processor introduces two negative side-effects: (1) in order to accommodate the programmability aspect, additional hardware resources might have to be allocated on the processor die which would otherwise be absent in a processor with hardwired circuits, (2) the extra cost incurred in configuring such programmable resources to emulate the application specific instruction I synthesized by the compiler. Our research was motivated by the belief that these negative side-effects can be overcome by the benefits of application specific customization.

Application specific instruction set synthesis has been attempted before [73, 72, 71, 122, 69, 12, 138, 60]. However, the problem addressed by past efforts is of a different nature than the one we are attempting here. The key difference being, in our case, the processor allows dynamic reconfiguration and hence the instruction set synthesis problem has to also address the costs and benefits of such a capability; not to mention the problem of determining what type of processors support such a capability.

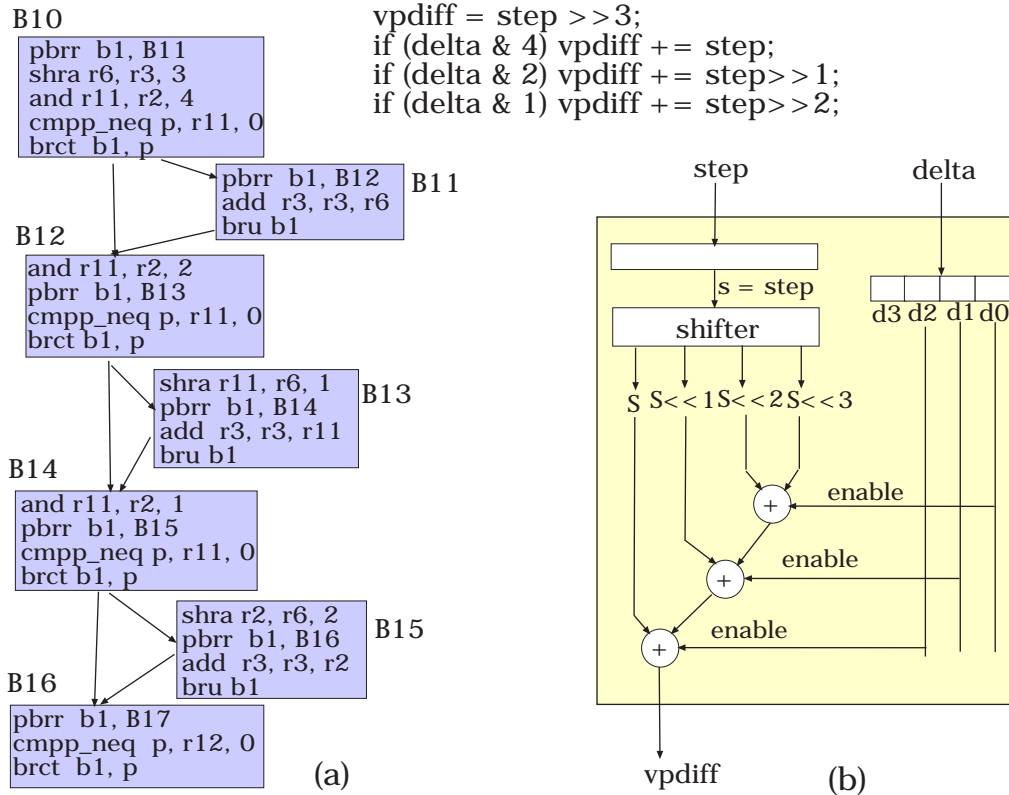


Figure 1.10: IR and circuit for lines 6-9

1.3.2 Challenges

1.3.2.1 Architecture Challenges

Micro-architectures for customization. The key question is what kind of hardware resource is required for a given application domain? FPGA like programmable logic is one candidate. Type of this base resource not only determines the set of computations that can be mapped to it, but also the difficulty in mapping the computation, the size of the “program” to configure the resource to perform the desired computation and the cost of the device.

Overheads of customization. Application specific customization has its benefits. However, the overheads of performing the customization have to be considered. Let us consider an example. Table 1.12 lists the code size required to perform the Inverse Discrete Cosine Transform (IDCT) function - a popular kernel used in several image/video data processing applications. The first column gives the base hardware. The first two rows correspond to a regular EPIC architecture [84] while the last two rows correspond to the FPGAs from Xilinx [180]. The second column gives the performance of the architecture (once it is configured) and the last two columns give the number of instructions and the code size (total size of the instructions). It is clear that the performance on Xilinx architecture is far superior to that on the EPIC processor. However, the overhead of programming the machine is prohibitively high in the case of Xilinx machines. Masking this overhead is a serious concern if the benefits of customization are to affect the final performance.

Source lines	Average EPIC code schedule length	Application specific instruction cycles/iteration	Reduction in total cycles
1	4	1	442560
2-3	5.1	1	600778
4-5	2	1	147520
6-9	11.1	3	1191760
10-11	2.5	1	219406
12-13	4	1	442560
14	4	1	442560
Total reduction in cycle count			3487144
Total cycles consumed			2221266

Figure 1.11: Reduction in cycle counts due to application specific instructions

Architecture	Cycles	Code	Code Size
HPL-PD EPIC	12127	184 ops	<2KB
HPL-PD EPIC	6633	196 ops	<2KB
Xilinx XC4K	544	4920 CLBs	>100Kb
Xilinx Virtex	26	6140 slices	>1Mb

Figure 1.12: Configuration/code size for IDCT

1.3.2.2 Compilation Challenges

Identify, synthesize, use. The key tasks for the compiler in the front-end are to identify the suitable candidates of the input program that may be grouped together and mapped to the programmable resource as an application specific instruction. Having identified such candidates, the next task is to convert them into “instructions” (configurations) that configure the processor to perform that computation and eventually use such configurations during the code translation. These tasks are usually not present in compilers for fixed ISA processors.

Code generation and optimization. Once the desired application specific instructions are synthesized, the compiler back-end needs to generate appropriate code that uses the synthesizes the instructions and code that configures the processor at the right time so that when a certain instruction is executed, the machine is already configured to perform that instruction. The essential task is to reduce the critical path through the program (in other words generate optimal schedules). Unlike traditional compilation, here the compiler has to consider the costs of reconfiguration in performing any of the back-end tasks.

1.3.3 Overview And Motivation For Our Approach

In the past, several “reconfigurable” processor architectures based on programmable logic have been proposed [174, 178, 127, 52, 58, 65, 80]. These processors or machines were also designed to be configured in an application specific manner to reduce computation time. However, most of these efforts placed great emphasis at the micro-architectural level without any thought into how they would be programmed. We believe that this *bottom-up* approach is one of the main reasons why it has not been possible to develop compilers that can efficiently target these machines. This is the reason why these machines have largely remained as “exotic toys”, programmed manually, confined to research laboratories, instead of gaining acceptance as a viable processor technology for

general purpose applications.

In contrast, this thesis takes a *top-down* approach that starts with an abstract model of a “reconfigurable processor”, whose computation can be related to a known computation model for which efficient compilation techniques are understood, and then successively refines it. *The guiding principle at each stage of the refinement being that of exposing the flexibility offered by programmable logic as much as possible without sacrificing the ability to compile efficiently to such a machine.* The proposed abstract model called *Adaptive Instruction Level Parallel* (AILP) Processing describes a class of processor architectures whose data-paths allow multiple instructions to be processed on each cycle and allow application programs to dynamically alter the functional unit composition of the data-path of the processor using the programmable logic resources provided by the base hardware.

Research in instruction-level parallel processing over the past decade has had a great impact both on the architectural front in the form of super-scalar processing, as well as on compiler technology. While the former (fueled also by concerns of compatibility) has led the way, poor scalability of the control unit in the face of increasing parallelism is forcing a trend towards simpler, explicitly parallel ILP architectures with increasing burden being placed on the compiler to expose, enhance and exploit the available ILP. The resulting style of architectures has come to be known as Explicitly Parallel Instruction Computing (EPIC). HPL-PD [84] and the more recent Intel’s IA-64 [42] exemplify the EPIC style. One of the main reasons for the success of traditional RISC style micro-processor architectures is due to the simple interface presented by the control unit. Reconfigurable architectures, as exemplified by a canonical FPGA, demand a much greater (low) level of control over the micro-architectural elements. This extra degree of control has made it enormously difficult to program these machines. The availability of large amounts of fine-grained parallelism and explicit control over resource allocation has been shown to yield impressive performance gains warranting further investigation of suitable programmable logic based processor architectures. By suitability we mean—being able to automatically compile without losing much of the performance gains exhibited through “hand” compilation.

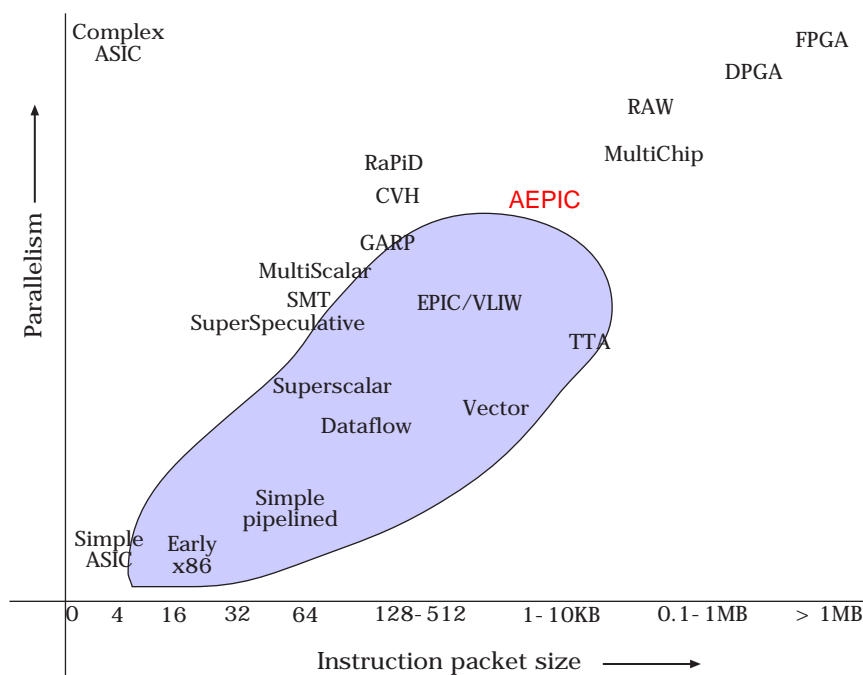


Figure 1.13: Control, parallelism and compability

A subset of the AILP space of architectures called **Adaptive Explicitly Parallel Instruction Computing (AEPIC)** architectures is chosen as the basis for our research infrastructure. AEPIC architecture is motivated by a desire to combine the advantages of the EPIC style of architectures (simpler architectures, known compilation technology) and those of programmable logic (fine-grained parallelism, explicit control over micro-architectural features). Its definition represents a collection of ideas intended to address the problems of micro-architectural data-path reconfiguration.

Figure 1.13 illustrates several known interesting architectures placed in a two dimensional space. The X-axis represents the size of an instruction packet that needs to be issued to initiate processing—it represents the amount of “control” program has over the architecture. FPGA’s and ASIC’s represent the two extremes on this axis - the former exposes the maximum amount of control leading to an extremely large instruction packet size while ASICs are hardwired designs hence are not programmable. The Y-axis represent the amount of parallelism in the machine that is exploitable by the instruction packet. The shaded region within the plot represents a set of architectures for which efficient compilation techniques are known. AEPIC class is a good candidate for research since it is at the boundary of what is known to be efficiently and automatically compilable class of architectures. We believe that these architectures are at the right level of granularity for automatic compilation (unlike many of the purely FPGA based machines) and yet yield many of the performance benefits of programmable logic.

1.4 Summary Of Main Contributions

Key contributions of this dissertation are as follows.

1. The proposal of an abstract model called *Adaptive Instruction Level Parallel* (AILP) Processing that describes a class of processor architectures whose data-paths allow multiple instructions to be processed on each cycle and allow application programs to dynamically alter the functional unit composition of the data-path of the processor using the programmable logic resources provided by the base hardware.
2. A novel, parameterized description of a subset of the AILP space of architectures called **Adaptive Explicitly Parallel Instruction Computing (AEPIC)**. AEPIC architecture is chosen as the basis for our research infrastructure. Its definition represents a collection of ideas intended to address the problems of micro-architectural data-path reconfiguration.
3. In the second part, we present a compilation framework targeting the proposed AEPIC architecture. We define the compilation problems that need to be addressed, such as partitioning, configuration selection, resource allocation and scheduling and present efficient algorithms for several of them.
4. Finally, we describe the design of a simulation and performance monitoring framework for AEPIC architectures. How such architectures can be used to improve application performance is demonstrated using a set of programs from the SPEC and MediaBench benchmarks [93]. Experimental results also indicate how the novel architectural features of AEPIC may be used to mask the runtime overheads of micro-architectural reconfiguration.

1.5 Organization Of This Dissertation

The rest of the dissertation is organized as follows.

Chapter 2 provides the necessary definitions and background on programmable logic and on the structure of machines that contain programmable logic in some form. It gives an insight into the structure and composition, and the key parameters of devices based on reconfigurable logic. An understanding of the

device level parameters discussed here will provide better intuition for more realistic system level models discussed in later sections.

Chapter 3 presents a brief historical perspective of reconfigurable computing research followed by a somewhat encyclopedic survey of the past and ongoing research in reconfigurable computing systems—work that is related to our own.

Chapter 4 defines Dynamic Instruction Set (DIS) architectures. After a brief background on Instruction Level Parallel (ILP) processing, we shift our focus to a subclass of DIS architectures that in addition are also ILP architectures. This subclass is referred to as Adaptive Instruction Level Processing (AILP) architectures. Basic structure and features along with a taxonomy of AILP architectures is presented. AILP space itself is quite large to serve as a starting point for detailed investigation. Hence, a specific subclass of AILP architectures called Adaptive Explicitly Parallel Instruction Computing architectures (C_{AEPIC}) is identified for further investigation.

Chapter 5 describes a specific instance of (C_{AEPIC}) referred to as AEPIC architecture. AEPIC forms the core of our architectural contribution. Defining aspects of AEPIC architecture, key features and the motivation behind them, its computation and machine models are described here. Details such as architectural state, memory hierarchy, instruction set, architectural parameters are also presented.

Chapter 6 presents a basic compilation framework targeting AEPIC processors. Key compilation problems such as partitioning, operation synthesis, configuration selection, configuration allocation and instruction scheduling are defined and efficient algorithms are presented for several of them.

Chapter 7 discusses the design of a simulation and performance monitoring framework—useful components of any compiler research infrastructure targeting AEPIC processors.

Chapter 8 presents the results of our experiments to evaluate the performance of AEPIC processors. Application domain, experimental methodology, compilation environment, machine configurations, are described.

Chapter 9 concludes the dissertation highlighting the achievements and possible future directions.

Chapter 2

Background

2.1 Introduction

Section 2.2 defines reconfigurable logic devices and other terms specifically related to this domain. Section 2.3 gives an insight into the structure and composition, and the key parameters of reconfigurable devices. An understanding of the device level parameters discussed here will provide better intuition for more realistic system level models discussed in later sections. Reconfigurable computing systems could be based purely on such reconfigurable components or on some hybrid mix of reconfigurable and standard processor components. This issue is explored in section 2.4. In section 2.5, we list a class of applications for which reconfigurable computing holds promise of orders of magnitude better performance/cost ratio compared to conventional technologies. In the same section, we examine application characteristics which make them suitable for a “reconfigurable” solution.

2.2 Definitions/Terminology

A simple model for a conventional uni-processor (Figure 2.1) consists of a data-path and a control unit and is externally defined by the *instruction set architecture* (ISA). ISA specifies the interface between the physical device and the software system. Typically, the ISA consists of a set of instructions each of which *configures* the processor hardware through a set of control signals generated by the control unit. These signals enable/disable specific portions of the integrated circuit such as ALU’s (specify the particular operation), registers (select sources and destinations for operands), etc.

Definition 1 *A programmable logic device is an integrated circuit that provides a programmable interface through which the user can dynamically instantiate and emulate almost **any** desired set of hardware images on the available circuit.*

Figure 2.2(a) shows a typical programmable logic device consisting of an array of fine grained processing elements (Figure 2.2(c)) embedded in a mesh of programmable interconnect resource. The interconnect is composed of *data interconnect* and *configuration interconnect* (Figure 2.2(b)). The configuration interconnect is used to route the *configuration* bit-stream. The configuration data (1) decides the functionality of each of the processing elements and (2) establishes the topology of the communication network between the processing elements, by programming the customization points. The *data interconnect* routes data to/from the external pins and the PE’s.

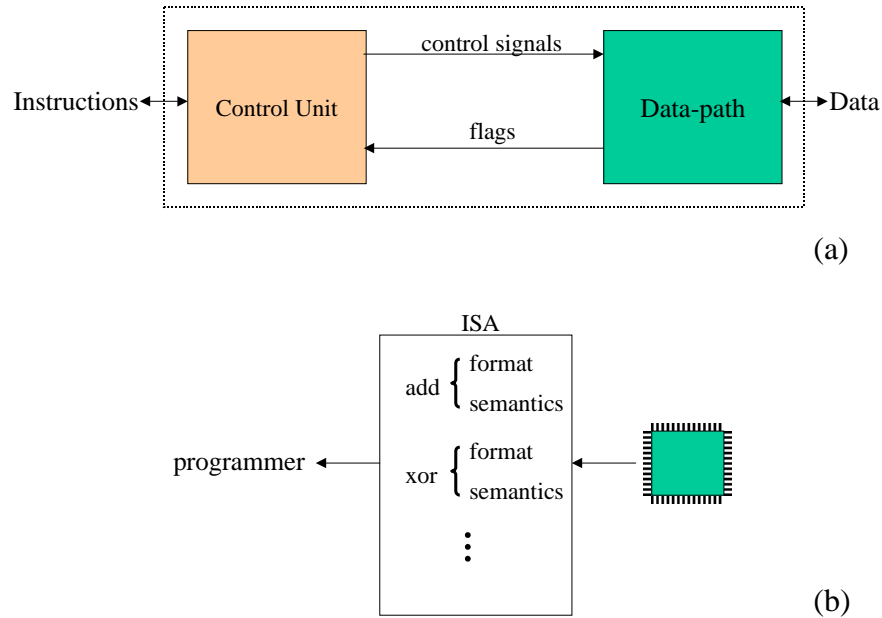


Figure 2.1: General purpose processor model

Programmability of the PE functionality and routing structure is achieved through customization points. In today’s technology, these customization points typically are memory cells (most often SRAM based) or anti-fuses. Anti-fuses are one-time programmable devices (Figure 2.3(a)), which when “blown” create a connection between two conducting materials, and not otherwise. In the case of FPGA’s, the control bit Q (Figure 2.3(b)) can be programmatically set any number of times through the *data* input. Q is typically used as input to a multiplexer.

Definition 2 A **configuration** is defined to be the set of program bits needed to specify the behavior of all the customization points. Alternately, it may also refer to the hardware image realized on the programmable logic device as a result of the customization.

Clearly, the number of configurations that can be specified depends on the number of customization points and the number of valid settings for each customization point. In order to provide the user an extremely large set of configurations to choose from, these devices typically contain

- a large number of independently controllable processing elements and,
- a rich programmable interconnection network.

It is conceivable to create a programmable logic device which does not fit the above description. However, for the sake of simplicity, we consider only those devices that are composed of a *regular array of identical “configurable” processing elements* (PE). All state used for re-timing, like latches, registers, etc., is considered part of the processing elements. We do not consider devices which can self modify their own controllers. Almost all devices proposed till date, research prototypes as well as the commercial ones fall into this category.

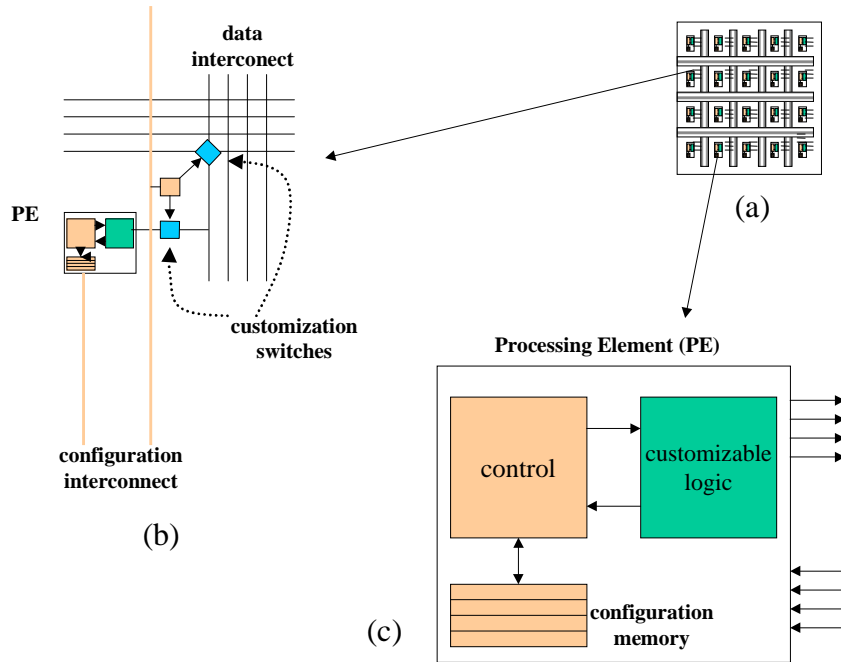


Figure 2.2: Generic programmable logic device

Definition 3 *The ability to reconfigure the device during the lifetime of a single execution is referred to as **run-time reconfigurability**.*

Note that anti-fuse based devices are not run-time reconfigurable; in fact they are only “one-time” configurable.

Definition 4 Partial reconfiguration *refers to the ability to reconfigure only a select portion of the device.*

A vast number of terms have been used in the literature to refer to various kinds of programmable logic devices. A classification of these terms is shown in Figure 2.4. Simple PLD’s are distinguished from High Capacity PLD’s based on capacity - those devices with a logic capacity of 600 gates or less. While CPLD’s have a continuous interconnect resource, FPGA’s are based on segmented interconnect. The segments are joined through programmable switches.

2.3 Programmable Logic Device: Issues

In this section, we present an argument wherein reconfigurability is viewed as composed of three orthogonal dimensions. A taxonomy of reconfigurable devices based these dimensions of reconfigurability is presented. Later, we characterize the space of reconfigurable logic devices through a small set of architectural parameters. This parameter set has a direct relationship to the performance and cost of the corresponding device and would help the designer pick a suitable candidate from the space of reconfigurable devices for a given application domain.

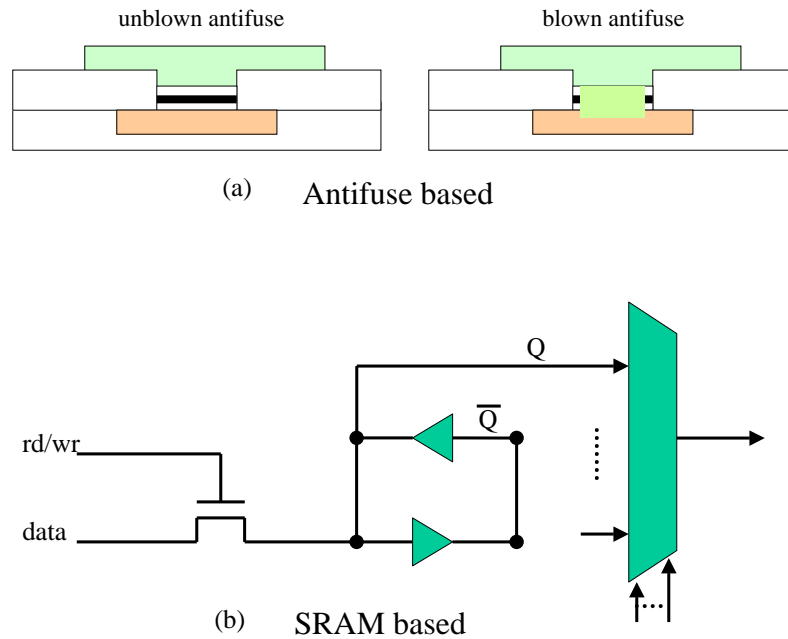


Figure 2.3: Customization Points of A Reconfigurable Device

2.3.1 Dimensions of Reconfigurability

1. **Temporal reconfigurability** refers to the ability to repeatedly reconfigure the device overtime. The two options in this case are run-time reconfigurability and static reconfigurability.

2. **Spatial reconfigurability** concerns the ability to reconfigure select portions of the device. The two choices are partial and/or full reconfigurability.

3. **Type of reconfigurability** refers to the type and the extent to which different elements of the device can be reconfigured. For example, in a particular reconfigurable device only the processing elements might be reconfigurable while the interconnect is not configurable.

Figure ?? gives a pictorial representation of the three dimensions of reconfigurability. In Table 2.1 we give a taxonomy of the reconfigurable devices based on these three dimensions.

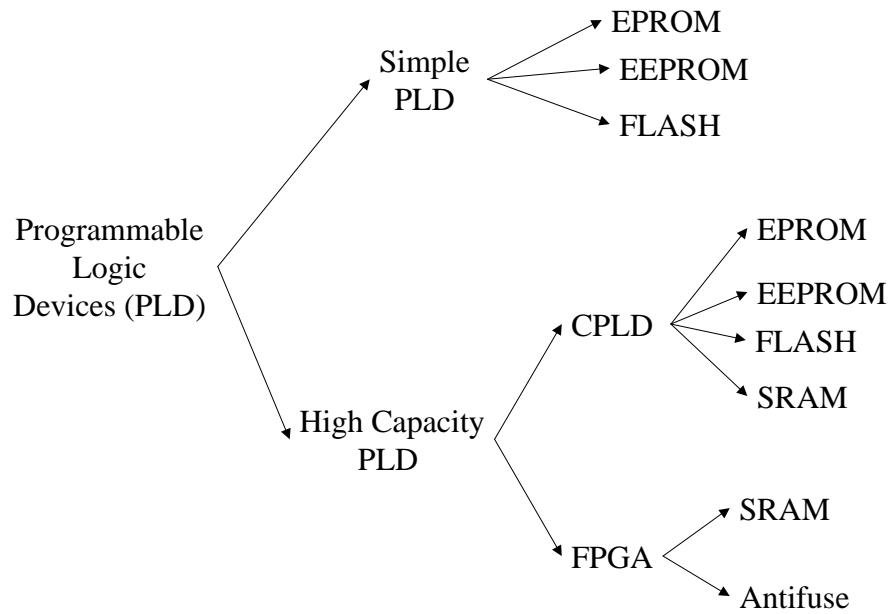


Figure 2.4: A Classification of Programmable Logic Devices

- PR* = Partially Reconfigurable
- FR* = Only Full Reconfigurability
- RTR* = Run-time Reconfigurable
- OT* = One-time Reconfigurable (static)
- N* = Only Network Reconfigurable
- P* = Only Processing Element Reconfigurable
- NP* = Both Network and PE Reconfigurable

Table 2.1: Taxonomy of Reconfigurable Devices

Device	Spatial	Temporal	Type
Xilinx XC6200	PR	RTR	NP
ORCA 2Cxx	PR	RTR	NP
QLogic QL3K	FR	OT	NP
ATMEL 6K	PR	RTR	NP
NAPA1K	PR	RTR	NP
Altera Flex10K	FR	RTR	NP

Table 2.1: Taxonomy of Reconfigurable Devices

Device	Spatial	Temporal	Type
Xilinx XC4K	FR	RTR	NP
Actel SX	FR	OT	NP
Triptych	PR	RTR	P
DPGA	PR	RTR	NP
MATRIX	PR	RTR	NP
GARP	PR	RTR	P
RaPiD	PR	RTR	N
PAM	FR	RTR	NP
DISC	PR	RTR	P
PRISC	PR	RTR	P
Chimaera	PR	RTR	NP
CVH	PR	RTR	NP
RAW	PR	RTR	PN

2.3.2 Programmable Logic Device Key Parameters

1. **Processing element type and granularity.** Processing element (PE) granularity is determined by the *set of operations performed* by the PE and the *bit width* of the input operands. For example, most FPGA devices operate on single bit operands and the set of operations performed are typically K-input, N-output truth tables where K and N are small integers (typically in the range : 2-6). Choices for the PE granularity depend on the computational requirements of the application domain. For example, most image processing filters operate on a large number of bit or byte level values and hence a fine grained reconfigurable array would be most efficient. On the other hand, most digital signal processing applications perform linear filtering on fixed point values of higher bit widths. In such cases, PE's designed to efficiently implement fixed point arithmetic are more suitable. A granularity mismatch between the application and the PE can lead to a very inefficient design.
2. **Interconnect.** Richness of the interconnect resources and the proportion of it that is reconfigurable determines the routability of a given application (class of program graphs that can be mapped to the reconfigurable device.) An interconnection network that is a complete graph on the processing elements can accommodate any program graph on the processing elements. However, for a given area constraint, richer the interconnect resource, less is the area available for processing elements, configuration control and memory elements and, larger the configuration instruction size. Some of the proposed interconnect types are : crossbar, mesh (regular, hierarchical), nearest-neighbors. Another factor that needs to be addressed is whether the interconnect itself is programmable. When the interconnect is not programmable, application specific routing is achieved by using some of the processing elements as switch boxes, whose function is to route specific inputs to specified outputs.
3. **Configuration distribution mechanism.** This refers to the allocation of bits of each *configuration* to individual processing elements and the dispersal mechanism used to transmit those bits to the associated customization points from the input pins. Issues that are of concern here are : (1) whether the allocation of configuration bits is done dynamically or statically, (2) whether the dispersal is sequential, parallel or some hybrid mix and, in each case whether it is full or partial. Consider case (1). If the allocation is static, then some form of instruction encoding should be in place to map configuration bit-streams to the customization points in the processing elements and interconnect. This would increase the bandwidth requirements to

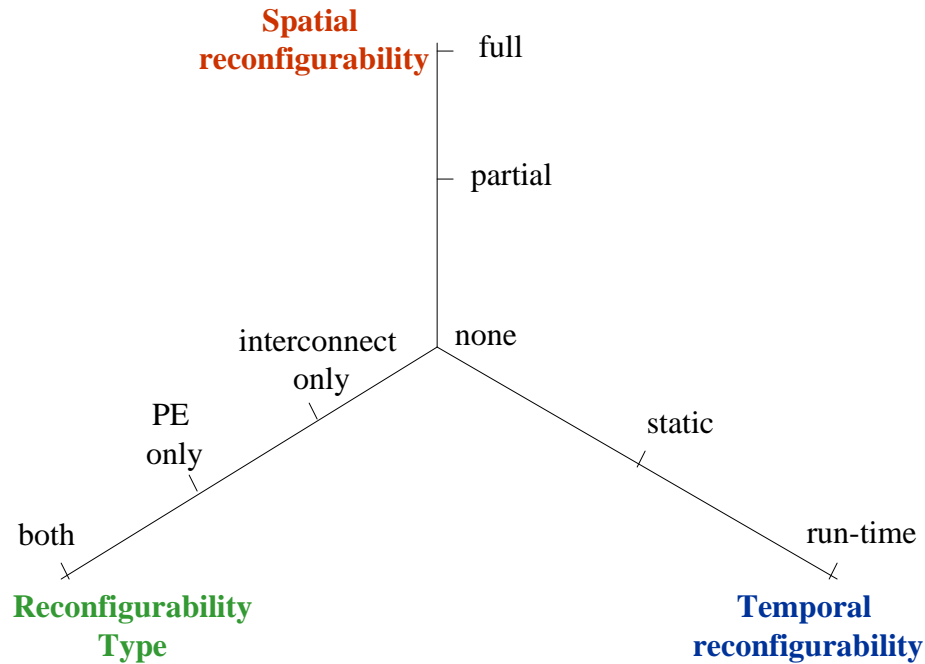


Figure 2.5: reconfig-dimensions-figure

the configuration store. On the other hand, if the allocation is dynamic, then the reconfigurable device complexity increases since it has to include hardware for dynamic allocation.

4. **Configuration context depth.** The number of *configurations* associated with each customization point is referred to as the *configuration context depth*. The configuration words associated with each customization point are ordered. The set of configuration words of all the customization points with the same position in the order is referred to as a configuration context. Multi-context devices are useful due to the fact that their contexts can be switched rapidly. The mechanism is also beneficial when a particular configuration needs to be split into multiple smaller configurations which can be overlapped in space (on the device) but sequentialized in time (during runtime). This might be necessary when the size of the configuration is larger than the size of configuration of a single context. If the context depth is much larger than the application critical path length, then most of the context memory is not utilized. If the context depth is too small, the device would have to load configurations too many times leading to performance degradation.

2.3.3 Programmable Logic Device Taxonomy

A taxonomy of the various reconfigurable logic devices based on the above four parameters is given in Table 2.2. The values taken by each of the key parameters are given below.

PE :Processing Element Granularity

FG = Fine grained (1 bit PE)

MG = Medium grained (2-4 bit PE)

LG = Large grained (4-16 bit PE)

WG = Wide word PE (*i*, 32 bits)

IN :Interconnect Types

NN = Nearest Neighbor

HN = Hierarchical/hybrid Interconnect

CN = Crossbar Interconnect

CR = Channel Routing

CDM :Configuration Distribution Mechanism

SD/PD = Serial/Parallel Distribution

CCD :Configuration Context Depth

SC/MC = Single/Multiple Context

Table 2.2: Taxonomy Based on Key Parameters

Device	PE	IN	CDM	CCD
Xilinx XC6200	FG	HN	SD	SC
ORCA 2Cxx	MG	HN	SD	SC
QLogic QL3K	FG	NN	SD	SC
ATMEL 6K	FG	HN	SD	SC
NAPA1K	FG	HN/NN	SD	SC
Flex10K	FG	NN	SD	SC
Actel	FG	HN	SD	SC
DPGA	FG	HN	PD	MC
MATRIX	MG	HN	PD	MC
GARP	MG	HN	PD	MC
RaPiD	LG	CR	SD	MC
PAM	FG	NN	SD	SC
DISC	MG	?	SD	SC
PRISC	WG	-	SD	SC
Chimaera	FG	HN	?	SC
CVH	MG	HN	PD	MC
RAW	LG/WG	NN	PD	MC

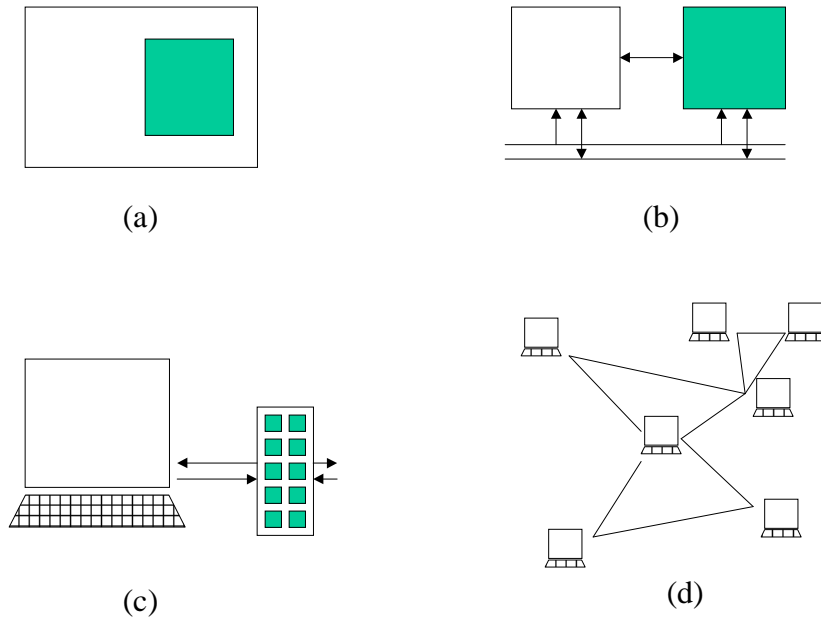


Figure 2.6: Processor-Reconfigurable Coupling Choices

2.4 System Level Issues

Here we present some of the system level issues to be tackled by the designer of a reconfigurable computing system. The designer's choices are eventually governed by a trade-off between system complexity and its performance.

1. **Coupling.** The various options are illustrated in Figure 2.6. The dark colored regions refer to programmable logic. In case (a), the programmable logic is tightly coupled with the processor core by being embedded on the processor die itself. And the coupling gets progressively weaker from (b) to (d). In a tightly coupled system, the semi-conductor budget for reconfigurable logic is very limited. While in a loosely coupled system (like (c)), it is possible to attach a large amount of reconfigurable logic. This enables mapping of large applications to be mapped to the reconfigurable logic. However, loose coupling implies additional communication costs due to limited bandwidth and large communication latencies. The choice is determined by the application domain of interest.
2. **Execution model.** Should the eventual system be viewed as a uni-processor or a multi-processor? While resource allocation issue and programming is very complicated for a multi-processing reconfigurable system, as discussed later in the research issues section, a uni-processing might not be a convenient model for some applications.
3. **Architectural interface.** This determines how the reconfigurable resource is exposed to the software interface. The resource could be oblivious to the user in which case, it is completely managed by the processor or it could be explicitly managed by the runtime system.

2.5 Application Domain

What does reconfigurability buy? Here, we list certain computational characteristics which, if present in an application, might indicate that a reconfigurable computing system based solution could outperform a conventional solution. Some of these “desirable” application features are :

2.5.1 Desirable application characteristics

What does reconfigurability buy? Here, we list certain computational characteristics which, if present in an application, might indicate that a reconfigurable computing system based solution could outperform a conventional solution. Some of these “desirable” application features are :

Non-standard data-path. A mismatch between application datapath requirements and those presented by the processor datapath could lead to inefficient use of processor resources. For example, most media processing involves computing on byte or word level input data and the wide data-path (32/64 bit) of general purpose processors are an inefficient match for such computation. This explains the proliferation of multimedia processors and the “media-extended” (MMX) processors. However, MMX style extensions are a temporary fix to this problem since the number of applications which fall in this category has been growing steadily.

Pseudo static data, adaptive precision, dynamic range. If some of the inputs to the computation are either static or change infrequently, then such factors can be taken advantage of in specializing the processor datapaths to either decrease operation latency or improve on power/area usage. An example of this is the use of constant co-efficient multipliers in most signal processing filtering applications.

Fault tolerance, real-time adaptation, threat sensitive adaptation. In certain applications like cryptographic systems, due to high throughput and computational requirements, it is desirable to perform as much of the computation in hardware as possible through custom datapath encoding. In extreme cases, some of the algorithm inputs (e.g. keys, etc) themselves are hard-coded. However, a custom chip (ASIC) is a poor choice for implementing such application since the chip cannot be altered (in this case, allow key changes) once it has been manufactured. Other examples are complex systems which become faulty during their usage (due to radiation or other environmental hazards). In such cases, instead of replacing the entire component, it is cheaper and more efficient to “rewire” the circuitry around the faults. In all these situations, programmable logic based processors provide the necessary capabilities to solve the problem.

High-performance multifunction portables. In this category of applications, it is not a good idea to include a custom chip (for high performance) for every possible functionality offered by the “multi-function” device. An example of such a device is one which includes multiple demanding functions such as voice recognition, handwriting recognition, graphic display, basic text processing as in a handheld device. A programmable logic based device can be more efficient since it can retain the performance advantage without sacrificing the functionality nor consuming additional power or space (when compared to a multi ASIC solution). This is because several of these functions are not required simultaneously and hence can be “paged” into the programmable logic on demand.

Regularity and concurrency. Repeated operations of similar types on large regular datasets are an ideal candidate for programmable logic implementations. Regularity in operations imposes less demands on the instruction bandwidth and improves configuration load times and the fine-grained parallelism typically provided by programmable logic arrays is a good match for these kinds of computations.

Low latency, high throughput. Response critical applications as in network interface cards, radio modems, the kernels can be fine tuned for input data specific operation providing even more performance than can be obtained from an ASIC solution (which are typically designed to be input data independent.)

Fortunately, the kind applications that are likely to dominate in the near future do exhibit characteristics most suitable for a reconfigurable computing system based solution. These are the applications that exhibit many of the above characteristics—rely heavily on library codes, are compute intensive with high data throughput requirements often with real time constraints, are communication and signal processing oriented. Fortunately, the kind applications that are likely to dominate in the near future do exhibit characteristics most suitable for a reconfigurable computing system based solution.

Chapter 3

Related Work

Section 3.1 gives a brief historical perspective of reconfigurable computing research. Several applications have been mapped to reconfigurable logic devices. A summary past application studies is given in Section 3.2. Section 3.3 presents a somewhat encyclopedic survey of the past and ongoing research in reconfigurable computing systems.

3.1 History

The earliest known computing system based on reconfigurable devices was proposed and implemented by Gerald Estrin at UCLA [45]. It is a hybrid machine consisting of a general purpose processor augmented with high speed logic devices (ALU's, memories) which were interconnected via application specific interconnect. Due to a lack of enabling technology, the reconfiguration was done manually. Mario Schaffner's Circulating Page Structure (CPS) machine [139] implemented a form of hardware paging scheme where the application task was partitioned into pages which circulate through the programmable hardware to compute the task.

The introduction of a field programmable gate array (FPGA) devices by Xilinx in the mid 80's [180, 165] spurred a flurry of research in the development of FPGA based reconfigurable computing engines. PRISM [7] developed at Brown University demonstrates substantial speedup in the case of large binary operations. PAM, a universal reconfigurable hardware co-processor developed by researchers at DEC Paris Research Labs [173] has been used to demonstrate superior performance/cost ratio compared to every other existing technology of its time on a dozen applications ranging from computer arithmetic, cryptography, image analysis, neural networks, video compression, high-energy physics, biology and astronomy. Another such reconfigurable co-processor board developed by Super Computing Research Center at Maryland called SPLASH-2 [52] has been used to achieve two orders of magnitude speedup on genome sequence matching compared to supercomputers of that time (Cray2). The cover story of the recent issue of Scientific American [171] written by researchers at UCLA outlines some novel applications of reconfigurable devices. A complete survey of the past and current ongoing research will be dealt with in subsequent sections of this survey. From Figure 3.1, it is clear that as a field, reconfigurable computing is rather new, but it is gaining momentum.

3.2 Application Studies

An exhaustive list of applications that have been mapped to processors which include some form of programmable logic as part of their construction are listed here. Wireless communications, spread-spectrum communications, IQ

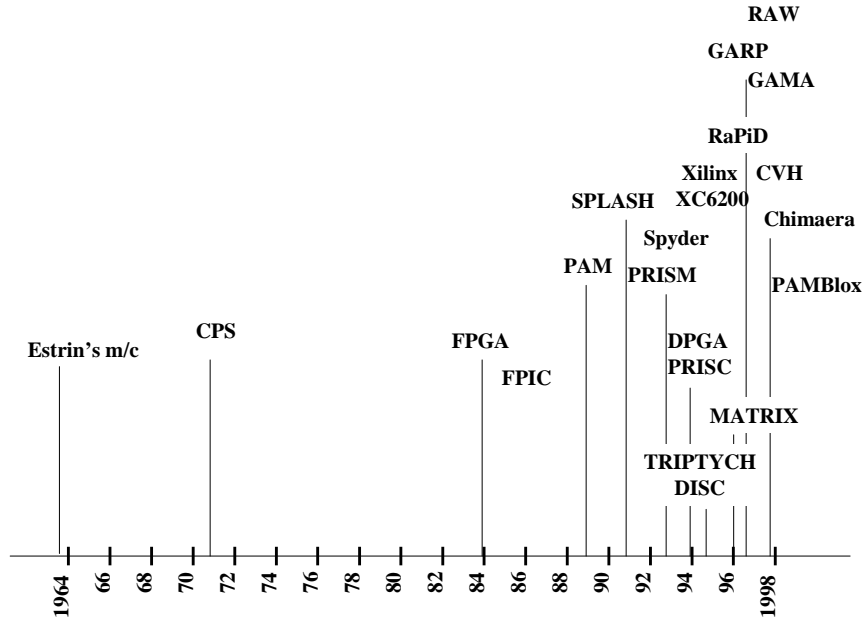


Figure 3.1: Reconfigurable Computing Time-line

demodulation [104, 145, 82]. Genetic Algorithms [54, 68, 154, 99, 56, 55]. SAR,ATR [130, 133, 129, 170]. Image coding, compression [150, 49, 145, 1, 16, 173, 137, 37, 41, 18]. DCT,FFT,filters [151, 35, 179, 118, 81, 119, 149, 89]. Viterbi decoder [183]. Parallel object recognition, geometric hashing [119]. Digit-recurrence division, square-root [102, 101]. Various (big-num, algebra, etc) [16]. Polynomial evaluations [44]. On-line arithmetic [163]. Floating-point arithmetic [46]. CORDIC [6, 106]. Character recognition [167]. DSP [30, 13, 121, 95, 108]. Genome sequence matching [96, 100]. Engineering, sciences applications [16].

3.3 A Survey of the State of the Art

In this section, we present brief summaries of some of the major reconfigurable computing research efforts. For each of the research projects described here, our intent is to give a broad overview highlighting goals and achievements with respect to architecture, systems, language and compilation issues and application studies if any.

3.3.1 Programmable Reduced Instruction Set Computer (PRISC)

The Programmable Reduced Instruction Set Computer (PRISC) architecture was proposed by Razdan and Smith at Harvard University [127, 128]. The goal of this project is to augment the base instruction set of standard RISC architectures so that it meets the instruction set needs of any given application better. The constraints are to minimally impact the RISC cycle time and be able to provide automatic compilation to PRISC.

