High Level Compilation for Gate Reconfigurable Architectures

by

Jonathan William Babb

S.M.E.E., Massachusetts Institute of Technology (1994)
B.S.E.E., Georgia Institute of Technology (1991)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2001

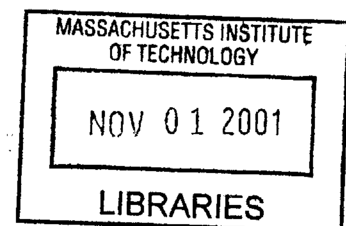
© Massachusetts Institute of Technology 2001. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
August 2001

Certified by.... ....
Anant Agarwal
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

BARKER



High Level Compilation for Gate Reconfigurable Architectures

by
Jonathan William Babb

Submitted to the Department of Electrical Engineering and Computer Science
on August 2001, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

A continuing exponential increase in the number of programmable elements is turning management of gate-reconfigurable architectures as "glue logic" into an intractable problem; it is past time to raise this abstraction level. The physical hardware in gate-reconfigurable architectures is all low level — individual wires, bit-level functions, and single bit registers — hence one should look to the fetch-decode-execute machinery of traditional computers for higher level abstractions. Ordinary computers have machine-level architectural mechanisms that interpret instructions — instructions that are generated by a high-level compiler. Efficiently moving up to the next abstraction level requires leveraging these mechanisms without introducing the overhead of machine-level interpretation. In this dissertation, I solve this fundamental problem by specializing architectural mechanisms with respect to input programs. This solution is the key to efficient compilation of high-level programs to gate reconfigurable architectures.

My approach to specialization includes several novel techniques. I develop, with others, extensive bitwidth analyses that apply to registers, pointers, and arrays. I use pointer analysis and memory disambiguation to target devices with blocks of embedded memory. My approach to memory parallelization generates a spatial hierarchy that enables easier-to-synthesize logic state machines with smaller circuits and no long wires. My space-time scheduling approach integrates the techniques of high-level synthesis with the static routing concepts developed for single-chip multiprocessors.

Using DeepC, a prototype compiler demonstrating my thesis, I compile a new benchmark suite to Xilinx Virtex FPGAs. Resulting performance is comparable to a custom MIPS processor, with smaller area (40 percent on average), higher evaluation speeds (2.4×), and lower energy (18×) and energy-delay (45×). Specialization of advanced mechanisms results in additional speedup, scaling with hardware area, at the expense of power. For comparison, I also target IBM's standard cell SA-27E process and the RAW microprocessor. Results include sensitivity analysis to the different mechanisms specialized and a grand comparison between alternate targets.

Thesis Supervisor: Anant Agarwal
Title: Professor of Computer Science and Engineering

Acknowledgments

I am indebted to my advisor, Anant Agarwal, both for inspiration and patience. Professors Martin Rinard and Srinivas Devadas served as readers on my thesis committee. Professors and researchers Saman Amarasinghe, Krste Asanović, Norman Margolus and Tom Knight also provided helpful advice anytime I solicited it. The bio-circuit research of Tom and my friend and fellow windsurfer Ron Weiss has kept me humble.

Many pieces of the DeepC compilation system were contributed or co-developed by other researchers. The pointer analysis package was developed by Radu Rugina. The bitwidth analysis package was implemented by Mark Stephenson. The space-time scheduling algorithm is an extension of Walter Lee's algorithm for the Raw Machine. This algorithm is partly based on scheduling for Virtual Wires, which was developed at MIT and extended by Charlie Selvidge and others at Virtual Machine Works, a company I founded with Anant and Russell Tessier. The memory disambiguation algorithms were developed by Rajeev Barua for his PhD. The Verilog generator is built on the data structures of VeriSUIF (Robert French, Monica Lam, Jeremy Levitt, Kunle Olukotun). All these passes make extensive use of Stanford's SUIF infrastructure. In addition, many of the benchmarks in DeepC are leveraged from the Raw Benchmark Suite, which also inspired the DeepC Project. Benchmark authors include: Matthew Frank, Victor Lee, Elliot Waingold, Rajeev Barua, Michael Taylor, Jang Kim, and Devabhaktuni Srikrishna. The SOR benchmark is contributed by Benjamin Greenwald, who also supported many of the Raw system utilities required to generate my results. The Raw simulator was written and supported by Elliot Waingold and Michael Taylor. Alex Kupperman implemented the latest host interface used to realize some of my results. Many other members of MIT's compiler and computer architecture groups indirectly contributed to the success of DeepC, including: Gleb Chuvpilo, Albert Ma, Matthew Frank, Fae Ghodrat, Jason Kim, Samuel Larsen, Diego Puppin, Maria Cristina Marinescu, Darko Marinov, Jason Miller, Michal Karczmarek, Chriss Kappler, Vivek Sarkar, David Wentzlaff, Kevin Wilson, Michael Zhang, and Csaba Andras Moritz. I enjoyed sharing an office with Andras who also provided much additional guidance. Special thanks to Matt Frank for valuable feedback on an early draft. The MIT Writing Center was also an incredible help when it came time to communicate my results.