The PRISC (Figure 3.2) consists of a RISC core whose datapath is augmented with configurable logic in the form of programmable functional units (PFU's) in addition to the regular "fixed function" functional units. Even though PFU's offer limited hardware-programmable resources (due to the design constraints mentioned earlier), these resources reside inside the chip, thus, minimizing the costs of loading and accessing the PFU. *The claim is that, using this approach, performance benefits beyond those captured by pipelining and multiple issue techniques are achievable through pipelining operations at a granularity that is smaller than the existing cycle time.* In the following paragraphs, we briefly outline the PFU micro-architecture, instruction set extensions and the compilation techniques adopted by the PRISC project.

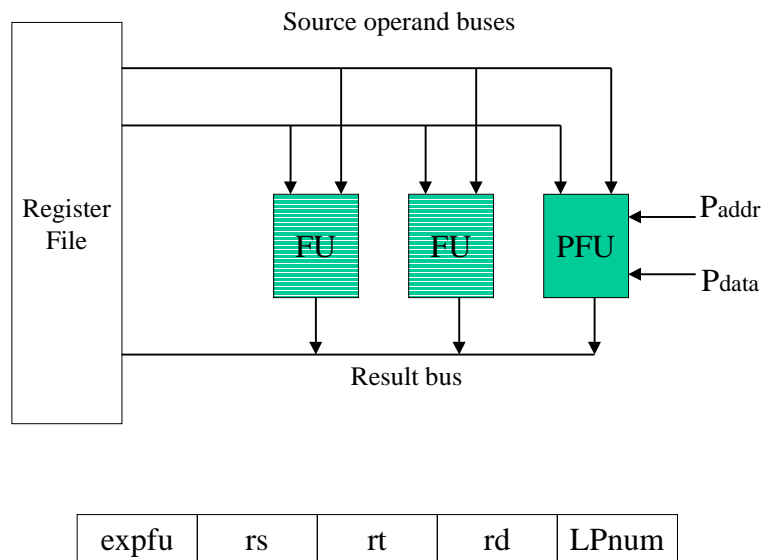


Figure 3.2: PRISC Data-path

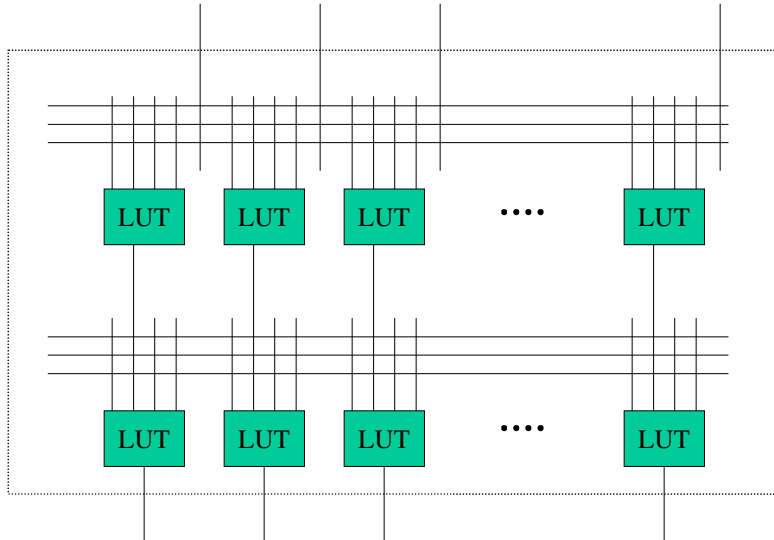


Figure 3.3: PRISC PFU

Figure 3.3 illustrates the proposed micro-architecture for PFU's. A PFU is composed of two basic components: (i) a programmable interconnection matrix and (ii) logic evaluation units implemented as Look-Up Tables (LUT). Each possible intersection point in the interconnection matrix is controlled by a memory cell whose data bit determines whether it is a short or an open connection. Programming the PFU to implement a particular operation (function) consists of loading the appropriate values into the interconnection matrix switches and the LUT memory cells.

A single new instruction *expfu* (Figure 3.2) has been added to the instruction set (MIPS) of the base processor to initiate executions on the PFU. The *rs* and *rt* fields specify the source operands, the *rd* the destination register and the *LPnum* is the number indicating the logical PFU function to execute. A *Pnum* register is associated with each PFU which holds the *LPnum* of the logical function currently programmed into the the PFU. Consecutive execution of the same *LPnum* PFU instruction need not initiate a “PFU programming” phase. The programming information for each logical PFU function is stored as part of the data segment from where it is loaded when necessary by the runtime exception handler. For the proposed PRISC-I processor, they used a 32-bit wide instruction for *expfu*, allowing 11 bits for *LPnum* and estimated that programming a PFU costs anywhere between 100 and 600 processor cycles depending on the number of programmable switches to be set/reset inside the PFU to emulate the desired custom instruction.

The proposed PRISC compilation is an extension of a standard compilation framework with an additional “hardware extraction” phase inserted after code generation. First, sets of sequential (RISC core) instructions that could potentially benefit from a mapping to the PFU referred to as PFU-LOGIC operations are identified. Then, depending on the expected benefit, equivalent new micro-instructions are synthesized for groups of PFU-LOGIC instructions to be executed on the PFU in their place. All the hardware (PFU programming bits) and software

images are linked into one executable. The goal is to identify those instructions which can probably be evaluated using only a portion of the PFU logic. In this way, many such PFU-LOGIC operations may be compressed into one PFU instruction which can be executed in one cycle. They define the notion of *density* to determine likely PFU-LOGIC candidates. *Density* is determined by the *complexity* of the instruction which is purely a function of the opcode and *function_width* which is an estimate of the maximum number of bits of the result affected by the opcode. Instruction semantics are required to compute this information. Since in most cases, the values of input operands are not known at compile time, the *function_width* is an estimate. Once the compiler has identified and marked all of the potential PFU-LOGIC operations, sequences of these operations are merged into one *expfu* instruction using a suite of optimizations. Further details can be obtained in razdan:94.

Experiments on the SPECint92 benchmark suite indicated performance gains of about 22% on a PRISC processor consisting of a single PFU. Razdan's thesis ([128]) contains other examples for which impressive performance gains were attained.

3.3.2 Gate Array Reconfigurable Processor (GARP)

GARP, proposed by the BRASS research group at UC Berkeley [66], is a hybrid architecture combining a reconfigurable logic array with a standard MIPS processor core on the same die. As can be seen in the Figure 3.4, the reconfigurable array and the processor core share the same data cache while the reconfigurable array also has direct access to the memory subsystem. Use of the reconfigurable array is exclusively under the control of the process executing on the MIPS core and entirely optional for the current computation. It is expected that programs will switch to the array temporarily to speedup critical sections of the code.

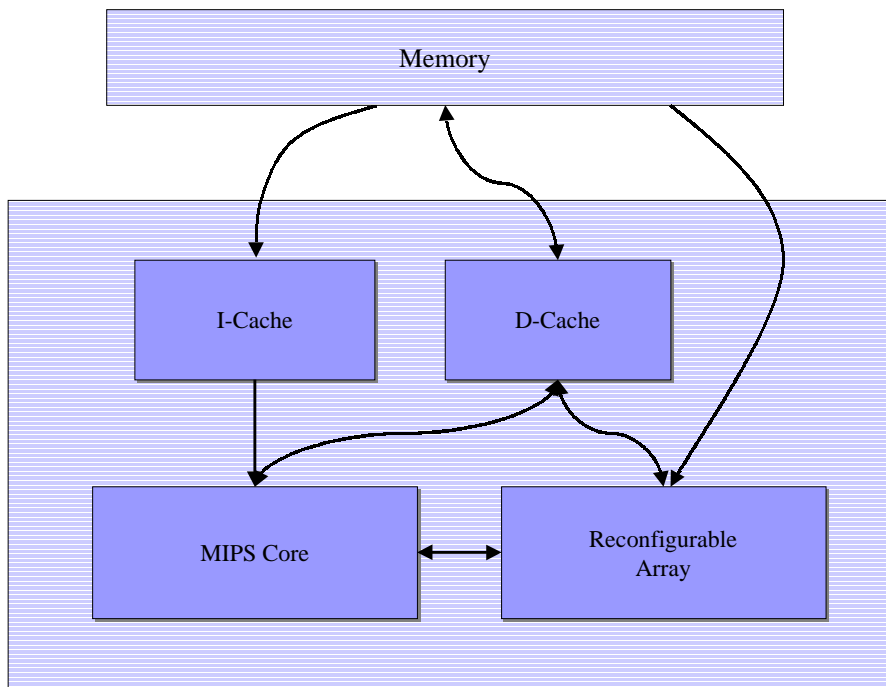


Figure 3.4: Basic Organization of GARP Processor

Execution model. The MIPS instruction set has been extended for the purpose of controlling the reconfigurable array execution. These instructions serve the purpose of loading configurations, for copying data between the array and the processor registers, for manipulating the array clock counter and for saving and restoring array state on context switches. Table 3.1 lists the new instructions. Details about the architecture and the instruction formats can be found in [65].

Table 3.1: GARP - Array Specific Instructions

Instruction	Description
<i>gaconf reg</i>	Load (or switch to) configuration at address given by <i>reg</i>
<i>mtga reg, array-row-reg, count</i>	Copy <i>reg</i> value to <i>array-row-reg</i> and set array clock counter to <i>count</i> .
<i>mfga reg, array-row-reg, count</i>	Copy <i>array-row-reg</i> value to <i>reg</i> and set array clock counter to <i>count</i> .
<i>gabump reg</i>	Increase array clock counter by value in <i>reg</i> .
<i>gastop reg</i>	Copy array clock counter to <i>reg</i> and stop array by zeroing clock counter.
<i>gacinv reg</i>	Invalidate cache copy of configuration at address given by <i>reg</i> .
<i>cfga reg, array-control-reg</i>	Copy value of array control register <i>array-control-reg</i> to <i>reg</i> .
<i>gasave reg</i>	Save all array data state to memory at address given by <i>reg</i> .
<i>garestore reg</i>	Restore previously saved data state from memory at address given by <i>reg</i> .

At present, the configurations are manually programmed. The user has to specify the data transports, interconnections and the operations to be implemented by the logic block. This configuration information (text file) is compiled by a *configurator* module and converted into the appropriate configuration bit-stream to be downloaded into the GARP reconfigurable array. This bitstream is linked into the main executable as part of its data segment. Example 1 shows (i) the code segment which includes the configuration bit-stream into the source file, and, (ii) the code for loading the configuration which adds three operands. The names *v0*, *a0*, *a1*, *a2*, and *ra* refer to ordinary MIPS registers; *la* is the MIPS “load address” instruction. The symbols *\$zi* and *\$di* indicate the *Z* and *D* registers of array row *i*. The MIPS subroutine calling convention passes the first three subroutine arguments in registers *a0*, *a1*, and *a2* and the return value being passed back in *v0*. With this assembly language stub, a program can add any three values *a*, *b* and *c* using the reconfigurable array by executing the ordinary subroutine call *add3(a,b,c)*.

Micro-architecture. The reconfigurable array (Figure 3.5) is organized as a two dimensional matrix of processing elements called *blocks*. While the number of columns is fixed at 24, the number of rows is implementation dependent and is expected to be at least 32. Blocks from the first column are called *control blocks* while the rest *logic blocks*. *Control* blocks serve as liaisons between the array and the external world. Among other things, *control* blocks can initiate memory transfers or interrupt the core. *Logic* blocks are primarily meant for implementing application logic. All blocks, *control* as well as *logic*, have a processing granularity of 2-bits and the interconnection wires run in pairs. Four memory buses run vertically through the array for information into and out of the array rows. The information transfer can be between the array and the processor core (through the register file) or between the array and the memory. In addition to the memory buses, a wire network carries signals between array blocks. An individual configuration covers some number of complete rows of the array,

Example 1 GARP Example : 3-Input Adder

```
char config_add3[] = #include "add3.config";
```

```
function add3;
    add3: la v0, config_add3;
           gaconf v0;
           mtga a0, $z0;
           mtga a1, $d0;
           mtga a2, $d1, 2;
           mfga v0, $z1;
           j ra;
end function;
```

which may be less than the total number of physical rows available. Distributed within the array is a cache of recently used configurations, so that programs can quickly switch between configurations without the cost of reloading them from memory each time. However, management of the configuration cache is not under program control (like in traditional caches). The array registers are latched synchronously under the control of *array clock* whose frequency is fixed by the implementation. It is required that all configurations be designed so that their critical delay path is less than that which the *array clock* can satisfy. This imposes certain restrictions on the allowed designs for the configurations. An *array clock counter* governs the array execution. A zero value in the counter stalls the array execution. It is up to the main processor to initialize the counter depending upon the configuration to be executed. Each logic block in the GARP array can implement a function of up to four 2-bit inputs. Operations on wider data types can be formed by adjoining logic blocks along a row.

Software environment. The BRASS group is in the process of building a SUIF based compiler for targeting GARP. In addition to the traditional back-end code generators, the GARP compiler also includes a data-path mapping tool called GAMA [25] for mapping the program intermediate code to the custom instructions on the GARP array. GAMA takes as input a data-flow graph (G) where each intermediate node in G is some multi-bit arithmetic operation. GAMA's task is to implement the given data-flow graph on the reconfigurable array with the goal to either minimize the number of array logic blocks consumed or the critical path length of the mapping.

A complete GARP hardware implementation does not exist at the time of this writing. However, a simulator has been built, using which impressive speedups have been demonstrated on three benchmark applications : DES encryption, image dithering and sorting of a large (1 million) number of records. The speedup factors reported ([66]) are 24, 9.4 and 2.1 respectively.

3.3.3 Dynamic Instruction Set Computer (DISC)

The DISC architecture [178] was one of the first to propose support for demand-driven modification of a processor's instruction set. Every DISC instruction is a custom instruction and every application is expected to be compiled to an instruction set suited to its own computational needs. It is also expected that the control and data-path mappings for each instruction used by the application also be specified along with the application code. Instructions are implemented as partial configurations and individually configured on demand. DISC extensively relies on partial reconfiguration to implement custom instruction caching. The claim is that this ability to partially reconfigure the DISC processor for the subset of custom instructions currently being used, has the benefit of substantially reducing the hardware requirements as well as the reconfiguration time overheads. Two novel ideas,

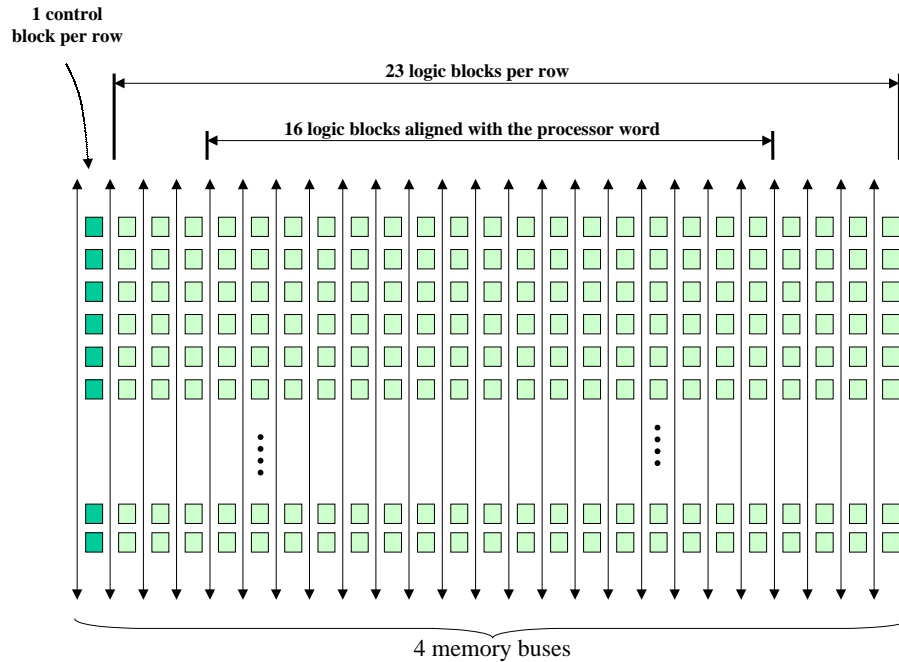


Figure 3.5: GARP Array

namely, *relocatable-hardware* and *linear hardware space* were proposed to achieve the above goal.

Relocatable Hardware. Relocatable hardware, implementable only in partially configurable architectures provides the ability to *relocate* or make placement decisions of the configurations corresponding to the custom instructions at *runtime*. As noted before, this can substantially improve hardware utilization. This is achieved by designing each custom instruction module for multiple locations on the base architecture (FPGA). These multiple locations are defined around a firmly defined global context. The global context provides physical placement positions and a communication network necessary for these modules to operate correctly independent of their actual location at runtime. In order to design instruction modules that fit within the global context, all DISC instructions must be physically independent of each other and the physical layout of any module must not have any effect on the layout of any other module.

Linear Hardware Space. *Linear hardware space* forms the basis for implementing relocatable hardware (custom instructions) in the DISC system. The two-dimensional grid of configurable logic cells of a DISC processor are organized as a linear array of rows. Each of the custom (relocatable) instructions location is specified by vertical position (row number) and by its height in number of rows. The linear hardware space consists of a *global controller* and a uniform *communication network*. The communication network provides access to global resources for all instruction modules and performs inter-module communication (through the global controller). To gain access to all the global signals, modules are designed horizontally across the width of the linear hardware space. The global controller specifies the communication protocol, controls global signals (such as I/O and global state) and monitors circuit execution.

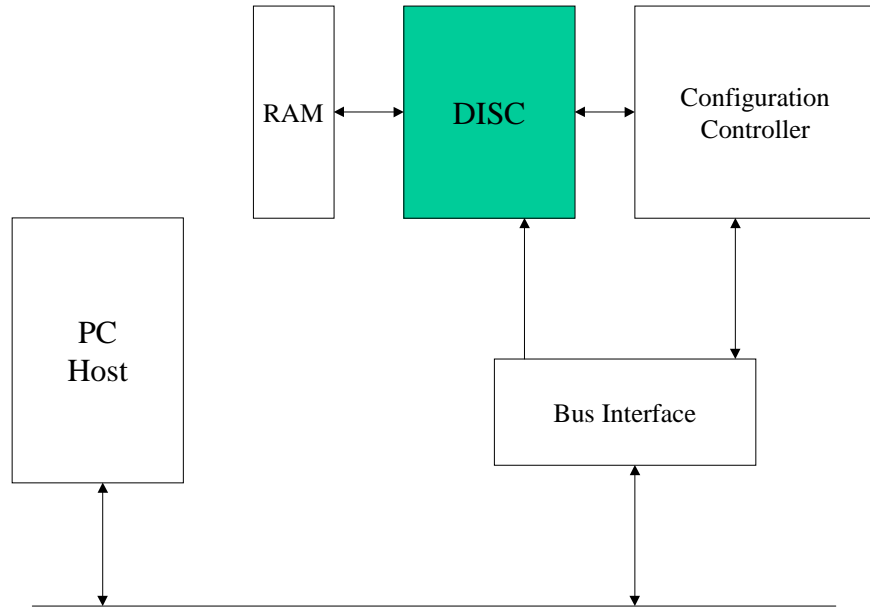


Figure 3.6: The DISC System

The proposed DISC architecture was implemented on a single CLAy31 FPGA [135] coupled to an external RAM (Figure 3.6). The DISC processor is attached to a PC-bus, a Configuration controller also implemented on another CLAy31 and a RAM. The configuration controller monitors the DISC processor execution and requests instructions from the host PC. The RAM consists of the application program as a sequence of custom instructions. The DISC execution model is illustrated through the flowchart is shown in Figure 3.7. A DISC application is initiated first by loading the program memory (RAM) with the target application and configuring the global controller on the DISC processor. During execution, the processor validates the presence of each instruction requested by the application program. If the requested instruction is not loaded on the DISC, the DISC processor enters a halting state and requests the instruction module from the host (through the configuration controller). Before issuing an instruction load, the global controller checks if enough hardware resources are available for the new instruction. If not, then it swaps out one or more currently loaded instructions to make room for the new instruction.

The main achievement of the DISC project was to demonstrate the concept of tackling the density issue of configurable devices through novel ideas like *relocatable hardware* and *linear hardware space*. Preliminary experiments on an image filtering application indicated substantial (factor 23 on the median filter) speedup.

3.3.4 Programmable Active Memories (PAM)

Programmable Active Memories (PAM's) [15, 16, 173], one of the first configurable logic based hardware accelerators, was developed by researchers at the DEC Paris Research Labs. At the time of development, they were able to exhibit a dozen applications for which PAM accelerators proved superior in both *performance and cost* categories to every other existing technology, including supercomputers, massively parallel machines and custom hardware

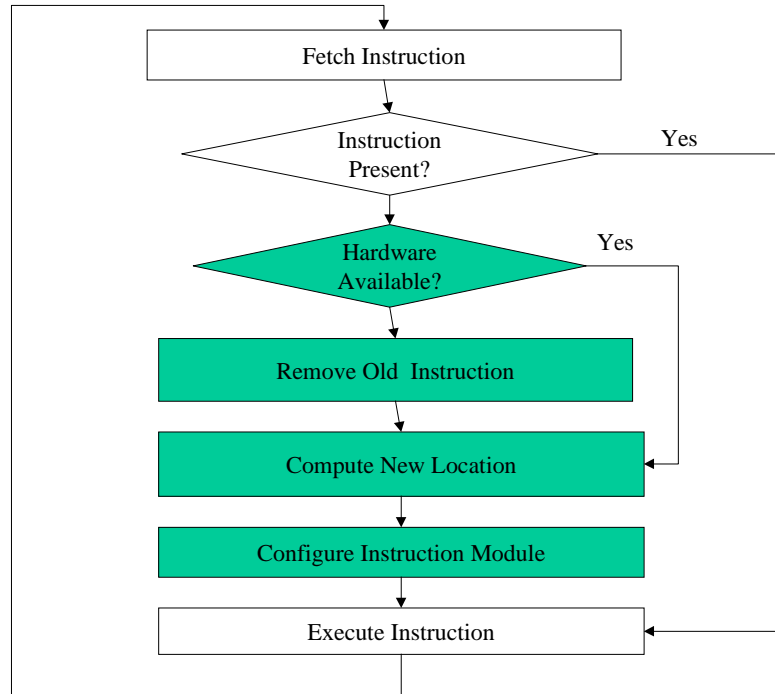


Figure 3.7: DISC Execution Flowchart

of their time. The applications covered diverse fields like long integer arithmetic [147, 36], cryptography [148], high energy physics [107], image analysis, video compression, sound synthesis [47], thermodynamics, biology and astronomy [173]. Here we present the motivations, design ideas and the achievements of the PAM project.

The purpose of PAM is to implement a *virtual machine* which can be dynamically configured as one of a large number of specific hardware devices. Figure 3.8 shows the PAM and its external interface. The inputs/outputs of the PAM could be from/to the external i/o devices or from/to a host controller. The local RAM attached to the memory interface can be used by the PAM to buffer and re-order local data or to implement specialized look-up tables. The PAM array itself is composed of a two dimensional matrix of *Programmable Active Bits* (PAB) connected in a nearest neighbor fashion. Each PAB maps four inputs to four computed inputs. Figure 3.9 shows a complete system consisting of the PAM board attached to a host computer which controls the PAM program, a local RAM and channels to external data sources. A first generation PAM system called DecPerle-1 consisted of an array of 16 Xilinx FPGA's and 7 other FPGA's for memory access control, program down-load and data transfer between the host computer and PAM. The FIFO's are used for data burst control.

A PAM program consists of three components:

- *Driver module* written in C++ which runs on the host machine and controls the PAM hardware.
- *Logic equations* that describe the synchronous hardware implemented on the PAM board.
- Placement and routing directives that guide the implementation of the logic equations on the PAM board.

The PAM circuit designs are described algorithmically at the structural level, and the structure can be annotated with geometry and routing information to help generate the final physical design. This middle ground approach

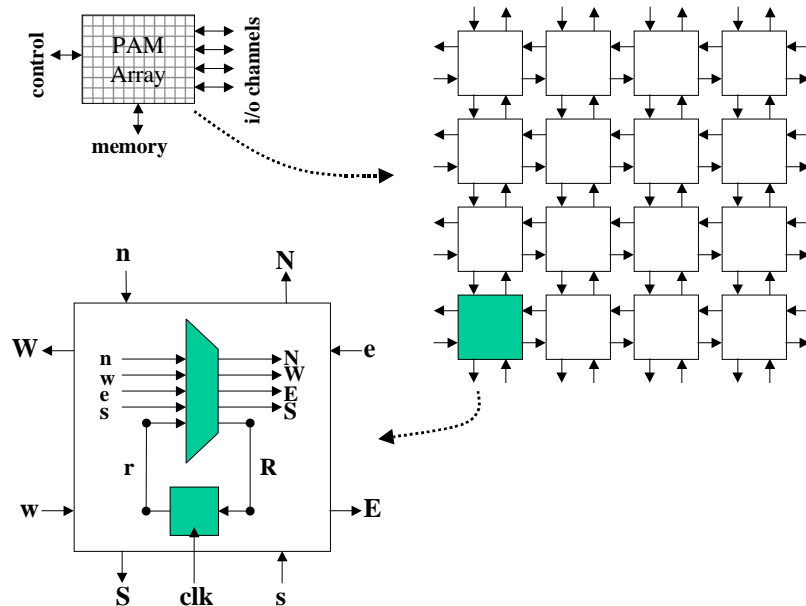


Figure 3.8: PAM Array

has been taken to alleviate the laborious process of schematic capture and to avoid compute intensive CAD synthesis process. Algorithmic description of synchronous circuits is done using an minor extended version of C++. A new type called `Net` is introduced to capture the effect of signal nets, the boolean operators are overloaded to describe combinational logic, and a primitive type is added for the synchronous register. Example 3 shows the code for a ripple carry adder written using these extensions. To specify that the ripple-carry adder should be aligned vertically, with the paired carry and sum bits generated by the same logic block, addition of the *placement* function to the description of the adder would suffice.

3.3.4.1 PAM-Blox, PaModules

PamBlox [111] are parameterizable templates of primitive hardware objects such as counters, shifters, adders, registers described in a hierarchical manner. A *PamBlox* hardware object describes the structure, placement, functionality and the interface of a digital circuit. A generic *PamBlox* object described in C++ is shown in Example 3. *PaModules* are complex, fixed layout circuits implemented as C++ objects. *PaModules* are composed of multiple *PamBlox* and are optimized for specific data-path width. Examples are constant co-efficient multipliers, Coordinate Rotation Digital Computer (CORDIC) circuits. A collection of *PamBlox*, *PaModules* can be obtained from the Stanford University adaptive computing research group website (see [111] for details).

Example 2 Circuit Description in C++

```
function void placement(Net<N>& s, Net<N>& c)
  for (i = 0; i < N; i++)
    c[i] <<= s[i];
    s[i+1] <<= s[i] + OFFSET(0,1);
  end for
end function;

template<int N>
class RippleAdder: Block {
  RippleAdder(): Block("RippleAdder"){};
  void logic(Net<N>& a, Net<N>& b, Net<N>& c,
    Net<N>& sum, Net<N>& carry) {
    input(a); input(b); input(c);
    output(sum); output(carry);
    for (int i = 0; i < N; i++) {
      sum[i] = a[i] ^ b[i] ^ c[i];
      carry[i] = (a[i] & b[i]) | (c[i] & b[i]) | (c[i] & a[i]);
    }
  };
};
```

Example 3 PAM-Blox Circuit Description in C++

```
class HWobject : public parent {
  public:
    /* internal wire declarations */
    /* constructor */
    HWobject(input parameters, optionals) { /* initialize inputs */ }
    out(output parameters, optionals) { /* internal logic */ }
    /* additional methods called by out */
    place(absolute placement parameters) { /* absolute placement */ }
    place() { /* relative placement */ }
};
```

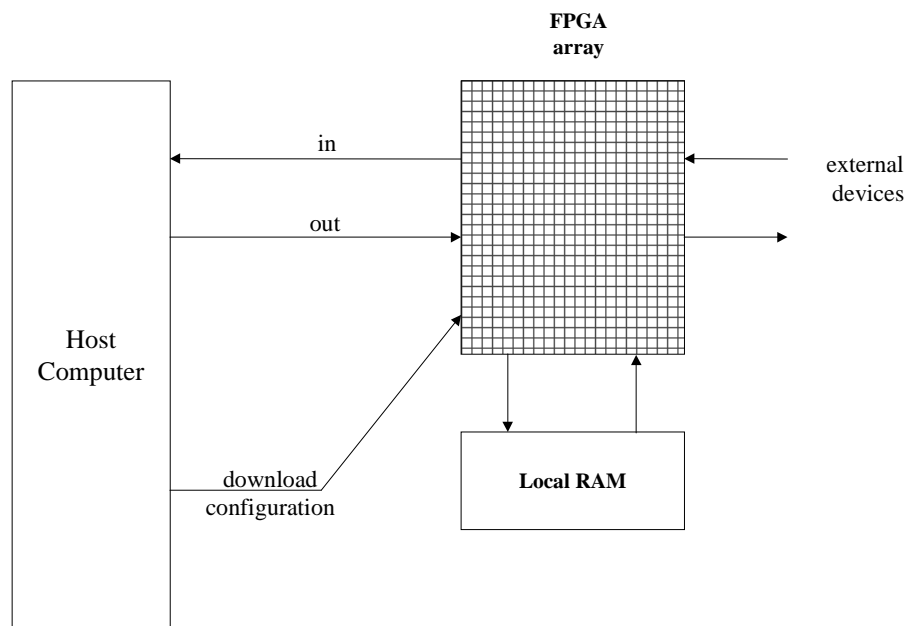


Figure 3.9: A PAM System

3.3.5 Reconfigurable Pipelined Data-path (RaPiD)

RAPID [58] is a coarse-grained reconfigurable architecture that allows deeply pipelined data-paths to be constructed dynamically from a mix of ALU's, multipliers, registers and local memories. The structure of data-paths constructed in RAPID is strongly biased towards linear arrays of functional units communicating mostly in a nearest neighbor fashion. In its capabilities, RAPID is more general than systolic arrays in that the pipeline stages could be heterogeneous both spatially and temporally. RAPID is coarser-grained than standard FPGAs in terms of processing element granularity and the interconnection data-path widths. Compared to general purpose architectures, it is finer grained containing smaller distributed memories instead of large instruction and data memories, distributed register file with limited interconnect instead of the regular register file with a crossbar to all the functional units.

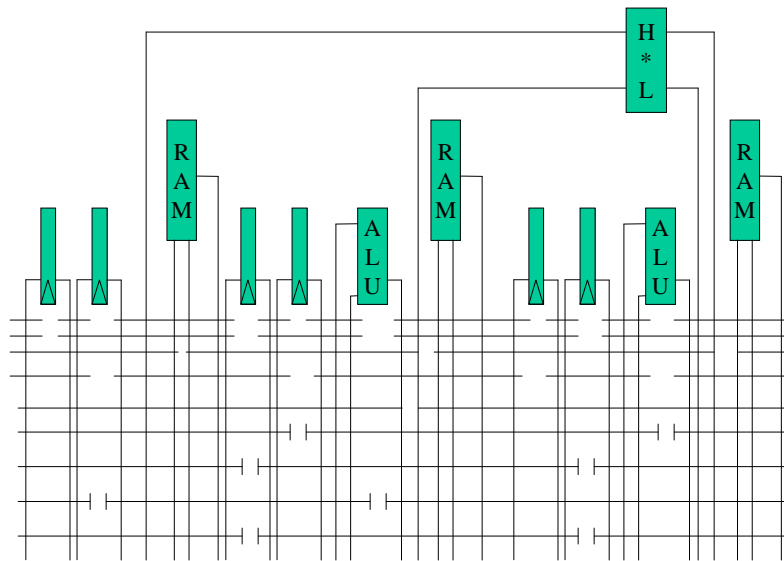


Figure 3.10: RAPID Cell

Architecture. RAPID is a linear array of cells consisting of uncommitted functional units. An array might have 16 to 32 of RAPID cells. Figure 3.10 shows the architecture of a cell. Each cell (Figure 3.10) comprises of an integer multiplier, two integer ALU's, six general purpose registers and three small local memories. These cells are interconnected using a set of ten segmented buses that run the length of the data-path. Each input to the functional unit is attached to a multiplexer which can be configured to select one of the eight input buses. Each output in turn is attached to a De-multiplexer comprised of tristate drivers, each driving one of eight buses. The buses in different tracks are segmented into different lengths so that bus tracks could be used efficiently and at the same time limiting the number of switches on longer paths. The limited amount of local memories are provided for saving and reusing data over many cycles. In many applications, the input or output data is segmented into blocks that are accessed once from the main memory, saved locally and reused as needed, and later discarded.

RAPID1 operates on 16-bit signed or unsigned data.

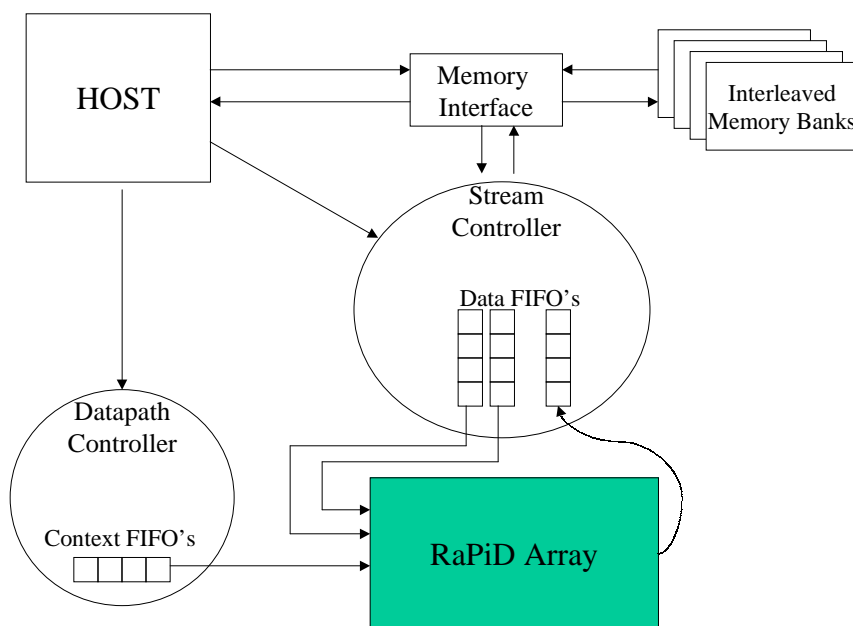


Figure 3.11: RaPiD System

A reconfigurable computer system based on the RAPID array is shown in Figure 3.11. Input and output data enter and exit from the array via I/O streams at each end of the data-path. The FIFO buffers are used to implement flow control on the i/o streams between the RAPID array and its external interface. Signals for controlling the array are divided into static control signals provided by configuration memory as in typical FPGA's, and dynamic control signals which must be provided on every cycle. RAPID is programmed for a particular application by first mapping the computation onto the data-path pipeline using the static programming bits to build this pipeline. A controller is programmed to generate the dynamic control signals through the control FIFO as shown in the Figure 3.11. Examples of application mapping to RAPID can be found in [58].

3.3.6 Cached Virtual Hardware (CVH), PipeRench

The objective of the CMU Cached Virtual Hardware (CVH) project is to create architectures, tools, and methodologies for what is termed *virtualization of hardware design* for reconfigurable computing. These virtual designs can be executed on any one of a family of upwardly-compatible FPGA-like hardware platforms. By virtualization they mean time-multiplexing of a large hardware design on a comparatively smaller physical device. This requires that the target device support rapid reconfiguration. Techniques proposed to achieve this are (i) widening the configuration bus width, (ii) reducing the access time to configuration data through configuration caching and (iii) by containing the portion of the device (design) to be reconfigured and finally (iv) by pipelining the reconfiguration process. The architecture proposed in [142] supports the above features. The basic unit of reconfiguration is a pipeline stage and is called the *stripe*. Application designs are mapped into a sequence of stripes, each stripe

representing a portion of the application logic corresponding to a pipeline stage of a set of independent pipeline stages. The assumption is that the reconfiguration at the granularity of a stripe can increase the throughput of an implementation, without significantly increasing the latency or the required storage for the configuration data.

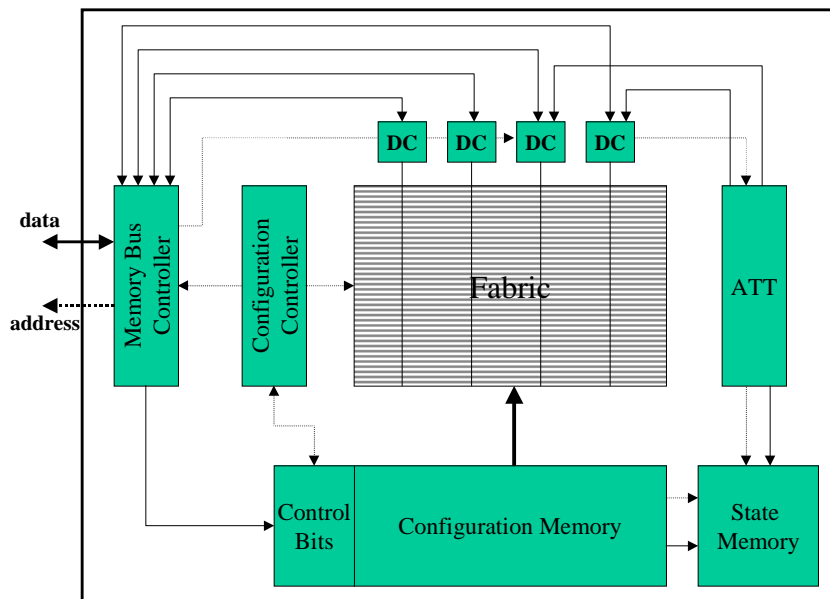


Figure 3.12: CVH Architecture

PipeRench Architecture. As shown in Figure 3.12, the proposed *PipeRench* architecture [23] is composed of an *FPGA fabric*, a configuration cache, control units for managing data and configurations, a memory interface and an interface to a standard processor (not shown in the figure). The design methodology for striped FPGA's allows any application to be broken up into a set of stripes which can be run on a compatible striped FPGA. The control mechanism manages the loading and unloading of stripes onto the striped-FPGA fabric depending on the needs of the application.

A *PipeRench* executable is composed of configuration words each of which includes configuration bits (for programming the desired section of the fabric), a next address field, and flags used by the configuration and data controllers. A *PipeRench* execution is equivalent to the evaluation of a sequence of configuration words (perhaps in a loop.) The configuration controller flags in the configuration word indicate whether the current word is the first/last of this sequence. The host specifies a start address (of the first configuration word) and the number of iterations before the start of each execution.

3.3.7 Chimaera

Similar to many other attempts, Chimaera ([63, 64]) consists of a reconfigurable logic array tightly integrated with a microprocessor core. The Chimaera architecture is shown in Figure 3.13. The reconfigurable array consists

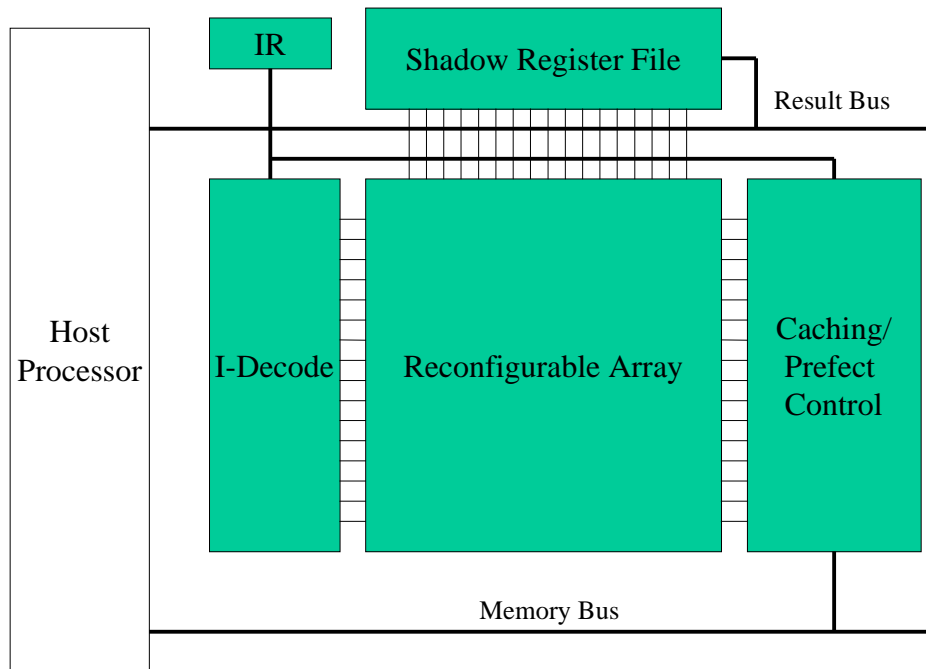


Figure 3.13: Chimaera Architecture

of FPGA-like logic and is used to implement *reconfigurable functional units* (RFU). In order to use instructions in the RFU, application code includes calls to the RFU, and the corresponding segments of RFU configuration bits are contained within the instruction segment of that application. The RFU calls are made in a two step process: (1) tell the cpu to trigger execution on the RFU and (2) specify the identity of the particular RFU to execute. The RFU calls are just like the execution of many other processor instructions, the inputs read from the register file, and the results are written back to the register file in the write-back cycle. RFU's inputs (number, locations) are specified as part of the RFU. The architecture allows for each RFU to read up to nine operands from the register file.

The Chimaera system (Figure 3.13) consists of a core processor, the reconfigurable array, instruction decode CAM (content addressable memory) and the caching/prefetch control. Each location in the CAM controls one row of the reconfigurable array, determining which of the loaded instructions are completed. Loaded instructions occupy disjoint rows of the reconfigurable array. If the triggered RFU operation is not present in the reconfigurable array, the caching/prefetch logic stalls the processor and initiates a RFU instruction load from the memory. Reconfiguration can be done on a per row basis with each RFU operation occupying one or more rows.

The reconfigurable logic array is composed of rows of logic cells between routing channels. Within each row, there is one cell per processor's memory word. This restricts the row size to equal the memory word size. All cells in a given column I have access to the I^{th} bit of the first nine registers. As shown in the Figure 3.14, each cell receives four inputs I_1, I_2, I_3, I_4 and generates four outputs O_1, O_2, O_3, O_4 . The logic block inside the cell itself can be configured as a 4-LUT, two 3-LUT's or a 3-LUT and a carry computation. Unlike most other reconfigurable logic arrays, em Chimaera reconfigurable array has no state holding elements (flip-flops or latches). This forces any sequential computation to be implemented by storing/retrieving intermediate values to/from the register file.

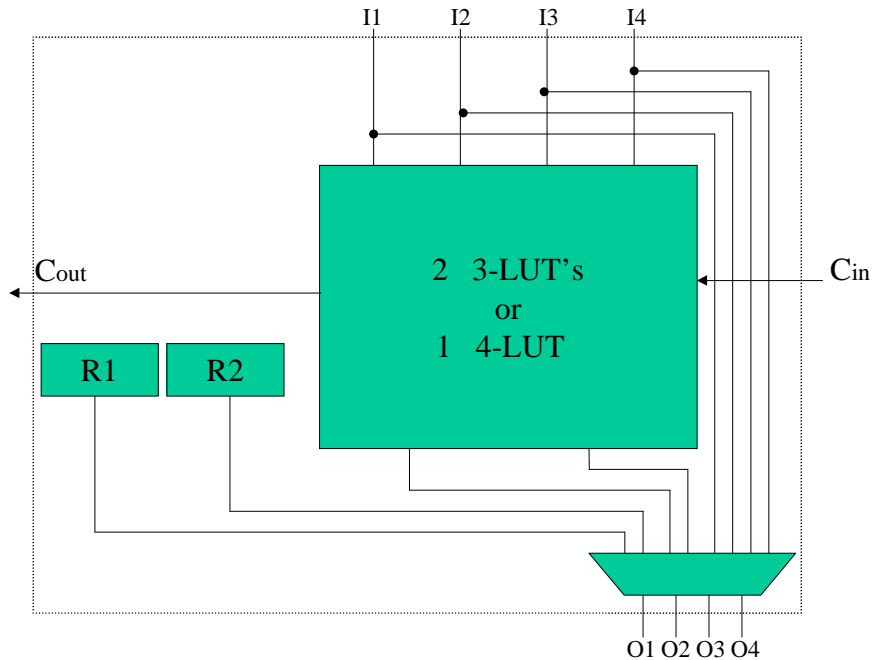


Figure 3.14: Chimaera Cell

Lack of pipelining latches also requires that the *values in the register file be stable as long as the RFU instruction execution is incomplete*. In many respects, Chimaera is very similar to the PRISC reconfigurable processor.

3.3.8 Reconfigurable Architecture Workstation (RAW)

Reconfigurable Architecture Workstation (RAW) proposed by Agarwal and his colleagues at MIT [174, 94, 4] was motivated by three main concerns : (1) the need to keep chip wires short so that clock speed scales with feature size, (2) economic constraints of quickly verifying new designs and (3) changing application workloads that emphasize stream-based multimedia computations. These factors lead to a design that takes an extreme position by distributing all the processor's resources such as instruction streams, register files, memory ports and ALU's. As shown in Figure 3.15, the RAW processor is a set of interconnected tiles each of which (Figure 3.16) contains instruction and data memories, an arithmetic-logic unit, registers, configurable logic, and a programmable switch that supports both dynamic and static (compiler orchestrated) routing. The tiles are interconnected with programmable interconnects. The tightly integrated, synchronous network interface of a RAW machine allows for inter-tile communication with short latencies similar to those of register accesses. The switch on a RAW tile contains a *static* and a *dynamic* network component. The static switch is programmable, allowing statically inferable communication patterns to be encoded in the instruction streams of the switches eliminating the overhead of composing and routing a bi-directional header. The dynamic switch is a worm-hole router that makes routing decisions based on message headers. The inter-tile communication ports are accessed as register operands allowing useful computation to be overlapped with a communication operation.

A typical RAW system would consist of a RAW microprocessor tightly coupled with off-chip memory and stream-

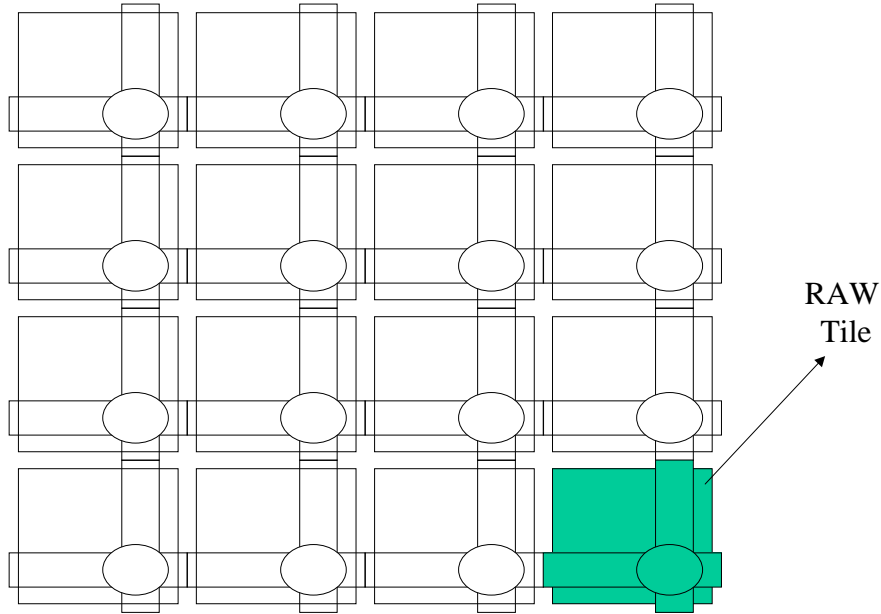


Figure 3.15: RAW

IO devices (Figure 3.17). It is interesting to note that reconfigurability in a RAW machine takes place at two levels. At the low level, it is possible to use the configurable logic inside each tile to build custom functional units for bit (or non-standard) data width operations. At a higher level, the RAW processor can reconfigure its communication network and the distribution of computation over the array of tiles. Compiling to a RAW machine is complicated by two factors:

- Unlike traditional super-scalars, a RAW processor does not bind specialized logic structures such as register renaming or dynamic instruction issue logic into hardware. Scheduling and resource allocation are the responsibility of the compiler.
- Communication patterns within the code need to be analyzed in order to schedule inter-tile communication.

Preliminary experience with compiling to the RAW machine can be found in [4, 94]. The authors also report impressive speedups [174] on a class of benchmark applications which are now referred to as the RAW benchmark suite.

3.3.9 Others

Many other groups have been pursuing reconfigurable computing research. However, these shall not be covered here for two reasons. Either the ideas are not substantially different from what has already been surveyed earlier or the research is too preliminary to report any outcome of value. Some of these are Map Oriented Machine (MoM XPuter), Splash-II, PRISM, HARP (Oxford), Spyder, NSC, FM (Hawaii). Interested reader can refer to the following references : PRISM [7] , Spyder [80], Xputer [61], Splash [52], Data parallel C [53], Harp [114] and Functional Memory [59].

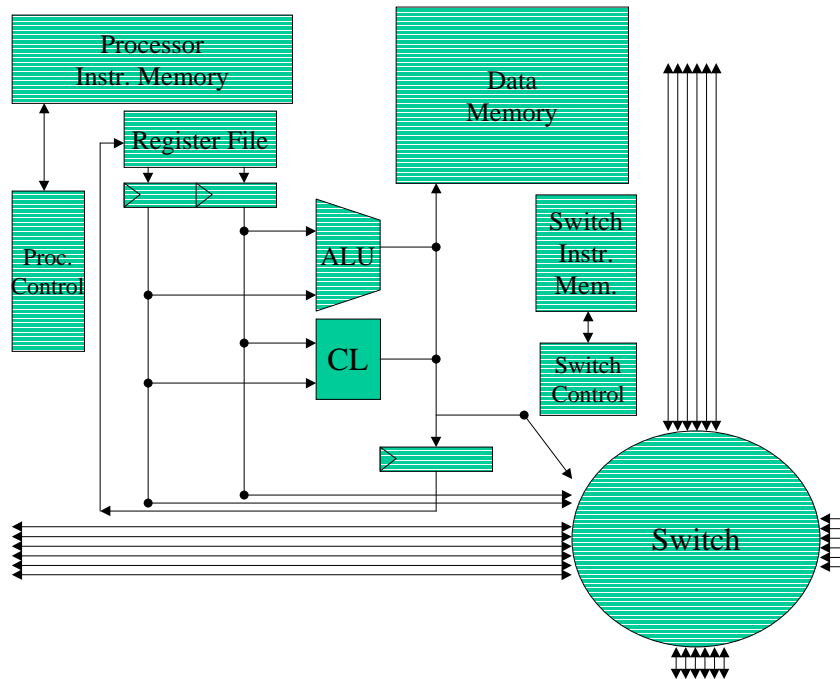


Figure 3.16: RAW Tile

3.4 Summary

In this survey, an overview of the major reconfigurable computing research efforts is presented. Due to a lack of a well accepted set of definitions, an attempt was made to clarify the terminology and identify key issues and parameters of reconfigurable computing systems. The main dimensions of reconfigurability and the key parameters of these devices have been identified and have been used to taxonomize extant reconfigurable computing systems. A list of applications for which reconfigurable systems hold promise have been presented.

So far, most research efforts have focused on the architectural aspects of reconfigurable systems. Little attention has been paid to compilation issues. Many performance studies have been done and impressive speedups were demonstrated. However, important issues like compilation times, target cost have been neglected.

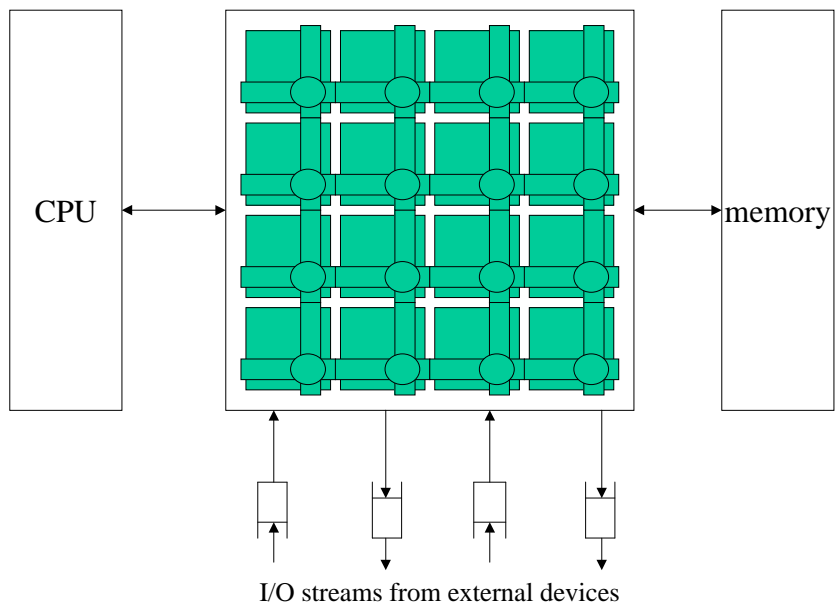


Figure 3.17: RAW Computing System

Chapter 4

Adaptive Instruction Level Parallel Processing

In this chapter we describe a novel variant of the Instruction Level Parallel (ILP) processing model called Adaptive Instruction Level Parallel (AILP) processing. The two key characteristics that define AILP processing are (a) instruction level parallel processing and, (b) dynamically specified instruction sets.

We start by presenting a brief background on ILP, highlighting key features and limitations. Next we present the *Dynamic Instruction Set Architecture* (DISA) model and contrast it with the *Static Instruction Set Architecture* model. The proposed AILP architectures are a subset of the DISA space of architectures. We describe the key features of AILP architectures and then present a taxonomy of the AILP architectures. A specific subset of this AILP set forms the basis for our architecture and compiler research.

4.1 Instruction Level Parallel Processing

4.1.1 Background

ILP processing is a set of processor and compiler design techniques intended to increase application performance by enabling parallel execution of multiple RISC style instructions extracted from a sequential instruction stream (Figure 4.1). In order to achieve this effect, ILP machines are composed of multiple, independent, possibly pipelined functional units (F_1, F_2, \dots, F_k in Figure 4.2) that communicate through a local memory space such as a register file.

The number and types of instructions that can be issued for execution is limited by the number and types of available functional units. In practice, constraints such as inter-instruction data or control dependences, resource conflicts, etc., might further restrict the number of choices for parallel instruction issue. To achieve this parallel record of execution (ROE), the compiler and processor, between them must perform the following tasks:

1. determine dependencies among instructions
2. determine independencies, reduce critical path
3. schedule, allocate and generate code
4. parallel issue

These tasks are normally performed in the above sequence. The task of determining dependencies is a step towards *exposing* ILP while the tasks of determining independent instructions, performing transformations to

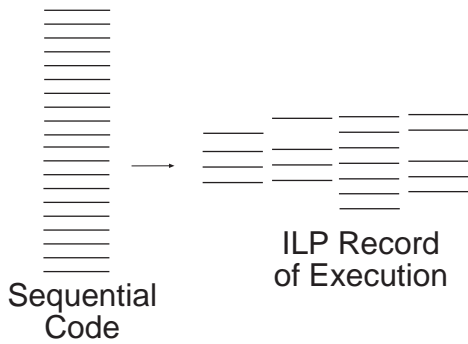


Figure 4.1: ILP computation

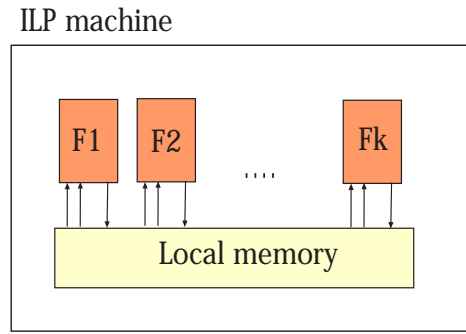


Figure 4.2: ILP machine model

reduce critical path length are steps towards *enhancing* ILP (see Figure 4.3). Scheduling, allocation and code generation finally *exploit* the available ILP exposed earlier. Parallel issue is typically performed by the processor and is the final step in realizing the parallel execution.

Unlike most other types of parallel computation, this type of parallel processing is transparent to the user (the software developer) and hence any advances in processor or compiler techniques to increase effective parallelism will provide automatic benefits to the user without their being even aware of it. Automatically and transparently improving application performance is a tremendously appealing and consequently a tremendous amount of research has gone into ILP processing [125], [3]-[77], [75]-[78].

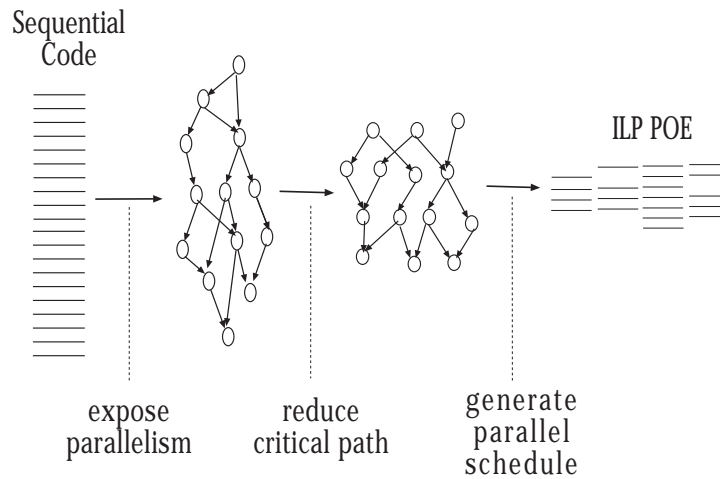


Figure 4.3: ILP processing steps

ILP research has led to two main styles of architectures: the **Very Long Instruction Word (VLIW)** and the **Superscalar**. The code for a VLIW processor is an explicit plan which specifies the set of operations to be issued on each machine cycle, which functional units to use to execute them and which registers to use as sources and sinks for input and output operands respectively for operations performed on those functional units. Borrowing the terminology from [140], this explicit plan is called **Plan of Execution (POE)**. On the other hand, the

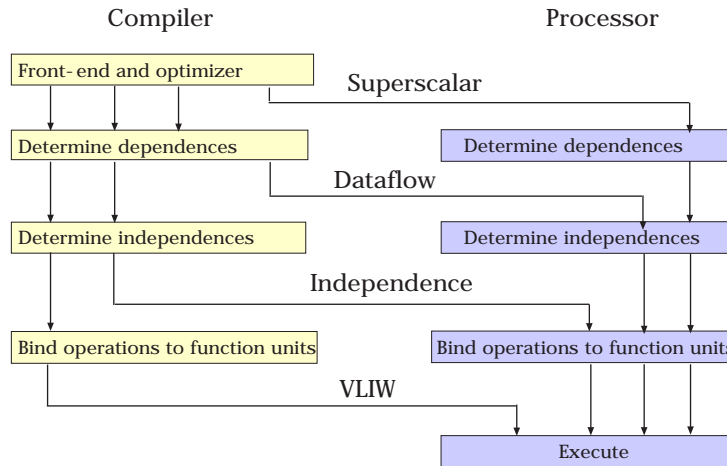


Figure 4.4: ILP taxonomy

program for a superscalar processor does not specify any such POE. The program submitted to a superscalar processor is a sequential stream of instructions, from which the processor dynamically generates a valid POE. A typical superscalar processor examines a window of instructions from the instruction stream, analyzes the chosen instructions for inter-instruction dependencies (and also checks for dependencies with instructions that have already been issued and not completed) and performs scheduling and resource allocation with the aim of extracting as much ILP as possible while processing those instructions [155].

In fact, several “intermediate” architectures are also feasible which are neatly captured in the ILP taxonomy proposed by Fisher and Rau [125]. This taxonomy categorizes architectures according to how the responsibility of performing various tasks of ILP processing are split between the processor and the compiler Figure 4.4. On the one extreme is the superscalar processor where all ILP processing steps (after the front-end processing and basic compiler optimizations are done) are performed by the processor. VLIW processors on the other hand are at the other extreme, performing only the final step of executing the instructions while the compiler performs all the ILP processing steps.

4.1.2 Limitations of current approaches to ILP

Several limitations of current approaches to ILP are highlighted to motivate the proposed ideas.

Limits on available ILP. Determining the limits of exploitable ILP in application programs can help in making processor design decisions and also indicate the limits of performance benefits one can expect from ILP processing. However, such studies can never be accurate as they have to make assumptions about the compilation and processor technology which may never be attained or (better yet) may be bettered in future. Early studies [164, 131] concluded that there is very limited parallelism in general purpose applications. Since then several “ILP limit studies” have been conducted [110, 83, 160, 158, 175, 176, 92]. Most of these studies concluded that, in general, the available ILP is limited, claiming that often the number of maximum instructions that can be issued on each cycle is typically less than 10. All these studies explain the general perception that *conventional* approaches to ILP may not improve performance dramatically. This is also evidenced by the fact

that several research groups have proposed alternative ILP processing approaches [159, 98, 174, 134, 112, 168].

Fixed Instruction Set Architecture. Current ILP processors are hardwired to support a *fixed* instruction set whether the application requires those instructions or not. For example, the floating point units on the processor are wasted if an application does not perform any floating point computations. This greatly limits the ability to improve performance if the instruction set needs of application change over the lifetime of its execution and result in poor utilization of processor resources.

Increasing complexity. Poor scalability (of primarily of the control circuitry) of dynamically scheduled architectures (superscalars) is a serious limitation to the amount of ILP that can be successfully exploitable. In a typical dynamically scheduled processor, dependence detection is an $O(n^2)$ operation while resource assignment is an $O(2^n)$ operation where, n is the number instructions processed at each step by the processor. In order to increase the ILP, a larger number of instructions would have to be examined, which in turn leads to ever more complex dynamic scheduling hardware. This affects the critical path of the design, increasing design time, design, verification and test costs [155, 115]. On the other hand, scalability is also an issue for VLIW processors as they rely heavily in multiple ports between the register files and the functional units (used to communicate several operands simultaneously on each cycle).

Media application shift. As general purpose processors become faster and cheaper, there has been a big push to perform traditional DSP and multimedia computations using purpose processors. However, the computational requirements of these applications do not match with those for which current ILP processors are most suitable. For example, several of these multimedia applications involve smaller bit-width, SIMD style, stream computations. Current architectures have been to extended (MMX) to take advantage of this shift. However, this is a temporary fix. As the variety of multimedia type applications increases, such fixes (which are additions to an already bloated ISA) add to the complexity of the micro-architecture.

4.2 Dynamic Instruction Set Architectures

We would like to consider architectures that can be customized on a per application basis. Clearly, some form of *programmable micro-architecture* seems necessary to allow it to be customized for the processing requirements of a particular application.

Dynamic Instruction Set Architectures (DISA) is a class of microprocessor architectures that provide an interface (a base ISA) that allows higher level software (such as the running program or the compiler) to extend the base ISA with additional instructions. The new instructions are presumably chosen to better support the needs of a particular application that is intended to be targeted to this processor. Here, extending the base ISA with new instructions not only means providing newer instructions that perform computations that are different from the ones supported by the operations from the base ISA, but also to provide efficient implementations for these new instructions directly in the micro-architecture instead of performing them using the base instructions.

A DISA compiler takes the base ISA and the program source code as inputs and generates the application specific ISA along with the executable code that makes use of instructions from the application specific ISA and the DISA processor performs the computations as specified in the executable in turn consulting the application specific ISA for instructions on how to implement custom instructions from the application specific ISA extensions. In contrast, compilers for **Static Instruction Set Architecture (SISA)** processors generate a single executable which uses instructions from the ISA and the machine is designed to recognize only the instructions from the published

ISA. This DISA model of extending instruction set architecture of processors is explored further in the remainder of this thesis.

4.3 Adaptive Instruction Level Parallel Processing

All ILP machines whether superscalar or VLIW, have one common characteristic: the number and types of operations that can be issued on each machine cycle is fixed. This set of operations is determined by the ISA and the subset of operations that can be issued on each cycle is smaller, limited by the available functional units (fixed in the processor) and other resource and data/control flow constraints.

In this section we introduce a new instruction level parallel processing model wherein the runtime system is allowed to reconfigure the micro-architecture of the base processor presenting a different ILP instruction set interface to the instructions following every processor reconfiguration phase. This class of architectures is referred to as *Adaptive Instruction Level Parallel* (AILP) architectures. A program intended for execution on an AILP machine is composed of *computation* and *reconfiguration* instructions leading to an execution that is composed of an alternating sequence of computation followed by reconfiguration phases (see Figure 4.5). During a computation phase the AILP machine behaves just like a traditional ILP machine. Each reconfiguration phase switches the AILP machine from one ILP machine configuration to another ILP machine configuration.

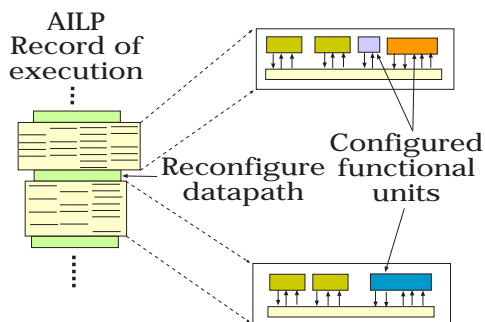


Figure 4.5: AILP Computation

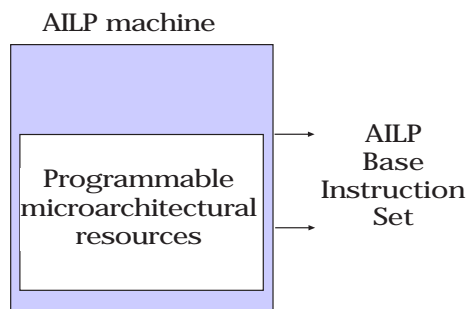


Figure 4.6: AILP high-level interface

Although there are several aspects of an ILP architecture that can be considered potential targets for reconfiguration, our main interest is on the investigation of architectures that allow controlling software to reconfigure the machine in terms of its *instruction set capabilities* at runtime.

In the following, we start by describing AILP class of architectures at a very abstract level. Later we provide a classification of the AILP space based on some key properties a given instance of an AILP architecture may have.

4.3.1 AILP processing

Figure 4.7 illustrates the basic transformations involved in transforming a sequential program to an executable intended for execution on an AILP machine. The starting point is an intermediate code that is obtained about lexical and syntactic analyses of sequential code written in some high-level language such as C or C++. The critical transformations specific to AILP processing are:

1. **Code partitioning.** The sequential intermediate code is partitioned into (not necessarily) disjoint regions of code. Each partition could represent a large piece of computation such as a loop-nest in the source

program or it could represent a simple abstract operation such as an operation that adds two fixed size data values. The goal of code partitioning step is to identify the most suitable operation set for the given application. Each code partition is intended to be replaced by an operation synthesized to perform the computation specified by that code partition.

2. **Instruction set synthesis.** During this phase, the candidate partitions identified in the earlier phase are synthesized into suitable micro-architectures (“implementations” of new instructions).
3. **Instruction selection.** Having synthesized the instruction set, the next step is to associate the most suitable instruction for a given code partition for each of the code partitions identified in step 1. It is quite possible that multiple implementations have been synthesized for each partition providing different tradeoffs in terms of performance and cost of resources consumed. Instruction selection phase considers the global instruction requirements and makes a decision about instruction assignment to each code partition.
4. **Generate AILP POE.** During this phase, the AILP compute and reconfigure instructions are inserted into the intermediate code and the intermediate code is scheduled and optimized for execution on the AILP processor. The compute instructions invoke the synthesized instructions associated with each code partition while the reconfiguration instructions reconfigure AILP micro-architecture so that it is capable of performing the operation invoked by the compute instructions. Clearly each compute instruction has to be preceded by at least one instance of reconfigure instruction that instantiates the implementation for the operation performed by the compute instruction.

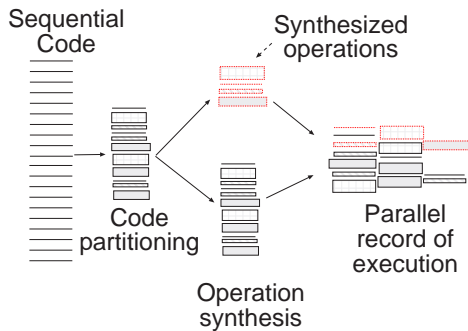


Figure 4.7: AILP processing steps

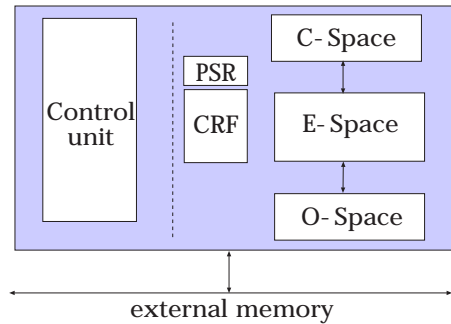


Figure 4.8: AILP machine state

The above description is intended to give a general idea about the steps involved in AILP processing. The specifics of each of how these steps are performed are better dealt within the context of a specific AILP architecture.

The AILP model represents a wide class of architectures. In order to gain a deeper insight into the architectural and compilation problems involved in AILP processing, we will proceed to refine the model to arrive at a specific AILP architecture. This AILP architecture will serve as our basis for AILP processing research.

Our approach for defining this AILP architecture follows these steps: (a) define an abstract machine model for AILP, (b) based on the machine model, a taxonomy of AILP architectures is presented and (c) a specific subclass of AILP architectures is identified for further investigation. In the rest of the thesis, we focus on a specific instance of this subclass of architectures.

4.3.2 AILP Machine Model

At an abstract level, an AILP machine is composed of (a) an architectural state possibly composed of programmable micro-architectural resources and (b) a base ISA (see Figure 4.6). The base ISA is used by the runtime system to reconfigure the AILP machine. The architectural state and the base ISA are described below.

4.3.2.1 Architectural state

The architecturally visible resources can be partitioned into the following basic categories (refer to Figure 4.8).

1. **E-Space** is the *Execution Space*. It is an abstraction of programmable logic resources that can be configured to implement desired functional units—the micro-architectures corresponding to synthesized instructions.
2. **C-Space** is the *Configuration Space*. It serves as a local memory for *configurations*. Configurations are the program bits that are used to configure the synthesized functional units on the E-Space.
3. **O-Space** is the *Operand Space*. O-Space is the local memory intended for storing either the input values or to provide locations for the output operands of operations performed on the functional units (resident on the E-Space). O-Space may be comprised of a collection of registers, FIFO's, vector registers, etc., depending on the specific instance of AILP architecture of interest.
4. **Configuration Register File (CRF)**. Each configured functional unit on E-Space is associated with a *configuration register* from the CRF. These configuration registers serve as *aliases* to the configurations currently resident inside the AILP machine and are used by instructions to refer to the configured functional units.
5. **Processor Status Register (PSR)**. PSR can be initialized with the execution status of a particular instruction on any of the configured functional units.

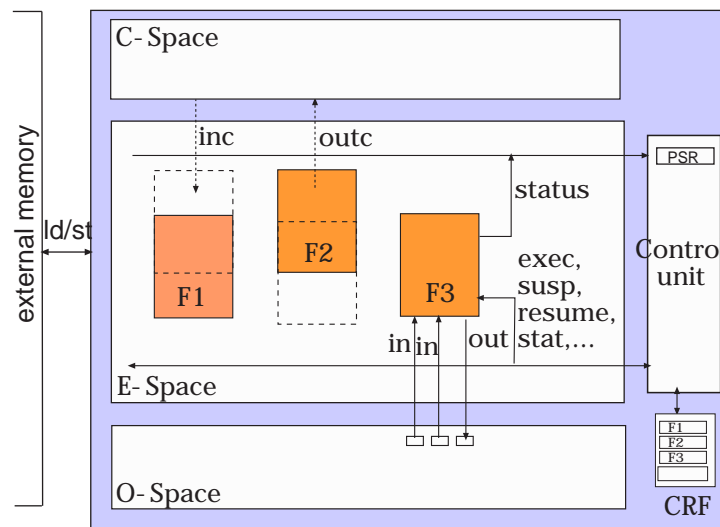


Figure 4.9: AILP machine state and instructions

4.3.2.2 Instruction set

The Base Instruction Set Architecture (Base ISA) consists of the instructions listed in Table 4.1 grouped into the following categories.

1. **Reconfiguration instructions.** Two instructions `inc`, `outc` swap configuration data in/out of E-Space from C-Space in essence, changing the functional unit composition of the data-path. Before functional units can be configured on the E-Space, the configuration data should be available in the C-Space. `alloc` and `dealloc` instructions allocate space on the C-Space for configuration data loaded from memory into C-Space.
2. **Instructions for data supply.** The instructions `in` and `out` are used to assign locations from O-Space to serve as data sources for input operands and destinations for output operands of configured functional units respectively.
3. **Computation instructions.** The basic instructions for controlling the execution of operations on configured functional units are: `exec`, `abort`, `susp`, `resume`, `reset`. These instructions initiate instruction processing, abort, suspend, resume instruction being processed currently. In general, configured functional units may be capable of performing multiple operations, in which case, `opid` specifies the specific operation to perform. Other computation instructions need not specify the `opid` since configured functional units are assumed to be capable of executing only one instruction at a time.
4. **Control flow instructions.** The branch instruction provides the necessary control flow change capability. The condition value could be specified as an immediate or through a register. Status of a particular instruction execution (on a configured functional unit) can be read into the PSR with the `pstat` instruction.
5. **Memory access.** Finally, load/store instructions move data between external memory and the C-Space and O-Spaces.

Table 4.1: AILP instruction set

Reconfiguration instructions	
<code>alloc r, cr</code>	Allocate a configuration and associate it with configuration register <code>cr</code> . Configuration data is loaded from memory location referenced by <code>r</code> into C-space.
<code>dealloc cr</code>	De-allocate a configuration associated with <code>cr</code> . All resources consumed by the configuration are freed for future allocations.
<code>inc cr, r</code>	Configuration associated with <code>cr</code> is brought into the E-Space at the location specified in <code>r</code> . Loading the configuration into the E-Space effectively creates a functional unit (the <i>Configured Functional Unit</i> (CFU)) on the E-Space and is available for processing instructions.
<code>outc cr</code>	Configuration associated with <code>cr</code> is moved from E-space to C-Space.

Data supply instructions	
<code>in cr, k, r</code>	Associate register <code>r</code> from the O-Space as a source for the k^{th} input operand values to the CFU associated with <code>cr</code> .
<code>out cr, k, r</code>	Associate register <code>r</code> from the O-Space as sink for the k^{th} output operand values to the CFU associated with <code>cr</code> .

Computation instructions

Table 4.1: AILP instruction set

<code>exec cr, opid</code>	Perform operation <code>opid</code> on CFU associated with <code>cr</code> .
<code>susp cr</code>	Suspend operation currently being processed on CFU associated with <code>cr</code> .
<code>resume cr</code>	Resume operation currently being processed on CFU associated with <code>cr</code> .
<code>reset cr</code>	Reset CFU associated with <code>cr</code> .
<code>abort cr</code>	Abort operation currently being processed on CFU associated with <code>cr</code> .
Control flow instructions	
<code>brc r, cond</code>	Perform a conditional transfer of control depending on the condition value <code>cond</code> to AILP instruction whose address is the value in <code>r</code> .
<code>pstat cr, cc</code>	Load the result condition of type <code>cc</code> of the most recent operation on CFU associated with <code>cr</code> and save it into the processor status register.
Memory transfer instructions	
<code>ldcc cr</code>	load configuration data into resources allocated for <code>cr</code> 's configuration.
<code>stcc cr, r</code>	store configuration data into memory at offset <code>r</code> of configuration associated with <code>cr</code>
<code>ld r, a</code>	load memory word at address <code>a</code> into register <code>r</code>
<code>st r, a</code>	store into memory at address <code>a</code> value from register <code>r</code>

The above description captures the essential features of the architectural interface of the AILP class of architectures.

4.3.3 AILP Taxonomy

A taxonomy of the AILP space based on five essential design decisions is presented here. Choices for these design decisions would significantly affect how these machines are programmed and their eventual cost-performance factors. Figure 4.10 lists the five dimensions and the choices available for each. Brief explanation for each follows.

4.3.3.1 Resource allocation

Resource allocation refers to the allocation of programmable logic resources from the E-Space for configured functional units. This function can be performed by the compiler (*static allocation*) or can be decided by the processor at runtime (*dynamic allocation*). In the former case, either the `inc` operation is expected to specify the E-Space region for the functional unit to be configured or the AILP instruction set should be extended with an additional instruction which communicates compiler's allocation decisions to the processor. For dynamic allocation, additional architectural resources may be required to maintain the E-Space resource usage map and extra logic to perform allocation/de-allocation operations on the E-Space.

4.3.3.2 Temporal reconfigurability

Temporal reconfigurability refers to the ability to alter AILP machine configuration over time. The two possible choices are *static reconfigurability* and *dynamic reconfigurability*. In the former case, the machine allows software to configure the machine only once: before the start of execution. In the latter case, software is allowed to repeatedly alter the machine configuration during runtime. While the latter provides greater flexibility, the programmable logic resource would be less complex in the case of static reconfigurable machines.

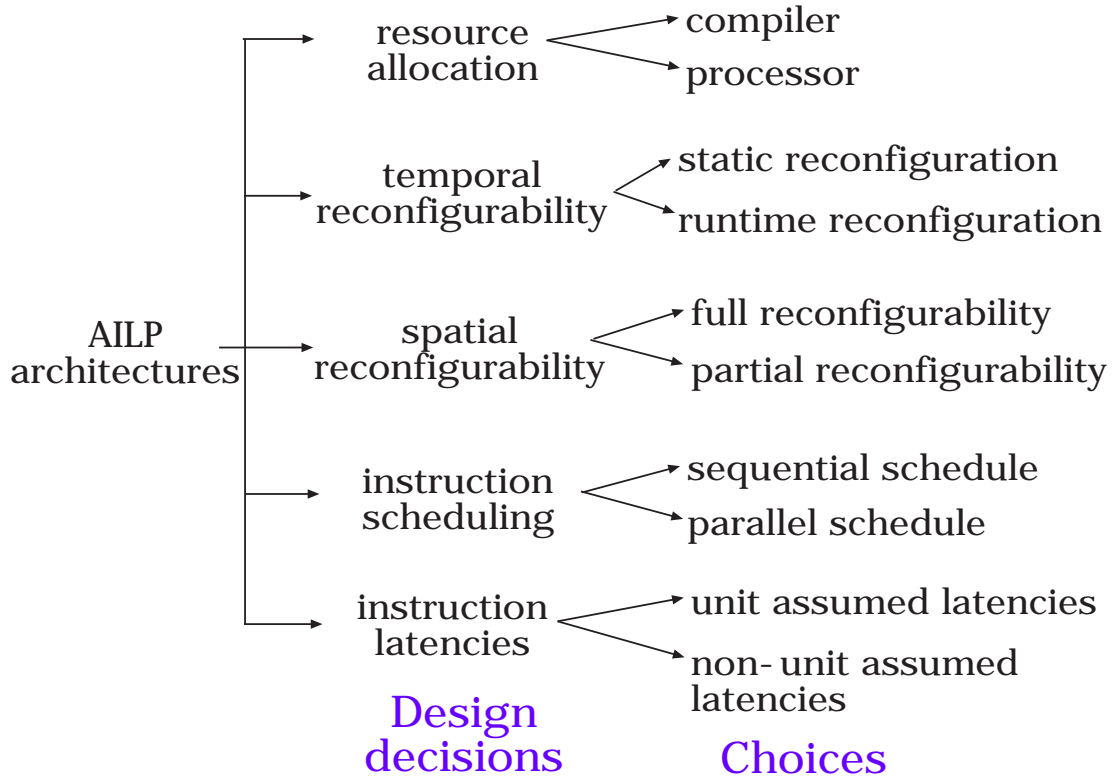


Figure 4.10: AILP taxonomy

4.3.3.3 Spatial reconfigurability

Here we are concerned with the ability to reconfigure the available programmable resources in a piecemeal manner in which case, the AILP machine is said to be *partially reconfigurable*. Partial reconfiguration requires that the machine allow software to address all programmable cells of configurable resources. This could significantly increase the area consumed by the programmable resources. On the other hand this feature improves performance it is cheaper to reconfigure only the necessary portion of the device.

4.3.3.4 Instruction Scheduling

Here the distinction is carried over from the ILP domain - namely how the parallel execution (on the multiple configured functional units) is specified and communicated to the machine. Two classes arise—the statically scheduled and the dynamically scheduled AILP architectures. A statically scheduled machine is analogous to the VLIW or EPIC machine. In this case the program specifies (as determined by the compiler) when (the exact cycle) and where (the functional unit) the operations execute. Similarly, the dynamically scheduled AILP architecture is analogous to the superscalar architecture - dependence checking among operations and their binding to functional units is performed in hardware at runtime.

4.3.3.5 Instruction latencies

Here we are concerned about the compiler’s view of operation latencies. The two significant choices are *unit latencies* and *non-unit latencies*. Since these latencies need not correspond to the true latencies of these operations, they are usually referred to as *unit assumed latencies* (UAL) and *non-unit assumed latencies* (NUAL). These

assumed latencies are part of the contract (architecture specification) between the compiler and the processor. Compiler assumes these as the actual latencies when making its scheduling and allocation decisions while processor should provide the illusion that operations actually exhibit such latencies even if the actual latencies are different from the assumed ones. Most sequential ILP architectures are UAL while typically the statically scheduled ILP architectures are NUAL. For statically scheduled machines, since the compiler generates the parallel POE, a more accurate view of operation latencies can help create compact schedules. Since the actual latencies may be unknown or non-deterministic, the compiler is only provided with an assumed latency (that hopefully approximates the actual latencies quite well).

In all, the above categories yield 32 classes of AILP architectures. Several of these architectures are either impractical or infeasible. A few interesting ones are shown in Table 4.2.

<i>RA</i> = resource allocation policy	<i>RR</i> = Runtime Reconfigurable
<i>TR</i> = Temporal Reconfigurability	<i>SC</i> = Statically Configured
<i>SR</i> = Spatial Reconfigurability	<i>PR</i> = Partially Reconfigurable
<i>IS</i> = Instruction Scheduling	<i>NP</i> = Non-partially Reconfigurable
<i>LA</i> = Latency Assumption	<i>SS</i> = Statically Scheduled
<i>C</i> = Compiler	<i>DS</i> = Dynamically Scheduled
<i>P</i> = Processor	

Table 4.2: A few AILP architectural subclasses

Architecture	RA	TR	SR	IS	LA
AILP-1	C	RR	NP	SS	NUAL
AILP-2	C	RR	PR	SS	NUAL
AILP-3	C	SC	NP	SS	NUAL
AILP-4	P	RR	NP	SS	NUAL
AILP-5	P	RR	PR	SS	NUAL
AILP-6	P	RR	NP	DS	UAL
AILP-7	P	RR	PR	DS	UAL

4.4 Adaptive Explicitly Parallel Instruction Computing

The space of AILP processing architectures is quite large and hence is not a good starting point for a detailed investigation. Our goal is to pick a suitable member from the AILP space and define it to the extent that a compiler may be built to target machines of that architecture. Our intent is to learn about the capabilities of such an architecture. The subset of architectures we shall focus on correspond to the AILP-5 category in Table 4.2. This set of architectures we call *Class of Adaptive Explicitly Parallel Instruction Computing* architectures and refer to it as C_{AEPIC} . Figure 4.11 captures the relationship between the different classes of architectures introduced so far. The AILP architectures form a subset of the Dynamic Instruction Set Architectures while the C_{AEPIC} class is a subset of the AILP set of architectures.

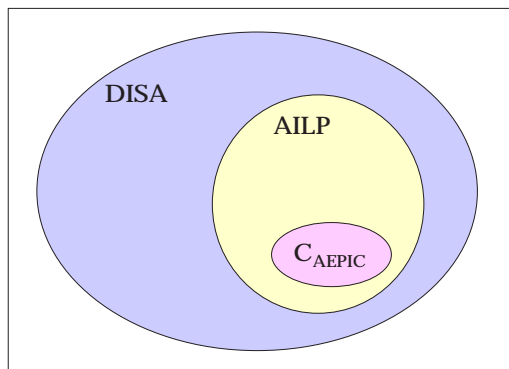


Figure 4.11: Dynamic instruction set architecture space

4.4.1 The C_{AEPIC} Class

A program intended for execution on a machine from the C_{AEPIC} AILP subset, would consist of a sequence of **MultiOp** instructions where each MultiOp instruction specifies a set of AEPIC operations that are to be issued for execution concurrently (at the start of a machine cycle). A new MultiOp instruction is issued for execution every machine cycle. A compiler for targeting an C_{AEPIC} machine determines which AEPIC operations can be issued during the current cycle and then packages them into a MultiOp. The number and types of AEPIC operations in a MultiOp is dictated by the capabilities of the specific instance of an AEPIC architecture. To summarize, a C_{AEPIC} machine is defined by:

- instruction scheduling is performed by the compiler
- compiler assumes non-unit latencies for operations
- instructions in the program explicitly specifies latencies
- the AILP machine is runtime reconfigurable
- the AILP machine is partially reconfigurable

4.5 Summary

We presented a novel variant of the Instruction Level Parallel (ILP) processing model called Adaptive Instruction Level Parallel (AILP) processing. The two key characteristics that define AILP processing are (a) instruction level parallel processing and, (b) dynamically specified instruction sets. AILP is a subset of a larger space of architectures called *Dynamic Instruction Set Architectures*. Key features of AILP architectures are described and then a taxonomy of the AILP space is presented based on these key features. A specific instance of the C_{AEPIC} class forms the basis for our architecture and compiler research and is the subject of the next chapter.

Chapter 5

Adaptive Explicitly Parallel Instruction Computing (AEPIC)

Don't you have a machine that puts food into the mouth and pushes it down?
–Nikita S. Khrushchev

In the previous chapter we introduced Dynamic Instruction Set Architectures (DISA) and presented a taxonomy of a subset of DISA called Adaptive Instruction Level Parallel (AILP) processing architectures. This taxonomy partitions AILP space into 32 subclasses one of which is referred to as C_{AEPIC} . In this chapter, we specify an instance of the C_{AEPIC} class called **AEPIC** in detail. AEPIC architecture is motivated by a desire to

- enable efficient reconfiguration of processor data-path at runtime,
- allow compiler to determine such reconfiguration decisions in a flexible and efficient manner and,
- allow AEPIC researchers to study a wide variety of AEPIC machine instance configurations.

Rest of the chapter is organized as follows. Key issues of to be considered in the design of dynamic instruction set architecture are presented in Section 5.1. The computation and machine models of the proposed AEPIC architecture are given in Section 5.2. Also, a summary of the key features of the architecture and how these features address the issues raised in Section 5.1 are presented in Section 5.2. AEPIC machine state details are in Section 5.3. A summary of AEPIC instruction set vis presented in Section 5.4 and the architectural parameters are listed in Section 5.5. Issues related to interrupts and exceptions and other variants of AEPIC machines are discussed in Section 5.6. Section 5.7 summarizes the chapter.

5.1 Dynamic Instruction Set Architectures: Issues

5.1.1 Long Reconfiguration Times

Let us consider the IDCT kernel, used in several multimedia applications. Table 5.1 lists the code size and reconfiguration time on various architectures. In the case of the EPIC processors, the reconfiguration time is simply the number of cycles required to fetch the instructions while in the case of the Xilinx programmable logic arrays, it is the number of cycles to load the configuration bit-stream. Note that for both cases, we assume equal memory bandwidth and access latencies (64 bits/cycle).

Architecture	Computation Cycles	Code Size	Reconfiguration cycles
HPL-PD EPIC	12127	<2KB (184 ops)	92
Xilinx XC4K	544	>100Kb (4920 CLBs)	≈ 1600
Xilinx Virtex	26	>1Mb (6140 slices)	≈ 16000

Figure 5.1: Configuration/code size for IDCT

From this table it is clear that even though the execution time is greatly reduced using programmable logic (column 2 in the table), the overhead of reconfiguring the machine reduces the benefit. We would like to mask the overheads of reconfiguration or use programmable logic when the costs of reconfiguration are negligible compared to the gains in execution time (say, if the IDCT kernel were to be executed a lot more times).

5.1.2 Large And Non-uniform Configuration Sizes

From Table 5.1, it is clear that configuration sizes can be quite large. The sizes also vary depending on the computation that is mapped onto the programmable logic. In the case of traditional processors instructions and data values have uniform sizes. So these machines can be built with identical storage elements. This implies that it does not matter which locations are used to store data values in the processor ¹. In the case of the configurations, we need fast and efficient schemes to allocate *variable* amount of storage.

5.1.3 Context Switching Overheads

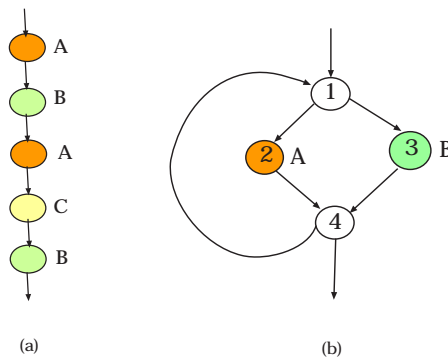


Figure 5.2: CFU context switching

Context switching is the swapping configurations in/out of programmable logic as the needs CFU needs of the application change. Context switching costs are the same as that for the reconfiguration. However, in some cases the slack for masking the reconfiguration overhead is minimal. This situation is illustrated through the example in Figure 5.2(b). If the computation alternates between the two paths 1-2-4 and 1-3-4 and that the available programmable logic resources cannot accommodate both the configurations (A and B) simultaneously, the processor has to be reconfigured rapidly (on every iteration) to switch between A and B configurations.

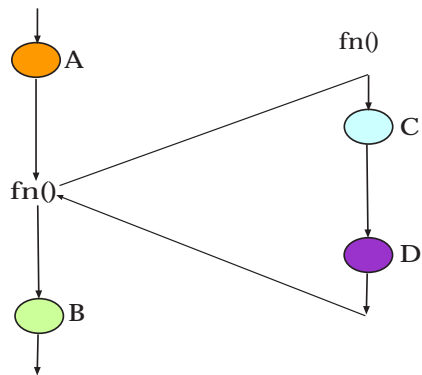


Figure 5.3: Effect of function calls

5.1.4 Modular Software Development

In Figure 5.3, function fn could be an external function linked by the current application. We would like fn to make use of programmable logic for improving its performance. However, when fn is developed, we cannot foresee all the contexts in which it might be used and hence cannot assume anything specific about which resources are available for configured functional units instantiated by fn . We would like fn to consider that all the resources are available and that after return from a call to fn , the resource usage map of the calling function is as if the call to fn did not cause any changes.