The original inspiration for this work began with the Virtual Wires Project (Russell Tessier, Matthew Dahl, Silvina Hanono, David Hoki, Ed Oellette, Trevor Bauer, and later Charlie Selvidge and other staff at Virtual Machine Works) and with ideas for reconfigurable computing that Russ and I have batted back and forth for many years. Russ also helped interface DeepC to the VPR place and router (contributed by Vaughn Betz, Jonathan Rose and others at Toronto) and developed the static router implementation for the network overclocking study. André Dehon has also been a valuable sounding board for reconfigurable ideas during our overlap at MIT.

This research was funded in part by numerous ARPA and NSF grants. The VirtuaLogic emulation system was donated by Ikos Systems. Cadence, Synopsys, and Xilinx contributed many of the CAD tools used under their outstanding university programs.

Thanks to Viktor Kunčak, Patrick Lam, and Karen Zee, the fortunate last group of students to share an office with me. Ed Hurley and Joost Bonsen have kept up my entrepreneurial spirit while finishing. Last, but not least, my close family is always available when advice is needed and has tolerated my absence from some important family events in the past year to get this done. Finally, all errors are mine.

Contents

1 Introduction **17**

1.1 Compiler/CAD Convergence 18
1.2 My Thesis: How DeepC Closes The Gap 19
1.2.1 Summary of Contributions 21
1.3 Motivation for Specialization 22
1.3.1 Five Classes of Mechanisms to Specialize 22
1.3.2 The Big Picture: Specialization in General 24
1.3.3 Counterarguments 28
1.4 Previous Work 30
1.4.1 My Previous Work 30
1.4.2 Computer-Aided Design (CAD) 32
1.4.3 Computer Architecture 34
1.4.4 Compiler and Language Implementation 36
1.4.5 Summary 38
1.5 Guide to Dissertation 39

2 Problem and Analysis **41**

2.1 The Old Problem: Compiling to Processors 41
2.2 The New Problem: Compiling to Gate Reconfigurable Architectures 42
2.3 Theory of Evaluation Modes 45
2.4 A Special Relationship 49
2.5 Summary of Result 50

3 Specialization of Basic Mechanisms **51**

3.1 Specializing Combinational Functions 52
3.1.1 Approach 52
3.1.2 Examples 56
3.1.3 Ease of Implementation 58
3.2 Specializing Storage Registers 58
3.2.1 Approach 58
3.2.2 Examples 62
3.2.3 Ease of Implementation 64
3.3 Specializing Control Structures 64
3.3.1 Approach 64
3.3.2 Examples 70
3.3.3 Ease of Implementation 75
3.4 Summary 77

4 Specialization of Advanced Mechanisms **79**
4.1 Specializing Memory Mechanisms 80
4.1.1 Approach 80
4.1.2 Examples 86
4.1.3 Ease of Implementation 86
4.2 Specializing Communication Mechanisms 87
4.2.1 Approach 90
4.2.2 Examples 93
4.2.3 Ease of Implementation 94
4.3 Summary 95

5 DeepC Compiler Implementation **97**
5.1 Compiler Lineage and Use 97
5.2 Overview of Compiler Flow 98
5.2.1 Traditional Frontend Phase 98
5.2.2 Parallelization Phase 100
5.2.3 Space-Time Scheduling Phase 102
5.2.4 Machine Specialization Phase 105
5.2.5 CAD Tool Phase 107
5.3 Target Hardware Technologies 109
5.4 Simulation and Verification Environment 110
5.4.1 Direct Execution on Workstation 112
5.4.2 Parallel Simulation with RawSim 112
5.4.3 Compiled-code Simulation with HaltSUIF 113
5.4.4 RTL Simulation with Verilog Simulator 114
5.4.5 Verification with a Logic Emulation 116
5.5 Summary 116

6 Results **117**
6.1 Experimental Setup 117
6.2 The Deep Benchmark Suite 119
6.3 Basic Results 121
6.3.1 *Mode 0* Basic Results 122
6.3.2 *Mode I* Basic Results 124
6.4 Basic Sensitivity Analysis 128
6.4.1 *Mode 0* Basic Sensitivity 130
6.4.2 *Mode I* Basic Sensitivity 131
6.5 Advanced Results 133
6.5.1 *Mode 0* Advanced Results 135
6.5.2 *Mode I* Advanced Results 135
6.6 Advanced Sensitivity Analysis 136
6.6.1 Memory Sensitivity 140
6.6.2 Communication Sensitivity 143
6.7 Grand Finale: Comparing All Modes 146
6.8 Summary 150

7 Conclusions **153**
7.1 Wrap Up 153
7.2 Future Prospects 153
7.3 A New Philosophy for Architecture Design 155
7.4 Parting Words to the Compiler Community 155

A Complete Evaluation Modes **157**
A.1 High-Level Compilation Modes 157
A.2 High-Level Interpretation Modes 160
A.3 Hybrid Modes 161

B Emulation Hardware **163**
B.1 DeepC Compilation Results and Functional Verification 163
B.2 Host Interfaces 165
B.2.1 WILD-ONE Host Interface 166
B.2.2 SLIC Host Interface 168
B.2.3 Transit Host Interfaces 169

C Data For Basic Results **171**
C.1 *Mode 0* Basic Results 171
C.2 *Mode I* Basic Results 173
C.3 Basic Sensitivity Results 173

D Data For Advanced Results **177**
D.1 *Mode 0* Advanced Results 177
D.2 *Mode I* Advanced Results 179
D.3 Advanced Sensitivity Results 179

E VPR Data and FPGA Layouts **183**
E.1 Input Data and Results 183
E.2 Wire Length Distribution and Layouts 183

List of Figures

1-1 Convergence of progress in compilers and CAD 18
1-2 DeepC compiler usage and flags 20
1-3 My thesis for compiling without instructions 21
1-4 Example of combinational function specialization 22
1-5 Example of storage register specialization 23
1-6 Example of control structure specialization 24
1-7 Example of memory mechanism specialization 25
1-8 Example of communication mechanism specialization 25
1-9 Specialization in time 26
1-10 Specialization in space 27
1-11 Summary of previous Raw Benchmark results 31

2-1 Matrix multiply inner loop 42
2-2 SPARC assembly for inner loop of integer matrix multiply 43
2-3 DLX pipeline 43
2-4 Xilinx XC4000 interconnect 44
2-5 Xilinx XC4000 CLB 45
2-6 Basic evaluation modes 46
2-7 Evaluation dominos 47
2-8 Review of denotational semantics 48

3-1 Specialization of combinational logic 53
3-2 Specialization of combinational adders 54
3-3 IEEE 32-bit floating point format 55
3-4 Example of eliminating a logical OR 56
3-5 Example of address specialization 57
3-6 A subset of transfer functions for bi-directional data-range propagation . . . 61
3-7 Example of turning a register into a wire 62
3-8 Sample C code illustrating bitwidth analysis 63
3-9 Example for bitwidth sequence detection 63
3-10 Simplified control flow of a von Neumann Architecture 65
3-11 Controller and datapath 66
3-12 Structure of generic while program (in Verilog) 67
3-13 Incrementer PC 68
3-14 Zero-hot PC 68
3-15 Forward, cycle-driven, critical-path-based list scheduler 71
3-16 Loop nest with three cycle body 72
3-17 Transitions for loop nest with three cycle body 72

3-18 IF-THEN-ELSE construct . . . 73
3-19 State transitions for IF-THEN-ELSE construct . . . 73
3-20 Predication example: IF-THEN-ELSE construct . . . 74
3-21 Predication example: Verilog after eliminating control flow . . . 74
3-22 Another predication example . . . 74
3-23 Another predication example: resulting Verilog . . . 74
3-24 Procedure call example . . . 75
3-25 While Program with procedure call (in Verilog). . . . 76