5.1.5 Large Variation In Operation Formats

Formats of operations depends on the code partition that is synthesized as the configured functional unit which performs that operation. A large variation in operation formats is expected. Hence we would like to design the processor that allows such variation in operation formats.

5.1.6 Non-deterministic Latencies

In fixed instruction set processors, instruction latencies (when operands are read/written with respect to issue time) are known at design time and processors are “hard-coded” with this knowledge. In the case of dynamic instruction set processors, the instruction set is not known and hence the latency information has to be conveyed to the processor at (or before) runtime.

5.1.7 Other Architecture Desirables

In addition to the issues highlighted above, we would like our “dynamic instruction processor” to satisfy several other constraints.

- Simplicity in design - simpler architectures reduce design and verification time and are also cheaper to manufacture.
- Compilability - architecture should support efficient compilation. This is one of the key requirements and the motivation behind the choices we made for the proposed architecture.

¹Not all registers are the “same” in certain DSP processors. But for the most part, the variations/exceptions are minor.

- It has been observed that several of the benchmark applications declare arrays of constants which are often referenced in compute intensive loops. Our architecture should support local caching of constant arrays as it can have a big impact on eventual performance.
- It is unlikely we would be able to get configuration right the first time. In fact such a configuration may not exist (application domains might have different needs).

5.2 Adaptive Explicitly Parallel Instruction Computing

Adaptive Explicitly Parallel Instruction Computing (AEPIC) architecture is motivated by a desire to (1) enable efficient reconfiguration of the processor data-path at runtime, (2) allow compiler to determine the reconfiguration decisions in a flexible and efficient manner and (3) allow AEPIC researchers to study a wide variety of AEPIC machine instance configurations. These motivations and the concerns listed in Section 5.1 crystallized into a set of architectural features that characterize the AEPIC architecture.

The architectural description is restricted to those aspects of the machine that are needed by a compiler to generate correct code. Hence for instance, implementation details of a micro-architecture, though important, are not discussed.

5.2.1 AEPIC Computation Model

Conceptually one can view program execution on an AEPIC machine as being composed of an alternating sequence of *reconfigure* followed by *execute* phases. During the *execute* phase, the AEPIC processor is functionally equivalent to an EPIC processor with similar functional unit composition. Each *reconfigure* phase can be viewed as a transition between two EPIC machines which differ only in their functional unit composition. In practice, for an AEPIC machine these two phases may overlap in time dictated by considerations of efficiency (explained later).

The memory space of an AEPIC process is partitioned into multiple segments. One particular segment contains the program code. Other memory segments contain the set of *configurations* (segment marked C in the executable of Figure 5.4), global data, dynamically allocated memory and the program stack.

The set of functional units on the AEPIC data-path can be partitioned into two sets:

1. *Hardwired functional units.* These functional units execute instructions from the AEPIC ISA. In Figure 5.4, F_1, F_2, \dots, F_5 are the hardwired functional units.
2. *Configured functional units.* These are the functional units that have been configured into the data-path by some of the “reconfiguration” instructions from the AEPIC ISA (which are executed on the hardwired functional units). Operations performed by the configured functional units are triggered by specific AEPIC instructions (described in Section ??) invoked on the hardwired functional units. Operation executed by the configured functional units are not part of the AEPIC ISA. In Figure 5.4, C_1, C_2, C_3 are the configured functional units.

An AEPIC processor relies on the compiler to specify the exact set of operations to be issued on each machine cycle and on which functional unit they are expected to execute. The mechanism used to communicate this explicit Plan Of Execution (POE) is called the **MultiOp** [140]. On each machine cycle, a single MultiOp is issued containing exactly one operation to be processed by each of the *hardwired* functional units. Some of the operations of the MultiOp might trigger reconfiguration of the processor data-path by changing the functional unit composition of the configurable portion of the data-path. Subsequent MultiOps may contain operations that

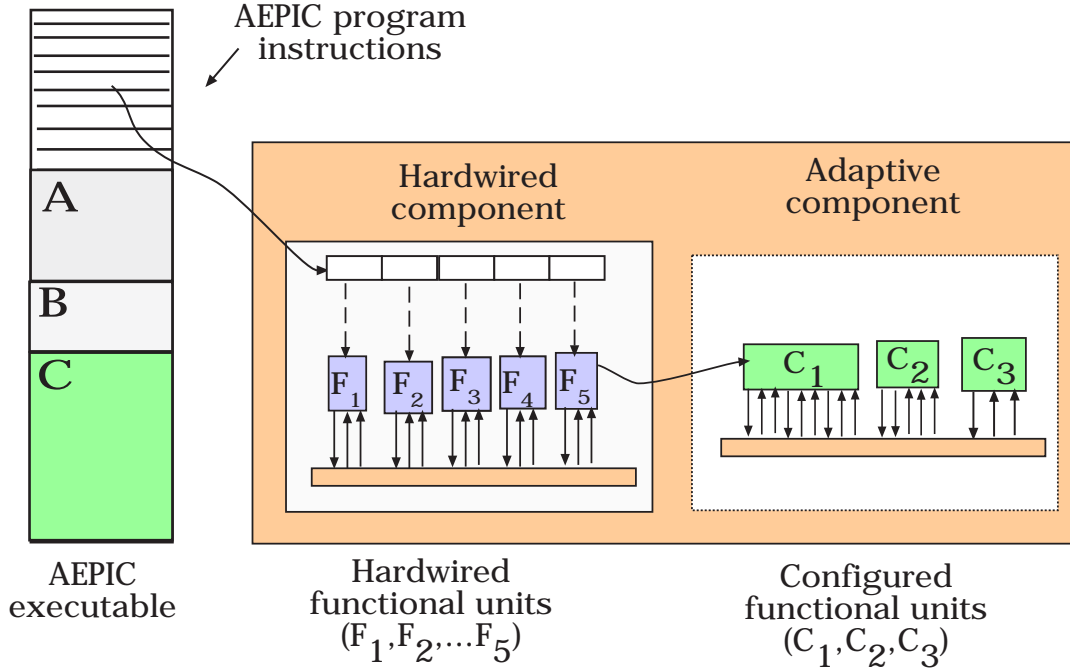


Figure 5.4: AEPIC executable and abstract data-path

trigger execution on the CFUs. These computations are indirectly triggered through the execution of special operations of the AEPIC ISA (described later) on the hardwired functional units.

AEPIC hardwired functional units are intended for performing non data-processing tasks that are common to all applications such as control flow changes (branching), data memory transfers, standard arithmetic and logic operations, exception processing. And some of the hardwired functional units are used to execute operations that reconfigure the data-path or trigger execution on the configured functional units. The configured functional units are intended to provide application specific advantages in terms of speedup, cost or power reduction - depending on the application performance requirements.

5.2.2 Machine Model

An abstract model of an AEPIC machine is shown in Figure 5.5. The processor can be viewed as composed of two sets of components: (1) the EPIC core and (2) the adaptive extension.

5.2.3 The Adaptive Extension

The adaptive component of the AEPIC processor consists of the **Configuration cache hierarchy**, **Multi-context Reconfigurable Logic Array (MRLA)** and **Array Register File (ARF)** connected together via bus interconnect (Figure 5.5). The MRLA provides the programmable logic resources to host the Configured Functional Units (CFUs). The *C-Cache* serves as a temporary cache for *configurations* before they are instantiated on the MRLA. This is analogous to the way registers serve as storage for program values. Rest of the

configuration cache hierarchy consists of the **C1 cache** connected to external memory. The **Configuration Register File (CRF)** consists of a set of **configuration registers (CRs)**. Each CR serves as an alias to either a configured functional unit or a configuration allocated in the *C-Cache*. Most of the AEPIC instructions take a configuration register as an operand. These are the AEPIC instructions that perform operations such as delete a CFU, etc. The instruction refers to the CFU by its alias—the configuration register. For example, the delete CFU operation in AEPIC is **DEL_C cr, L, p**. Here, **cr** is the configuration register associated with the desired CFU, **L** the latency assumed by the compiler for this operation and **p** the predicate guard for this operation. CRF details and their usage are described in 5.3.2.

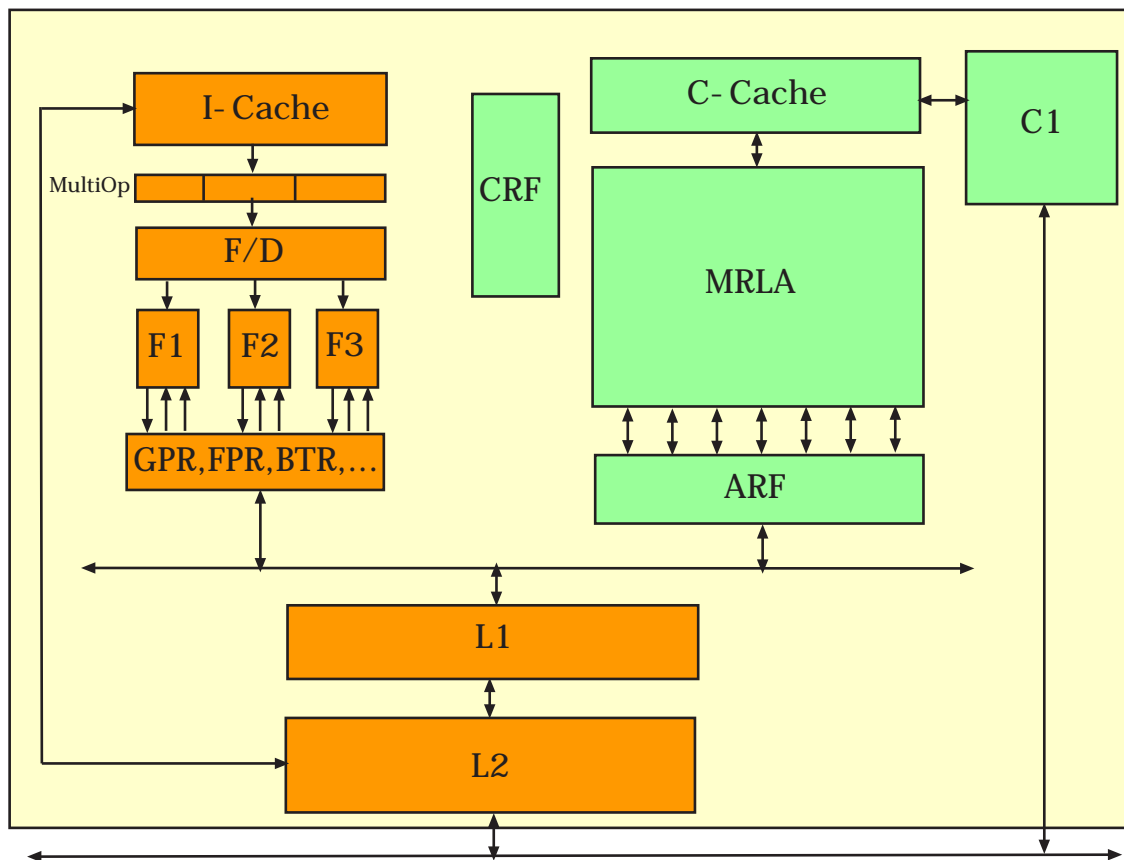


Figure 5.5: AEPIC machine model

5.2.4 The EPIC Core

The hardwired portion of the AEPIC data-path executes the AEPIC ISA. This hardwired component is composed of the EPIC architecture as embodied in [140] with additional functional units to execute the non-EPIC instructions of the AEPIC ISA. Execution of some of these non-EPIC instructions causes CFU instruction execution or state changes on the adaptive component.

The instruction cache, the standard VLIW/EPIC instruction processing engine and the generation of control signals for the various control ports of the data-path comprise the control unit. Control logic is not only re-

sponsible for the fetching and processing of instructions in program order, but is also responsible for flushing or stalling the pipelines of the functional units caused by program events such as branch operations or interruptions. Apart from the assumption that a new MultiOp instruction is fetched for issuance every machine cycle, no other aspect of the control unit is visible to the compiler and is not the subject of any further investigation in this thesis.

Referring back to the AILP model described in Section 4.3.2, the *C-Cache* corresponds to the *C-Space*. MRLA along with the hardwired functional units correspond to the *E-Space*. The ARF and the standard register files that are connected to the various functional units comprise the *O-Space*.

5.2.5 Summary Of Key Features

5.2.5.1 Architectural Support For Runtime Reconfiguration

In order to support efficient runtime reconfiguration, AEPIC architecture provides the following support:

Compiler specified resource allocation. Here we are referring to resources that are intended for hosting configured functional units (CFUs). AEPIC delegates to the compiler the task of specifying which regions of the program code will execute on the programmable logic and when they are allocated (deallocated) to (from) the programmable resources on processor. Allocation of programmable logic resources to configurations is analogous to allocation of registers to program values. This is explored further in Section 6.5.

Architecturally transparent resource assignment. Although the AEPIC compiler decides which particular piece of computation should be performed on the programmable logic and at which instant (part of the resource allocation activity mentioned above), exactly which region of the programmable logic resource is utilized for hosting the CFU is determined by the processor itself. This is because the programmable logic resource availability information is more accurate at runtime. Efficient hardware resource assignment schemes are presented in Section 6.5.12.

Support for efficient context switching and modular software development. AEPIC architecture allows multiple CFUs to be instantiated simultaneously and group them in different sets where at any instant, a particular set of CFUs is considered active—these are the ones on which operations can be executed. Architecture also provides special instructions to alter these CFU sets or switch active set. This mechanism can also be used to support modular software development.

Explicitly controlled configuration cache hierarchy. AEPIC provides architectural mechanisms to explicitly control the data placement in the configuration cache hierarchy. These mechanisms can be used by the compiler to override default hardware cache management policies when the compiler has sufficient information about the configuration memory access patterns and is able to determine a more efficient policy than that provided by the hardware. This feature is a natural extension of the explicitly controlled data cache hierarchy mechanisms provided in some EPIC architectures [140, 84, 42]. It is expected to play an even more significant role in the AEPIC processing where the costs of configuration cache misses can be very expensive. On the other hand, since applications are expected to have a much smaller number of configurations than the number of program values (which go through the traditional cache hierarchy), explicit control of configuration data placement is expected to be feasible and advantageous.

Implicitly specified operands for configured functional units. Unlike typical RISC operations, some of the operations performed by CFUs may take a large number of input/output operands. In order to simplify the instruction decode logic and to keep the instruction format simple, operands for CFU operations are not specified

as part of the instruction itself. Instead, AEPIC architecture specifies “operand assignment” operations that associate specified registers as sources (destinations) for input (output) operands for CFU operations.

Explicit parallelism MultiOp frees the hardware from performing dependence checking and resource allocation tasks at runtime by packaging as one unit the operations to be issued concurrently, and, encoding the functional unit on which they are expected to execute as part of the MultiOp packet itself. The POE is then an ordered sequence of MultiOps.

Non-Unit Assumed Latencies (NUAL) Although the POE specifies that a new MultiOp is to be issued on each machine cycle, it does not imply that the operations in the MultiOp actually finish before the next MultiOp is issued. Constituent operations of a MultiOp could take several cycles. This is especially true in the context of reconfigurable computing where the application specific operations are usually large pieces of computation. In fact, operations of the same MultiOp may generate their results at different times. In contrast, the complete operation is considered atomic in the case of superscalar architectures. *This non-atomicity implies that AEPIC operations exhibit non-unit latencies.* The latency information should be available to both the compiler which uses this information to generate a valid and optimal POE, and to the processor so that it ensures a correct interpretation of the compiler specified POE. The NUAL terminology was introduced in the context of the EPIC architecture work [140, 126].

Explicitly specified latencies In the case of fixed instruction set architectures such as EPIC, both the compiler and the processor are built with the knowledge of the operation latencies, obviating the need for communicating the latency information through the POE. However, in the case of AEPIC, the operation set is determined only at compile time and the processor cannot be built with the knowledge of all possible instruction sets that a compiler may synthesize. Hence the processor has no knowledge of operation formats, their latencies or their semantics. Consequently, the compiler has to communicate the latency information to the processor. There are several possible ways to communicate latency information [141]. The mechanism we advocate is one where the latency of each operation is explicitly specified as part of the operation itself. Often the custom operations (performed by the CFUs) not only have long latencies, but the latency behavior is not deterministic and depends on the values of the input operands. Explicitly specifying latency as part of the instruction field yields the greatest amount of flexibility since every invocation of the operation can specify a different latency value—the value that the compiler assumed for this particular instance of the operation. Depending on the difference between the actual latency and the compiler communicated latency, the processor may take the appropriate action—whether to introduce stall cycles on the functional unit pipelines or to delay committing the generated results.

Architectural features inherited from EPIC In addition to the above features, AEPIC architecture inherits several EPIC architectural features. Later we shall see how these features are useful in the context of reconfigurable computing. Details can be found in [140, 84, 9, 8].

1. *Speculative execution.* Speculation is a technique to break certain types of dependencies between operations in order to enhance parallelism [157].
2. *Predication.* Predication is a technique to enforce program control flow without the use of branch operations. In some cases predication is an efficient way to implement conditional branches and also provides greater freedom for code motion [103, 141, 10].
3. *Decoupled branch architecture.* Decoupled branches permit processing of branch operations in stages some of which may be performed as soon as the required data is available so that the branch target pre-fetching can be initiated as soon as possible.

4. *Efficient boolean reductions.* Critical path reduction is a crucial technique for enhancing available parallelism in program codes. Efficient boolean reductions is a collection of architectural features that allow rapid computation of boolean functions that may be required to evaluate branch conditions and predicate expressions guarding non-branch operations.
5. *Programmatic data cache management.* The architecture supports explicitly controlled cache hierarchy as in the HPL-PD architecture [84]. The key features are compiler control of data placement in the cache hierarchy, runtime memory disambiguation and ordered processing of memory operations within the MultiOp.

5.2.5.2 Parameterized Architecture

AEPIC is a parametric processor architecture. This means that several aspects of the AEPIC architecture are not concretely specified. For example the amount of programmable logic resource available for configured functional units is specified as a parameter. However, for a particular compilation, concrete values are required. Concrete values for all the parameters are specified in a *machine description database* which is read by the various modules of the compiler and simulator. A parametric description of the architecture facilitates exploration of a wide range of processors configurations with varying amount of resources.

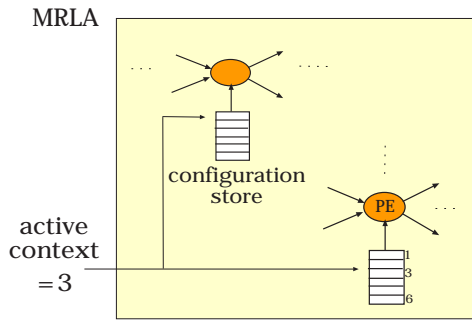


Figure 5.6: Structure of MRLA

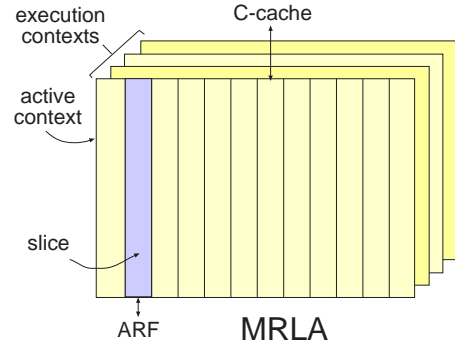


Figure 5.7: MRLA multiple contexts

5.3 Details Of The Architecture

5.3.1 Multi-context Reconfigurable Logic Arrays

Multi-context Reconfigurable Logic Array (MRLA) is the primary resource used for hosting the configured functional units. Like a typical Field Programmable Gate Array (FPGA), the MRLA is a two dimensional region of the processor die that is composed of programmable logic and interconnect blocks. We shall use the term **Programmable Element (PE)** to refer to both the programmable logic block as well as the programmable interconnect block. Each PE is associated with a *configuration instruction* (its “program”) which determines the behavior of that programmable element. Any given logic design can be emulated on the MRLA by supplying suitable *configuration instructions* for all the programmable elements of the array. In the context of AEPIC machines, these logic designs are the CFUs.

In a standard FPGA only one configuration instruction is associated with each programmable element. This implies that only one logic design can be resident on the array until the device is reconfigured (i.e., a new set of configuration instructions are associated with the programmable elements of the array). In an MRLA, each programmable element can be associated with multiple configuration instructions. This allows multiple logic designs

(CFUs) to be simultaneously resident on the MRLA. The desired logic design can be activated by selecting the appropriate configuration instruction for each of the programmable element.

Configuration instruction slots for each PE (called the *configuration memory*) are stored in an ordered sequence and all PEs have the same number (D) of configuration instruction slots. MRLA takes an input called *context_id* which can take values from 1 to D . A value of k to the *context_id* input selects the k^{th} configuration instruction from the configuration memory as the instruction for each PE. The k^{th} configuration instruction is referred to as the **active configuration instruction** for that PE.

The set of configuration instructions with identical index in the configuration memory of a PE is referred to as an **execution context**. The execution context that is associated with currently active configuration is called the **active context**. MRLA can be effectively viewed as an array of FPGAs, one array per execution context; and the *context_id* serves as the index into this array. Selection of an execution context, makes all the CFUs of that context available for instruction processing by subsequent instructions.

We elaborate on a few other restrictions that dictate how the MRLA execution contexts are viewed by the compiler. The AEPIC compiler views each execution context of the MRLA as composed of a linear sequence of fixed size **slices** and a CFU is expected to consume an integral number of *consecutive* slices. The signals that supply data to the CFUs on the MRLA are grouped into fixed width ports which are connected to the Array Register File (ARF). Each slice is associated with one port. The number of slices required for a given CFU might depend on either the programmable logic resource or the number of i/o ports required by that CFU.

5.3.1.1 Desired Properties Of A Programmable Logic Array

Clearly, there are several possible choices for the micro-architecture of the programmable logic array. However, since the focus of this thesis is on the compiler’s view of the machine, micro-architectural details of the programmable logic array (in our case the MRLA) such as the type of embedded interconnect, the granularity and structure of the programmable logic elements, etc., will not be discussed. We do discuss those features that may be useful from a compilation perspective.

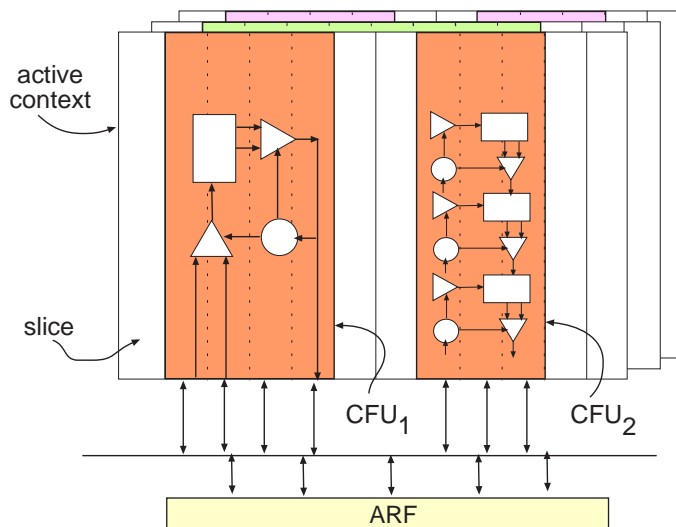


Figure 5.8: CFUs on MRLA

1. ***Partial, run-time reconfigurability*** Reconfiguring the entire programmable logic array is a slow and inefficient method to alter the CFU mix of the data-path. Often it is sufficient to evict only a subset of the currently configured CFU set in order to bring in new CFUs provided the programmable logic resources are inadequate to also include the new CFUs into the data-path. This highlights the first required property of an MRLA: **partial, run-time reconfiguration**—the ability to reconfigure subregions of MRLA at runtime to accommodate the changing CFU requirements of the program.
2. ***Location independent placement of CFUs*** The placement of CFUs on the MRLA could lead to fragmented use of MRLA resources. In order to utilize the MRLA resources efficiently, CFU’s may have to be *relocated* to consolidate unused segments of MRLA so that they may be assigned to a CFU that may be instantiated during a subsequent *reconfiguration* phase. Clearly, we wish to retain the semantics of the CFU even after it is relocated. This leads us to the second desired property of an MRLA: *location independence*. Location independence guarantees that a CFU may be mapped onto the MRLA at any position subject to the resource requirements of the CFU. Location independence for hardware, called *relocatable hardware* was first proposed in [178]. Subsequently, similar ideas have been proposed in the CVH architecture [142] and the Chimera architecture [62].
3. ***Multiple contexts and efficient context switching*** Applications CFU requirements may change rapidly during runtime. This implies that the programmable resource should be rapidly reconfigured from one set of CFUs to another in quick succession. Typically *configurations* are very large (orders of magnitude larger than typical EPIC instructions) and hence the reconfiguration time may be substantially larger than machine cycle time. In order to gain the advantages of custom functional units, the reconfiguration overhead has to be masked. This calls for a design in which the MRLA can simultaneously host multiple CFU’s such that (a) a subset of these may be designated to belong to the current execution context (those that may be used by the instructions until the next reconfigure phase) and (b) the MRLA allows rapid switching between two different sets of CFU’s so that context switching to a different set of CFUs (those that are required by subsequent phases of the computation) is very fast and, (c) die area of MRLA is not impacted significantly.

This leads us to the final desirable property of an MRLA: *multi-context programmable logic array*. Multiple contexts greatly reduces the number of data transfers that might be necessary to keep the MRLA “supplied” with configurations in a runtime reconfigurable processor such as AEPIC, allowing much faster switching between CFU configurations, reducing “idle time” while the system waits for the CFU set to be altered.

The idea of multiple configuration stores in FPGAs was first published in [136]. DeHon later based his Dynamically Programmable Gate Array (DPGA) architecture around this idea [39, 162] and did a study of the benefits of multiple configuration stores in various applications [38]. An earlier use of multiple configuration stores in a different context was described by Snyder [34]. In [166] the architecture of a time-multiplexed FPGA is proposed. Eight configurations of the FPGA are stored in on-chip memory. This inactive on-chip memory is distributed around the chip, and accessible so that the entire configuration of the FPGA can be changed in a single cycle of the memory.

5.3.2 Configuration Register File

When configuration data is present in the C-cache or in the MRLA, configuration is architecturally visible; in other words, the application is aware of the presence of this particular configuration and where it is located (whether it is in the C-cache or on the MRLA). Configuration data may also be distributed across the rest of the configuration memory hierarchy. However in that case, it is invisible to the application. When architecturally

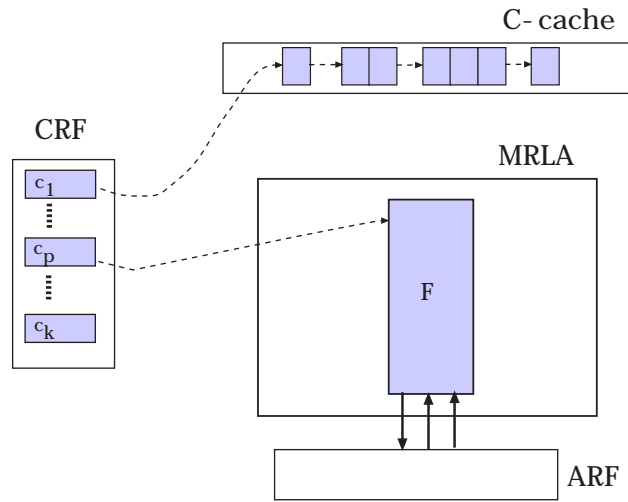


Figure 5.9: Configuration register file and associated configurations

visible, a configuration is uniquely identified by a *configuration register*—this is the only way an application may refer to an on-chip configuration. The configuration register is an alias for the architecturally visible configuration. Most AEPIC instructions (described in ??) refer to the configuration by specifying the configuration register associated with it. For example the instruction to invoke an operation *opid* on a CFU is **exec cr, opid** where **cr** is the configuration register associated with the CFU which performs the operation. In addition, a configuration register also stores several pieces of information pertaining to the associated configuration, some of which are listed here.

- *Resource allocation map for configurations on the MRLA.* This is the bitmap which identifies the resources allocated on the MRLA to the CFU.
- *Resource allocation map for configurations in the C-cache.* Similar to the above, this map identifies C-cache blocks allocated to the CFU.
- *Status of the associated CFU.* Status indicates whether the CFU is in use (executing an operation), idle, suspended, etc.
- *Latency assumptions of invoked operation on the associated CFU.* The latency information is supplied by the operation itself. If the operation generates multiple result operands, then a latency number is specified for each of them.
- *Operand-register association.* This association informs the CFU from which registers it should obtain the values for its input operands and in which registers it should deposit values of its output operands.

We shall explain the role played by these pieces of information of the configuration register as and when necessary.

5.3.3 Register Files

5.3.3.1 Array Register Files

AEPIC provides a large number of registers grouped into register files. Each MRLA is associated with a single register file called the Array Register File (ARF). Other register file types are described later. The ARF register

set is partitioned into scalar and FIFO registers. The FIFO registers are intended for use in applications which involve processing of large amounts of streaming data or alternatively may be used as rotating registers for software pipelined loop bodies that are mapped onto the MRLA. The data types of values stored in these registers are determined by the application itself. Only the register widths (number of bits in each register datum) are fixed and are assumed to be identical for all registers. The number of scalar and FIFO registers in a register file are architectural parameters.

5.3.3.2 Control Register File

Each AEPIC machine has exactly one control register file. Control register file structure follows the pattern set by the HPL-PD architecture [84]. The control registers include the program counter (PC), stack pointer (SP), frame pointer (FP), processor status word (PSW), context pointer (CP), registers for maintaining software pipelined code status and a few registers used as aliases for groups of predicates or for groups of speculative tag bits.

5.3.3.3 Other Register Files

Branch target register file. Control flow in AEPIC machines is implemented using the decoupled branch architecture model proposed in [84]. Here the different pieces of branch instruction are computed separately and as early as possible. One of the pieces of information that is computed in this model is the branch target address. The *branch target register* is used to store the branch target address and the static prediction bit which indicates whether the branch will be taken or not so that instructions that pre-fetch from the predicted target are initiated concurrent to the issuance (or as early as possible) of the associated branch operation.

Predicate register file. Predicates are one bit values. Predicate register file is composed of 1-bit registers and is partitioned into the static and the rotating predicate registers. Predicate registers are used to store predicate values that serve as guards for performing conditional execution of operations.

General purpose integer and floating point register files. General purpose and floating point register files are the traditional register files meant for storing intermediate values of integer and floating point arithmetic operations on the dedicated functional units for that purpose. These register files are also split into static and rotating parts. Details regarding branch target, predicate and general purpose register files can be found in HPL-PD architecture specification [84].

5.3.4 Instruction And Data Memory Hierarchy

The memory system of AEPIC architecture consists of the standard instruction and data cache memory hierarchies similar to the ones in standard EPIC processors. In addition, AEPIC architecture supports a novel memory system for *configuration* data.

The instruction cache and the data cache are disjoint at the highest level of the memory hierarchy but share subsequent levels of the memory hierarchy. The first level cache is followed by a larger second level cache and then the main memory (which is usually external to the processor). The structure of the caches is not architecturally visible. Unlike in the memory hierarchy of HPL-PD, a data-prefetch cache is currently not included. However, just as in HPL-PD, the architecture provides explicit control over the placement of data in the data and configuration memory hierarchies. The explicit control is provided in the form of additional tags associated with the memory operations that indicate the memory regions in which the data is to be saved or read from. These tags are merely hints to the processor as to where the data is expected to be found rather than strict conditions that the memory controller should enforce. These tagged memory operations also serve as a mechanism to convey to the processor

the latencies assumed by the compiler for those memory access operations when it (the compiler) generated the ILP POE. Note that instruction fetching memory access operations are not visible architecturally.

5.3.5 Configuration Memory Hierarchy

The configuration cache memory hierarchy is composed of C-Cache at the highest level followed by a larger next level cache called C1 cache which is followed by the main memory. Since configurations are immutable data, there is never any need to write back cached configuration data back to the main memory. However, configuration data may be transferred between the MRLA and C-Cache or between the C-Cache and C1 cache.

If a CFU is to be instantiated on the MRLA for a particular computation, it is necessary that the whole of the configuration data be available and loaded onto the MRLA before any operation is issued to it. Unlike instructions and data values, configurations are much larger in size and need not be of the same size. This calls for a cache organization that is slightly different from traditional data/instruction caches.

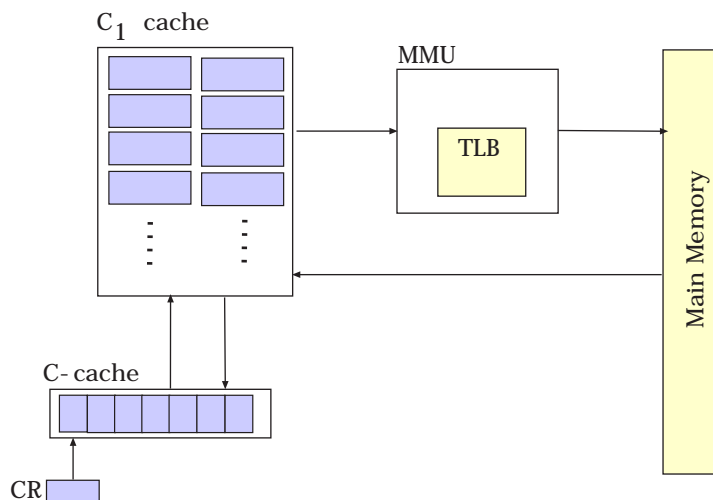


Figure 5.10: Configuration cache memory hierarchy

C-cache is a linear memory space segmented into *blocks*. A *block* is the basic unit of allocation for configuration data on the C-cache. Blocks in the C-cache are numbered sequentially. A *configuration* is allocated an integral number of blocks with monotonically increasing block addresses in the C-cache (allocated blocks need not be consecutive). In other words, configuration word order in C-cache is identical to its word order of that configuration in the virtual memory of the application process containing that configuration. This C-cache allocation for the configuration is saved with the configuration register associated with that configuration. Each configuration register contains a bit-vector called **Block Allocation Vector (BAV)** of length B . If *block* i in the C-cache is allocated to configuration associated with configuration register cr , then $cr.BAV[i] = 1$. If the configuration register has not been allocated to any configuration, then the *BAV* values are 0.

A configuration word once brought on chip, is stored in exactly one of three places: (1) C1 cache or (2) C-cache or (3) MRLA. However, before the first word corresponding to a configuration is written into the C-cache (MRLA), the architecture specifies that enough space be allocated and reserved for the entire configuration on the C-cache (MRLA). As configuration data is moved between C-cache and MRLA, the allocated space on either unit may

be reclaimed (incrementally or in one shot, depending on the implementation). Hardware support for a rapid C-cache allocator/de-allocator and memory allocation policies are discussed in Appendix 3.

Data in the C1 cache is also stored grouped into blocks. Here the blocks are referred to as *pages*. Each page can host fixed amount of data from contiguous locations of the external memory. When a request for a particular configuration is not met by the C-cache and the C1 cache, the processor issues requests for the appropriate pages to be loaded from the external memory. The Memory Management Unit (MMU) translates page addresses of virtual pages to physical addresses in the external memory. The Translation Look-aside Buffer (TLB) is a cache of virtual page address to physical page address map.

If a desired physical page is not available in the external memory then a page-fault occurs and the page-fault handler loads the page from the disk and updates the TLB if necessary. The operation of the MMU and the TLB are similar to their corresponding analogue used in the instruction/data memory hierarchies of standard microprocessors.

5.4 Instruction Set

Currently, the AEPIC ISA is composed of the HPL-PD ISA extended with a small set of instructions specifically intended for reconfiguring the processor, that we refer to as the **Adaptive Extension (AE) ISA**. HPL-PD ISA is the *EPIC* component of the AEPIC ISA. On any machine cycle, the operations being processed are either from the EPIC ISA or the AE ISA or are operations performed by some CFU.

We provide an assembly level description of the AEPIC instruction set. The parametric nature of the AEPIC architecture does not permit an explicit machine level instruction format for AEPIC instructions. In order to understand the architecture, however, such a format is not necessary. The AEPIC instruction set can be partitioned into two groups.

- The EPIC style instruction set
- The specific AEPIC instructions that are intended to handle the adaptive component of the processor are referred to collectively as **Adaptive Extension ISA**.

5.4.1 EPIC ISA

The former is based on the HPL-PD ISA and we refer the reader to the HPL-PD architecture specification for details [84]. For the remainder of this section, we focus only on the AEPIC Extension ISA.

5.4.2 Adaptive Extension ISA

The specific AEPIC instructions that are intended to handle the adaptive component of the processor can be grouped into the following categories. Details about these instructions are in Table B.1 (Page 141).

1. **Resource allocation.** Here we are referring to instructions that perform resource allocation for configuration data in the C-cache as well as on the MRLA. The *malloc* instruction allocates space for CFUs on MRLA and *calloc* allocates C-cache space. An architecturally visible configuration is present in either the C-cache or on the MRLA. Never on both. Hence one *free* instruction is sufficient to deallocate the space allocated for the configuration whether it is currently resident on the C-cache or on the MRLA. Both *malloc* and *calloc* should be provided with an unallocated configuration register *cr*. This *cr* is initialized with the resource allocation map of the CFU. *Free* resets the Allocation maps of the configuration register *cr*.

2. **Data-path reconfiguration.** Data-path reconfiguration involves two basic tasks: (a) add functional unit and, (b) remove functional unit. The AEPIC instruction *inc* is used for instantiating new CFUs and the instructions *outc* and *delc* are used for deleting instantiated CFUs. Latencies of these operations are dependent on the size of the configuration being instantiated. The sizes of configurations exhibit a large range, and since the compiler has full knowledge of the actual size, it is appropriate that the compiler convey the expected latency for every instance to these operations. These two operations perform: (a) the actual transfer of configuration data from the source to the destination (C-cache or MRLA) and (b) the associated deallocation of the resource consumed in the source region. The deallocation operation need not necessarily be atomic. We leave this aspect open for further investigation.
3. **Managing execution contexts.** The *setctx* operation deactivates the currently active context and *activates* the execution context referred to by the operand *cid*. This is the most efficient mechanism to switch the CFU mix of the data-path from one set to another. Both the sets of CFUs must already be instantiated on two different contexts of the MRLA. The *clrctx* instruction deallocates the space allocated for all the CFUs on context *cid*. The corresponding configurations are deleted permanently. This operation is equivalent to performing *delc* on each of the CFUs on the context *cid* individually. This is a fast mechanism to drop an execution context if it is known that none of the CFUs on that context will be needed again (or at least not in the immediate future). The *switchctx* instruction moves the configured functional unit associated with *cr* to the new context identified by *cid*. Space allocated for *cr* CFU on its current context is freed. The input/output operand associations to the CFU remain the same. The *pushctx* (*popctx*) instructions are used to allocate (deallocate) execution contexts. These instructions are used before (after) making a call to a function which might also instantiate CFUs.
4. **Memory access.** Memory operations serve two purposes: (a) to transfer data between the register files (ARF, GPR, FPR, BTR, PR) and the main memory, (b) to transfer data between the C-Cache and the main memory. In the latter case, the data are the configuration bit-streams while in the former the data are the input and output operand values for operations on the functional units (CFUs and the EPIC functional units). Data transfer operations between ARF and main memory are identical to those of that transfer data between the other register files (those that are used by the hardwired functional units) and main memory. The semantics of memory operations that transfer configuration data between C-cache and main memory are different from that of the conventional memory operations due to the nature of the type of data being transferred. This is due to the (typically) large sizes of configurations and their immutability.
5. **Operand association.** Before initiating an operation on a CFU, the CFU input/output interface should be associated with concrete locations in the processor state. These locations would serve as the sources (destinations) of input (output) operands of the CFU operations. The processor locations intended for this purpose are the ARF registers. Strictly speaking, the CFU inputs and outputs need be associated with ARF registers only for the read and write intervals of these operands respectively. Note that the same ARF register may be associated as a source location for input operands of multiple CFUs or with multiple input operands of the same CFU. An output location may be associated as the destination for multiple output operands if the write times of the corresponding output operands do not overlap in time. The AEPIC instructions for associating source locations for input operands are *inp* and *inpr* while the operations for making the association for output operands are *outp* and *outpr*.
6. **CFU computation.** CFU operations may be invoked once (a) the operand association for that CFU has been performed and, (b) the locations of source operands have the desired data during the input sampling intervals for those operands. The *opid* input for *exec* operation specifies the operation to be executed on the CFU from its opcode set. For the rest of the AEPIC instructions, the *opid* parameter is not necessary since only one operation can be issued at a time and that has already been specified by the *exec* operation (which has to issued earlier than any of the other operations).

5.5 Architectural Parameters

Many aspects of the AEPIC architecture are left unspecified since such details are not relevant to the compilation centric research problems we address here. Hence, issues such as instruction formats, micro-architectural details of various components of the machine are not addressed in this work. Also, several aspects of the AEPIC architecture are described in a parametric manner. For example, the width of the array in MRLA is a parameter. It is expected that only extensive experimentation will yield the right values for these parameters. Also, such flexibility is necessary for us to explore a large space of AEPIC machines to determine the most suitable configuration for various application domains. A list of all the identified architectural parameters that are interesting from the point of view of architectural exploration is given in Table C.1 (Page 146).

5.5.1 MRLA Parameters

Clearly the size and type of MRLA plays a significant role in determining the amount of computation that can be mapped onto it. *Array width* is an upper-bound on the size of any given configuration. The parameters *number of contexts* and *array width* together determine the total amount of computation can be simultaneously resident on the MRLA at any instant. *Number of contexts* dictates the maximum function call nesting that can be efficiently supported without resorting to CFU eviction (spill) from MRLA when a function call is made. This feature is explained in the chapter on AEPIC compilation. *Context switch latency* is the number of cycles required to switch from one execution context to another. The latency parameters for loading (unloading) configuration data into (out) of MRLA play a significant impact on the reconfiguration policies adopted by the compiler. The MRLA port parameters determine the register bandwidth available for configured functional units for the CFU operands.

5.5.2 Memory System Parameters

Structures of the various memory hierarchies are not architecturally visible. But it is unclear what types and sizes of cache memories are suitable for a given application domain to which AEPIC machines may be targeted. Since C-cache and the C1 caches are expected to have the greatest impact on the performance compared to that of the other caches, a thorough investigation into various choices for C-cache and C1 cache parameters is required.

5.5.3 Fixed Core Parameters

By fixed core, we mean the hardwired functional units of the EPIC core that process the AEPIC instruction set and the register files attached to those hardwired functional units. Most of these parameters have been used in the context of EPIC research.

5.6 Additional Notes

5.6.1 Multi-cluster Machines

As the size of the MRLA is increased, the number of ports to the ARF may need to be increased to support the expected increase in register bandwidth due to additional CFUs that might be mapped on the MRLA. A large number of ports to a register file can impact the processor cycle time [33]. One way to avoid this problem is to partition the register file into multiple register banks connected to separate MRLAs. An AEPIC machine with more than one register file of the same type connected to separate MRLAs is called a *multi-cluster* AEPIC machine. Note that registers from two different register files of the same type connected to the same MRLA are logically equivalent.

Multi-cluster AEPIC machines introduce the problem of keeping the register files from different clusters coherent. Coherency can be maintained by introducing copy operations between register clusters. Compiling to clustered machines requires additional steps such as assigning data and operations to clusters and scheduling copy operations between registers from different clusters, while simultaneously optimizing the overall schedule. This is inherently more complex than compiling to single cluster machines.

5.6.2 Heterogeneous Machines

It is unlikely that a single type of MRLA is optimal for hosting all the computations from a given application domain. Hence it might be advantageous to explore AEPIC machines containing multiple MRLAs of different types. For example, a single AEPIC machine could host a fine grained programmable logic array for image processing applications and a larger grained programmable logic array intended for SIMD type computations found in multimedia applications. Granularity being the word sizes of the data processed by the processing elements of the MRLA. We call an AEPIC machine that has more than one type of MRLA, a *heterogeneous* AEPIC machine.

5.6.3 Interrupts And Exceptions

Interrupts to program execution could be caused due to external factors such as hardware faults, power failure or could be caused by the program execution such as arithmetic overflows, memory protection violations, I/O calls or operating system calls, etc. Several solutions to the interrupt problem have been proposed for the interrupt problem for out-of-order processors such as the reorder buffer, the history buffer, the future file [156], checkpoint repair [74] and current-state buffer [113]. All issues related to exceptions and exception handling mechanisms are left for future work on AEPIC architecture.

5.7 Summary

AEPIC is a candidate architecture from the C_AEPIC space of architectures that we have identified as a suitable platform for reconfigurable computing research. It embodies several innovative architectural features specifically addressed to tackle the problems of reconfiguration in a way that is suitable for compiling to such a machine. The characteristic features of this architecture are:

- Compiler specified resource allocation.
- Architecturally invisible resource assignment.
- Explicitly controlled configuration cache hierarchy.
- Implicitly specified operands for configured functional units.
- Explicitly (compiler) specified operation latencies.
- Architectural support for efficient context switching.
- Parametric description of the architecture.
- EPIC features such as MultiOp, NUAL, speculation, predication, decoupled branches, programmatic caches and efficient boolean reductions.

Chapter 6

Compiling For AEPIC Targets

After ecstasy, the laundry.