4-1 Memory specialization example . . . 81
4-2 Logic for partial unification example . . . 84
4-3 Partial unification example . . . 84
4-4 Equivalence class unification . . . 86
4-5 Modulo unrolling in two steps . . . 87
4-6 Tiled architecture . . . 88
4-7 Example of a static router . . . 89
4-8 Example of router operations . . . 92
4-9 Global binding example . . . 93
4-10 Space-time scheduling example . . . 94
4-11 Local binding example . . . 95

5-1 DeepC compiler lineage . . . 98
5-2 Overview of DeepC compiler flow . . . 99
5-3 Steps of Parallelization Phase . . . 100
5-4 Steps of Space-Time Scheduling Phase . . . 102
5-5 Steps of Machine Specialization Phase . . . 105
5-6 Steps of CAD Tool Phase . . . 108
5-7 VirtuaLogic Emulator . . . 111
5-8 A newer generation VirtuaLogic Emulator . . . 111
5-9 Simulation by direct execution on a workstation . . . 112
5-10 Simulation with the Raw Simulator . . . 113
5-11 Simulation with HaltSUIF . . . 114
5-12 RTL simulation . . . 115
5-13 RTL power estimation . . . 116

6-1 *Mode 0* versus *Mode II* . . . 121
6-2 Runtime comparisons between DeepC, Raw, and an UltraSPARC . . . 122
6-3 IBM SA-27E ASIC non-memory power and area . . . 123
6-4 *Mode I* versus *Mode II* . . . 124
6-5 Speedup, LUT and register area for Virtex-E-8 . . . 125
6-6 Absolute and relative power reported by the Xilinx Power Estimator 1.5 . . . 126
6-7 Energy and energy-delay comparison . . . 126
6-8 Minimum track width to place and route each benchmark . . . 127
6-9 adpcm layout and wire length distribution . . . 128
6-10 Components of cycle count speedup . . . 129
6-11 Benefits of bitwidth reduction in *Mode 0* . . . 130
6-12 *Mode 0* performance normalized to one Raw tile. . . . 134
6-13 *Mode 0* speedup . . . 135

6-14 Performance as a function of LUT area. 137
6-15 Performance as a function of power 138
6-16 Performance as a function of total energy 139
6-17 Clock frequency versus number of states 141
6-18 Long-distance messages 142
6-19 Jacobi inner loop before and after blocking transformation 142
6-20 Improvements after blocking for sixteen-tile Jacobi 143
6-21 Performance improvements after increasing communication ports 145
6-22 Improvements from overclocking the routing network 146
6-23 Sensitivity to variations in network speed 147
6-24 Grand comparison of performance as a function of non-memory area 149
6-25 Grand comparison of performance as a function of non-memory energy . . . 151

A-1 Complete evaluation dominos 158
A-2 Dominos arranged on the corners of a cube 159

B-1 Gatecount after synthesis to VirtuaLogic Emulator 164
B-2 Verification with a logic emulator 165
B-3 Host interface to emulator 166
B-4 Annapolis Micro Systems WILD-ONE with custom connector card 167
B-5 Back of WILD-ONE 167
B-6 Components of host interface 168
B-7 SLIC interface 169

E-1 VPR architecture file, virtex.arch 185
E-2 Continuation of VPR architecture file 186
E-3 Settings for VPR 186
E-4 adpcm wire length distribution 187
E-5 bubblesort wire length 188
E-6 convolve wire length 188
E-7 histogram wire length 188
E-8 intfir wire length 188
E-9 intmatmul-v1 wire length 189
E-10 jacobi wire length 189
E-11 life-v2 wire length 189
E-12 median wire length 189
E-13 mpegcorr wire length 190
E-14 parity-v1 wire length 190
E-15 pmatch-v1 wire length 190
E-16 sor wire length 190
E-17 adpcm layout 191
E-18 bubblesort layout 192
E-19 convolve layout 193
E-20 histogram layout 194
E-21 intfir layout 195
E-22 intmatmul-v1 layout 196
E-23 jacobi layout 197
E-24 life-v2 layout 198

E-25 median layout 199
E-26 mpegcorr layout 200
E-27 parity-v1 layout 201
E-28 pmatch-v1 layout 202
E-29 sor layout 203

List of Tables

5.1 Heuristically determined timing estimates 103
5.2 Simulators 110

6.1 Description of DeepC experimental components 118
6.2 Other experimental tools 118
6.3 DeepC Benchmark Suite 119
6.4 Sensitivity of LUT area to specialization flags 132
6.5 Sensitivity of register bit area to specialization flags 132
6.6 Sensitivity of clock frequency to specializations 132
6.7 Minimum track width 144
6.8 Assumptions and estimates for important parameters 148