–Anonymous

An AEPIC processor is designed to enable fast and efficient reconfiguration of its data-path—a key requirement in order to avail the benefits of micro-architecture customization without paying for the overheads of performing the customization. The main features provided by the processor intended for this purpose are the adaptive extension instructions and the various micro-architectural resources for hosting application specific logic. However, the burden of realizing the benefits of application specific customization rests mostly on the compiler targeting these processors since most of the decisions such as which instructions an application requires, when and how the processor should be reconfigured to emulate these instructions are left entirely to the compiler.

In this chapter, we address the issues involved in targeting an application to AEPIC processors. The key AEPIC specific compilation problems are partitioning, instruction synthesis, configuration selection, resource allocation and instruction scheduling. We start by describing a basic compilation framework in Section 6.1. Partitioning is the problem of identifying code sections that may benefit by mapping them on to the programmable logic resources. Partitioning is discussed in Section 6.2. The instruction synthesis phase generates suitable implementations for the candidates partitions and updates the machine description database with the “new” instructions. Issues related to instruction synthesis are presented in Section 6.3. Configuration selection (Section 6.4) is the problem of narrowing down the choices of which synthesized instruction (from the set generated by the instruction synthesis phase) to use for each of the code regions that will be mapped to programmable logic. Configurations are just like program data values that need to be allocated on-chip resources before their use. Configuration allocation is the problem of deciding when and which configurations to load into processor local resources from external memory in the presence of control flow effects and limited programmable logic resource constraints. Compilation techniques addressing this problem are presented in Section 6.5. Section 6.6 deals with the instruction scheduling problem in the presence of synthesized operations.

6.1 A Basic Compilation Framework For AEPICs

Inputs to an AEPIC compiler are (a) the application source program written in a high level language such as C/C++, (b) a description of the particular instance of the AEPIC processor described in a machine description language (c) a library containing parameterized configurations for popular computational routines such as FFT, DCT, etc. The input source code may be instrumented with pragmas intended to give partitioning/mapping hints to the compiler. These cues are communicated to different phases of the compilation process through the

intermediate code representations.

The first phase is a standard lexical and syntactic analysis phase. The partitioning module will help delineate portions of the program that might benefit from execution on the configurable portion of the target. The partitioning phase is followed by two independent phases that may be performed in parallel: the high-level optimization phase and the operation synthesis phase. In the high-level optimization phase, the code to be executed on the hardwired functional units (which execute instructions from the AEPIC ISA) is optimized as is typically performed in a standard ILP compiler. The instruction synthesis phase generates suitable mappings of the code partitions identified by the partitioning phase. Each of these mappings are packaged as “custom operations”. The machine description is updated with the synthesized instructions. The instruction synthesis phase may generate multiple implementations for the identified partitions reflecting different performance/device-area tradeoffs. The configuration selection phase tags different regions of the intermediate code with semantically equivalent custom operations. At the end of the configuration selection phase, every region of the code identified for mapping on to the configurable part of the target has a unique configuration associated with it.

Subsequent phases of the compilation are similar in structure to back-end phases of a typical ILP compiler suitably adapted to consider the special characteristics of configurations/CFUs.

In most ILP compilers, register allocation is usually not performed prior to scheduling. This is because, register allocation in its attempt to optimize register usage, introduces too many inter-instruction dependencies that constrains scheduling. On the other hand, register allocation also introduces additional code (spill and shuffle code) that needs to be scheduled requiring a scheduling phase post register-allocation. Hence, a typical ILP compiler back-end comprises of (at the minimum) the following phases, in sequence: pre-pass scheduling, register allocation, post-pass scheduling and code generation. In addition to these, AEPIC compilation introduces an extra phase: *configuration allocation*. The configuration allocation phase is aimed at optimizing allocation of resources for configurations—a task analogous to that of register allocation. The resources meant for configuration are the C-cache and the MRLA. These resources are independent of those intended for register allocation. Hence, configuration allocation may be performed in parallel or in any order with respect to register allocation. The main task of the scheduler is to reduce the critical path through the code by masking reconfiguration overheads.

In the above framework, the scheduled and allocated code is transformed into machine code, which is then translated to object code for simulation. Simulation yields correctness as well as performance data for the program on the given input data. The execution profile can be re-instrumented into the IR for profile based optimizations.

In subsequent sections, we define the problems encountered in each of the phases of the compilation path identified above. In some cases we provide techniques that may be used to solve these compilation problems. Substantial research still needs to be done and we do not make any claims on whether any of these techniques are sufficient to fully exploit the capabilities of AEPIC processors.

6.2 Partitioning

Partitioning is the task of determining the set of code segments (referred to as “candidates”) in the application which may be synthesized as application specific instructions. These application specific instructions are implemented on the MRLA as configured functional units.

The partitioning module takes as input (a) an intermediate code representation of the source program and (b) the machine description of the target AEPIC processor and identifies regions of the intermediate code that can benefit

from mapping to the MRLA. Although not necessary, the partitioner can only benefit from an execution profile of the application. The execution profile gives execution frequencies of different regions of the intermediate code. This information can be obtained by compiling to a base EPIC architecture and re-instrumenting the intermediate code with the profile data from the simulator. Since the reconfiguration overhead can be quite large, it may not pay to reconfigure the processor for a certain segment of the code if it is known that this section will rarely be executed. The execution profile can be used by the partitioner to eliminate such code segments from consideration for mapping onto the MRLA.

The steps of the partitioning process are described here. (1) Initial run of the compiler on the MPEG2 decoder application targeting an EPIC processor followed by simulation generates the execution profile for the application. (2) The profile re-instrumentation module (PIM) updates the IR of the application with the execution frequency information. (3) Partitioner module traverses the IR and tags intermediate nodes that can benefit from mapping to the MRLA. The partitioner reads the AEPIC machine description file to determine the resource constraints and the type of programmable logic—information that is used to determine suitability of selected candidates. In the case of MPEG2 decoder, the partitioner identifies four candidates as suitable candidates for mapping onto the MRLA: function IDCT, basic block 4 of function Form_Component, hyperblock 16 of Add_Block and the Saturate function. (4) The operation synthesis phase processes these tagged partitions and generates the relevant configurations that can perform the computations of the identified partitions.

Not all candidates are profitable since the overheads of reconfiguring the processor may annul the benefits of application specific instructions. Considering that the task of synthesizing application specific instructions is a very compute intensive one (as we shall see later), we would like the partitioner to choose candidates intelligently. The goal of partitioner is to identify as many potentially beneficial candidates as possible in order to enable enough opportunities for optimization during subsequent phases of AEPIC compilation and perform this task fast.

6.2.1 Partitioning Considerations

What factors determine if a candidate partition can be mapped to MRLA? And if a candidate can be mapped to the MRLA, will it be profitable to do so? At what stage of the compilation phase should the partitioner be invoked? These are some of the issues discussed in this section.

6.2.1.1 Machine Constraints

The type and amount of MRLA resources available limit the size and number of candidate partitions that can be accommodated. The resource consumption of a partition is available only after the instruction synthesis phase (Section 6.3). However, the partitioner can be provided with heuristics for estimating the resource requirements for a given piece of intermediate code. For example, given the resource estimates for various types of high-level operations and a measure of the reconfiguration overhead, it might be possible to develop heuristic techniques to estimate the resource requirements and reconfiguration overheads for a given partition.

6.2.1.2 Application Considerations

Application factors that determine the suitability of a given piece of code for implementation on a programmable logic array are may be considered by the partitioner to determine if a code section is a suitable candidate are , operation mix, word length requirements, execution profile, performance requirements, regularity, parallelism, resource constraints, task granularity

6.2.2 A General Framework For Code Partitioning

In order to maximize the number potentially beneficial candidates, our experience indicates the following to be the most important requirements of a code partitioner.

1. An ability to examine application code at multiple intermediate levels of representation.
2. Availability of execution profile for each intermediate representation.
3. Resource and reconfiguration overhead estimators for high-level language constructs and other primitives of the intermediate codes (semantic operations).
4. Accurate type and data-width (bit-width) information for program variables.
5. Machine parameters.

The proposed code-partitioner is composed of four main steps as illustrated in Figure ???. These steps are performed in sequence on each intermediate representation that is created during the compilation process.

Profile. The profile phase is composed of four steps:

1. *Instrument:* The IR nodes are tagged with code for gathering the execution profile for each node. So for example, in a basic block IR, each basic block will be associated with a “execution frequency” variable and a piece of code at the entry of the basic block which updates this variable whenever control enters the basic block.
2. *Translate:* During this step the instrumented IR is translated to C language code which is then compiled using the native compiler to yield a semantically equivalent (to the application program) executable with one additional attribute: this executable also gathers the runtime properties of the program as specified by the instrumentation code.
3. *Execute:* The execute step runs the instrumented and compiled application using the inputs associated with the application program.
4. *Re-instrument:* After the execute step, the application generates the execution profile (written out by the instrumented code) and re-instruments the starting IR with the actual execution profile values (such as the frequency of execution of a basic block, etc.)

Analyze. After the **profile** step, the IR is tagged with the execution profile. This information is used to identify the most frequently executing regions of the code. The **analyze** phase traverses the IR and determines various properties of the program that may be useful in determining if a region is a candidate for partitioning. Some of these properties are bit-widths/types of variables, whether certain arrays are constants or are bounded of known dimensions, etc.

Identify. The **identify** step performs a bottom-up traversal of the IR and tags each region as a candidate for mapping using certain heuristics based on control structure, operation types and other attributes identified in the **analyze** phase. The identify phase also considers machine resource constraints (from the machine description database) and any user supplied cues transmitted through the IR, in deciding whether a region of the IR is a feasible candidate for mapping.

Coalesce. The coalesce phase merges adjacent regions marked by the **identify** phase if the merged regions can still be accommodated on the MRLA assuming the given machine constraints.

6.3 Instruction Synthesis

Instruction synthesis is a systematic technique for defining new instruction set for a given micro-architecture. The process typically involves analyzing the benchmark(s) to infer the most suitable operation repertoire based on its computational characteristics for the intended micro-architecture. In certain cases the micro-architecture itself is synthesized in parallel with the instruction set.

The Instruction Synthesis (IS) module takes a list of candidate partitions that have been identified for mapping onto the programmable logic and synthesizes a set of functional units that can implement all the computations of the code partitions. In addition to the candidate partitions list, the IS module may also take as input, a library of pre-synthesized macros for various basic operators and frequently used kernels such as FFT, DCT, FIR/IIR filters, etc. The purpose of this library is to speedup the process of mapping a given partition to the MRLA. It helps to keep these pre-synthesized macros generic so that they are applicable to a wide range of the target programmable logic parameters, and also accommodate variations in the structure of the input partitions.

6.3.1 Instruction Synthesis Techniques

There are two approaches for the instruction synthesis problem that are relevant to AEPIC style processors: handling this case:

1. Full synthesis: In this method, the intermediate code of the partition is converted into any of the Hardware Description Languages RTL format. The RTL code and the target parameters are then provided as input to EDA synthesis tools whose output is mapping of the original partition onto the given target.
2. Synthesis by construction: Synthesis by construction is a semi-automatic technique wherein, the IS module generates configurations by composing pre-computed configurations for subproblems (low-level primitives). For instance, one could have configurations for basic trigonometric operations based on the CORDIC [172] evaluation technique. For a candidate partition, which might contain trigonometric and other arithmetic functions, the IS module combines the configurations for each individual primitive without violating any of the dependencies between the primitive operations within the candidate partition. The initial operation set that an IS module uses can be (1) often used operations such as integer arithmetic (2) high-level language statements (configuration bits for an *if-stmt*) (3) implementations of popular kernels, etc. In addition to *composing* primitive configurations, other methods for synthesizing configurations can be based on (1) specialization, (2) generalization, (3) decomposition and (4) compaction of mappings of known primitives.

6.4 Configuration Selection

After the partitioning and instruction synthesis phase, all the regions of the intermediate code that may be mapped to the MRLA have been identified. Each of the candidate partitions can now be replaced with one of the equivalent configurations synthesized by the IS phase. Configuration Selection (CS) is the problem of determining which semantically equivalent synthesized configuration (custom instruction) to associate with each candidate partition that is mapped to MRLA.

To illustrate the CS phase we continue with the MPEG2 decoder example used earlier to illustrate partitioning. The task of the CS module is to associate suitable mappings for the four candidates identified by the partitioner. A typical partition might have multiple synthesized mappings reflecting to a variety of performance/area tradeoffs. In such cases, semantically equivalent configuration templates (mappings on the MRLA) are grouped together and an appropriate tag is associated with the group. In Figure ??, the IDCT kernel is shown to have two

alternate implementations : *IDCT_Distributed_Arithmetic* and *IDCT_Systolic_Array*. These two implementations correspond to the same “semantic opcode” (the IDCT function) represented as *IDCT_Configuration_Operation*. Figure ?? also shows other information that is stored as part of each opcode description (for both custom as well as standard opcodes) – the *I/O_Format*, *Resource_Usage* and *Latency_Summary*. Together, these describe the input/output formats for the configuration, the actual resources used by them during their lifetime (from issue time to retirement stage) on the AEPIC processor and a summary of their latencies. For each partition, the CS module obtains various parameters of all mappings associated with each partition from the machine description database (where the results of the synthesis phase are saved).

Configuration size impacts code size as well as reconfiguration times. For high throughput applications, it may be beneficial to use heavily pipelined (which implies larger) configurations. In practice, most partitions may be associated with a single synthesized configuration. In which case, the selection step is a trivial operation.

We extend the EPIC code generation path to include configuration selection. The key difference compared to traditional instruction selection is that the selection can happen at multiple levels in the IR. Configuration selection can happen for leaf level operations as well as higher level structures such as groups of instructions, program statements, loops, functions, etc.

6.5 Configuration Allocation

6.5.1 Configuration Allocation Problem

After configuration selection, some of the nodes of the intermediate code may be tagged with application specific instructions. The configuration selection module *does not consider availability of resources on chip* when it decides on which configuration to assign with each partition that is synthesized into application specific instruction. Any realistic AEPIC processor would have limited programmable logic resources available for CFUs and hence it is quite possible that all the desired configurations cannot be accommodated on chip simultaneously.

Configuration Allocation (CA) is the problem of optimally allocating/de-allocating on-chip resources that are intended for holding configurations (on C-cache) or CFUs (on MRLA). Poor allocation of resources for configurations can lead to long periods of processor stalls caused either due to waiting for the configurations to load into the MRLA or due to excessive thrashing in the configuration memory hierarchy. Hence, optimal allocation of configurations to C-cache and MRLA resources is critical for achieving high performance on an AEPIC processor. In addition, we would like the allocation algorithm itself to be time and space efficient.

This problem is analogous to the problem of implementing virtual memory on systems with limited physical memory or to the problem of allocating registers for program variables performed in most standard compilers. In the remainder of this section, we present a formal definition of the configuration allocation problem and relate it to the well studied problem of register allocation. We show how extensions to the register allocation techniques yield solutions to the configuration allocation problem.

6.5.2 Similarities With Register Allocation

Register Allocation (RA) allocates program variables (also referred to as temporaries) to registers in order to minimize the overall number of accesses to external memory. Fundamentally, both register allocator and configuration allocator perform the same task—allocation of on-chip resources for program values (variables in the case of RA and configurations in the case of CA). Configuration allocation differs from register allocation in the following ways:

1. *Non-uniform allocation units.* Sizes of configurations are typically much larger and *vary widely* compared to the sizes of data values stored in registers which are much smaller and almost always uniform in size.
2. *Heterogeneous resources to be allocated.* There are two types of local memories for configurations: (1) the C-cache and (2) the MRLA. These resources differ in their capacities, access (read/write) costs and sizes of allocation units. There is only one type of resource to be allocated in register allocation - the register set.
3. *Immutable values.* Currently, AEPIC architecture configurations are immutable. So memory write back of configurations is not an issue.
4. *No copies or moves.* AEPIC does not provide any architecturally visible features to create copies or move configurations with in the MRLA or the C-cache.

6.5.3 Machine Model

Here we discuss the resources provided by AEPIC machines for hosting configurations and the instructions from the AEPIC ISA intended for managing configurations. Two types of storage classes are available for hosting configurations: (1) C-cache and (2) MRLA. Allocated configurations are present in exactly one of these storage classes. Every allocated configuration whether it is on the MRLA or in the C-cache is associated with a distinct configuration register from the configuration register file (CRF). See Section 5.3.2 (Page 73) for details. The unit of allocation in the C-cache is a C-cache *block* and on the MRLA it is a *slice*. All configurations are constrained to consume an integral number of consecutive slices in the MRLA. Configuration data for each MRLA slice requires an integral number of blocks in the C-cache. The set of AEPIC instructions intended for performing configuration allocation/de-allocation are shown in Table 6.1.

Table 6.1: Instructions for configuration management

Resource allocation on MRLA	malloc, gcc, gcall, free
Configurations from/to memory	ldcc, ldccns, stcc, stccns
Context allocation on MRLA	setctx, clctx, clralctx, switchctx, pushctx, popctx
CFU instantiation	inc, incns, outc, outcns, delc

The basic steps involved in using configurations are:

1. Allocate a configuration register with the configuration to be loaded.
2. Allocate requisite number of blocks in the C-cache for the configuration. If the C-cache is insufficient for the configuration, then the allocation fails and application is aborted. If the configuration is smaller than the C-cache size but the total free space is less than that which the configuration requires, then some of the resident configurations are evicted (since configurations are immutable, they are simply deleted and not written back to memory).
3. Schedule the loading of configuration data into the C-cache into the allocated blocks.
4. Allocate consecutive slices on the MRLA to load the configuration that was loaded into the C-cache (to instantiate the CFU corresponding to the configuration).
5. Schedule transfer of configuration data from the C-cache to MRLA.
6. One or more operations are executed on the CFU.

7. At some point if the MRLA resources are required for another CFU or if this configuration will not be used any more, then it is evicted to the C-cache (if it may be used again) or just deleted from MRLA.
8. Allocated configuration register is freed - it can be allocated to a new configuration.

Note that the same configuration register refers to the configuration data when it was in the C-cache and to the corresponding CFU when loaded onto the MRLA. Once the configuration is completely moved to MRLA from the C-cache, the C-cache resources allocated for the configuration may be freed since it is wasteful blocking those resources as long as the configuration data is available on the MRLA. However, in certain cases it might be useful to architecturally expose the deallocation operation. For example, if multiple instances of the CFU corresponding to the configuration are needed on the MRLA, it might be more efficient to copy the configuration data from the C-cache to the MRLA once for each of the CFU instances instead of loading from external memory into the MRLA to make copies of the CFU already available on the MRLA.

6.5.4 Cost Model

Configuration allocation costs include the costs of operations that move configuration data between external memory and the processor local resources (C-cache and MRLA) and, the cost of instructions that perform configuration moves between different resources on the processor itself. The overhead is measured against a perfect allocation scheme that assumes a machine with unlimited resources. These overheads fall into three categories:

1. **Spill cost.** This is the cost of moving configurations between processor local resources and external memory. The actual cost depends on the size of this configuration. Since configurations are assumed to be immutable, they need not be written back to memory. However, a configuration on the MRLA may be copied back to C-cache for later reuse. Configurations on C-cache are never written back to memory. Load costs from memory to C-cache and from C-cache to MRLA depend on the latency and bandwidth constraints between memory and C-cache and that between C-cache and MRLA. Spill MRLA to C-cache: $\text{cost} = (S_c/W) * L_{MC}$. S_c is the size of the configuration in bytes, W the number of bytes transferred from MRLA to C-cache per transfer operation and L_{MC} is the minimum latency between successive MRLA to C-cache moves.
2. **Call cost.** This is the cost of free/restore of resources allocated for configurations performed upon procedure entry and exit. This cost is influenced by two factors: (1) the procedure calling convention employed by the compiler, (2) any architecture specific issues that affect allocation decisions such as passing procedure parameters through resources reserved for such purposes and, (3) the number of contexts in MRLA that are available for allocation for called functions.
3. **Shuffle cost.** This is the cost of moving allocated configurations between different resources on the processor, typically from one live range to another. The proposed version of AEPIC architecture does not support moves or performing copies of configurations between different locations on MRLA. And in the C-cache, there is no use creating duplicates of configurations. Hence shuffle cost does not figure in the configuration allocation cost function.

6.5.5 Simplified AEPIC Allocation Model

For the remainder of this section we consider a simplified version of the configuration allocation problem. In the simplified version, the machine is assumed to contain N configuration registers and each “virtual” configuration in the program intermediate code specifies an integer k which is the number of physical configuration registers it requires. The only difference between this problem and the conventional register allocation is that, in the conventional register allocation problem, each virtual register (also called program temporary) is assigned to a

single physical register.

6.5.6 Allocation Techniques Background

Graph coloring is a powerful technique used for register allocation. Chaitin et al. first abstracted the register allocation as a graph coloring problem [27, 28]. Nodes in the graph are the *live ranges* representing variables (temporaries or virtual registers) used in the intermediate code. The edges in the graph represent interferences between two live ranges. A live range is said to *interfere* with another live range if the variables corresponding to the two live ranges are simultaneously live at some point in the program intermediate code. Allocating the two variables to the same physical register might violate program semantics. In other words, we desire to associate registers with the nodes of the graph such that no two adjacent nodes are not assigned the same register. The graph coloring problem is to assign colors to the nodes such that no two adjacent nodes are assigned the same color. Clearly, any valid graph coloring yields a valid register allocation for the variables. If the machine contains K registers, then we desire a K coloring of the graph. Since determining a K coloring of general graphs is NP-complete [51], polynomial time approximation schemes are often used. Chaitin's algorithm is one such polynomial time heuristic. If the number of available registers is less than the number of live ranges that need to be allocated, then the interference graph is modified either by eliminating some of the live ranges (called *spilling*) or by splitting the live range into smaller live ranges with the hope that the new interference graphs becomes colorable.

We adapt ideas from past work on register allocation for the configuration allocation problem. In the rest of this section, we present a coloring based configuration allocation scheme. We first describe how *live ranges* of configurations in the intermediate code and their interference graph are computed. Then we present a technique to transform the interference graph to enable better opportunities for coloring in case the coloring scheme blocks. This is followed by an algorithm for allocating configurations which draws from previous work by Chaitin [27, 28], Briggs [19] and Chow and Hennessy [32].

6.5.7 Interference

Live range construction. A live range is an isolated and connected group of nodes in the control flow graph that connects the definitions and uses of a given program variable. Live ranges are discovered by finding connected groups of *def-use chains*. A single def-use chain connects the definition of a virtual register to all of its uses. Multiple definitions may reach any use. Live ranges are typically computed through data-flow analysis; live-variables analysis and reaching definition analysis. A variable is live at block i if there is a direct reference of the variable at block i or at some point leading from block i not preceded by a definition. The reaching attribute is solved by forward iteration through the control flow graph. A variable is *reaching* in block i if a definition or use of the variable reaches block i . The live range $lr(v)$ of a variable v is constructed as $live(v) \cap reach(v)$.

Interference graph construction. Two live ranges interfere if one of them is live at the definition point of the other. If the smallest program units under consideration are instructions (basic blocks), then this definition point is an instruction (a basic block containing the instruction) that writes to the variable denoted by the live range. Two algorithms for interference graph construction for configurations, one whose point of definition is an instruction and the other whose point of definition is a basic-block are presented in 1.

Algorithm 1 Configuration interference computation

```
/* Point of definition based computation;  $I_c$  = configuration instructions */
 $\forall I_1, I_2 \in I_c : G[I_1, I_2] \leftarrow true;$ 
foreach ( $I \in I_c$ ) {
  foreach ( $I_1 \in Live(I)$ ) {
    foreach ( $I_2 \in Defined(I)$ ) {
       $G[I_1, I_2] \leftarrow true; G[I_2, I_1] \leftarrow true;$ 
    }
  }
}
/* First compute liveness for each BB using dataflow analysis techniques.
    $PostLive(B)$  is the set of live vars (corresp. to configurations) at the exit of B.
   Then perform BB liveness based computation of interference. */
 $\forall I_1, I_2 \in I_c : G[I_1, I_2] \leftarrow true;$ 
 $CurrentLive \leftarrow PostLive[B];$ 
foreach ( $\{I \in B \mid I \text{ is a configuration instruction}\}$ , in reverse order) {
  foreach ( $I_1 \in CurrentLive[I]$ ) {
    foreach ( $I_2 \in Defined[I]$ ) {
       $G[I_1, I_2] \leftarrow true; G[I_2, I_1] \leftarrow true;$ 
    }
  }
   $CurrentLive \leftarrow CurrentLive - Defined[I];$ 
   $CurrentLive \leftarrow CurrentLive \cup Used[I];$ 
}
```

6.5.8 Spilling And Splitting Of Live Ranges

Resource assignment is blocked when all legal resources have been allocated. *Live range spilling* is one technique where the value corresponding to the live range lr_i is assigned to a memory location and all references to the variable corresponding to lr_i are performed by LOAD/STORE memory access operations. *Live range splitting* is one alternative to spilling when no candidates are available and when we need to control the program points where compensation code gets inserted and which subset of references to the associated variable reside in registers/processor local memory.

- Spill heuristics aim to find better nodes to spill.
- Live range splitting techniques aim to reduce the length of live ranges with the expectation that smaller live ranges will interfere with fewer other live ranges.

6.5.9 Pruning For Configuration Allocation

If there are program regions where the total resource requirement of configurations that are live at that point exceeds the available resources on the processor, configuration allocation will fail. Analogous to the concept of *register pressure*, we define the total resource requirement at any program point the *resource pressure* at that point. *Pruning* is a technique of preprocessing the live ranges of configurations to ensure that the *resource pressure* never exceeds the total resources available on the machine.

Pruning involves (a) determining the set of live ranges to split and, (b) determining the right split points for the selected live ranges. Once a live range is split, compensation code needs to be inserted to fetch the values (configurations) for the uses encountered in the second (or later, if multiple splits were performed) part of the live range. One downside of reducing the resource pressure is the additional time taken for executing the compensation code. Hence pruning decisions cannot be made arbitrarily. In addition to the resource pressure, execution frequency should be taken into account before a selecting a live range for splitting since that determines the number of times the compensation code would be executed.

Pruning is not a new concept. However, it has been used only in the context of register allocation in the past. We adapt those techniques to work in the context of configuration allocation. First, we present a brief survey of past work related to pruning/live range range splitting. Chow et. al [32] first proposed live range splitting. When their register allocator fails to assign a color to a live range lr , it splits lr into smaller live ranges, each smaller piece spanning a single basic block over which it is live. To decrease the amount of compensation code, adjacent live ranges are combined if the register pressure due to the combined live ranges does not exceed the available number of registers. Briggs [19] in his thesis proposed splitting based on control flow structure. He avoids the problem of picking optimal live ranges and instead chooses split points based on structure of the control flow graph and then splits all live ranges that cross that point. Two of the proposals for split points are splitting around loops, at dominance frontiers or at reverse dominance frontiers. Hansoo [87] proposed a frequency based live range splitting algorithm which attempts to split along the least frequent edges in the control flow graph. Bernstein et al. [14] proposed live range selection heuristics. The heuristics give an estimate of a live range's contribution to the total resource pressure. Callahan and Koblenz [24] proposed a hierarchical method to heuristically prune the interference graph.

We propose a pruning technique based on the hierarchical technique proposed by Callahan and Koblenz [24]. In their algorithm, gaps between references to a register in a live range are identified. These are possible regions which can be spilled to memory. The maximal length live-range gap is referred to as *wedges* in [31]. Non-overlapping and maximal wedges are identified using the control tree [109]. The choice of wedges to prune is a

function of (a) the runtime cost of compensation code that would be added in the pruned region and, (b) size of the region of the program region that would benefit from the pruning decision. Our algorithm takes into account the special requirements of configurations (their non-uniform sizes - which implies non-uniform spill costs; immutability - which implies stores to memory are not required) and also enhances the scheme with regard to identifying spill candidates and spill locations based on execution profile as well. The algorithm, adapted from the register live range pruning algorithm from [31] and is described in Algorithm 3. The relevant parameters and the data-structures used by the algorithm are listed below.

$$\begin{aligned}
R &= \text{total number of recourse units available for allocation} \\
C &= \text{configurations (candidates) to be allocated} \\
T &= \text{Control tree of the program} \\
ResUnits[c] &= \text{number of resource units required by the configuration } c \\
Live[n] &= \text{set of configurations live in control node } n \\
Refs[n] &= \text{configurations that are referenced in control node } n \\
Wedges[n] &= \text{list of candidates with wedges that have heads at } n \\
Freq[e] &= \text{number of times control traversed along edge } e \\
Excess[n] &= \begin{cases} \sum_{c \in Live[n]} ResUnits[c] - R & n \in T.Leaves, \\ \text{Max}_{c \in T.Children[n]} \{Excess[c]\} & n \notin T.Leaves. \end{cases} \\
LiveUnits[n, C] &= \begin{cases} 1 & n \in T.Leaves \wedge C \in Live[n], \\ 0 & n \in T.Leaves \wedge C \notin Live[n], \\ \sum_{p \in T.Children[n]} LiveUnits[p, C] & n \notin T.Leaves. \end{cases}
\end{aligned}$$

The list of candidate live ranges in $Wedges[n]$ is sorted by the order in which they are most desirable for pruning. For example, candidate C with larger $LiveUnits[n, C]$ is more desirable for pruning since it contributes to the resource pressure over a larger program size.

Algorithm 2 Pruning

```

function InitPrune(N) {
  if ( $N \in T.Leaves$ ) {
     $LivePart[N] \leftarrow Live[N]$ ;
     $Excess[N] \leftarrow (\sum_{n \in Live[N]} ResUnits[n]) - R$ ;
     $\forall C \in Live[N] : LiveSize[C, N] \leftarrow |C|$ ;
  } else {
     $\forall M \in N.children : InitPrune(M)$ ;
     $LivePart[N] \leftarrow \cup_{M \in N.children} LivePart[M]$ ;
     $Refs[N] \leftarrow \cup_{M \in N.children} Refs[M]$ ;
     $Excess[N] \leftarrow Max_{M \in N.children} Excess[M]$ ;
     $\forall C \in LivePart[N] : LiveSize[C, N] \leftarrow \sum_{M \in N.children} LiveSize[C, M]$ ;
     $\forall C \in Refs[N] \wedge \forall M \in N.children$ :
    if ( $C \notin Refs[M] \wedge C \in LivePart[M]$ )
      NewWedge(C, M);
  }
}
function UpdatePressure (W, N) {
  if ( $N \in T.leaves$ ) {
     $Live[N] \leftarrow Live[N] - \{W\}$ ;
     $Excess[N] \leftarrow Max\{(\sum_{n \in Live[N]} ResUnits[n]) - R, 0\}$ ;
  } else {
     $Excess[N] \leftarrow Max_{m \in N.Children} \{UpdatePressure(W, m)\}$ ;
  }
  return  $Excess[N]$ ;
}

```

6.5.10 Graph Multi-coloring Configuration Allocator (GMCA)

Here we provide a simple configuration allocator called Graph Multi-coloring Configuration Allocator (GMCA), based on Chaitin's graph coloring register allocator [27, 28] with the spill and split decision modifications suggested by Hansoo and Leung [88] for the simplified AEPIC configuration resource model ???. All past graph coloring based register allocation schemes are based on the idea of *simplification* [86]. If a graph G contains a node v with fewer than K neighbors and if $G - v$ can be colored with K colors, then G is K colorable. In the case of *configuration allocation*, multiple colors may be allocated to each node and hence it calls for a stronger version of the simplification step. We first state and prove this *simplification lemma* and then show it is used in our GMCA algorithm.

Let $G(V, E, w)$ be an undirected graph where $w : v \rightarrow \mathbf{Z}$ is a weight function defined on the vertices. Let $C : v \rightarrow S$ be a function on the vertex set such that $S \subset \{1, \dots, K\}$. Then C is a valid K -multi-coloring of the graph if $|C(v)| = w(v)$ and $\forall e \in E, \text{ where } e = (u, v), C(u) \cap C(v) = \emptyset$.

Algorithm 3 Pruning (cont'd.)

```
function Prune (N) {  
  PrioritizeWedges(N);  
  while ( $Excess[N] > 0 \wedge |Wedges[N]| > 0$ ) {  
     $W \leftarrow Wedges[N].top()$ ;  
    PruneWedge(W);  
    UpdatePressure(W, N);  
  }  
   $\forall M \in N.Children \wedge Excess[N] > 0$ : Prune(M);  
}
```

Lemma 1 [Simplification lemma]

Let vertex $v \in V$ be such that $\sum_{u \in Adj(v)} w(u) \leq K - w(v)$. Let $G' = G - v$ be the subgraph obtained by removing v and its incident edges from G . If G' can be K -multi-colored, then so can G .

Proof: Let C be the K -multi-coloring of G' . Let $S_v = \bigcup_{u \in Adj(v)} C(u)$. By definition of C , $|C(u)| = w(u)$. This implies that $|S_v| \leq \sum_{u \in Adj(v)} w(u)$. Given that $\sum_{u \in Adj(v)} w(u) \leq K - w(v)$, it implies that $|S_v| \leq K - w(v)$. Consider $C_v = \{r | r \in \{1, \dots, K\}, r \notin C(u)\}$. Clearly, $|C_v| \geq K - w(v)$. Let $C_v^{w(v)}$ be any $w(v)$ sized subset of C_v . Let

$$C'(p) = \begin{cases} C(p) & \text{if } p \in V(G') \\ C_v^{w(v)} & \text{if } p = v \end{cases}$$

Then, C' is a valid K -multi-coloring of G . □

A vertex $v \in V$ such that $\sum_{u \in Adj(v)} w(u) \leq K - w(v)$ then the vertex is called *unconstrained* else it is referred to as a *constrained* node. Pseudo-code for the Graph Multi-coloring Configuration Allocator (GMCA) based on the simplification lemma is presented in Algorithm 4. The various phases of the allocator are described here.

Phases of graph multi-coloring configuration allocator.

1. *Build*: Live ranges are computed and the interference graph is constructed during this phase. The interference graph is maintained both as an adjacency list data-structure as well as an adjacency matrix.
2. *Coalesce*: The coalesce step removes any unnecessary move (copy) instructions effectively merging the live ranges corresponding to the values connected by the move instruction. The live ranges should not interfere with each other. In AEPIC since configurations are immutable and there are no copy/move instructions which take configuration registers as operands, this step is unnecessary. We however mention it here in case such instructions are later added to the AEPIC architecture.
3. *Simplify*: The *simplification lemma* is the basis for this step. *Unconstrained* nodes are selected and pushed onto the *color_stack* and removed from the interference graph. Since the *simplification lemma* guarantees that unconstrained nodes can always be colored if the reduced graph is colorable, they are removed from the graph. This step in turn might enable other *constrained* nodes to become *unconstrained*. If so, then it is repeated until either the graph is empty or all nodes are *constrained*.
4. *Prioritize*: Priorities are assigned to *Constrained* live ranges. Live ranges are selected for coloring in order of their priority. Priority functions capture the expected benefit of allocating the live range to on chip resources as opposed to external memory. Several priority functions have been proposed in the context of register allocation which have been adapted for the special nature of configurations. Details are given in ??.

5. *ProcessNode*: Highest priority node is selected for coloring. For each node, just as in [32], a *Forbidden* set is maintained which indicates the set of colors (resources) that have already been allocated and hence cannot be used for the current node. If there are enough available colors that can satisfy the color requirement for the current node, the node is colored and the *Forbidden* sets of its neighbors are updated to reflect the allocation. If the set of available colors cannot satisfy the demand for this node, then the node is either *spilled* or the live range *split* depending on which one is most beneficial.
6. *ProcessStack*: Unconstrained nodes eliminated (pushed onto the *color_stack*) during the simplify phase are colored in the reverse order in which they were removed from the graph. Original graph is incrementally reconstructed by adding one node at a time from the top of the *color_stack* and is assigned the color vector. The *simplification lemma* guarantees that enough colors are available to color the inserted node.

K = number of colors (resource units available for allocation)
 G = interference graph to be colored
 v = node G corresponding to a live range of a configuration
 $v.ColorReq$ = number of colors required by v
 $v.Assignment$ = K bit vector indicating colors assigned to v
 $v.Forbidden$ = K bit vector indicating colors that cannot be allocated to v

Algorithm 4 Graph multi-coloring configuration allocator

```

function GMCA(CFG cfg) {
   $G \leftarrow Build(cfg)$ ;
  while ( $G \neq \emptyset$ ) {
    while ( $\exists v | v$  is unconstrained) {
       $S.push(v)$ ;
       $G \leftarrow G - \{v\}$ ;
    }
    if ( $G \neq \emptyset$ ) {
       $ComputePriorities(G, h)$ ;
       $v \leftarrow HighestPriorityNode(G)$ ;
      if ( $IsColorable(v)$ ) {
         $Color(v)$ ;
      } else {
         $G \leftarrow Split(G, v)$ ;
      }
    }
  }
   $ProcessStack(S)$ ;
}

```

6.5.11 Effect of procedure linkage conventions

Procedure linkage conventions may specify additional constraints on register allocation by specifying that certain registers be saved by the caller and others by the callee. Typically spill costs for caller-saved register are different

Algorithm 5 Graph multi-coloring configuration allocator (cont'd.)

```
function IsColorable( $v$ ) {  
    return ( $v.ColorReq \geq (K - \sum_{0 < i \leq K} v.Forbidden[i])$ );  
}  
  
function Color( $v$ ) {  
     $availColors \leftarrow v.Forbidden$ ;  
     $P \leftarrow \underset{\{i | 0 < i \leq K\}}{Min} \{ \sum_{0 < k \leq i} availColors[k] = v.ColorReq \}$ ;  
     $mask \leftarrow 1^P 0^{N-P}$ ;  
     $v.Assignment \leftarrow v.availColors \wedge mask$ ;  
     $\forall u | u \in Adj(v) : u.Forbidden \leftarrow u.Forbidden \vee v.Assignment$ ;  
}
```

from callee-saved registers since the caller-callee execution patterns need not be identical. This implies that in order to optimize performance, register allocator cannot view the set of registers as a uniform resource and hence cannot make arbitrary decisions about allocating a register for a selected live range.

Similarly, procedure linkage convention may cause splits in live ranges of configurations that cross the procedure call instructions. However, we bypass the *call costs* of spilling configuration live ranges due to procedure call statements by simply allocating a new MRLA context to the called procedure and “pushing” the *execution context* of the calling procedure. The set of MRLA contexts are viewed as a stack of execution contexts. The *active context* is the context of the currently executing procedure. The called procedure uses the new execution context and when it returns, the pushed context of the calling procedure is “popped” back to become the *active context*. The *configuration registers* (the registers which serve as “aliases” to allocated configurations) are themselves viewed as grouped into register windows. One window being allocated for each called procedure. Some of the configuration registers are reserved as global configuration registers and are used to alias the configurations that have either been allocated or will be allocated to the C-cache.

6.5.12 Hardware support for allocation

Here we present a simple solution to the problem of allocating/de-allocating C-cache resources as configurations are swapped in/out of it. In other words, we provide efficient hardware implementations of the following instructions since these are the only instructions that are used to allocate/de-allocate space for configurations on the C-cache.

```
calloc  $cr, r, N$   allocate  $N$  blocks in C-cache for configuration located at  
                    address  $r$  and associate configuration register  $cr$  with  
                    the allocated configuration.  
free  $cr$          free the C-cache blocks allocated for configuration  $cr$ 
```

CC = denotes C-cache
 N = total number of C-cache blocks
 $CC.FV$ = N bit binary “free” blocks vector
 $CC.FV[i] = \begin{cases} 1 & \text{if } i^{th} \text{ C-cache block is unallocated} \\ 0 & \text{if } i^{th} \text{ C-cache block is allocated to some } cr \end{cases}$
 $CC.N$ = number of free blocks in C-cache
 cr = configuration register
 $cr.N$ = number of C-cache blocks allocated to cr
 $cr.AV$ = allocation vector for cr
 $cr.AV[i] = \begin{cases} 1 & \text{if } i^{th} \text{ C-cache block is allocated to } cr \\ 0 & \text{if } i^{th} \text{ C-cache block is not allocated to } cr \end{cases}$
 x^p = denotes p bit binary vector in which every bit is x where $x \in \{0, 1\}$
 $_k[X]$ = leftmost k bits of N bit vector X
 $[X]_k$ = rightmost k bits of N bit vector X
 $A.B$ = concatenation of binary vectors A and B

Algorithm 6 Semantics of **free** operation

```

function free( $cr$ )
{
   $CC.FV \leftarrow CC.FV \vee cr.AV$ ;
   $CC.N \leftarrow CC.N + cr.N$ ;

  /* Clear C-cache blocks allocated for  $cr$  (those that
     are marked “1” in  $cr.AV$ ) */
   $cr.AV \leftarrow cr.AV \wedge 0^N$ ;
   $cr.N \leftarrow 0$ ;
}

```

6.6 Instruction Scheduling

Consider the IDCT kernel used in many video compression applications (eg. MPEG.) The code size for this function using our compiler for a 9-issue EPIC processor ([70]) is less than 2KB. The compiler generated 184 operations. We assumed a simple encoding scheme using 32 bits per operation. In [124], it was shown that it would require 4920 CLB’s on a Xilinx 4K FPGA to implement a 16 bit fixed point version of the IDCT function. The configuration size for programming the 4920 CLB’s could run into tens of kilobytes. Clearly, it is very inefficient to load the IDCT configuration every time it is executed. The benefits of faster execution on the reconfigurable

Algorithm 7 Semantics of **calloc** operation

```
function calloc(cr, r, K)
{
  if ( $N < K \parallel CC.N < K$ )
    exit("error: insufficient number of available blocks.");
   $P \leftarrow \underset{\{i \mid 0 < i \leq N\}}{\text{Min}} \{ \sum_{0 < k \leq i} CC.FV[k] = K \};$ 
   $mask \leftarrow 1^P 0^{N-P};$ 
   $CC.FV \leftarrow CC.FV \wedge \overline{mask};$ 
   $CC.N \leftarrow CC.N - K;$ 
   $cr.AV \leftarrow CC.FV \wedge mask;$ 
   $cr.N \leftarrow K;$ 
  /* now initiate memory fetches for configuration at r into
     C-cache blocks allocated to cr (marked in cr.AV) */
}
```

logic array can be lost due to the high reconfiguration times. We propose the following techniques to address this problem:

- Decouple configuration loading from and configuration execution. Consider wherein, function *f3* executes configuration *C*. In order to hide the long configuration load time, our compiler would identify speculatively load the configuration data into the configuration cache (C-cache) preceding to the call to *C*. The architecture supports a *configuration-pointer* (as part of the configuration register). This is analogous to loop counters in DSP processors. It indicates to the configuration load instruction, the appropriate memory location to read, each time it is executed. This avoids the problem of separate load instructions for a configuration from reading the same memory location. The slots where the configuration load instruction is speculated are ranked according to the number of times they might be executed (obtained from an execution profile) and their proximity to the actual point of use of the configuration. We use this information to select appropriate slots for the speculated configuration loads. Loading a configuration too early is wasteful if the control flow never reaches the actual point of use and too late might not effectively mask the configuration load overhead.
- Since configurations can correspond to larger chunks of computations compared to standard RISC/VLIW style instructions, it is likely that the number of input/output operands to the operation performed by the configured functional unit is large. Hence we extend the above latency masking idea even further by decoupling supply of operands to the configuration instructions, from the triggering of their execution. This is one of the reasons why our architectural space is restricted to those processors for which operands to configurations are implicitly specified. Compiler schedules the operand assignment and data supply to the configured functional unit prior to its execution.
- In the EPIC core there is already a provision for specifying caching hints to load/store operations indicating choices for data placement within the cache hierarchy. We extend this feature to the load operations that fetch the configurations. We believe that an explicitly controlled *configuration cache* for intelligently caching configuration data between consecutive uses of the same configuration can vastly alleviate the problem of high reconfiguration overhead.

In this section, we describe the scheduling problems associated with AEPIC architectures in a formal setting. These scheduling problems are related to problems related to precedence constrained scheduling with non-unit latencies and multiple-pipelines, and to various forms of bin packing problems.

6.6.1 AEPIC Features Relevant To Scheduling

The specific features of an AEPIC processor that determine how instructions are scheduled for execution are discussed here.

MultiOp. An AEPIC processor relies on the compiler to specify the set of operations to be issued on each machine cycle and on which functional unit they are expected to execute. This set of operations is specified as a single packet called *MultiOp*. An AEPIC processor executable is composed of a sequence of MultiOps. A machine cycle is the smallest unit of time on the virtual time-line implicitly specified by the POE.

Dynamic ISA. Abstractly, an AEPIC processor data-path is composed of (a) a set of functional units each of which is capable of performing a set of operations, and (b) a set of register files with an interconnect network between the two. In addition, AEPIC provides architectural features to add/delete functional units and specify their interconnection to the register files, during runtime, in effect, varying the instruction set architecture of the machine dynamically. Functional units that are instantiated during runtime are called *configured functional units* (CFUs).

Non-Unit Assumed Latencies (NUAL). Although an AEPIC executable specifies that a new MultiOp is to be issued on each machine cycle, it does not imply that the operations in the MultiOp actually finish before the next MultiOp is issued. Operations could take several cycles (especially in the context of reconfigurable computing where the application specific operations are usually large pieces of computation) and in fact, operations of the same MultiOp may generate their results at different times. Following the EPIC philosophy which forms the core of AEPIC, it is actually the read/write events that are considered atomic instead of the instruction itself as is the practice in superscalar architectures. *This implies that AEPIC operations take non-unit latencies and that this latency information should be available to both the compiler which generates the executable and to the processor that ensures correct interpretation of the executable.* One of the motivations for exposing this non-unit latency behavior to the compiler is with the hope that the compiler will be able to generate compact schedules if given the exact time instants at which values are produced and consumed by instructions.