A.1 Complete evaluation modes with examples 157

B.1 VMW gate count 164

C.1 Cycle counts 171
C.2 Power and area estimates 172
C.3 Synthesis to Xilinx Virtex-E-7 process 173
C.4 Synthesis results after place and route to Xilinx Virtex-E-8 174
C.5 More synthesis results after place and route to Xilinx Virtex-E-8 174
C.6 Energy and energy-delay comparison 175
C.7 Cumulative contribution of specializations to cycle count reduction 176
C.8 Synthesis to IBM SA27E process without bitwidth analysis 176

D.1 Multi-tile cycle counts with and without router specialization 177
D.2 Estimated total cell area for IBM SA-27E 177
D.3 Estimated total power for IBM SA-27E 178
D.4 Clock frequency after parallelization 179
D.5 Number of LUTs 179
D.6 Number of registers 180
D.7 Estimated switching activity 180
D.8 Estimated power for Xilinx Virtex-8 180
D.9 Number of states used in scatter plot 181

E.1 VPR statistics 184
E.2 Expected value of 2-pin net lengths 187

Chapter 1

Introduction

> *Hypercomputer systems amalgamate sets of FPGA chips into a generally-useful critical mass of computing resource with variable communication and memory topologies to form what the company believes is the first massively-parallel, reconfigurable, third-order programmable, ultra-tightly-coupled, fully linearly-scalable, evolvable, asymmetrical multiprocessor.*
>
> — *from the website of StarBridgeSystems.com, a reconfigurable computing startup in Utah.*

The above cold fusion-like description by Star Bridge Systems may be over-exuberant; however, when compared to traditional computer architectures, systems of Field Programmable Gate Arrays (FPGAs), and more generally Gate Reconfigurable Architectures[1], *are* fundamentally different: they have no pre-ordained instruction set. That is, the GRA software interface is lower than computer instructions — it is bit-level, exposes a parallel geometry of gate-level logic and wiring, and supports multiple clocks at user-determined frequencies.

Lack of an instruction set is both a shortcoming and an advantage. On the one hand, without an instruction set, the only current users of GRAs are hardware designers who have the ability to handcraft applications at the *Register Transfer Level* (RTL). In contrast, any technologist can program a traditional computer; high-level programming languages and efficient compilers to support them are abundant and free. On the other hand, without an instruction set, users are relieved of the obligations, obstructions, and entanglement of decades of binary instruction compatibility; at the instruction level, users can easily adapt and "custom-fit" the hardware for each application. This flexibility is a result of gate- or logic-level programmability. Furthermore, silicon densities are approaching the extremes of physical limits. Consider future architectures based on quantum dots [90], chemically-assembled nanotechnology [62], and the *in vivo* digital circuits of Weiss et al. [139]. At these extremes, the simplicity of gate reconfigurable architectures will become a significant advantage over their instruction-programmable counterparts.

[1]This dissertation uses the term Gate Reconfigurable Architecture (GRA) to refer to the broad class of computing devices that can be modified at the gate or logic level. These devices have historically been likened to hardware that is capable of "rewiring" itself, although this claim is a stretch of the imagination. We restrict our study to the class of GRAs that is easily reprogrammable (SRAM-based FPGAs and not antifuse-based FPGAs).

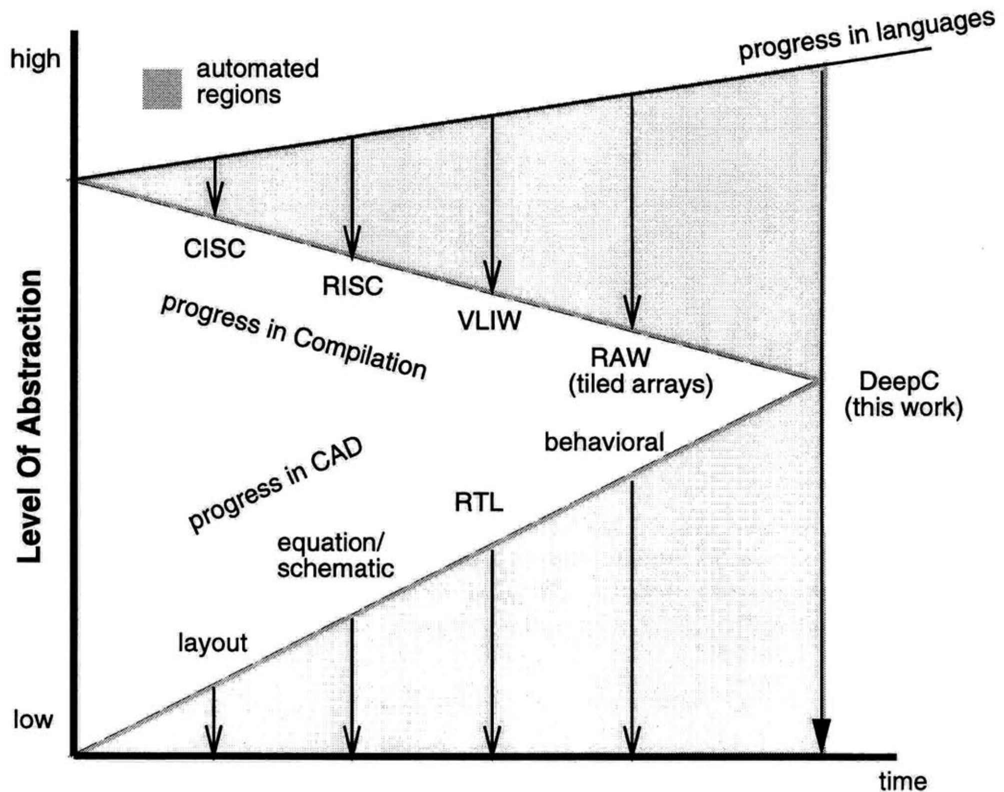


Figure 1-1: Convergence of progress in compilers and CAD

Improvements over time in compilers allow successively lower architectural abstractions levels, while CAD improvements raise the abstraction level of hardware design. DeepC lies at the convergence of these trends.

This work seeks to retain the advantages of gate reconfigurability while overcoming the limitations. The key to achieving this goal lies in the development of a compiler from high-level languages to gates, without the need for instruction interpretation — in essence repositioning the hardware-software dichotomy. Progress in compilers and progress in computer-aided design (CAD) tools are converging on a solution to this problem. The next section demonstrates this convergence and how my work completes the solution.

1.1 Compiler/CAD Convergence

Over time, advances in CAD tools have raised the hardware design abstraction while advances in compilers have allowed the abstractions for computer architecture to be lowered (Figure 1-1). The top line shows the continuing ascent of high-level programming languages. The arrows descending represent compilation to target machine architectures. The first architecture is a Complex Instruction Set Computer (CISC), followed by a Reduced Instruction set Computer (RISC) and a Very Long Instruction Word (VLIW) machine. While CISC architectures contain a large set of instructions, RISC architectures include only the most frequent instructions, enabling a simpler and therefore faster implementation. A VLIW approach further transfers complexity from hardware to software, processor

to compiler, by performing instruction scheduling in software. This approach leads to a simpler, more efficient, easier-to-design processor. Thus, as technology advances from one point to the next, the abstraction level is lowered and the compiler's work is increased.

The last instruction-level architecture is a Reconfigurable Architecture Workstation (Raw) [136], a wire-efficient architecture that interconnects tiles to scale with increasing transistor densities. Raw exposes even more hardware detail, including wires, the spatial arrange of function units, and the on-chip power grid (RawICE). Raw shares a common theme with previous architecture improvements: a lower target machine abstraction permits a simpler architecture, with higher performance, at the cost of a more complex compiler.

In contrast to descending high-level compiler targets, CAD tools are raising the abstraction level for designers, allowing them to do more with less. The first CAD tools raised the hardware level of abstraction from a knife and vellum to automatic layout tools. Designers with more than a few transistors to manage soon needed the next level: logic equations and schematic capture. Following this level are language-based abstractions (Verilog, VHDL) that support RTL design. The last stand-alone CAD level is the behavioral level, represented by languages such as Behavioral Verilog. This level enables an algorithmic description of hardware, but is much lower than high-level computer languages — it does not include high-level mechanisms such as complex data structures, pointers, and procedure calls.