Explicitly specified latencies. In the case of EPIC, the operation set is fixed. Hence both the compiler and the processor can be built with the knowledge of the operation latencies obviating the need for communicating the latency information through the executable. However, in the case of AEPIC, the operation set is determined only at compile time and somehow the compiler has to communicate this information to the processor. There are several possible ways to do this []. The mechanism we propose is to explicitly specify latency information as part of the instruction. Often the custom operations not only have long latencies, but also the latency behavior is not deterministic, most often depending on the values of the input operands. Being able to explicitly specify latencies as part of the operation itself yields the greatest amount of flexibility since every invocation of an operation can specify a different latency value — the value that the compiler itself assumed for this particular operation. The processor takes appropriate action (whether to delay the commitment of results or to stall the processor) depending on the difference between the actual latency and the compiler communicated latency.

6.6.2 Scheduling Model

AEPIC functional units To simplify the model and subsequent discussion, we assume that we have m' *homogeneous* functional units which can process all types of AEPIC instructions (which are basically EPIC-style instructions). These AEPIC functional units are numbered from 0 to $m' - 1$. In a more realistic setting, instructions and functional units are partitioned by their types — i.e. load/store, floating point, integer, branch

instructions, etc — and only instructions from the appropriate type can be assigned to a functional unit. We will assume that all these functional units are pipelined.

Configured functional units In addition to the m' non-adaptive, pipelined functional units, we assume that there are an additional m'' ALU data-path slices (corresponding to the *slices* of the execution contexts of the MRLA) which may be used by the *configured functional units* (CFU) numbered m' to $m' + m'' - 1$. Each CFU may consume an arbitrary number of these ALU slices while it is active on the AEPIC data-path.

Some of the AEPIC instructions may trigger computations on the CFUs. These instructions executed on the CFUs are the application specific custom instructions synthesized by the AEPIC compiler. Each such custom instruction i may utilize a CFU during its execution. CFUs and custom instructions are typed, such that a custom instruction i can only be executed on an CFU of type $type(i)$. CFUs, unlike the normal functional units, may not be pipelined, i.e. these instructions have non-unit processing times, and typically have non-zero latencies.

Precedence constraints As in previous work in this area, we assume that the basic block being scheduled is represented as a DAG, with nodes representing the instructions, and edges representing the *precedence* constraints between the instruction. We use $i \prec j$ to denote that i must precede j in all feasible schedules.

An instruction i requires p_i cycles of *processing time*. On a pipelined machine, processing times are used to model the number of *interlock* cycles for an instruction, i.e. if i is issued on some pipeline, then no other instructions may be issued on the same pipeline before p_i cycles have elapsed. For EPIC machines, it is natural to assume that $p_i = 1$ for all i , i.e. we have an *Unit Execution Time* (UET) model.

Latencies Results computed from one instruction may not be immediately available to subsequent instructions in the instruction stream. For example, if an instruction i is a load instruction, there might be additional delays due to access latencies to cache or main memory. An instruction j that is data-dependent on i may have to be delayed an additional l_i cycles. We call the l_{ij} the *latency* between i and j . This model has been developed in works such as [116, 97], where we assume that the latency is a function of both the producer of the value, and its consumers. As in EPIC/VLIW machines, which usually has simpler pipelines than superscalar architectures, it may be possible to assume that the latency is independent of the consumer even for AEPIC processors. In such cases we will drop the second subscript and denote the latency of i as l_i . In addition, instructions executing on some of the configured functional units may have data (input operand) dependent latencies. In such cases, we denote the set of latencies associated with i as $\{l_{ij}^1, l_{ij}^2, \dots, l_{ij}^k\}$.

6.6.3 The Resource Constrained AEPIC Scheduling Problem

The *resource constrained AEPIC scheduling problem* is to locate a feasible schedule (σ, ψ) . Here, $\sigma : \mathbf{I} \rightarrow \mathbf{N}$ maps each instruction to its start time in the schedule, and $\psi : \mathbf{I} \rightarrow \{0 \dots m - 1\}$ maps each instruction to the functional unit that it utilizes.

A schedule (σ, ψ) is feasible \iff it satisfies the following constraints:

latency For all instructions i, j such that $i \prec j$, $\sigma(i) + p_i + l_{ij} \leq \sigma(j)$ for all $i \prec j$.

resource For all time t , $|\{j \mid \sigma(i) \leq t \leq \sigma(i) + p_i\}| \leq m$

resource assignment For all instructions i, j , if $\psi(i) = \psi(j)$ then $[\sigma(i), \sigma(i) + p_i - 1] \cap [\sigma(j), \sigma(j) + p_j - 1] = \emptyset$.

The first two constraints are essentially the same constraints that are used to model traditional EPIC/VLIW machines. The resource assignment constraint is used to specify that the same CFU is not utilized by two custom

Problem	Reference
$P p_i = 1; prec C_{\max}$	Garey and Johnson [51]
$1 p_i = 1; prec(l_{ij}) C_{\max}$	Hennessy and Gross [67]
$1 p_i = 1; chains(l_i \in \{0, k_1, k_2\}) C_{\max}$	Palem and Simmons [116]
$P2 p_i \in \{1, 2\}, prec C_{\max}$	Ullman [169]

Figure 6.1: NP-hard problems from scheduling

Problems	Reference
$1 p_i = 1; prec(l_{ij} \in \{0, 1\}); r_i L_{\max}$	Leung et al. [97]
$P p_i = 1; int - order(mono l_{ij}); r_i L_{\max}$	
$P2 p_i = 1; prec(l_{ij} \in \{-1, 0\}); r_i L_{\max}$	
$1 p_i \geq 1; prec(l_{ij} \in \{0, 1\}) C_{\max}$	
$P p_i = 1;intree(l_{ij} = l) L_{\max}$	Bruno, Jones, So [21]
$P p_i = 1;outtree(l_{ij} = l); r_i C_{\max}$	
$P p_i = p; r_i L_{\max}$	Simmons [153]

Figure 6.2: Polynomially Solvable Instances

instructions at the same time step.

Intuitively, we can treat the AEPIC scheduling problem as precedence scheduling on heterogenous functional units with non-unit processing time and latencies.

6.6.4 Some Complexity Results

For brevity, we will extend the $\alpha|\beta|\gamma$ notation of Graham et al. [57], and Brucker and Knust [20] to describe AEPIC scheduling problems. We use $P|prec(l_{ij})|C_{\max}$ to denote the general problem of scheduling a precedence constrained AEPIC scheduling problem on multiple pipelines. Here, P specifies that there are multiple pipelines; $prec(l_{ij})$ specifies that general precedence constraints are allowed and general latencies are allowed. Finally C_{\max} states that the problem is to minimize the maximal completion time.

Clearly $P|prec(l_{ij})|C_{\max}$ is NP-hard, as it reduces to many simpler problems which are also NP-hard known from deterministic scheduling (see Figure 6.1).

Garey and Johnson [51] have shown UET precedence constrained scheduling on parallel processors is NP-hard. Dealing with only latencies, Hennessy and Gross [67] have shown that scheduling on one pipeline but with arbitrary latencies is also NP-hard. Note that restricting the latency model to the “producer-only” model does not make the problem easier, as Palem and Simmons [116] have shown that a more restricted problem with chains is also NP-hard. The problem with only 2 processors, without latencies, and only with processing times of 1 or 2, is also NP-hard, as Ullman [169] has shown. Note that this implies even the simplest AEPIC scheduling problems with the most restricted shapes of “tiles” are NP-hard.

In [97] Leung et al., have shown that a similar problem involving arbitrary processing times and latencies of 0 and 1

$$1|prec(l_{ij} \in \{0, 1\}); p_i \geq 1|C_{\max}$$

is polynomially solvable. However, the algorithm is restricted to 1 pipeline and does not generalize. Other

polynomially solvable instances related to pipelined scheduling, have also been identified in the paper [97]. All related polynomially solvable problems in the literature are summarized in Figure 6.2.

Chapter 7

AEPIC Simulation

7.1 Introduction

In this chapter we present the design of the AEPIC simulator. AEPIC simulator is based on the cycle-level simulator of the HPL-PD EPIC architecture [84] which is distributed with the Trimaran ILP compiler infrastructure [70].

In Section 7.1, AEPIC simulator design goals and requirements are presented. Relevant details of the EPIC simulator in Trimaran are discussed in Section 7.2. In Section 7.3, we review details of the semantics of AEPIC program execution, focusing on those aspects that are relevant when constructing the simulator. AEPIC specific extensions to the Trimaran’s simulator are also presented in this section.

7.1.1 Simulator Requirements

The following are the requirements of the AEPIC simulation environment.

- To convert code generated by AEPIC compiler into a format that can be executed on an existing architecture and yet simulate AEPIC execution semantics.
- To generate run-time information such as clock cycles taken for execution, average number of operations executed per cycle, static instruction counts, register allocation overheads, etc.
- To provide detailed information about the execution profile on the adaptive component of AEPIC such as time spent for data-path reconfiguration, computation time on MRLA, effectiveness of configuration caches, etc.
- To provide a framework that allows researchers to extend the functionality of the run-time performance analysis module with additional modules that implement algorithms intended to analyze AEPIC execution traces.

7.1.2 AEPIC Simulator Design Goals

Parameterized by machine description. The simulator should not be programmed with any inbuilt assumptions about the specific instance of AEPIC machine it might be simulating. In other words, all the architectural parameters of AEPIC are to be read from the AEPIC machine description itself before (or during) every simulation.

Platform independent design. The simulation system should not make any assumptions either about the operating system or about the architecture of the host platform on which the simulator executes.

Inter-operability. Here the intent is to allow free inter-mixing of AEPIC and host platform code. In other words, an executable program can consist of a mix of AEPIC object files and host platform object files. This allows the application that is being targeted to the AEPIC machine to invoke operating system or library functions of the host platform.

Hooks for performance monitoring. The AEPIC compilation framework is meant for performing research. Hence one cannot anticipate all the ways in which the infrastructure may be utilized. Allowing a developer to extend the performance monitoring framework is a key requirement for performing AEPIC compiler or architecture research. Examples of such tools include a data and configuration cache memory simulator, tools for determining runtime reconfiguration overheads and MRLA utilization.

7.2 Trimaran’s EPIC Simulator

7.2.1 Overview

The HPL-PD EPIC architecture simulator-generator is distributed as part of Trimaran [70]. In addition to the primary task of simulating the HPL-PD EPIC architecture, it provides run-time information on execution time, branch frequencies, and resource utilization. This information can be used to perform profile-driven optimizations as well as to validate new optimizations. Trimaran also comes with a Graphical User Interface (GUI) for configuring and running the Trimaran ILP compiler. Included in the GUI are tools for the graphical visualization of the program intermediate representation and of the performance results generated by the simulator. The simulator-generator is also parameterized by the machine description so that when machine configuration is altered, the simulator-generator need not be recompiled.

Behavior of the simulator can be controlled through several runtime options. Some of these options may be enabled to generate execution traces containing runtime information, a feature that is useful to study the performance of new architectural features. For example, all memory instructions and the memory locations they access can be output in the trace. Traces can contain billions of instructions and consume a lot of disk space depending on the level of detail one wishes to probe the execution. Hence, Trimaran employs an execution-driven simulation model which eliminates the need for producing and storing trace off-line, by dynamically generating the trace stream. The trace processing and profiling extensions can consume the trace as it is generated.

7.2.2 Simulator Code Generation Process

Simulation process is illustrated through Figure 7.1. Input to the simulator-generator is the benchmark application source code in C. Trimaran generated code for the intended EPIC target is “assembled” by the *Code Processor*. The output generated by the *Code Processor* is a collection of low-level C files which form the simulator “pseudo-executable”. These files are described in Section 7.2.3. This C pseudo-executable also contains patch-up code for entry and exit into the simulator world from external library function calls. Since all the code for the generated pseudo-executable is in C, a native compiler can be used to generate the equivalent host machine code. The C code is compiled and linked with the *Simulation Library*, and the *Cache Simulator* by the native C compiler. The *Simulation Library* contains the EPIC virtual machine interpreter and other simulation specific utilities. The EPIC interpreter is invoked on every application procedure entry. It simulates the instruction stream in a loop until the procedure returns. There is one simulation function for every HPL-PD operation. The *Cache Simulator* called *SMACHS* simulates the HPL-PD data cache hierarchy which consists of a streaming cache also called the

data pre-fetch cache (V1), a conventional first and second level caches (L1 and L2) and main memory which is external to the processor.

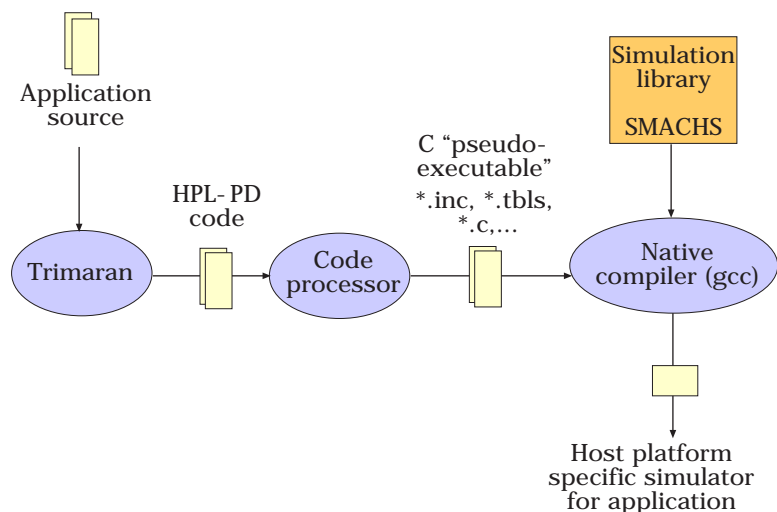


Figure 7.1: EPIC simulator generation in Trimaran

7.2.3 Structure Of The Simulator

The Trimaran simulator-generator generates a set of low-level C files corresponding to each of the application source files. These low-level C files are compiled by the native compiler and linked with the simulation library and the cache library. The generated executable is the simulator for *this particular application*. The low-level C files created for each application source file *xyz.c*, are as follows.

1. *xyz.inc*: External variable declarations, global data, modified structure and union declarations, global data used by the simulator at runtime are included here.
2. *xyz.tbl*: This file contains a set of emulation tables for each C function in the source file (*xyz.c*). An emulation table is a C array of *Op* structures. Each *Op* structure contains pointers to the source and destination operands of the operation and a pointer to the function in the emulation library that implements this operation type. In other words, it contains all the information necessary for the simulator to simulate this operation.
3. *xyz.c*: This file includes the *xyz.tbl* and *xyz.inc* files and also declares *vector functions* (described in Section 7.2.5) for all the functions invoked in the source file.

In addition, a *benchmark_data_init.simu.c* file is also generated for the entire application. It contains the definition of a function that initializes the global data of the application.

Example. Implementation of the subtract literal operation from the HPL-PD EPIC ISA is shown in Program 3 and is explained below. The simulator functions for most operations have a similar structure.

1. Each operation from the ISA is associated with a C function. The `PD_SUBL_W_reg_lit` function is associated with the `SUBL_W` HPL-PD operation.

2. Code in the function computes the operation and saves the result into local variable 'r'. In Program 3, the assignment statement computes the subtract literal operation and saves result into local variable r. This value is not reflected in the processor state until the operation latency has expired.
3. Schedule the update to the processor state to occur after L cycles where, L is the latency of the operation. In Program 3, the **PD_delay** function call schedules the update of processor register $op \rightarrow dest[0] \rightarrow r$ (the processor location that is affected by the execution of this operation) with the value in the local variable r to occur after $op \rightarrow lat[0]$ (= L) cycles.

Memory access operations make a call to the Cache_Manager to determine if the particular location is available in the cache and the number of stall cycles to introduce. Operations that perform procedure calls are described in Section 7.2.5.

Program 3 Simulator code of an HPL-PD operation

```
function void PD_SUBL_W_reg_lit(Op *op)
{
    REG r = (op → src[0] → reg) - (op → src[1] → lit);
    PD_delay(&r, &(op → dest[0] → r), op → lat[0]);
}
```

7.2.4 The Interpreter

The key data structure used by Trimaran's simulator is the *event queue* Q, which is a queue of pending action lists L_a ordered by virtual time. Action list is simply a list of *action items*. An *action item* (or action for short) is either a *write* action or a *function call* action. These actions are created every time an operation is fetched and processed. A *write* action copies a value from simulator local state to processor state and is implemented as a simple assignment statement. Actions that read values from processor state for input operands of instructions, are performed at the same time as when the operation is issued. Since all the input operands are available at issue time, the computation corresponding to the operation is also performed at issue time. Hence read actions are not scheduled on the event queue. Memory reads are also read actions. A *function call* action transfers control to the specified function, implicitly updating the processor state and the state of the application that is being simulated. The target function may be an external library function or a function defined in the application source. The actions performed by the simulator when the called function is external are different from the actions that need to be taken when the function is defined within the application source itself. Details are given in Section 7.2.5.

The interpreter fetches the operations in sequence from the table of operations (listed in the xyz.tbls files) associated with each source function, and schedules the actions specified by the operation on to the event queue, Q. All operations of a MultiOp are scheduled relative to the same issue time—the virtual time slot during which the EPIC processor issues the MultiOp. Operations belonging to adjacent MultiOps are separated by a MultiOp separator (dummy operation) called *ACLOCK*. When the interpreter encounters an *ACLOCK* operation, it commits all the pending actions scheduled for the current virtual time slot on Q and advances the virtual time as well as the program counter.

The function of the application that gets invoked first starts the interpreter. This function is typically the *main* function in the C source. Since the simulator allows linkage to external functions and the *main* function itself could be external and need not be the function that starts the interpreter. A simplified version of the interpreter loop in the absence of speculation, and inter-procedural jumps is shown in Program 4. The interpreter sequences

through the operations of the function (line 1) processing each operation in turn (line 3).

Program 4 EPIC Interpreter loop

```
function void PD_simulate(Op *opslst)
{
    Op* op_ptr;
1  for (int i=0, op_ptr = opslst; !done; op_ptr++) {
2      PD_pc = op_ptr → getPC();
3      *(op_ptr → op)(PD_pc);
4      op_ptr = (Op *)PD_pc → getOp();
    }
}
```

7.2.5 Handling Function Calls

A skeleton function called **vector function** is defined for each function *called* in the application's source code. It is not necessary for the called function to be defined within the application itself. The vector function (1) maps HPL-PD registers to native function parameters followed by (2) code to invoke the function with the transformed parameters. In Program 5, **_vector_test** is the vector function associated with the original source function **test**. The *f* and *b* parameters of source function **test** are mapped to the architectural registers *PD_Float_p1* and *PD_Int_p1*. The reverse mapping from native function parameters to HPL-PD registers is performed after the native function returns. This is illustrated by the example shown in Program 5. This is illustrated by the example shown in Program 5. *PDS_Int_Result*, *PDS_Float_Arg1* and *PDS_Int_Arg1* are macros that are mapped to specific registers in the HPL-PD architecture that have been reserved for passing function parameters.

The actual function that is invoked by the vector function is a **wrapper function** (defined below) for functions that are defined within the application source otherwise is a call to an externally defined function. The **wrapper function** for each function *defined* in the application source contains code that maps native function parameters to EPIC registers, followed by a call to simulation library function **PD_Simulate** and then code to perform the reverse mapping from EPIC registers back to native function parameters. In Program 5, the input parameters *f* and *b* are copied into the parameter passing registers *PD_Float_p1* and *PD_Int_p1*. The **PD_simulate** function from the simulation library invokes the interpreter on the “assembly” code for the **test** function which is stored in the C structure **_PD_tbl_test**. The return value of the original test function saved into the EPIC register *PD_Int_Result* is returned by the wrapper function.

Function calls in the HPL-PD architecture are performed using the *branch-and-link* (BRL) operation. In the simulator code, the source operand for a BRL operation is a register containing a pointer to the *vector function* associated with the target of BRL (the original function to be invoked). The simulator invokes the vector function, which in turn performs one of two tasks: (a) it calls an external function if the original function was not defined in the application source or (b) it invokes the *wrapper function* otherwise. The wrapper function connects back to the simulation world through a recursive call to the simulator.

Program 5 Source function and the associated wrapper and vector functions

```
/* original function definition */
function int test(float f, int b) ...

/* original definition of test is replaced with a wrapper function which invokes the
   simulator recursively to execute the code generated for the original function */
function int _test(float f, int b)
{
    /* copy parameters to processor state */
    PD_Float_p1 = f;
    PD_Int_p1 = b;
    /* invoke simulator on the code for test */
    PD_simulate(_PD_tbl_test);
    /* return the result of the computation */
    return (int) PD_Int_Result;
}

/* the vector function associated with test. */
function int _vector_test(void)
{
    *PD_Int_Result = _test(*PD_Float_p1, *PD_Int_p1);
}
```

Program 6 Pseudo-code generated by the simulator

```
/* code generated for function test is a "C" array of operations*/
Op _PD_tbl_test[315] =
{
    {1, __PD_start_procedure, {{(PD_REG *) "_test",0,0x0,0}, ...
    {0, __PD_prologue, {{(PD_REG *) 1568,0,0x0,0}, ...
    {353, __PD_ADD_W_reg_lit, {{(PD_REG *) __PD_RS,0,0x0,1}, ...
    {0, __PD_aclock, {{(PD_REG *) 1,0,0x0,0}, ...
    {354, __PD_S_W_C1_reg_reg, {{(PD_REG *) ...
    ...
}
```

7.2.6 Performance Monitoring Library

The **Performance Monitoring (PM)** library provides a C++ interface for building of performance monitoring tools. Examples of such tools are memory profiler and control flow profiler. The PM framework processes events generated by the simulator and filters events that are of interest to the user.

Trace events. Control flow information, procedure entry/exit, control block entry/exit, operation nullification, memory access information are some of the events generated by the simulator. The events the user wishes to monitor and the region of the program intermediate code (the viewing window) over which these events are to be monitored are specified in configuration files.

Trace processing. The PM framework includes basic classes for reading the trace and filtering events from the code region of interest. User specific trace processing tools can be built by extending the base classes. A few examples and further details about the PM library are included in the Trimaran distribution [70].

7.3 Design Of The AEPIC Simulator

AEPIC simulator design is based on the EPIC simulator in Trimaran. We discuss only the relevant extensions to the EPIC simulator. Our discussion is split into three parts: (1) modeling AEPIC specific extensions to the EPIC processor state, (2) implementing AEPIC ISA extensions to the EPIC ISA and, (3) extensions to Trimaran’s performance monitoring framework.

In the case of EPIC processor, one need only model the register files and the cache memory system. However, in the case of AEPIC, in addition to the EPIC core state, the simulator needs to model the MRLA, C-cache, configuration memory hierarchy, configuration register files and the set of CFUs. Modeling AEPIC state is discussed in Section 7.3.1.

Simulation of instructions from the EPIC subset of the AEPIC ISA does not pose any special problems—they are handled the same way as in the EPIC processor simulation. The adaptive extension instructions and CFU operations require special handling. We discuss these issues in Section 7.3.2. in Section ??.

7.3.1 Representing AEPIC Processor State

Since the EPIC core portion of the state is identical to Trimaran’s EPIC processor, we do not discuss that component of AEPIC in the AEPIC simulator. The remaining elements of interest are Multi-context Reconfigurable Logic Array (MRLA), configuration cache hierarchy including C-cache, Array Register File (ARF), Configuration Register File (CRF) and the Configured Functional Units (CFUs).

7.3.1.1 Configuration Register File (CRF)

CRF is a collection of configuration registers. In the simulator, it is represented as an instance of the CRF class whose interface is shown in Program 7. The *free_list* maintains the list of unallocated configuration registers and is updated by the *alloc* and *free* functions. AEPIC latency values have the internal type AE.LAT.

Allocated, configuration register serves as an alias to an allocated configuration. It contains all the information pertaining to that configuration—its resource allocation (whether on the MRLA or in the C-cache), its input/output assignment if it refers to a CFU, execution status of CFUs and temporary storage for buffering the values read (written) by the associated CFU before they are committed to the processor state. The configuration register is represented by a **ConfigReg** object as defined in Program 8. All the methods are invoked only when

Program 7 CRF interface

```
class CRF
{
    ConfigReg calloc(Reg r);
    void free(ConfigReg cr);
    void ldcc(ConfigReg cr, AE_LAT l);
    void ldcns(ConfigReg cr, AE_LAT l);
    void stcc(ConfigReg cr, AE_LAT l);
    void stccns(ConfigReg cr, AE_LAT l);

    ConfigReg ConfigRegisters[NUM_CONFIG_REGISTERS];
    int *free_list;
}
```

the configuration register refers to a CFU (in which case, the *is_cfu* field is set to true).

The *in_reg_refs*, *out_reg_refs* hold references to objects that represent the ARF registers that are intended to hold the input and output operands for the CFU associated with this configuration register. The *tmp_in_reg*, *tmp_out_reg* serve as temporary storage for values.

The *event_queue* data structure is identical to that of the *event queue* (Q) of the EPIC simulator. Since the events of each individual CFU may proceed asynchronously with respect to the global AEPIC event queue (explained later), each CFU maintains its own event queue. The *clock* field indicates the position of the virtual time in the CFU local event queue. This clock is incremented on every update to the global AEPIC clock except when the CFU execution has been suspended (by calling *susp* instruction on this CFU).

The *cfg_alloc* field holds the resource allocation map for the CFU. Recall that architecturally visible configurations are either allocated in the C-cache or on the MRLA (in which case, we refer to them as CFUs). If the configuration data associated with the configuration register is in the C-cache, then the *ccBAV* vector indicates the C-cache blocks allocated to this configuration. If the configuration is present on the MRLA, the *mrlaAlloc* field specifies the position, context and the number of slices of MRLA allocated to it. Since the slices allocated for CFUs on MRLA are required to be consecutive, the *left_slice_id*, *num_slices* and *context_id* are sufficient to determine a CFU's allocation on the MRLA. The *is_cfu* field, if true, implies that the configuration data is on the MRLA. If the *is_allocated* field is false, then this configuration register is available for allocation to a new configuration. The member functions of the ConfigReg are explained in Section 7.3.2.

7.3.1.2 Multi-context Reconfigurable Logic Array (MRLA)

The MRLA is represented by the interface in the simulator as shown in Program 10. The *malloc* function allocates requisite number of slices for configuration *cr*, on the MRLA context *cid*. If the *free_list* does not have a contiguous set of slices that can accommodate the desired configuration, an exception is raised. The *free* method deallocates the slices allocated to the CFU (*cr*) and adds them to the *free_list*. The method *clctx* (*clrallctx*) *frees* resources allocated for all CFUs on the context *ctx* (on all contexts). The *gcc* and *gcall* methods are intended for performing garbage collection on the MRLA. They are left for a future version of the AEPIC processor. The *pushctx* (*popctx*) allocate (deallocates) *num* contexts for the calling function so that the function may allocate CFUs on the allocated contexts. The *switchctx* function moves the CFU referred to by *cr* to the context *ctx*. AEPIC processor can issue

Program 8 ConfigReg interface definition

```
class ConfigReg
{
    bool is_allocated, is_cfu;
    AE_CfgStatus cfu_status;
    AE_CfgAlloc cfg_alloc;
    AE_Reg *in_reg_refs, *out_reg_refs, *pred_reg, *stat_reg;
    AE_Reg *tmp_in_reg, *tmp_out_reg;
    AE_Lat *lat_in_opnds, *lat_out_opnds;
    AE_Q *event_queue;
    AE_CFU_FN *cfu_fn;
    AE_CLK clock;
    int next_in, next_out;

    void inpr(AE_Reg r);
    void inp(AE_Reg r, int k);
    void outpr(AE_Reg r);
    void outp(AE_Reg r, int k);
    void pred(AE_Reg p);
    void stat(AE_Reg r);
    void exec(AE_OPCODE opc, AE_LAT l);
    void susp(); void resume(); void abort();
    void reset(); void step(); void tick();
}
```

Program 9 Configuration allocation data structure

```
typedef struct AE_CfgAlloc {
    union {
        bool ccBAV[CC_SIZE];
        struct AE_MRLA_Alloc {
            int context_id;
            int left_slice_id;
            int num_slices;
        } mrlaAlloc;
    } cfu_alloc;
}
```

CFU operations to only those CFUs that are on the **active context**. The *setctx* method can be used to switch to a different **execution context** and make it the **active context**.

Program 10 MRLA object interface

```
class MRLA
{
    /* MRLA resource allocation */
    void malloc(ConfigReg cr, int ctx);
    void free(ConfigReg cr);
    void gcc(int ctx);
    void gcall();

    /* managing MRLA contexts */
    void setctx(int ctx);
    void clrctx(int ctx);
    void clralctx();
    void switchctx(int ctx, ConfigReg cr);
    void pushctx(int num);
    void popctx(int num);

    struct AE_MRLA_AllocMap *alloc[NUM_CONTEXTS];
    struct AE_MRLA_AllocMap *free_list[NUM_CONTEXTS];
}
```

7.3.1.3 Configured Functional Units

Every CFU is associated with a configuration register. Hence CFU related data is stored within the ConfigReg object itself (Program 8). The *cfu_fn* is an external function that is linked with the simulator code. It is a pointer to the function which implements the semantics of the CFU operations. The *exec* member function of ConfigReg invokes the *cfu_fn* with the appropriate opcode. The read (write) latencies of CFU operands are saved in *lat.in.opnds* (*lat.out.opnds*) fields of the ConfigReg data structure.

7.3.1.4 Array Register File

ARF functionality in the simulator is identical to that of the conventional register files of the EPIC core and have identical representations as the conventional register files. Hence, we do not discuss it any further here. Current version of AEPIC processor's ARF does not include register FIFOs.

7.3.1.5 Configuration Cache Hierarchy

The configuration cache hierarchy consists of the *C-cache* followed by the *C1 cache* terminating in the *external memory*. All accesses to external memory are assumed to “hit”. The *C1 cache* model is identical to that of L1 cache of the regular data cache hierarchy with a few simplifications. Since the configurations are assumed to be immutable, there is never any need to write data back to external memory from the C1 cache. Hence there is no need for a data eviction (from C1 cache) policy. The simulator does not model the contents of the caches. Since we are only concerned with the “hit/miss” data, only the addresses should be cached. The contents themselves are stored in the simulator's own address space. The address cache is to be maintained as a hash map just like

that of the regular data cache. The hash map is updated with the addresses currently cached due to changes in cache contents caused by memory access operations.

7.3.2 Simulation Of Instruction Execution

Instructions processed by the AEPIC processor fall into three groups:

1. **EPIC core ISA.** These are the instructions of the HPL-PD ISA that form the EPIC core of the AEPIC machine. Section 7.2 describes how these instructions are handled in the simulator.
2. **Adaptive extension instructions** This is the set of instructions added to the EPIC core that handle reconfiguration of the adaptive component of the data-path and initiation of computation on CFUs. These instructions are listed in Table 7.1.
3. **CFU operations.** These are the operations performed by the CFUs. They are not part of the AEPIC ISA. However they need to be simulated since they affect the processor state.

We first discuss instruction processing semantics that are common to all these three categories of instructions. Then we discuss the specifics of each type separately.

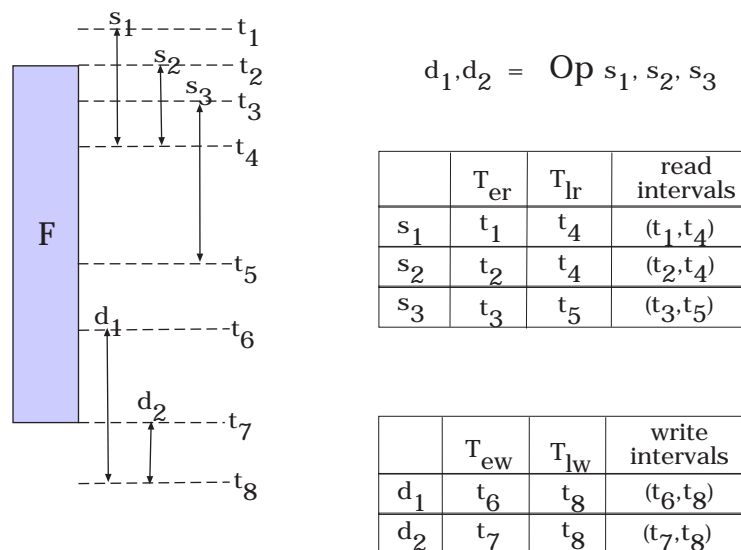


Figure 7.2: Read/write intervals of operands

7.3.2.1 Instruction Processing Semantics

Let us consider the execution of an instruction Op on an AEPIC functional unit F . For correct operation, F is expected to sample the source registers (for input operands) within their associated *read* intervals and is expected to deposit the results of the computation into the output destination registers within the *write* intervals for the corresponding output operands (see Figure 7.2).

Two possibilities for instruction processing styles arise within the above model. If the read and write events for the operands always take place at single precise instants in time relative to the issue time of the operation, then the instruction processing model is referred to as **Equals (EQ)** model. In this case, $T_{er}^o = T_{lr}^o$ for all input operands and $T_{ew}^o = T_{lw}^o$ for all output operands where, o is the operand. If the processor allows instructions to sample the inputs or write their outputs at any point in time within their allowed intervals then it is referred to as a **Less-than Equals (LEQ)** model of execution. Here, $T_{er}^o < T_{lr}^o$ for all input operands and $T_{ew}^o < T_{lw}^o$ for all output operands. A discussion of the merits and demerits of EQ and LEQ models is presented in [140].

We restrict our attention to the case where the operations obey NUAL semantics with EQ model of operation. In addition, we assume that $\underset{o \in src_operands}{MAX} \{T_r^o\} < \underset{o \in dest_operands}{MIN} \{T_w^o\}$ where, T_r^o (T_w^o) is the read (write) time for the input (output) operand o . In other words, each operation samples all of its inputs before writing any of its results—a strict separation between the set of read events and the set of write events on the virtual time-line. This assumption simplifies the construction of the simulator since the simulator itself can perform (simulation of) operation execution *instantaneously* (for example, as a call to a function that implements the semantics), once all the required inputs have been sampled.

Here we summarize operation processing on an AEPIC machine in three steps:

1. **Schedule read actions.** Each of the *read* actions reads the input operand from the associated processor register and saves it into a specific local variable of the simulator (in case of CFU operations, it is the local variable field in the associated ConfigReg: *tmp.in.reg*). This read action is scheduled to be performed at time $(T + L_{kr})$ where, L_{kr} is the input read latency of the k^{th} input operand and T is the issue time of the operation (on the virtual time-line of execution).
2. **Schedule execute action.** An *execute* action implements the semantics of the operation. However, it stores the results of the computation into local variables of the simulator (the *tmp.out.reg* field of ConfigReg for CFU operations). These results are committed to the processor state by the write actions. The execute action is performed “instantaneously”. This action may be performed at any time T such that $T \geq \underset{k}{MAX} \{T_{kr}\}$ where T_{kr} is the read time for the k^{th} input operand and $T \leq \underset{k}{MIN} \{T_{kw}\}$ where T_{kw} is the write time of the k^{th} output operand.
3. **Schedule write actions.** The *write* action writes the computed value of the output operand (which was saved in the simulator local variable associated with that output operand, by the execute action) to the processor registers associated with those output operands. The write action for the k^{th} output operand is scheduled to be performed at time $T + L_{kw}$, where, L_{kw} is the output write latency of the k^{th} output operand and T is the issue time of the operation (on the virtual time-line of execution).

Figure 7.3 illustrates AEPIC instruction execution by the simulator. Operation Op takes s_1, s_2, s_3 as input source operands and d_1, d_2 as the destination operands. All the input read latencies equal 0 ($L_{kr} = 0$). The output latencies are $L_{1w} = 3$ and $L_{2w} = 5$ cycles. If Op was issued at time T , all the input read actions are scheduled to be performed at time T while the write actions are scheduled to be performed at time instants $T + 3$ and $T + 5$ respectively. The execute action may be performed at T' where $T \leq T' \leq (T + 3)$. Note that all the actions to be performed at a particular time instant are added to the *action list* and performed sequentially. Clearly, the read actions should be ahead of the execute action on the *action list* if the execute action were scheduled to be performed at time T .

7.3.2.2 MultiOp Execution Specifics

Having discussed the semantics of single instruction processing, let us look at the MultiOp execution. Two issues are under consideration here: (1) In what relative order should the operations of the MultiOp be processed?

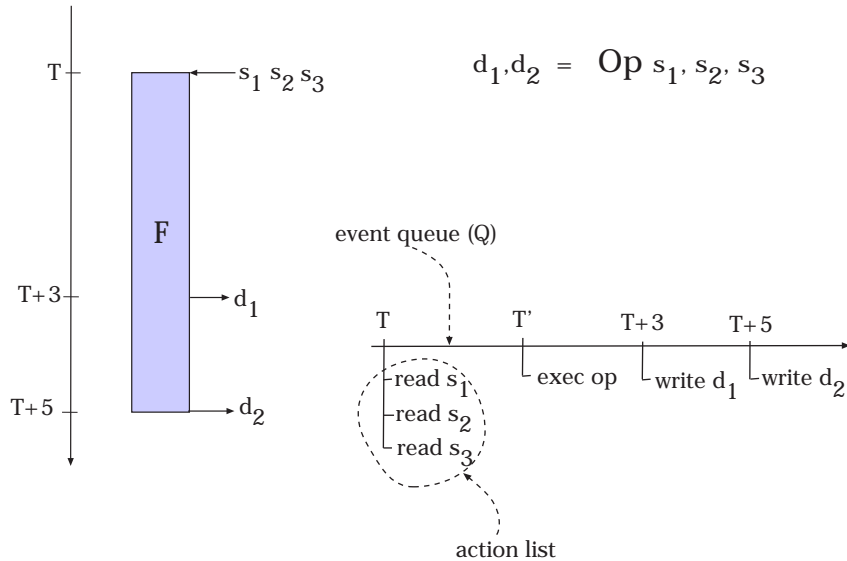


Figure 7.3: AEPIC instruction execution

(2) How are interruptions handled?

To recall, MultiOp is a mechanism to specify multiple operations that *can* be issued simultaneously along with information about which functional unit on which each of the operations is to be issued. This model permits two variations: the MultiOp-P and MultiOp-S [140]. In the former, all the operations are to be issued simultaneously. In the case of MultiOp-S, operations of the MultiOp may be issued sequentially. MultiOp-P permits operations that are anti-dependent, to be packed in the same MultiOp. However, in the case of MultiOp-S, bi-directional dependencies between any two operations of the MultiOp-S are not permitted. Since AEPIC compiler is based on Trimaran which currently supports MultiOp-S, we restrict to AEPIC machines that support MultiOp-S semantics for MultiOp processing.

Other issues. Memory operations within the MultiOp are processed in left-to-right order even if there are flow-dependencies among them. Just as in the HPL-PD EPIC architecture, multiple operations may simultaneously write to the same register. The architecture specifies that the final value in the register is defined when all the write operations deposit the same value. Otherwise the end result is undefined. Branch operations take effect after exactly k cycles after the issue time where k is the branch operation latency. Multiple branch operations may be scheduled in the same MultiOp. However, only one branch operation is permitted to be “taken”. This feature is useful in scheduling multiple branch operations when the compiler can be sure that only one of the branch operations will be taken (such as is the case in the C language *switch* statement).

7.3.2.3 Adaptive Extension Instructions

Adaptive extension instructions are listed in Table 7.1. For instruction formats and their semantics, see Appendix B.

Single-instance and multi-instance operations. The simulator handles most of the adaptive extension instructions the same way the EPIC instructions are handled. However, some of the instructions such as *ldcc[ns]*, *stcc[ns]*, *inc[ns]*, *outc[ns]*, require special handling. These instructions are actually multiple operations packed into one instruction and are referred to as **multi-instance operations**. For example, *ldcc* instruction (format: *ldcc cr, L*) loads data related to configuration whose address is stored in the configuration register *cr* into the C-cache. However, due to bandwidth restrictions, all of the configuration data may not be transferred to C-cache in one cycle. The complete data transfer is performed through a sequence of loads issued by the processor. Although the program code issued only one *ldcc* operation, the processor may convert this operation into several loads depending on the size of the configuration and the amount of data transferred by each load instruction. The multi-instance operations are: *ldcc*, *ldccns*, *stcc*, *stccns*, *inc*, *incns*, *outc* and *outcns*. Rest of the operations are referred to as **single-instance operations**.

Simulating multi-instance operations. Unlike single-instance operations, multi-instance operations are held in the execution pipeline and processed repeatedly until the compiler specified latency expires. All multi-instance operations of the MultiOp are appended to a special queue called the MultiInstanceFIFO. Instructions in the MultiInstanceFIFO are processed in FIFO order.

Table 7.1: Adaptive extension instructions

Instruction category	Instructions
CFU control	exec, susp, resume, abort, reset, step, stat
Operand assignment	inpr, outpr, inp, outp, predp, statp
Resource allocation	malloc, calloc, gcc, gcall, free
Memory access	ldcc, ldccns, stcc, stccns
Context allocation	setctx, clctx, clrallctx, switchctx, pushctx, popctx
Data-path reconfiguration	inc, incns, outc, outcns, delc

Stalling and non-stalling operations. Instructions whose names end in **ns** are referred to as **non-stalling** instructions otherwise they are **stalling**. This classification applies to only the multi-instance instructions. In the case of non-stalling instructions, the processor re-issues the multi-instance operation (perhaps with different operands) as many times as can be accommodated within the compiler specified latency. Consider the *ldccns* instruction with the actual instance of *ldccns* that was issued as *ldccns cr, L*. If the configuration data is not completely loaded within the compiler specified latency *L*, the *ldccns* instruction is abandoned. However, if the compiler chose to use a stalling version of the load operation (*ldcc cr, L*), the processor would be stalled if the complete data has not been loaded within *L* cycles. The rest of the unfinished load is performed during the stall period.

It is expected that the compiler will issue *ldcc* operation when it knows that the subsequent operations cannot be issued unless the full configuration is loaded into the processor. On the other hand, the compiler may issue *ldccns* operations to fetch pieces of the configuration speculatively without having to stall the machine when it knows that the fully loaded configuration is not immediately required.

Non-stalling multi-instance operations are removed from the MultiInstanceFIFO once the compiler specified latency expires. Whereas a stalling multi-instance operation is first completed (even if it means that the processor needs to be stalled) and only then it is removed from the MultiInstanceFIFO. Note: there is a minor issue of how to handle the unfinished operation of a multi-issue operation that is outstanding when the assumed latency expires. Since it does not matter whether the outstanding operation is finished or not, we do not think it would

matter which policy is adopted for this case (whether to stall the machine until the pending operation is finished or to abort the operation). We leave it for future investigation.

7.3.2.4 CFU Operation Processing

Unlike the operations that execute on the hardwired functional units of the AEPIC data-path, execution of the CFU operations can be controlled i.e., their execution can be suspended and resumed or even aborted on any given machine cycle. This additional controllability means that the times at which the input(output) operands are read(written) occur, may vary depending on whether or not the CFU has been interrupted during the operation execution. Hence, each of the CFUs maintains its own *event_queue*. The read/write/execute events of the CFU operation are first scheduled on the CFU local *event_queue*. The CFU local time (*clock*) is initialized to the global issue time of the operation and is updated depending on the status of the CFU (whether it is suspended or in execution). Events on the CFU *event_queue* at the current local clock value are processed as if they are scheduled at that instant on the global event queue.

7.3.3 AEPIC Simulation Framework

7.3.3.1 Overall Architecture

Design of the AEPIC simulation framework is explained here. The optimized, scheduled and allocated AEPIC code from the compiler is *assembled* by the AEPIC simulator generator. The AEPIC simulator “pseudo-executable” which is actually AEPIC assembly in C similar in structure to the EPIC simulator code is compiled by a native compiler (such as *gcc*) and linked with the *Configuration Library*, *AEPIC Simulation Library* and the *Data Cache Library* and *Configuration Cache Library* to generate the equivalent host machine code—the final application specific simulator executable.

7.3.3.2 The Simulator

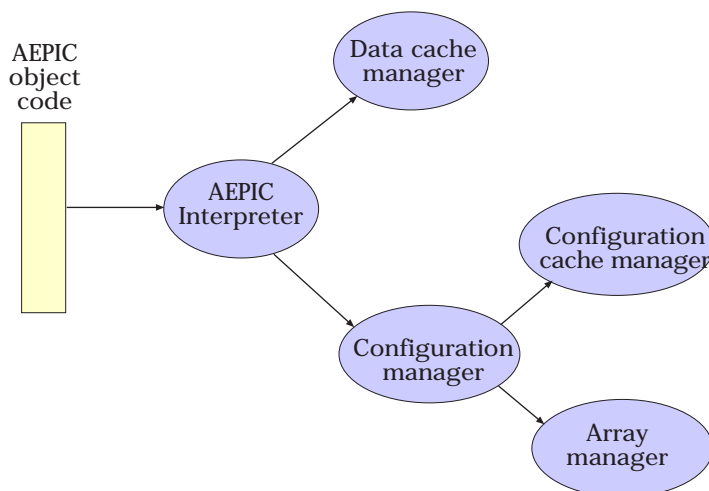


Figure 7.4: AEPIC simulator components

The AEPIC simulator is composed of five key modules (Figure 7.4). Each of the modules processes a different subset of the AEPIC ISA. The *AEPIC Interpreter* reads in the AEPIC code and processes instructions

Algorithm 8 AEPIC simulator algorithm

```
procedure AE_simulate(code_table)
{ MultiOp m;
  /* 1. process the multi-ops in the code_table (AEPIC code for a function) */
  while (m = code_table.FetchMultiOp(gPC)) {
    /* 2. process current multiop */
    OpsList ops = m.OpsInMultiOp();
    for (op ∈ ops) {
      if (isMultiInstanceOp(op)) {
        gMIopsFIFO.AppendOp(op);
      } else {
        ScheduleEvents(Qg, op);
      }
    }
  }
  /* 3. process ops in the multi-instance ops FIFO that are ready */
  OpsList ops = gMIopsFIFO.ReadyOps();
  for (op ∈ ops) {
    ScheduleEvents(Qg, op);
  }
  /* 4. commit ready events on the global event queue */
  EventList elist = Qg.GetEvents(gClock);
  for (e ∈ elist) {
    e.ProcessEvent();
  }
  ...
}
```

Algorithm 9 AEPIC simulator algorithm(contd.)

```
...
/* 5. commit ready events on the CFU event queues that are ready */
for (c ∈ CFU_List) {
  elist = c.ReadyEvents();
  for (e ∈ elist) {
    e.ProcessEvent();
  }
}
/* 6. update global as well as CFU local clocks */
gClock ++;
for (c ∈ CFU_List) {
  if (c.isActive()) {
    c.UpdateClock();
  }
}
}
```

in sequence as dictated by the compiler generated POE. Non-memory related AEPIC instructions which are from the EPIC subset of the AEPIC ISA are processed by the interpreter itself. Non-configuration data related memory operations are processed by the *Data_Cache_Manager*. Adaptive extension instructions are processed by the *Configuration_Manager*. Among these instructions, those that deal with the configuration cache hierarchy are processed by the *Configuration_Cache_Manager* while those that deal with array (MRLA) reconfiguration are processed by the *Array_Manager* and the rest by the *Configuration_Manager* itself.

The simulator algorithm is shown in Algorithm 8. The two key data-structures are (1) the global event queue: Q_g , which contains events ordered by virtual time and, (2) the queue which holds the multi-instance operations to be processed in FIFO order: *gMIopsFIFO*. The algorithm describes operation on the code generated for a single source function. The code is passed in the parameter *code_table*. The *while* loop fetches the MultiOps from the code in sequence, based on the program counter *gPC*. Each operation in the MultiOp is either a single-instance operation or a multi-instance operation. Processing a single-instance operation is equivalent to scheduling their *read*, *execute* and *write* events on to Q_g . Multi-instance operations are queued onto the *gMIopsFIFO* to be processed in FIFO order. In step 3, all the multi-instance operations that can be issued in the current cycle are also processed—meaning, their events are also scheduled onto the Q_g . Note that a multi-instance operation is not removed from the queue until either the time allowed for it expires (its assumed latency) if it is a non-stalling operation or until its operation is completed if it is a stalling operation. In steps 4 and 5, all the ready events are processed. Ready events are from either the global event queue Q_g or are from the CFU local event queues. Events scheduled on Q_g for the current virtual time (*gClock*) are considered “ready”. Similarly, events on CFU local event queues for the times referred to by the CFU local clocks are considered ready. Step 6 updates the global clock as well as CFU local clocks (only if the CFU is “Active”, meaning, it is not suspended and has a operation in process on its pipeline).

7.4 Summary

In this chapter, AEPIC simulator design based on Trimaran’s EPIC simulator is presented. Key simulation issues regarding AEPIC instruction processing are discussed and the various data-structures for representing AEPIC machine state are presented.

Chapter 8

Performance Evaluation

8.1 Introduction

In this chapter, we present a summary of the initial results of our AEPIC research. We start with a description of the applications used for benchmarking purposes in Section 8.2. This is followed by a our research methodology where we explain the compiler infrastructure the machine configuration for our experiments and the type of experiments we conducted in Section 8.3. Results of our experiments are presented in Section 8.4.

The purpose of these experiments is to determine if compilers can efficiently target various common applications from various application domains to AEPIC processors and yield improved performance compared to the performance on equivalent traditional processors. Some of the questions we would like to answer are as follows.

- Are there regions in application code that if mapped to application specific instructions can potentially improve application performance?
- How many such candidates are “good” (those that can be mapped to the MRLA).
- How many such good candidates can be identified by our compilation techniques?
- How many of these candidates are eventually profitable?
- Can the compiler map them to MRLA efficiently?
- What is the effect of MRLA structure on CFU performance?
- How effective are various AEPIC architectural features?
- What are good choices for AEPIC architectural parameters?
- Which application domains are most suited for AEPIC style architecture?

In Section 8.2, we list the set of applications that were used for our experiments. We also discuss their computational characteristics. The Trimaran [70] based compilation environment used for our experiments is described in Section 8.3.1. In Section 8.3.2, we describe the targeted machine configurations, listing the values used for the relevant architectural parameters. The methodology and results of experiments are presented in Section 8.3.

8.2 Application Domain

We drew applications from various domains. Some of these applications are from the SPEC benchmark suite [161], several are from the MediaBench [93] benchmark set and the rest were collected from multiple sources. Some of these remaining applications were obtained from the Trimaran distribution [70]. The list of applications along with a brief description for each, is given in Table 8.1.

Table 8.1: Applications used for performance evaluation

Benchmark applications and their descriptions
132.ijpeg: Integer intensive compression/decompression of image files.
023.eqntott: Translates a logical representation of a boolean equation to a truth table.
026.compress: Reduces the size of input files by using Lempel-Ziv coding. Compress is a pointer-intensive lossless compression scheme. Input to the application is a text file of size 50KB which gets compressed to about 20KB.
052.alvinn: Trains a neural network using back propagation. The program operates on single precision numbers.
G721: ANSI-C language reference implementations of the CCITT G.721 voice compression scheme [26] (Source released by Sun Microsystems, Inc. to the public domain.)
MPEG2: Moving Picture Experts Group (MPEG) [43] is a family of standards used for coding audio-visual information (e.g., movies, video, music) in a digital compressed format. The version of the compression standard used in this implementation is MPEG-2. The implementation used is distributed as part of the MediaBench suite of benchmarks [93].
NBRADAR: Narrow Band Tracking Radar (NBRADAR) [40] identifies targets from a sequence of synthetic radar images. Program receives a $c * r * d$ sized complex number array ($c=4, r=512, d=10$) every 5ms. Program outputs a $d * w$ sized 0/1 array (1's indicate potential targets, $d=10, w=40$). The application computes on random image data generated for 50ms (10 data cubes). It is an FFT-intensive computation. Contains many non-perfectly nested loops (with constant bounds).
(I)DCT: (Inverse) Discrete Cosine Transform ((I)DCT) is an 8×8 matrix transformation operating on fixed-point/floating point values. It is a heavily used kernel in audio/image/video compression codes like JPEG/MPEG/H.261. The application source is from the Software Simulation Group's (SSG) implementation of MPEG2 video signal codec. This implementation is a floating point intensive code. Note that there are other implementations of more efficient IDCT's (which operate no fixed point values). http://www.mpeg.org/MPEG/MSSG/#source
ADPCM: Adaptive Differential Pulse Code Modulation (ADPCM) [26] is a speech compression and decompression algorithm. It takes 16-bit linear PCM samples and converts them to 4-bit samples, yielding a compression rate of 4:1. The ADPCM code used is the Intel/DVI ADPCM code which is being recommended by the IMA Digital Audio Technical Working Group. ADPCM codec implementation consists of two separate benchmarks, the coder and the decoder: rawcaudio and rawdaudio. This is part of the MediaBench [93].
EPIC: Efficient Pyramid Image Coder (EPIC) [152] is an image data compression utility written in C. The compression algorithms are based on wavelet decomposition and run-length / Huffman entropy coder. The filters have been designed to allow extremely fast decoding on non-floating point hardware, at the expense of slower encoding and a slight degradation in compression quality (as compared to a good orthogonal wavelet decomposition). This is part of the MediaBench [93].

ZLIB: Zlib [50] is a lossless data-compression library. Unlike the LZW compression method used in Unix compress and in the GIF image format, the compression method currently used in zlib essentially never expands the data. Zlib's memory footprint is also independent of the input data.
IDEA: International Data Encryption Algorithm (IDEA) [144] operates on 128 bit key for encrypting a stream of data. It is a pointer-intensive integer code. The sample application encrypts and decrypts an array of 8 byte data.
DES: Data Encryption Standard (DES) [79] is an algorithm designed to encipher and decipher blocks of data consisting of 64 bits under control of a 64-bit key.
RC2: RC2 [132] is a conventional secret-key block encryption algorithm from RSA labs. The input and output block sizes are 64 bits and the key length varies from 1 to 128 bytes. The algorithm consists of three steps. (1) Key expansion. This takes a (variable-length) input key and produces an expanded key consisting of 64 16-bit values $K[0], \dots, K[63]$. (2) Encryption. This takes a 64-bit input quantity and encrypts it in place. (3) Decryption. This is an inverse of the encryption step.
A5: A5 [182] is a stream cipher used to encrypt GSM (Group Special Mobile) data. The implementation is from Bruce Schneier's Applied Cryptography [143].
CORDIC: Co-Ordinate Rotation Digital Computer (CORDIC) is an iterative algorithm to compute two-dimensional vector rotation. It was first developed by Volder [172]. CORDIC techniques are based on arithmetic shifts and adds and hence are good candidates for mapping to programmable logic. CORDIC techniques can be used to compute trigonometric, hyperbolic, exponential functions, natural logarithm, square root, Givens rotation and several other "hard" to compute functions. The benchmark code reads in a list of initial vector positions and rotation angles and outputs a list of rotated vectors. All values are represented as 16-bit fixed point values.
POLYPHASE: The polyphase filter bank is a multirate filter, which decomposes a particular frequency spectrum into sub-bands, which can later on be used for a variety of signal processing tasks.
WC: UNIX word-count utility
DAG: Simple if-then-else control structure in a for loop.
EIGHT: Similar to DAG but with labels and goto-s.
IFTHEN: Very similar to DAG.
HYPER: Simple one sided if-then with continue in a for loop.
FIB: Computes Nth Fibonacci number.
STRCPY: Initializes and copies a 1K character array.
NESTED: Simple nested loop used to test software pipelining.
MM: Initializes two static float matrices of a given size, multiplies them, then sums up the elements of the final matrix. Tests loop-based code and modulo scheduling.
BMM: Uses statically declared matrices of double values in above but with a blocked-multiply algorithm.
WAVE: Simple 2D wavefront calculation. Initializes the left column and the upper row of an array, then computes the remaining elements as the sum of left and upper neighbors, and finally sum all the elements together.
SQRT: Uses Newton-Raphson method to compute the square root of a number saving all partial results in an array.

<p>PARAFFINS: Computes the number of paraffin isomers with a given number of carbon atoms. Tests pointer manipulation, structures and complex control structures over multiple procedures. Perhaps the shortest non-trivial test case.</p>
<p>FIR: Finite Impulse Response. Implements a 32-tap FIR filter on 128 floating point data values. Both the number of taps and the number of data items are compile-time constants.</p>

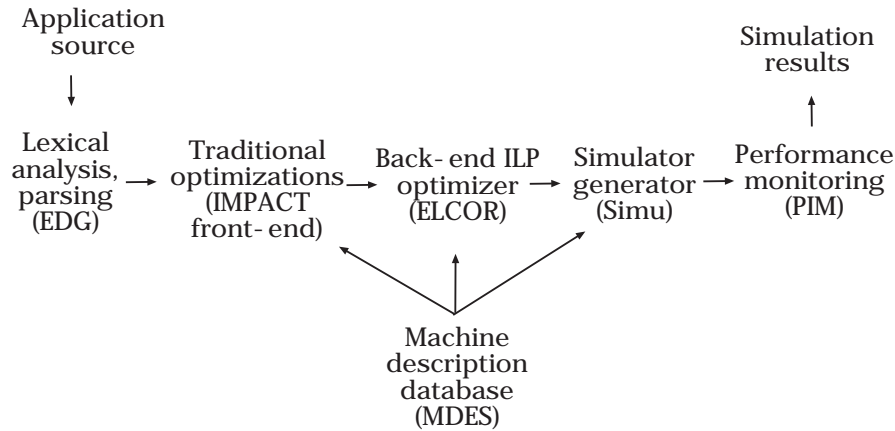


Figure 8.1: Components of compiler infrastructure

8.3 Methodology

We describe the compilation environment used for performing the experiments in Section 8.3.1 and the machine configurations targeted in Section 8.3.2. In Section 8.3.3, details of the experiments are described.

8.3.1 Compilation Environment

The compiler framework targeting the AEPIC class of processors, currently under development, is based on the Trimaran [70] ILP research compiler targeting the HPL-PD [70, 84] EPIC architecture.

The infrastructure used for experiments is shown in Figure 8.1. The major components are as follows. The EDG front-end which performs lexical analysis and parsing on the source program (written in C). The Impact front-end performs several traditional high-level optimizations, basic ILP optimizations and region formation. The Elcor module performs ILP optimizations, primarily, register allocation and scheduling. Simulator assembles the code generated by Elcor module and instruments it for execution profile collection. The Performance Monitoring framework collects traces generated during the application simulation and also reads in the execution profile generated by the simulator and re-instruments it into the IR used by Elcor (called REBEL). The compilation flow is illustrated in Figure 8.2. The major compilation paths that were used for the experiments are 1-2-3-6, 1-2-4-3-6, 1-2-5-3-6. Except for the EDG front-end parsing (first step in 1), 1,2,3,4,5 comprise the Impact front-end. The various region types are created towards the end of 1 and in 2, 4 and 5 (basic-blocks, super-blocks and

hyper-blocks).

The compiler uses several types of Intermediate Representations (IR). The type of IR used depends on the stage of the compilation pipeline. Earlier stages use the **Pcode** and **Hcode** formats [177]. These representations are closer to the source code representation and are used to perform source level transformations. Some of the partitioning techniques are applicable during these stages. Subsequent phases use the **Lcode** [29] and **REBEL** [70] (mostly in the step 6 of Figure 8.2) intermediate representations. Most of the back-end phases exclusively operate on the code represented in REBEL, the low-level machine code equivalent representation. REBEL is based on a graph-based intermediate language. It is an extensible IR that allows any number of attributes to be associated with nodes in the graph for use by various modules of the compiler. For example, the IR can support information about a program statement's resource needs, including attributes such as available device area, that are useful in making decisions regarding configuration selection.

The compiler supports execution profile generation and re-instrumentation at each of these intermediate representation levels. This activity is performed as follows. Desired nodes of the intermediate code are instrumented for profile gathering (for e.g., at the basic block level, each basic block is associated with a variable that keeps track of the number of times the basic block is entered during runtime.)

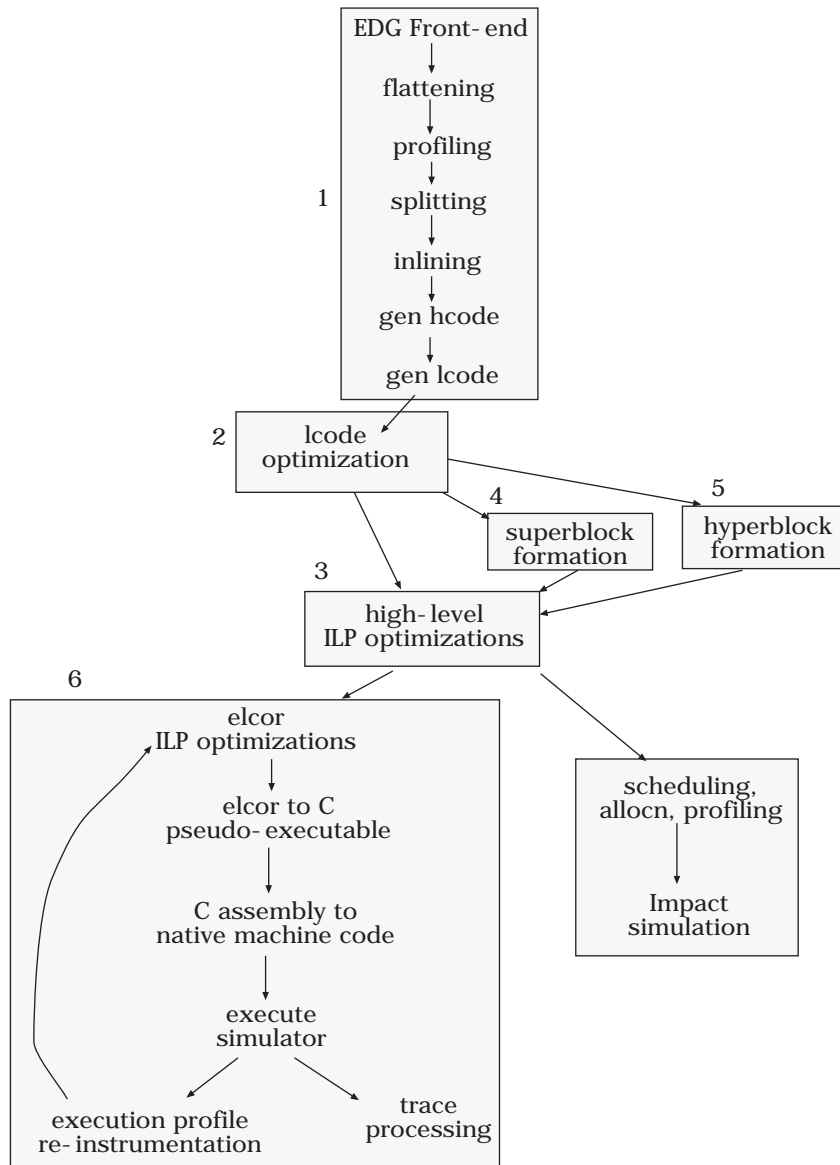


Figure 8.2: Compilation steps

8.3.2 Machine Configurations

We chose three main machine types for our experiments. These are (1) an EPIC architecture $EPIC_{4221}$, (2) an AEPIC machine with realistic resource bounds whose MRLA is based on the Chimaera Reconfigurable Functional Unit [63] which we refer to as $AEPIC_{2221-2C}$ and, (3) an AEPIC machine with unbounded Xilinx FPGA [181] style programmable logic resources for the MRLA referred to as $AEPIC_{\infty}$. Together, these machines represent current technology ($EPIC_{4221}$), what our research suggests ($AEPIC_{2221-2C}$) and the outer reaches of the potential benefit of our research direction ($AEPIC_{\infty}$) respectively. The parameters for the three machine configurations are shown in Table 8.2. We assume that the programmable logic arrays can be clocked at the same rate as the processor core. Note that this does not imply that the configured functional unit itself can be clocked at that rate. The rate at which a configured functional unit (on MRLA) can be clocked is given by the critical path of the design on the programmable logic that implements the CFU. Since none of these processors have been manufactured, we do not comment how they compare in terms of their hardware resource requirements.

Table 8.2: Machine configurations

$EPIC_{4221}$	
ISA	HPL-PD [84]
Issue width	9 instructions per cycle
Functional units	4 integer, 2 floating point, 2 memory and 1 branch units
Registers	128 general purpose registers, 128 floating point registers, 320 predicate registers, 16 branch target registers. Integer and floating point registers are equally split between static and rotating registers. Of the 320 predicate registers, 64 are rotating registers.
Latencies	Most integer arithmetic operations: 1 cycle, integer/floating-point multiply 3 cycles, integer/floating-point divide 8 cycles, L1 cache hit 2 cycles, L2 hit 7 cycles and 35 cycles external memory latency. All branch latencies are 1 cycle.
Caches	We do not model caches (all memory accesses are assumed to “hit”).

$AEPIC_{2221-2C}$	
ISA	AEPIC ISA (Appendix C) which is based on the HPL-PD ISA [84].
State	128 general purpose registers, 128 floating point registers, 320 predicate registers, 16 branch target registers. Integer and floating point registers are equally split between static and rotating registers. Of the 320 predicate registers, 64 are rotating registers. 64 configuration registers (at most 64 CFUs can be resident at one time on the MRLA). 128 array registers. The MRLA is composed of 2 Chimaera Reconfigurable Functional Units (C-RFUs) [63]. Each C-RFU is a programmable logic array composed of 32 rows of 32 programmable logic cells per row. Each row can be configured to implement most integer arithmetic operations and requires 1674 bits of configuration data to configure it. Further details of the C-RFU are in [63].
Issue width	9 instructions per cycle. 2 instructions execute on the AEPIC units.

Functional units	2 integer, 2 floating point, 2 memory, 1 branch and 2 aepic units. Each aepic unit executes instructions from the adaptive extension of the AEPIC ISA Appendix B. In addition, functional units can be configured onto the MRLA (chimaera RFU array). However, only 2 CFUs can be simultaneously active (one on each RFU of the MRLA).
Latencies	Most integer arithmetic operations: 1 cycle, integer/floating-point multiply 3 cycles, integer/floating-point divide 8 cycles, L1 cache hit 2 cycles, L2 hit 7 cycles and 35 cycles external memory latency. All branch latencies are 1 cycle. Latencies to load/store values into ARF same as latencies to load/store values into GPR. Configuration load/store latencies are proportional to the size of the configuration.
Caches	We do not model caches (all memory accesses are assumed to “hit”).

<i>AEPIC_∞</i>	
ISA	AEPIC ISA (Appendix C) which is based on the HPL-PD ISA [84].
State	128 general purpose registers, 128 floating point registers, 320 predicate registers, 16 branch target registers. Integer and floating point registers are equally split between static and rotating registers. Of the 320 predicate registers, 64 are rotating registers. 64 configuration registers (at most 64 CFUs can be resident at one time on the MRLA). The number of array and configuration registers is assumed to be infinite as well as the sizes of the MRLA and configuration caches. The MRLA is based on Xilinx XC4000 FPGA.
Functional units	2 integer, 2 floating point, 2 memory, 1 branch and 2 aepic units. Each aepic unit executes instructions from the adaptive extension of the AEPIC ISA Appendix B. In addition, functional units can be configured onto the MRLA. Any number of CFUs can be simultaneously active. However, at most two CFU operations can be issued one each cycle.
Latencies	Most integer arithmetic operations: 1 cycle, integer/floating-point multiply 3 cycles, integer/floating-point divide 8 cycles, L1 cache hit 2 cycles, L2 hit 7 cycles and 35 cycles external memory latency. All branch latencies are 1 cycle. Latencies to load/store values into ARF same as latencies to load/store values into GPR. Configuration load/store latencies are proportional to the size of the configuration.
Caches	We do not model caches (all memory accesses are assumed to “hit”).

8.3.3 Experiments

Here, we describe some of the experiments performed.

Experiment 1. Computational characteristics of applications. The purpose of this experiment is to determine static and dynamic opcode distributions, available parallelism in applications, performance hot-spots and other basic information related to the chosen benchmark applications. This information will be used to categorize applications, and also give us cues on what the architectural parameters for the AEPIC machine should be. It is also expected to highlight some of the problems with the fixed ISA architectures. For this experiment, we chose *EPIC₄₂₂₁* as the target machine.

Experiment 2. Having identified limitations of traditional compilation targeting fixed ISA machine ($EPIC_{4221}$), we would like to determine what the potential benefit of a programmable logic based dynamic ISA machine such as AEPIC would be. In order to explore the limits of the performance potential of AEPIC machines, we consider an AEPIC machine with unlimited resources ($AEPIC_{\infty}$).

Experiment 3. Here we consider performance on machines with realistic configurations and practical compilation techniques. The machine configuration used is the $AEPIC_{2221-2C}$ Chimaera-RFU based AEPIC machine. Although the ultimate goal is to see if performance gains as expected can be achieved, this experiment is also intended to give us an insight into the type of application computations that may be targets for customized processing on the MRLA and the benefits of the various features of AEPIC architectures.

8.4 Results And Discussion

8.4.1 Application Characteristics

Table 8.3: Application sizes

Application	#lines	Application	#lines	Application	#lines
008.espresso	14850	eight	24	epic	3057
023.eqntott	3466	052.alvinn	272	fib	35
085.cc1	91050	fir	110	g721encode	1591
a5	398	compress	525	g721decode	1593
cjpeg	26865	grep	458	mpeg2decode	9832
bmm	106	djpeg	26102	mpeg2encode	7605
dag	30	mm	48	ifthen	30
cordic	335	des	6668	idea	1232
hyper	20	nbradar	457	paraffins	388
nested	24	polyphase	549	rawaudio	314
rawaudio	314	rc2	1253	wc	255
sqrt	34	zlib	8281		

Table 8.4: Dynamic opcode distribution

Benchmark	%br	%ld	%st	%ia	%fa	%cmpp	%pbr
023.eqntott	11.7	12.6	1.8	39.2	0	15.6	18.9
052.alvinn	8.7	25.3	10.7	28.3	9.3	8.8	8.8
a5	8.1	4.1	5	64.3	0	6.3	10.9
bmm	8.4	16.8	1.4	46.1	7.7	9.6	9.9
cc1-no	7.3	18.6	13.9	40.5	0	7.9	10.9
cjpeg	5.7	18.2	12.5	47.4	0	7.5	8.5
compress	5.8	20.3	17	45	0	5.6	6.1
cordic	2	19.8	17.1	52.3	2.2	1.9	4.4
dag	12.6	0	0.1	54.1	0	12.5	20.6
des	0.3	22.7	18.9	57.3	0	0.3	0.4
djpeg	1.5	21.7	18.3	54.9	0	1.6	2
eight	13.7	0	0.2	48.4	0	15.4	22.3

Table 8.4: Dynamic opcode distribution

Benchmark	%br	%ld	%st	%ia	%fa	%cmpp	%pbr
epic	9.2	9.3	1.3	42.4	15	10.2	12.7
fact2	19	3.8	10.1	55.7	0	2.5	7.6
fib	14.7	2.8	10.1	60.6	0	3.7	7.3
fir	14.9	6.9	7.2	31.2	1.1	14.4	24.2
g721decode	9.9	12.7	7.1	44.1	0	11.5	14.2
g721encode	10.2	12.6	6.9	43.4	0	11.8	14.5
grep	13.2	12.8	8.7	31.5	0	15.9	17.9
hyper	15.2	1	2.2	47.9	0	15.8	18
idea	7.4	9.4	5.2	64.3	0	5.8	7.7
ifthen	12.5	0	0.1	48.3	0	16.8	22.2
mm	8.8	17.2	1.3	45.3	7.8	9.2	10.3
mpeg2dec	5.8	19.9	5.1	47.9	10.6	4.5	6.2
mpeg2enc	13.7	4.1	7	38.5	0.4	15.9	20.2
nbradar	2	16.1	13.6	50.3	12.9	2.4	2.7
nested	9	11.3	9.1	45.3	4.3	7.3	13.6
paraffins	4.5	12.1	22.5	49.7	0	5	6.1
polyphase	3.3	21	15.1	53.6	0	3	4
rawcaudio	14.1	4.8	0.7	34.6	0	17.9	27.9
rawdaudio	13.3	4.2	1.7	36	0	16.7	28.1
rc2	4.8	14.2	8.4	63.6	0	4.3	4.6
sqrt	6.3	11.4	6.5	39.8	22	6.7	7.2
strcpy	10.4	10.5	10.5	47.8	0	10.4	10.4
switch_test	22.4	7.8	3.5	22.7	0	17.5	26
unepic	6.6	15.7	13	43.2	1.7	9	10.8
wave	8.5	15.5	4.2	40.6	13.6	8.8	8.9
wc	15.5	9.9	4.5	19.3	0	24.5	26.3
zlib	6.8	17.9	9.5	47.7	0	8	10.1

8.4.2 Standard ILP Processor Performance

Table 8.5: Performance on 9-issue EPIC processor

Application	IPC	DIC	SIC
023.eqntott	3.67	730034092	25925
hyper	3.27	1256	196
052.alvinn	1.75	3039488026	6698
idea	1.3	14823	3206
a5	2.94	93412	3415
ifthen	2.2	2566	92
bmm	3.46	34651	976
mm	3.36	36905	585
cc1-no	1.57	239924075	270091
mpeg2dec	2.19	4904133874	33840

cjpeg	1.68	21646188	41085
mpeg2enc	2.5	28238	9484
compress	1.44	48612509	7774
nbradar	0.91	451805415	8931
cordic	1.18	14618890	11298
nested	2.46	1498	392
dag	1.88	2966	87
paraffins	2.07	141238	1736
des	1.08	3404516	38622
polyphase	1.29	1242579	4442
djpeg	1.25	10076349	27170
rawcaudio	2.7	4260830	1003
eight	1.42	2748	72
rawdaudio	2.87	3853722	581
epic	2.29	32993344	19977
rc2	1.84	11619	1413
fact2	1.36	58	79
sqrt	1.28	2277	406
fib	2.02	54	105
strepv	2.46	8006	278
fir	2.47	181325	1941
switch_test	1.55	11833	632
g721decode	2.32	186395439	6497
unepic	1.77	9770797	23028
g721encode	2.32	205860897	8229
wave	1.92	11847	511
grep	2.22	159445	1966
wc	2.65	940443	894
zlib	2.06	278235	9421

8.4.3 Hot-spot Distribution

Table 8.6 shows the percentage of code that contribute to a large fraction of application execution time. These percentages are shown in terms of the number of regions (such as basic-blocks/super-blocks or hyper-blocks). It is clear from the data that for a majority of the applications, less than 10% of the regions consume more than 90% of the execution time. In fact, for 16 of these benchmarks, less than 10 regions need be examined if the focus is on the code contributing 90% towards the total execution time.

Table 8.6: Hot-spot distribution

Application	#R	% $R_{90\%}$	# $R_{90\%}$
052.alvinn	90	1.1	1
rc2	72	12.5	9
wave	31	3.2	1
switch_test	65	13.8	9
sqrt	20	5	1

compress	153	7.2	11
strcpy	10	10	1
rawaudio	36	33.3	12
wc	69	2.9	2
grep	175	7.4	13
nbradar	107	2.8	3
polyphase	119	10.9	13
eight	11	27.3	3
rawcaudio	43	37.2	16
hyper	8	37.5	3
djpeg	1899	0.9	17
des	534	0.7	4
epic	514	3.7	19
mm	32	12.5	4
epic	456	4.2	19
dag	12	33.3	4
a5	76	25	19
fact2	9	44.4	4
paraffins	115	21.7	25
nested	14	35.7	5
mpeg2dec	1140	2.4	28
fib	11	45.5	5
023.eqntott	143	20.3	29
bmm	63	9.5	6
unepic	278	10.8	30
ifthen	13	46.2	6
cordic	54	57.4	31
rc2	66	12.1	8
g721encode	203	25.1	51
fir	45	17.8	8
g721decode	195	25.6	51
grep	170	5.3	9
cjpeg	1635	3.2	53
g721decode	218	28.9	64
g721encode	227	31.3	71
mpeg2enc	1853	3.8	72
zlib	641	11.9	76
idea	210	39	82

Table 8.7: Examples of candidates for partitioning in various applications

Application	Characteristics
cjpeg	DCT, lookup in array of constants
rc2	hammocks, logical/arithmetic/shift ops

mpeg2encoder	DCT, hammocks, operations on constant arrays
grep	logical operations, constant array lookups
g721encoder	logical/shift arithmetic, constant array lookups
EPIC	operations on bounded arrays and array of constants
djpeg	IDCT, color transformation kernels
compress	arith/shift operations, operations on bounded arrays
CORDIC	diamonds, arithmetic/shifts, constant operands
wc	constant array bounds, small types, compact if/switch statements
eight	compact if-switch statements, small types
fib	“atoi” kernel, internal registers (avoid register port conflicts)
eqntott	compact if-switch statements, high-overhead branches
adpcm	constant arrays, shift/logical arithmetic operations
fir	constant coefficient multiplications
polyphase	constant coefficient multiplications

Program 11 ADPCM sample code

```

{
  if (sign) valpred -= vpdiff;
  else valpred += vpdiff;

  if (valpred > 32767) valpred = 32767;
  else if (valpred < -32768) valpred = -32768;

  index += indexTable[delta];
  if ( index < 0 ) index = 0;
  if ( index > 88 ) index = 88;
  step = stepsizeTable[index];

  vpdiff = step >> 3;
  if ( delta & 4 ) vpdiff += step;
  if ( delta & 2 ) vpdiff += step>>1;
  if ( delta & 1 ) vpdiff += step>>2;
}

```

8.4.4 Performance on $AEPIC_{\infty}$

Table 8.8 shows the results of targeting five different applications to $EPIC_{4221}$ and $AEPIC_{\infty}$ machines. Columns 2 and 3 give the dynamic instruction counts on the $EPIC_{4221}$ machine for both the standard compilation and when all suitable optimizations are enabled in the Trimaran compiler. Column 4 gives the instruction count for the $AEPIC_{\infty}$ machine and column 5 the speed up on the $AEPIC$ machine with respect to the performance on the $EPIC_{4221}$ machine with optimized compilation. The parameters of the two machines are described in Table 8.2.

In order to target $AEPIC_{\infty}$, the application is first compiled and simulated on $EPIC_{4221}$ and then all the program hot-spots are identified. For each program hot-spot, its FPGA implementation is compared with the performance

Program 12 Pegwit sample code

```
#define R0(v,w,x,y,z,i) z+=((w&(x^y))^y)+blk0(i)+0x5A827999+rol(v,5);w=rol(w,30);
{
    /* Hash a single 512-bit block.
       This is the core of the algorithm. */
    ...
    R0(a,b,c,d,e, 0);
    R0(e,a,b,c,d, 1);
    ...
    R4(b,c,d,e,a,79);
    ...
}
```

on the fixed *EPIC* core. If found beneficial, the hot-spot is mapped to the MRLA (Xilinx 6200 FPGA [181]). The FPGA mapping is performed manually (not by the Trimaran compiler). For example, three such mappings were determined to be beneficial for the MPEG2Decoder application. These correspond to the *IDCT*, *Add_Block* and *Saturate* functions in the application source. Since MRLA is assumed to be infinite, we do not consider the costs of swapping in/out of configurations due to resource constraints. For most of the hot-spots, the synthesizing the mapping is not very complex. However, for the IDCT function, we used the technique described by Sikstrom et.al., in [151]. The FIR is a finite impulse response filter implementation. The filter is composed of 32 taps. The taps are synthesized based on the constant co-efficient multiplier implementations for XC6200 described in [85]. The IDEA encryption is executed on a single block of data. The NBTR application samples inputs 10 times for the run.

It is unreasonable to expect to build a machine with infinite amount of programmable logic. However, we found that the actual resource usage is not impractical to consider in a realistic machine and considering the potential gains in performance, it is certainly worthwhile to explore more realistic machine configurations.

Table 8.8: AEPIC with FPGA array

Application	<i>EPIC</i> ₄₂₂₁	<i>EPIC</i> ₄₂₂₁ (<i>Opt</i>)	<i>AEPIC</i> _∞	<i>AEPICSpeedup</i>
MPEG2Decoder	–	439486198	80686602	5.4
IDEA	118	118	18	6.4
FIR	31533	13491	384	35
NBTR	22529978	13731573	532800	25.8
IDCT	12127	6633	544	12.2

8.4.5 Performance on *AEPIC*_{2221–2C}

Table 8.9 presents results of targeting several applications on to the Chimaera RFU based AEPIC machine. Column 2 gives the number of configured functional units that were found beneficial for each of the applications considered. Column 3 gives the speedup achieved on the given application compared to the performance on *EPIC*₄₂₂₁ processor. The *AEPIC*_{2221–2C} machine configuration is described in Table 8.2. Standard input datasets were used for all of these applications (datasets as supplied by the benchmark set from which these applications were obtained.)

Table 8.9: AEPIC with Chimaera RFUs

Application	#CFUs	Speedup
RAWCAUDIO	12	1.23
RAWDAUDIO	19	1.89
MPEG2ENC	8	1.12
COMPRESS	28	1.13
G721	8	1.08
PEGWIT	23	1.13
CORDIC	26	2.61
MPEG2DEC	72	2.34
DJPEG	53	1.61

The following are the steps taken to target the applications to $AEPIC_{2221-2C}$ machine.

1. *Profile.* All the applications are first targeted to $EPIC_{4221}$. The application is profiled at each intermediate stage of the compilation and the execution profile is re-instrumented into the IR of that stage. So for example, before the Pcode is converted to Hcode, the application is profiled and the execution profile instrumented into the Pcode. At the end of the simulation, all the intermediate stages have execution profile data associated with their IR nodes.
2. *Partition.* The partitioning step is performed in some of the major intermediate forms (Pcode, Hcode and REBEL) to tag candidates that might be suitable for an MRLA implementation. The partitioner first makes an initial traversal of the IR and gathers the hot-spots in sorted order. For each hot-spot, depending on the type of IR, various heuristics are applied to determine the suitability of mapping to MRLA. For example, regions involving memory accesses or floating point operations are discarded. These heuristics are specific to the Chimaera RFU. For example, an if-statement guarding a single assignment statement guarded by a boolean variable or a constant (a common occurrence) can be mapped to a single row of the RFU. RFUs do not hold state and only generate one output even though 9 input operands may be supplied. These constraints further restrict the number of partitions that can be mapped to MRLA.
3. *Code generation.* Once the partitions have been determined, only those partitions that are actually beneficial are selected to be incorporated into the output program for use in the generated application code. A partition is considered beneficial if it can execute faster (produce results in less number of cycles compared to the number of cycles on the EPIC core) and the reconfiguration overhead can be sufficiently masked so that even after considering the reconfiguration time, the eventual performance is improved. Once the partition is determined, the size of the configuration is obtained from a simple heuristic which takes into account which set of instructions can be mapped to a single row of the RFU and then packs the partition onto as many rows as required on the RFU. Each row requires 1674 bits of configuration data. Size of the CFU is simply the product of the number of RFU rows used by the partition with 1674. In order to determine if the reconfiguration overhead can be masked, the IR is traversed in reverse order starting at the first use of the CFU and configuration load instructions are speculatively inserted. Each speculative configuration load will load $(f * B * L_c)/L$ number of bits. Here, f is the number of times the speculated configuration load executes, B the number of bits transferred on each load, L_c the compiler specified latency for this load instruction and L the memory access latency.

8.5 Summary

Our initial experiments indicate that there is potential for achieving substantial improvements in performance on a large class of applications through AEPIC processors and compilation techniques that are not too far removed from known techniques. Clearly, there is scope for further improvements considering that the MRLA we considered is very limited in capability and that the compiler optimizations are still unexplored let alone used to achieve the above results.

Chapter 9

Concluding Remarks

“...embedded computing will introduce a new theme into computing: the automation of computer architecture.”

—B. R. Rau [11]

Given an application program, traditional approaches to improving performance involve one of two approaches—improve the performance either of the microprocessor or of the generated code through better compiler optimizations. Both these approaches are constrained by the fact that they have to conform to a *fixed* interface between the processor and the software that executes on it—the Instruction Set Architecture (ISA). The ISA is designed to be suitable for *all* applications that will be targeted to the microprocessor. A fixed ISA offers many advantages: compatibility, uniformity and simplicity. However, due to the very nature of its generality, one can expect that the instruction mix offered by the ISA need not necessarily be the perfect match for a given application. In addition to the fixed ISA, poor scalability of dynamically scheduled architectures and limited available “traditional” instruction level parallelism has motivated us to look for alternative approaches to improving microprocessor performance on general purpose applications.

On the other hand, several new opportunities have presented themselves. We list some of these here. (1) According to the international technology road map for semiconductors [146], the number of transistors per device will reach 500 million by the year 2008, 1.5 billion by 2011 and cross 4 billion by 2014. This represents an order of magnitude increase in transistor capacity by 2008 compared to year 2000 chip capacities. (2) Advances in programmable logic devices. Due to the increased demand for rapid prototyping of circuit design there has been a tremendous progress both at the hardware level in terms of improving programmable logic designs but also in our understanding of how applications can be mapped beneficially to these devices. (3) There has been a shift in the types of applications that form a major part of current general purpose computational workload. These applications are primarily multi-media oriented. Multi-media processing involves a large amount of fine grained parallel processing on streaming data and involve the use of a small set of frequently used kernels such as discrete cosine, wavelet and Fourier transforms, color conversion routines etc.

The above observations motivated us to take a radically new approach to microprocessor architectures. Traditional approaches to processor architectures are based on the needs of a large class of applications. The advent of programmable logic and advances in semiconductor technology resulted in a vast amount of hardware resources at the disposal of an architect. In this context, one is tempted to ask if one can build processors based on programmable logic in a way that a compiler can customize them for a given application. Performance advantages due to application customization on programmable logic is an established fact now. However, two problems have remained, hindering the widespread adoption of programmable logic based processors.

1. The costs of reconfiguration are extremely high and hence applications requiring rapid reconfiguration may

lose all the advantages of customized processing.

2. It has been notoriously difficult to map an application to programmable logic, especially in *reasonable compilation times*.

There has been very little progress in developing processors based on programmable logic that provide convenient abstractions to the compiler to alleviate the two key problems mentioned above.

In this dissertation, we have proposed a novel classes of microprocessors that allow application programs to add and subtract functional units yielding a dynamically varying instruction set interface to the running application. We focus on a small subset of these *dynamic instruction set architectures* called *Adaptive Explicitly Parallel Instruction Computing (AEPIC)* architectures whose definition represents a collection of ideas intended to enable **efficient reconfiguration of processor data-paths**. While AEPIC processor reconfiguration is affected by the executing program at runtime, the decisions of when and how to reconfigure are determined by the compiler and embedded in the application's executable.

The AEPIC class of architectures is a good candidate for research since it is at the boundary of what is known to be efficiently and automatically compilable class of architectures. We believe that these architectures are at the right level of granularity for automatic compilation (unlike many of the purely FPGA based machines) and yet yield many of the performance benefits of programmable logic. This is evidenced by the similarity between the compilation techniques targeting conventional ILP architectures and the ones we have proposed for AEPIC architectures. Our preliminary results also indicate that these architectures are worthwhile direction to pursue.

Appendix A

AEPIC Architectural Parameters

Table A.1: Architecture parameters

Architecture parameters	
Parameter	Notes
MRLA Parameters	
Number and types of MRLAs	for now AEPIC is restricted to only one MRLA
Number of contexts	depends on the application CFU requirements
Array width (slices per context)	determines the maximum size of CFU that can be instantiated
Context switch latency	for tradeoff between context switching and datapath reconfiguration
Slice load/reset latencies	load latency depends on bandwidth to C-cache; determines reconfiguration time
Context reset latency	1-k cycles
Array reset latency	1-k cycles
Array compaction latency	determines how often garbage collection can be invoked on the array
Allocation/deallocation latencies	has very little impact on reconfiguration time since it can be performed efficiently (1-2 cycles). See Appendix C.
Number of data ports to C-cache	bandwidth to C-cache
C-cache parameters	
Block size	determines the complexity of the allocation/deallocation circuitry
Total blocks	determines total amount of configuration data that can be cached
Allocation/deallocation latencies	has very little impact on reconfiguration time since it can be performed efficiently (1-2 cycles). See Appendix C.
Ports to C1	bandwidth to C1 cache
C1 Cache parameters	
Number of pages	impacts the total configuration data that can be stored
Page size	internal memory fragmentation depends on page size and average configuration size; also impacts the

Data cache (L1 and L2) parameters	
Cache type and sizes	associativity, number of lines, line width, port sizes, number of ports, replacement policy
operation latencies	Load/store hit/miss latencies
Fixed core parameters	
Int,Float,Branch and Memory FUs	depends on available ILP and how much of it is mapped to CFUs
AEPIC extension FUs	determines how many AEPIC extension instructions can be executed in parallel. Note: this is not counting the CFUs which are executing custom instructions and not AEPIC extension ISA
GPRs, FPRs, BTRs and Predicates	
ARF scalar and FIFO registers	
CRF size	total number of configurations in C-cache and MRLA is upper bounded by the number of available configuration registers
Number of ARF ports	bandwidth to CFUs
Operation latencies	latencies of operations from the AEPIC ISA and not the CFU operations

Appendix B

AEPIC Instruction Semantics

Table B.1: Semantics Of Adaptive Extension Instructions

Instruction	Description
Instructions for controlling CFU execution	
<i>exec cr, opid</i>	Trigger the operation <i>opid</i> on CFU associated with <i>cr</i> .
<i>susp cr</i>	Suspend the operation in progress on CFU associated with <i>cr</i> .
<i>resume cr</i>	Resume the operation in progress on CFU associated with <i>cr</i> .
<i>abort cr</i>	Abort the operation in progress on CFU associated with <i>cr</i> .
<i>reset cr</i>	Reset the CFU associated with <i>cr</i> .
<i>step cr</i>	Step the operation in progress on CFU associated with <i>cr</i> for one machine cycle and then suspend the operation until it is resumed or aborted.
Associate locations for source/sink operands of CFUs	
<i>inpr cr, r</i>	Associate register <i>r</i> from the ARF as the k^{th} input operand of the CFU associated with configuration register <i>cr</i> . It is assumed that $k - 1$ <i>inpr</i> operations have been invoked prior to this call to <i>inpr</i> , all of which take the same <i>cr</i> as the first operand. Currently we assume that all operations performed by a CFU have the same input/output format. Hence this specification is CFU opcode independent.
<i>outpr cr, r</i>	Associate register <i>r</i> from the ARF as the output location for the k^{th} operand of the CFU associated with configuration register <i>cr</i> . It is assumed that $k - 1$ <i>outpr</i> operations have been invoked prior to this call to <i>outpr</i> , all of which take the same <i>cr</i> as the first operand. Currently we assume that all operations performed by a CFU have the same input/output format. Hence this specification is CFU opcode independent.
<i>inp cr, r, k</i>	Associate register <i>r</i> from the ARF as the k^{th} input operand of the CFU associated with configuration register <i>cr</i> . Currently we assume that all operations performed by a CFU have the same input/output format. Hence this specification is CFU opcode independent.
<i>outp cr, r, k</i>	Associate register <i>r</i> from the ARF as the output location for the k^{th} operand of the CFU associated with configuration register <i>cr</i> . Currently we assume that all operations performed by a CFU have the same input/output format. Hence this specification is CFU opcode independent.

<i>predp cr, p</i>	Associate predicate register <i>p</i> from the PRF as source for predicate operand of the operations performed on CFU associated with <i>cr</i> . This input will be ignored if a particular operation does not take any predicate inputs.
<i>statp cr, r</i>	Associate register <i>r</i> as the destination for saving status of CFU associated with <i>cr</i> .
MRLA and C-cache resource allocation	
<i>calloc cr, r</i>	allocate adequate number of blocks in the C-cache for the configuration located at memory address stored in <i>r</i> and associate configuration register <i>cr</i> with the configuration. The number of blocks required is obtained from the configuration header stored along with the rest of the configuration data in the process's address space in memory. Blocks in C-cache allocated for the configuration data need not be consecutive.
<i>malloc cr, cid</i>	allocate requisite number of slices on MRLA on context <i>cid</i> for the configuration associated with <i>cr</i> . The number of slices required by the CFU is obtained from the configuration information stored in <i>cr</i> .
<i>gcc cid</i>	Compact context <i>cid</i> of the MRLA. All CFUs are assigned consecutive slices so that all free slices are merged into one free block.
<i>gcall</i>	Compact all contexts of the MRLA. This is equivalent to calling <i>gcc</i> on all the contexts of MRLA.
<i>free cr</i>	Deallocate (free/reclaim) space allocated for configuration referred to by the configuration register <i>cr</i> . Note configuration is in either the C-cache or on the MRLA. If the configuration is in the C-cache, the C-cache blocks allocated to the configuration are freed and added to the C-cache free block vector. If the configuration is on the MRLA, then either of two actions can take place: (a) the configuration is moved to C-cache and the MRLA slices assigned to the configuration are freed or (b) the configuration is deleted and the MRLA slices reclaimed. We adopt the latter semantics for the <i>free</i> instruction when the configuration is on the MRLA.
Load/store configurations from/to memory	
<i>ldcc cr, l</i>	“Stalling till completion” configuration load instruction. Initiate fetch operations for remaining configuration words from memory. The location of the next word to load, the number of words to load and the destination for the loaded word are stored in configuration register <i>cr</i> . The latency assumed by the compiler for this instance of the load operation is <i>l</i> .
<i>ldccns cr, l</i>	Non-stalling configuration data load instruction. Initiate fetch operations for <i>as many as possible</i> configuration words from memory within the allotted <i>l</i> cycles since the operation issue time. The location of the next word to load, the number of words to load and the destination for the loaded word are stored in configuration register <i>cr</i> . Unfinished load is either aborted or completed (by stalling the processor). The choice is implementation dependent.

<i>stcc cr, r, l</i>	Initiate store operations for the remaining configuration words in the C-cache to memory location starting at address given in <i>r</i> . The location of the next word to store, the number of words to store are maintained in configuration register <i>cr</i> . The latency assumed by the compiler for this instance of the store operation is <i>l</i> . If the rest of the configuration is not saved to memory within the assumed latency interval, the processor is stalled until the operation completes. If the configurations are assumed to be immutable, this operation is not of much use.
<i>stccns cr, r, l</i>	Non-stalling version of <i>stcc</i> operation. Initiate store operations for as many configuration words as can be transferred from the C-cache to memory within the stated latency. The location of the next word to store, the number of words to store are maintained in configuration register <i>cr</i> . The starting destination address is given in <i>r</i> . The operation aborts any uncompleted stores after <i>l</i> cycles.
<i>stccns cr, r, l</i>	Non-stalling configuration store operation. Initiate store operations for configuration words in the C-cache to memory. The location of the next word to store, the number of words to store are maintained in configuration register <i>cr</i> . The starting destination address is given in <i>r</i> . The latency assumed by the compiler for this instance of the store operation is <i>l</i> .
Manage execution contexts	
<i>setctx cid</i>	Make execution context corresponding to <i>cid</i> the active context.
<i>clrctx cid</i>	Free all configured functional units allocated on execution context referred to by <i>cid</i> .
<i>clrallctx</i>	Free configured functional units from all contexts of the MRLA.
<i>switchctx cr, cid</i>	Move configured functional unit from its current context to context <i>cid</i> . An exception is raised if context <i>cid</i> has inadequate resources to host the CFU associated with <i>cr</i> . Latency assumption????
<i>pushctx k</i>	Allocate <i>k</i> consecutive execution contexts to the calling thread.
<i>popctx k</i>	De-allocate the top <i>k</i> consecutive execution contexts.
Configure functional units on MRLA	
<i>inc cr, l</i>	Transfer configuration data associated with <i>cr</i> , from C-cache to MRLA. Once the entire configuration data is transferred, the MRLA effectively hosts a new CFU corresponding to that configuration. The latency assumed by the compiler for this instance of the <i>inc</i> operation is <i>l</i> cycles. Configuration data is loaded from C-cache. Memory used by the configuration in C-cache is freed once the configuration is moved to MRLA. The execution context and the specific MRLA slices allotted for the configuration are already specified in the configuration register <i>cr</i> during the <i>malloc</i> call which allotted that space for this configuration. If the <i>inc</i> operation requires more than <i>l</i> cycles, the processor is stalled until the entire configuration data is moved to MRLA. Note: every call to <i>inc</i> instruction should be preceded by a call to <i>malloc</i> to that <i>cr</i> .

<i>incns cr, l</i>	This operation is identical to the <i>inc</i> operation except that the configuration data transfer is abandoned after the assumed latency expires. Processor makes “best effort” in transferring configuration data to MRLA within the allotted <i>l</i> cycles. Incomplete transfers are aborted. Presumably, subsequent <i>inc/incns</i> operations resume and complete the configuration data transfer. Note: every call to <i>incns</i> instruction should be preceded by a call to <i>malloc</i> to that <i>cr</i> .
<i>outc cr, l</i>	Configuration data corresponding to the CFU on MRLA associated with <i>cr</i> removed from MRLA and saved in the C-cache. The MRLA space used by the CFU is freed and necessary space in C-cache is allocated for the configuration. The configuration in the C-cache is still associated with the same configuration register <i>cr</i> . The latency assumed by the compiler for this instance of the <i>outc</i> operation is <i>l</i> cycles. If the operation does not complete in <i>l</i> cycles, the processor is stalled until the operation completes. Note: every call to <i>outc</i> instruction should be preceded by a call to <i>inc</i> to that <i>cr</i> without an intervening call to free or <i>delc</i> that take the same <i>cr</i> as operand.
<i>outcns cr, l</i>	This operation is identical to <i>outc</i> except that the CFU configuration data transfer is abandoned after the assumed latency expires. Processor makes “best effort” in transferring configuration data to C-cache from MRLA within the allotted <i>l</i> cycles. Incomplete transfers are aborted. Presumably, subsequent <i>outc/outcns</i> operations resume and complete the configuration data transfer.
<i>delc cr, l</i>	Configured functional unit associated with the configuration register <i>cr</i> is removed from MRLA. Unlike <i>outc</i> , the configuration is not moved to the C-cache. The MRLA space is freed for future allocation and the configuration register <i>cr</i> is available for future configurations. The latency assumed by the compiler for this instance of the delc operation is <i>l</i> cycles. Note: every call to <i>delc</i> instruction should be preceded by a call to <i>inc</i> to that <i>cr</i> without an intervening call to <i>free</i> or <i>outc</i> that take the same <i>cr</i> as operand.

Appendix C

AEPIC ISA Summary

C.1 Notes On Instruction Format

Since the AEPIC ISA is an extension of the HPL-PD ISA, we use the same notation as in the description of the HPL-PD architectural interface in [84]. We repeat the description given in [84] here in brief. Format of the operation description (one row of the table below) is as follows:

$\langle \textit{Opcode} \rangle \langle \textit{Operation} \rangle \langle \textit{IOFormat} \rangle \langle \textit{Sp} \rangle \langle \textit{Semantics} \rangle$

- **Opcode.** This field specifies the major opcode Op and the modifiers $A, B \dots$ in the format $Op_{A|B|\dots}$. The opcode for an instruction is the major opcode followed by exactly one of the modifiers (if present).
- **Operation.** A short description of the actions performed by the operation are described in this column of the row.
- **IOFormat.** The IOFormat lists the explicitly specified input and output operands in the following format:

$\langle \textit{predicated} \rangle \langle \textit{source} \rangle \dots \langle \textit{source} \rangle : \langle \textit{destination} \rangle \dots \langle \textit{destination} \rangle$

If the “predicated” field is P? then the operation takes a predicate register as input otherwise it is an unpredicated operation. *Source* and *destination* are the source and destination operands. They can be one of I, F, P, B, L, C, A and X. These denote registers from the integer (or general purpose), floating-point, predicate, branch, literal (supplied through the operation), control, array and configuration register files respectively.

- **S.** The **S** field denotes whether the operation has a speculative version (Y) or not (N).
- **Semantics.** For most operations, a brief description about the effects of execution—the actions performed by the operation. The semantics for the adaptive extension instructions is given in Appendix B.

C.2 Adaptive Extension Instructions

Table C.1: Adaptive extension ISA

AEPIC Instruction Repertoire			
Opcode	Operation description	I/O description	S
CFU operation execution			
EXEC	initiate CFU operation execution	P? X, L :	N
SUSP	suspend currently executing operation on CFU	P? X :	N
RESUME	resume currently suspended operation on CFU	P? X :	N
ABORT	abort issued operation on CFU	P? X :	N
RESET	for stateless CFUs, same as ABORT	P? X :	N
STEP	execute suspended operation on CFU for one cycle and suspend again	P? X :	N
STAT	save CFU status into the associated status register from ARF	P? X :	N
CFU input/output association			
INPR	associate ARF register as source for k^{th} input operand for CFU where $k - 1$ INPR operations have already been issued to this CFU	P? X, A :	N
OUTPR	associate ARF register as destination for k^{th} output operand for CFU where $k - 1$ OUTPR operations have already been issued to this CFU	P? X, A :	N
INP	associate ARF register as source for k^{th} input operand for CFU	P? X, A, L :	Y
OUTP	associate ARF register as destination for k^{th} output operand for CFU	P? X, A, L:	Y
PREDP	associate predicate register as source for the predicate operand of CFU	P? X, P :	Y
STATP	associate ARF register as the destination for saving CFU status	P? X, A :	Y
MRLA and C-cache resource allocation			
MALLOC	allocate space on MRLA on specified context for CFU	P? X, I :	Y
CALLOC	allocate adequate number of blocks in the C-cache for a new configuration	P? X, I:	Y
GCC	compact the CFUs on the specified context of the MRLA	P? L :	Y
GCALL	compact the CFUs on all the contexts of the MRLA	P? :	Y
FREE	deallocate space allocated for configuration	P? X :	Y
Load (store) configurations from (to) memory			
LDCC	“Stalling till completion” configuration load instruction.	P? X, L :	N
LDCCNS	Non-stalling configuration load instruction.	P? X, L :	N

STCC	“Stalling till completion” configuration store instruction.	P? X, I, L :	N
STCCNS	Non-stalling configuration store instruction.	P? X, I, L :	N
Manage execution contexts on MRLA			
SETCTX	make the specified context the active context of the MRLA	P? L :	Y
CLRCTX	free all resources allocated to CFUs on specified context	P? L :	Y
CLRALLCTX	clear all contexts of MRLA	P? :	Y
SWITCHCTX	switch CFU context in MRLA	P? X, L :	Y
PUSHCTX	allocate contexts	P? L :	N
POPCTX	de-allocate contexts	P? L :	N
Configure functional units on the MRLA			
INC	configure functional unit on the MRLA	P? X, L :	Y
OUTC	remove configured functional unit from MRLA	P? X, L :	N
DELC	delete configured functional unit from MRLA	P? X, L :	N

Bibliography

- [1] A. L. Abbott, P. M. Athanas, L. Chen, and R. L. Elliott. Finding lines and building pyramids with Splash 2. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 155–161, Napa, CA, April 1994.
- [2] ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society. *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, California, October 5–8, 1987. *Computer Architecture News*, 15(5), October 1987; *Operating Systems Review*, 21(4), October 1987; *SIGPLAN Notices*, 22(10), October 1987.
- [3] ACM SIGMICRO and IEEE Computer Society TC-MICRO. *Proceedings of the 19th Annual Workshop on Microprogramming*, New York, New York, October 15–17, 1986. *SIGMICRO Newsletter*, 17(4), December 1986.
- [4] Anant Agarwal, Saman Amarasinghe, Rajeev Barua, Matthew Frank, Walter Lee, Vivek Sarkar, Devabhaktuni Srikrishna, , and Michael Taylor. The RAW compiler project. In *Proceedings of the Second SUIF Compiler Workshop*, pages 21–23, Stanford, CA, August 1997.
- [5] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, tools*. Addison-Wesley, Reading, MA, USA, 1986.
- [6] Ray Andraka. A survey of CORDIC algorithms for FPGA based computers. In *Proceedings of the ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays (FPGA-98)*, pages 191–200, New York, February 22–24 1998. ACM Press.
- [7] P. M. Athanas and H. F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *IEEE Computer*, 26(3):11–18, March 1993.
- [8] D. August, D. Connors, S. Mahlke, J. Sias, K. Crozier, B. Cheng, P. Eaton, Q. Olaniran, and W. Hwu. Integrated predicated and speculative execution in the IMPACT EPIC architecture, June 27–July 1, 1998.
- [9] D. August, K. Crozier, J. Sias, P. Eaton, Q. Olaniran, D. Connors, and W. Hwu. The impact epic 1.0 architecture and instruction set reference manual, 1998.
- [10] D. August, W. Hwu, and S. Mahlke. A framework for balancing control flow and predication, 1997.
- [11] B. R. Rau. The Era Of Embedded Computing. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis For Embedded Systems, CASES 2000*, page 119, 2000.
- [12] J. P. Bennet. *A Methodology for Automated Design of Computer Instruction Sets*. PhD thesis, University of Cambridge, 1987.

- [13] N. W. Bergmann and J. C. Mudge. Comparing the performance of FPGA-based custom computers with general-purpose computers for DSP applications. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 164–171, Napa, CA, April 1994.
- [14] D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines, 1991.
- [15] P. Bertin, D. Roncin, and J. Vuillemin. Introduction to programmable active memories. In J. McCanny, J. McWhirther, and E. Swartslander Jr., editors, *Systolic Array Processors*, pages 300–309. Prentice Hall, 1989.
- [16] P. Bertin, D. Roncin, and J. Vuillemin. Programmable active memories: a performance assessment. In G. Borriello and C. Ebeling, editors, *Research on Integrated Systems: Proceedings of the 1993 Symposium*, pages 88–102, 1993.
- [17] James F. Blinn. Jim Blinn’s corner: Fugue for MMX. *IEEE Computer Graphics and Applications*, 17(2):88–93, March/April 1997. Makes several cogent comments about deficiencies in the Intel MMX pixel-processing instruction set [120] for use in image compositing.
- [18] Gordon Brebner and John Gray. Use of reconfigurability in variable-length code detection at video rates. In W. Moore and W. Luk, editors, *Field-Programmable Logic and Applications. 5th International Workshop on Field-Programmable Logic and Applications*, pages 429–438, Oxford, UK, September 1995. Springer-Verlag.
- [19] Preston Briggs. *Register allocation via graph coloring*. Phd thesis, Rice University, 1992.
- [20] P. Brucker and S. Knust. Complexity results for single-machine problems with positive finishstart time lags, 1998.
- [21] John Bruno, John W. Jones, III, and Kimming So. Deterministic scheduling with pipelined processors. *IEEE Transactions on Computers*, C-29(4):308–316, April 1980.
- [22] Doug Burger and James R. Goodman. Guest editors introduction: Billion-transistor architectures. *Computer*, 30(9):46–49, September 1997.
- [23] Srihari Cadambi, Jeffrey Weener, Seth Copen Goldstein, Herman Schmit, and Donald E. Thomas. Managing pipeline-reconfigurable fpgas. In *Proceedings ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, February 1998.
- [24] D. Callahan and B. Koblenz. Register allocation via hierarchical graph coloring. In *Proceedings of the ACM SIGPLAN ’91 Conference on Programming Language Design and Implementation*, volume 26, pages 192–203, Toronto, Ontario, Canada, June 1991.
- [25] Timothy J. Callahan, Phillip Chong, André DeHon, and John Wawrzynek. Fast module mapping and placement for datapaths in FPGAs. In *Proceedings of the ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays (FPGA-98)*, pages 123–132, New York, February 22–24 1998. ACM Press.
- [26] CCITT. Adaptive differential pulse code modulation, 1984.
- [27] G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register allocating via coloring, 1981.
- [28] Gregory J. Chaitin. Register allocation and spilling via graph coloring. *SIGPLAN Notices (Proceedings of the SIGPLAN ’82 Symposium on Compiler Construction, Boston, Mass.)*, 17(6):98–105, 1982.
- [29] P. Chang and W. Hwu. The lcode language and its environment, 1991.

- [30] C. Chou, S. Mohanakrishnan, and J. B. Evans. FPGA implementation of digital filters. In *Proceedings of the Fourth International Conference on Signal Processing Applications and Technology*, pages 80–88, Santa Clara, CA, 1993.
- [31] F. Chow, K. Knobe, A. Meltzer, R. Morgan, and K. Zadeck. Register allocation.
- [32] Fred C. Chow and John L. Hennessy. Register allocation by priority-based coloring. In *Proceedings of the ACM SIGPLAN 84 Symposium on Compiler Construction*, pages 222–232, New York, NY, 1984. ACM.
- [33] Robert P. Colwell, Robert P. Nix, John J. O'Donnell, David P. Papworth, and Paul K. Rodman. A VLIW architecture for a trace scheduling compiler. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems [2]*, pages 180–192. *Computer Architecture News*, 15(5), October 1987; *Operating Systems Review*, 21(4), October 1987; *SIGPLAN Notices*, 22(10), October 1987.
- [34] R. Cypher, J. Sanz, L. Snyder, O. The, N. complexity, and N. by. mesh array architectures, 1987.
- [35] A. Dandalis and V. Prasanna. Fast parallel implementation of dft using configurable devices, 1997.
- [36] M. Dumas, J. M. Muller, and J. Vuillemin. Implementing on line arithmetic on PAM. In R. Hartenstein and M. Z. Servit, editors, *Field-Programmable Logic: Architectures, Synthesis and Applications. 4th International Workshop on Field-Programmable Logic and Applications*, pages 196–207, Prague, Czech Republic, September 1994. Springer-Verlag.
- [37] Marcelo Alves de Barros and Mohamed Akil. Low level image processing operators on FPGA: Implementation examples and performance evaluation. In *International Conference on Pattern Recognition D: Parallel Computing*, pages 262–267, Jerusalem, Israel, 1994. International Association for Pattern Recognition.
- [38] A. DeHon. Dpga utilization and application, 1996.
- [39] A. DeHon. Multicontext fieldprogrammable gate arrays, 1997.
- [40] P. Dinda, T. Gross, D. O'Hallaron, E. Segall, J. Stichnoth, J. Subhlok, J. Webb, and B. Yang. The CMU task parallel program suite, 1994.
- [41] Matias Djunatan and Tati. Mengko. A programmable real-time systolic processor architecture for image morphological operations, binary template matching and min-max filtering. In *1991 IEEE International Symposium on Circuits and Systems*, volume 5 volumes, pages 65–68, June 1991. IEEE Catalog Number 91CH3006-4 Library of Congress Number 80-646530(softbound) ISBN 0-7803-0051-3 (casebound) ISBN 0-7803-0052-1 (microfiche).
- [42] Carole Dulong. The IA-64 architecture at work. *Computer*, 31(7):24–32, 1998.
- [43] S. Eckart and m play. MPEG software simulation group, 1994.
- [44] M. Ercegovac, J. M. Muller, and A. Tisserand. FPGA implementation of polynomial evaluation algorithms. In J. Schewel, editor, *Proceedings of the International Society of Optical Engineering (SPIE). Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing.*, volume 2607, pages 177–188, Philadelphia, PA, October 1995.
- [45] G. Estrin. Organization of computer systems – the fixed plus variable structure computer. In *Proceedings of the Western Joint Computer Conference*, pages 33–40, 1960.
- [46] B. Fagin and C. Renard. Field programmable gate arrays and floating point arithmetic. *IEEE Transactions on Very Large Scale Integration [VLSI] Systems*, 2(3):365–367, September 1994.

- [47] O. Faugeras, B. Hotz, H. Mathieu, T. Viéville, Z. Zhang, P. Fua, E. Théron, L. Moll, G. Berry, J. Vuillemin, P. Bertin, and C. Proy. Real time correlation-based stereo: algorithm, implementations and applications. Technical Report 2013, Institut National De Recherche en Informatique et en Automatique (INRIA), 06902 Sophia Antipolis, France, 1993.
- [48] Joseph A. Fisher. Walk-time techniques: Catalyst for architectural change. *Computer*, 30(9):40–42, September 1997.
- [49] F. Furtek. A field-programmable gate array for systolic computing. In G. Borriello and C. Ebeling, editors, *Research on Integrated Systems: Proceedings of the 1993 Symposium*, pages 183–199, 1993.
- [50] J. Gailly and M. Adler. ZLIB documentation and sources.
- [51] M. Garey, D. Johnson, and C. intractability. a guide to the theory of np-completeness, 1979.
- [52] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweely, and D. Lopresti. Building and using a highly parallel programmable logic array. *IEEE Computer*, 24(1):81–89, January 1991.
- [53] M. Gokhale and B. Schott. Data parallel C on a reconfigurable logic array. *Journal of Supercomputing*, 9(3):291–313, 1994.
- [54] P. Graham and B. Nelson. A hardware genetic algorithm for the travelling salesman problem on SPLASH 2. In W. Moore and W. Luk, editors, *Field-Programmable Logic and Applications*, pages 352–361, Oxford, England, August 1995. Springer.
- [55] P. Graham and B. Nelson. Genetic algorithms in software and in hardware — A performance analysis of workstation and custom computing machine implementations. In J. Arnold and K. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 216–225, Napa, CA, April 1996.
- [56] P. S. Graham. A description, analysis, and comparison of a hardware and a software implementation of the SPLASH genetic algorithm for optimizing symmetric traveling salesman problems. Master’s thesis, Brigham Young University, Provo, UT, 1996.
- [57] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Ann. Discrete Mathematics*, 5:287–326, 1979.
- [58] C. Ebeling D. C. Green and P. Franklin. RaPiD – reconfigurable pipelined datapath. In R. W. Hartenstein and M. Glesner, editors, *Field-Programmable Logic: Smart Applications, New Paradigms, and Compilers. 6th International Workshop on Field-Programmable Logic and Applications*, pages 126–135, Darmstadt, Germany, September 1996. Springer-Verlag.
- [59] R. Halverson, Jr. and A. Lew. Programming with functional memory. In Dharma P. Agrawal, editor, *Proceedings of the 23rd International Conference on Parallel Processing. Volume 1: Architecture*, pages 85–92, Boca Raton, FL, USA, August 1994. CRC Press.
- [60] F. Haney. ISDS-A program that designs computer instruction sets. In *Fall Joint Computer Conference, 1969*, 1969.
- [61] R. W. Hartenstein, A. G. Hirschbiel, M. Riedmuller, K. Schmidt, and M. Weber. A novel ASIC design approach based on a new machine paradigm. *IEEE Journal of Solid-State Circuits*, 26(7):975–989, July 1991.

- [62] S. Hauck. The chimaera reconfigurable functional unit. In *Proc. of IEEE Symp. on FPGAs for Custom Computing Machines, Napa Valley, California, 1997*, pp. 87–96., 1997.
- [63] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. The chimaera reconfigurable functional unit. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 87–96, 1997.
- [64] S. Hauck, M. M. Hosler, and T. W. Fry. High-performance carry chains for fpgas. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 223–233, 1998.
- [65] John R. Hauser. The GARP architecture. Technical report, University of California at Berkeley, Berkeley, CA, USA, October 1997.
- [66] John R. Hauser and John Wawrzynek. GARP: A MIPS processor with a reconfigurable coprocessor. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 12–21, Napa, CA, April 1997.
- [67] J. L. Hennessy and T. R. Gross. Code generation and reorganization in the presence of pipeline constraints. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 120–127, New York, NY 10036, USA, January 1982. ACM Press.
- [68] D. T. Hoang. Searching genetic databases on Splash 2. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 185–191, Napa, CA, April 1993.
- [69] Bruce K. Holmer and Alvin M. Despain. Viewing instruction set design as an optimization problem. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pages 153–162, Albuquerque, New Mexico, November 18–20, 1991. ACM SIGMICRO and IEEE Computer Society TC-MICRO.
- [70] <http://www.trimaran.org>. Trimaran ILP Research Infrastructure, 1998.
- [71] I. Huang and A. Despain. Generating instruction sets and microarchitectures from applications, 1994.
- [72] I. Huang and A. Despain. Synthesis of application specific instruction sets, 1995.
- [73] I.-J. Huang and A. M. Despain. Generating instruction sets and microarchitectures from applications. In *International Conference on Computer-Aided Design*, pages 391–396, Los Alamitos, Ca., USA, November 1994. IEEE Computer Society Press.
- [74] Wen-mei Hwu and Yale N. Patt. Checkpoint repair for out-of-order execution machines. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 18–26, Pittsburgh, Pennsylvania, June 2–5, 1987. IEEE Computer Society TCCA and ACM SIGARCH. *Computer Architecture News*, 15(2), June 1987.
- [75] IEEE Computer Society and ACM SIGARCH. *Proceedings of the 9th Annual Symposium on Computer Architecture*, Austin, Texas, April 26–29, 1982. *Computer Architecture News*, 10(3), April 1982.
- [76] IEEE Computer Society TC-MICRO and ACM SIGMICRO. *Proceedings of the 26th Annual International Symposium on Microarchitecture*, Austin, Texas, December 1–3, 1993.
- [77] IEEE Computer Society TC-MICRO and ACM SIGMICRO. *Proceedings of the 31st Annual International Symposium on Microarchitecture*, Dallas, Texas, November 30–December 2, 1998.
- [78] IEEE Computer Society TCCA and ACM SIGARCH. *Proceedings of the 26th Annual International Symposium on Computer Architecture*, Atlanta, Georgia, May 2–4, 1999.
- [79] D. Integrity. National institute of standards and technology, 1989.

- [80] C. Iseli and E. Sanchez. Spyder: A reconfigurable VLIW processor using FPGAs. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 17–24, Napa, CA, April 1993.
- [81] T. Isshiki and W. Dai. High-level bit-serial datapath synthesis for multi-FPGA systems. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 167–173, Monterey, CA, February 1996.
- [82] C. C. Jong, Y. Y. H. Lam, and L. S. Ng. FPGA implementation of a digital IQ demodulator using VHDL. *Lecture Notes in Computer Science*, 1304:410–??, 1997.
- [83] Norman P. Jouppi and David W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *3rd Symposium on Architectural Support for Programming Languages and Operating Systems, Computer Architecture News*, pages 272–282, Boston, MA, April 1989. Published as 3rd Symposium on Architectural Support for Programming Languages and Operating Systems, Computer Architecture News, volume 17, number 2, DECWRL.
- [84] V. Kathail, M. Schlansker, and B. Rau. HPL PlayDoh Architecture Specification Version. *Technical report HPL-93-80, Hewlett-Packard Laboratories, Technical Publications Department, 1501 Page Mill Road, Palo Alto, CA 94304.*, 1994.
- [85] T. Kean, B. New, and B. Slous. A fast constant coefficient multiplier for the XC6200. *Lecture Notes in Computer Science*, 1142:230–??, 1996.
- [86] A. Kempe. The geographical problem of the four colors. *Amer. J. Math.* 2, 193–200., 1879.
- [87] H. Kim, K. Gopinath, and V. Kathail. Register allocation in hyper-block for EPIC processors, 1999.
- [88] H. Kim and A. Leung. Frequency based live range splitting. *Technical report, ReaCT-ILP Laboratory, New York University*, 1999.
- [89] Bjørn Klock, Lisbet Utne, and John Håkon Husøy. Implementation of filter banks in field programmable gate arrays (FPGA). In *Proc. Int. Conf. Signal Proc. Appl. Tech.*, pages 441–445, Dallas, U.S.A., October 1994.
- [90] Krishna V Palem and Surendranath Talla. Reduced Configuration Space Processor: Architecture and Compilation Techniques. In *High Performance Embedded Computing (HPEC) Workshop, MIT Lincoln Lab, Lexington, MA*, pages 241–261, September 1997.
- [91] Krishna V Palem and Surendranath Talla and Patrick W. Devaney. Adaptive Explicitly Parallel Instruction Computing. In *Australasian Computer Architecture Conference, Auckland, New Zealand*, pages 61–74. Springer-Verlag, Singapore, January 1999.
- [92] Monica S. Lam and Robert P. Wilson. Limits of control flow on parallelism. In *The 19th Annual International Symposium on Computer Architecture (ISCA)*, pages 46–57, 1992.
- [93] C. Lee. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems, 1997.
- [94] Walter Lee, Rajeev Barua, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, Saman Amarasinghe, and Anant Agarwal. Space-time scheduling of instruction-level parallelism on a RAW machine. *MIT/LCS Technical Memo TM-572*, December 1997.
- [95] M. Leeser, R. Chapman, M. Bertone, and A. Wenban. Implementing filters with programmable logic. In W. Moore and W. Luk, editors, *More FPGAs. Oxford International Workshop on Field-Programmable Logic and Applications*, pages 192–201, Oxford, England, August 1993. Abingdon EE&CS Books.

- [96] E. Lemoine and D. Merceron. Run time reconfiguration of FPGA for scanning genomic databases. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 90–98, Napa, CA, April 1995.
- [97] Allen Leung, Krishna V. Palem, and Amir Pnueli. A fast algorithm for scheduling time-constrained instructions on processors with ILP. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*, pages 158–166, Paris, France, October 12–18, 1998. IEEE Computer Society Press.
- [98] M. Lipasti. Value locality and speculative execution, 1997.
- [99] D. P. Lopresti. Rapid implementation of a genetic sequence comparator using field-programmable gate arrays. In C. Sequin, editor, *Advanced Research in VLSI: Proceedings of the 1991 University of California/Santa Cruz Conference*, pages 138–152, Santa Cruz, CA, March 1991.
- [100] D. P. Lopresti. Rapid implementation of a genetic sequence comparator using FPGAs. *Adv. Res. VLSI*, pages 139–152, 1991.
- [101] M. E. Louie and M. D. Ercegovac. A digit-recurrence square root implementation for field programmable gate arrays. In *IEEE Workshop on FPGAs for Custom Computing Machines*, April 1993.
- [102] M. E. Louie and M. D. Ercegovac. On digit-recurrence division implementation for field programmable gate arrays. In E. E. Swartzlander, M. J. Irwin, and J. Jullien, editors, *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 202–209, Windsor, Canada, June 1993. IEEE Computer Society Press, Los Alamitos, CA.
- [103] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *25th Annual International Symposium on Microarchitecture (MICRO-25)*, pages 45–54, 1992.
- [104] T. Mathews, S. C. Gibb, L. E. Turner, and P. J. W. Graumann. An FPGA implementation of a matched filter detector for spread spectrum communications systems. *Lecture Notes in Computer Science*, 1304:364–??, 1997.
- [105] Wen mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7:229–248, 1993.
- [106] U. Meyer-Base, A. Meyer-Baese, and W. Hilberg. Coordinate rotation digital computer (CORDIC) synthesis for FPGA. *Lecture Notes in Computer Science*, 849:397–??, 1994.
- [107] Laurent Moll, J. Vuillemin, and P. Boucard. High energy physics on DECPeRLe-1 programmable active memory. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 47–52, Monterey, CA, February 1995.
- [108] S. Monaghan and C. P. Cowen. Reconfigurable multi-bit processor for DSP applications in statistical physics. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 103–110, Napa, CA, April 1993.
- [109] S. Muchnick. Advanced compiler design and implementation, 1997.
- [110] Alexandru Nicolau and Joseph A. Fisher. Measuring the parallelism available for very long instruction word architectures. *IEEE Transactions on Computers*, C-33(11):968–976, November 1984.

- [111] M. Flynn O. Mencer, M. Morf. Pam-blox: High performance fpga design for adaptive computing. *IEEE Symposium on FPGAs for Custom Computing Machines*, 1998.
- [112] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. *ACM SIGPLAN Notices*, 31(9):2–11, September 1996. Co-published as SIGOPS Operating Systems Review **30**(5), December 1996, and as SIGARCH Computer Architecture News, **24**(special issue), October 1996.
- [113] E. Özer, S. W. Sathaye, K. N. Menezes, S. Banerjia, M. D. Jennings, and T. M. Conte. A fast interrupt handling scheme for VLIW processors. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*, pages 136–141, Paris, France, October 12–18, 1998. IEEE Computer Society Press.
- [114] I. Page. The HARP reconfigurable computing system. Technical report, Oxford University Hardware Compilation Group, October 1994.
- [115] S. Palacharla, N. Jouppi, and J. Smith. Complexity-effective superscalar processors, 1997.
- [116] K. Palem and B. Simons. Scheduling time-critical instructions on risc machines, 1990.
- [117] Krishna Palem. End-to-end Solutions for Reconfigurable Systems: The Programming Gap and Challenges. In *Proceedings the 30'th HICSS, Hawai'i.*, 1997.
- [118] G. Panneerselvam, P. J. W. Graumann, and L. E. Turner. Implementation of fast fourier transforms and discrete cosine transforms in FPGAs. In W. Moore and W. Luk, editors, *Field-Programmable Logic and Applications. 5th International Workshop on Field-Programmable Logic and Applications*, pages 272–281, Oxford, UK, September 1995. Springer-Verlag.
- [119] Heonchul Park and Viktor Prasanna. A class of optimal VLSI architectures for computing discrete fourier transform. In Quentin F. Stout, editor, *Proceedings of the 1992 International Conference on Parallel Processing. Volume 3: Algorithms and Applications*, pages 61–68, Ann Arbor, MI, August 1992. CRC Press.
- [120] Alex Peleg, Sam Wilkie, and Uri Weiser. Intel MMX for multimedia PCs. *Communications of the ACM*, 40(1):24–38, January 1997. See also Blinn's comments [17] about MMX instruction set deficiencies.
- [121] R. J. Petersen and B. L. Hutchings. An assessment of the suitability of FPGA-based systems for use in digital signal processing. In W. Moore and W. Luk, editors, *Field-Programmable Logic and Applications*, pages 293–302, Oxford, England, August 1995. Springer.
- [122] I. Pyo, C.-L. Su, I.-J. Huang, K.-R. Pan, Y.-S. Koh, C.-Y. Tsui, H.-T. Chen, G. Cheng, S. Liu, S. Wu, and A. M. Despain. Application-driven design automation for microprocessor design. In *Proceedings of the 29th Conference on Design Automation*, pages 512–517, Los Alamitos, CA, USA, June 1992. IEEE Computer Society Press.
- [123] Lawrence R. Rabiner and Ronald W. Schafer. *Digital Processing of Speech Signals*. Prentice Hall, Englewood Cliffs, New Jersey, 1978.
- [124] S. Rajamani and P. Viswanath. Accelerating the risc processor using programmable logic, 1995.
- [125] B. Rau and J. Fisher. Instruction-level parallel processing: History, overview, and perspective., 1993.
- [126] B. Ramakrishna Rau. Dynamically scheduled VLIW processors. In *Proceedings of the 26th Annual International Symposium on Microarchitecture* [76], pages 80–92.

- [127] R. Razdan and M. D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 172–80. IEEE/ACM, November 1994.
- [128] Rahul Razdan. *PRISC: Programmable Reduced Instruction Set Computers*. PhD thesis, Harvard University, May 1994.
- [129] M. Rencher and B. Hutchings. Automated target recognition on splash 2. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 192–200, Napa, CA, April 1997.
- [130] Michael Rencher. A comparison of FPGA platforms through SAR/ATR algorithm implementation. Master’s thesis, Department of Electrical and Computer Engineering, Brigham Young University, Provo, Utah, 1996.
- [131] E. M. Riseman and C. C. Foster. The inhibition of potential parallelism by conditional jumps. In *IEEE Transactions on Computers*, volume C-21, pages 1405–1411, 1972.
- [132] R. Rivest. Description of the RC2 encryption algorithm.
- [133] Richard Ross. An FPGA implementation of ATR using embedded RAM for control. Master’s thesis, Department of Electrical and Computer Engineering, Brigham Young University, Provo, Utah, 1997.
- [134] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors, 1997.
- [135] C. R. Rupp. *CLayFUNtm Reference Manual*. National Semiconductor, September 1994. Version 2.00.
- [136] Michael Saleeba. A dynamically reconfigurable computer architecture.
- [137] C. Sanz, L. De Zulueta, and J. M. Meneses. FPGA implementation of the block-matching algorithm for motion estimation in image coding. *Lecture Notes in Computer Science*, 1142:146–??, 1996.
- [138] J. Sato, M. Imai, T. Hakata, A. Alomary, and N. Hikichi. An integrated design environment for application-specific integrated processors. In *Proc. IEEE Int. Conf. on Computer Design*, pages 414–417, Rochester (New York, U.S.A.), 1991.
- [139] Mario R. Schaffner. Processing by data and program blocks. *IEEE Transactions on Computers*, 27(11):1015–1028, November 1978.
- [140] M. Schlansker and B. Rau. EPIC: An architecture for instruction-level parallel processors, 2000.
- [141] M. Schlansker, B. Rau, S. Mahlke, V. Kathail, R. Johnson, S. Anik, and S. Abraham. Achieving high levels of instruction-level parallelism with reduced hardware complexity. *Technical Report HPL-96-120, Hewlett Packard Laboratories, Feb. 1997.*, 1997.
- [142] H. Schmidt. Incremental reconfiguration for pipelined applications. In *Proceedings of the IEEE Symp. On FPGA for Custom Computing Machines, Napa, CA, 1997.*, 1997.
- [143] B. Schneier. *Applied Cryptography*. John Wiley & Sons, 1994.
- [144] Bruce Schneier. The IDEA encryption algorithm. *Dr. Dobb’s Journal of Software Tools*, 18(13):50, 52, 54, 56, 106, 1993.
- [145] B. Schoner, C. Jones, and J. Villasenor. Issues in wireless coding using run-time-reconfigurable FPGAs. In P. Athanas and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 85–89, Napa, CA, April 1995.

- [146] Sematech. International technology roadmap for semiconductors, 1999.
- [147] M. Shand, P. Bertin, and J. Vuillemin. Hardware speedups in long integer multiplication. *Computer Architecture News*, 19(1):106–114, 1991.
- [148] M. Shand and J. Vuillemin. Fast implementations of RSA cryptography. In *11th IEEE Symposium on COMPUTER ARITHMETIC*, 1993.
- [149] N. Shirazi, P. M. Athanas, and A. L. Abbott. Implementation of a 2-D fast fourier transform on an FPGA-based custom computing machine. In W. Moore and W. Luk, editors, *Field-Programmable Logic and Applications. 5th International Workshop on Field-Programmable Logic and Applications*, pages 282–292, Oxford, UK, September 1995. Springer-Verlag.
- [150] R. Shoup. Parameterized convolution filtering in an FPGA. In W. Moore and W. Luk, editors, *More FPGAs. Oxford International Workshop on Field-Programmable Logic and Applications*, pages 274–280, Oxford, England, August 1993. Abingdon EE&CS Books.
- [151] B. Sikstrom, L. Wanhammar, M. Afghahi, and J. Pencz. A high speed 2-d discrete cosine transform chip. In *INTEGRATION, The VLSI Journal*, May 1987.
- [152] E. Simoncelli and E. Abelson. EPIC: an efficient pyramid image coder, 1990.
- [153] Barbara Simons. A fast algorithm for multiprocessor scheduling. In *21st Annual Symposium on Foundations of Computer Science*, pages 50–53, Syracuse, New York, 13–15 October 1980. IEEE.
- [154] N. Sitkoff, M. Wazlowski, A. Smith, and H. Silverman. Implementing a genetic algorithm on a parallel custom computing machine. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 180–187, Napa, CA, April 1995.
- [155] J. Smith and G. Sohi. The microarchitecture of superscalar processors, 1995.
- [156] James E. Smith and Andrew R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, 37(5):562–573, May 1988.
- [157] M. Smith. The interaction of compilation technology and computer architecture, 1994.
- [158] M. Smith, M. Johnson, and M. Horowitz. Limits on multiple instruction issue, 1989.
- [159] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proc. of the 22nd Annual International Symposium on Computer Architecture (22nd ISCA'95) ACM SIGARCH Computer Architecture News*, pages 414–425, Santa Margherita, Italy, June 1995. Published as Proc. of the 22nd Annual International Symposium on Computer Architecture (22nd ISCA'95) ACM SIGARCH Computer Architecture News, volume 23, number 6.
- [160] Gurindar S. Sohi and Sriram Vajapeyam. Tradeoffs in instruction format design for horizontal architectures. In *3rd Symposium on Architectural Support for Programming Languages and Operating Systems (3rd ASPLOS'89), Computer Architecture News*, pages 15–25, Boston, MA, April 1989. Published as 3rd Symposium on Architectural Support for Programming Languages and Operating Systems (3rd ASPLOS'89), Computer Architecture News, volume 17, number 2, U WI.
- [161] SPEC Benchmark Specifications, 1995.
- [162] E. Tau, I. Eslick, D. Chen, J. Brown, and A. DeHon. A first generation DPGA implementation. In *Proceedings of the Third Canadian Workshop on Field-Programmable Devices*, pages 138–143, 1995.

- [163] A. Tisserand and M. Dimmler. FPGA implementation of real-time digital controllers using on-line arithmetic. *Lecture Notes in Computer Science*, 1304:472–??, 1997.
- [164] Garold S. Tjaden and Michael J. Flynn. Detection and parallel execution of independent instructions. *IEEE Transactions on Computers*, 19(10):889–895, October 1970.
- [165] Stephen M. Trimberger. *Field-Programmable Gate Array Technology*. Kluwer Academic Publishers, 1994.
- [166] Steve Trimberger, Dean Carberry, Anders Johnson, and Jennifer Wong. A time-multiplexed FPGA. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 22–28, Napa, CA, 1997.
- [167] K. W. Tse, C. H. Leung, and K. F. Cheng. Implementation of pre-processing and feature extraction of Chinese characters with FPGAs. In W. Moore and W. Luk, editors, *More FPGAs: Proceedings of the 1993 International workshop on field-programmable logic and applications*, pages 307–314, Oxford, England, September 1993.
- [168] Dean Michael Tullsen. *Simultaneous multithreading*. Thesis (ph.d.), University of Washington, Seattle, WA, USA, 1996.
- [169] J. D. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384–393, June 1975.
- [170] J. Villasenor, B. Schoner, K. Chia, and C. Zapata. Configurable computing solutions for automatic target recognition. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 70–79, Napa, CA, April 1996.
- [171] John Villasenor and William H. Mangione-Smith. Configurable computing. *Scientific American*, pages 66–71, June 1997.
- [172] Jack E. Volder. The CORDIC Trigonometric Computing Technique. *IRE Transactions on Electronic Computers*, EC-8:330–334, 1959.
- [173] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard. Programmable active memories: Reconfigurable systems come of age. *IEEE Transactions on VLSI Systems*, 4(1):56–69, 1996.
- [174] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: RAW machines. *IEEE Computer*, pages 86–93, September 1997.
- [175] D. Wall. Limits to instruction level parallelism, 1989.
- [176] David W. Wall. Limits of instruction-level parallelism. Technical Report DEC-WRL-93-6, Digital Equipment Corporation, Western Research Lab, November 93.
- [177] N. Warter and G. Haab. Pcode manual, 1991.
- [178] M. J. Wirthlin and B. L. Hutchings. DISC: The dynamic instruction set computer. In J. Schewel, editor, *Proceedings of the International Society of Optical Engineering (SPIE). Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing.*, volume 2607, pages 92–103, Philadelphia, PA, 1995.
- [179] R. D. Wittig and P. Chow. OneChip: An FPGA processor with reconfigurable logic. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 126–135, Napa, CA, April 1996.

- [180] Xilinx, San Jose, CA. *The Programmable Logic Data Book*, 1994.
- [181] Xilinx, San Jose, CA. *Xilinx 6200 Preliminary Data Sheet*, 1996.
- [182] S. Xu, D. He, and X. Wang. An implementation of the gsm general data encryption algorithm a5, 1994.
- [183] D. Yeh, G. Heygin, and P. Chow. RACER: A reconfigurable constraint-length 14 Viterbi decoder. In P. Athanas and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 60–69, Napa, CA, april 1996.