In Figure 1-1, compilation and CAD progress converge at the point labeled DeepC. The system in this work, pronounced "Deep Sea", lies here. All the ideas, premises, and arguments that support my thesis are embodied in this working compilation system. DeepC spans the abstraction gap between compilers and CAD tools. DeepC also integrates existing compilers, CAD tools, and simulators. To illustrate the depth and functionality of DeepC, which internally contains over 50 passes, Figure 1-2 shows the usage and feature flags. I do not discuss these features now — you will learn about them as you read this dissertation and study the resulting performance metrics.

Closing the abstraction gap permits high-level compilation to GRAs. But why close the gap? Why compile to gates? The advantage of gate-level computing, without an instruction set, is that architectural structures can be matched to the algorithm, resulting in cheaper, faster, smaller, and lower power implementations. Furthermore, these implementations will more easily scale to future technologies. Although I demonstrate cases where gate reconfigurable architectures outperform traditional architectures, my goal is not to prove that gate reconfigurable computers are *better* than other classes of machines. Instead, my goal is to show *how* to program them.

1.2 My Thesis: How DeepC Closes The Gap

The topic of this dissertation is *how* DeepC closes the abstraction gap, *how to* compile without instructions, and *how* such compilation relates to the traditional mechanisms of instruction set architectures. My solution is to *specialize* architectural mechanisms. Specialization is a general technique in which a program or system is customized for a given input, state, or environmental condition. Given this technique, a precise statement of my thesis (Figure 1-3) is: *specialization of architectural mechanisms with respect to an input program is the key to efficient compilation of high-level programs to gate-reconfigurable architectures.* This statement is broken down as follows. *Specialization of architectural mechanisms* refers to a compilation approach in which mechanisms more traditionally used for interpreting processor instructions are instead customized. Section 1.3 discusses why specialization is

```
Usage: deepc <flags> <infile> [ <outfile> ]

Flags:
 -nprocs       number of tiles
 -O<i>         optimization level (1-6)

Feature Flags:
 -fno-addr    address calculation optimization
 -fbit        bitwise optimization
 -fbeopt      additional backend porky optimization passes
 -fbram       enable Virtex on-chip block RAM synthesis
 -fno-ecmap   equivalence class mapping
 -ffeopt      additional frontend porky optimization passes
 -floop       loop invariant optimization
 -fmacro      macro instruction identification
 -fmeopt      additional machine porky optimization passes
 -fopt        all additional frontend/backend/machine optimization passes
 -fno-part    partitioning logic into tiles
 -freg        enable register allocation
 -froute      enable router synthesis
 -fno-vliw    vliw scheduling

 -cports       <i> maximum number of ports per communication channel
 -cportwidth   <i> data width of comm ports
 -imul         <i> maximum number of integer multiplier (per tile, default 1)
 -max-unroll   <i> unroll inner loops at most <i> times total
 -max-width    <i> maximum datapath width (default 32)
 -min-unroll   <i> unroll inner loops at least <i> times total
 -min-unrollx  <i> unroll inner loops at least <i> times per processor
 -mports       <i> maximum number of ports per memory
 -netmult      <i> factor to overclock routing network
 -nmems        <i> number of memories, if larger than number of processors

Target Flags:
 -t soft      execute in software on localhost
 -t raw       raw multiprocessor compilation
 -t rawsim    raw multiprocessor simulation
 -t haltsim   compiled-code basic-block level simulation
 -t rtlsim    rtl verilog simulation
 -t syn       synthesize
 -t gatesim   post-synthesis simulation
 -t virtex    Xilinx Virtex place and route
 -t vpr       VPR place and route
 -t vwires    virtualwires multi-FPGA compilation and timing re-synthesis
 -t vlerun    load and run on IKOS VirtuaLogic Emulator

Synthesis Flags:
 -fsynopt          optimize RTL synthesis
 -fsynarea         area optimizations
 -max-area         target gate/cell area per tile (parsyn)
 -fsynspeed        speed optimizations
 -clock-speed      target clock speed in MHz
 -fsm-style <s>    fsm enoding style: none | binary | one_hot | gray
 -resource <s>     resource allocation: none | area | constraint_driven (default)
 -technology       target hardware: ibm27 | xc4000 | vmw | vpr | virtex (default)
 -part             part number for Xilinx
```

Figure 1-2: DeepC compiler usage and flags

Specialization of traditional architectural mechanisms with respect to an input program is the key to efficient compilation of high-level programs to gate-reconfigurable architectures.

Figure 1-3: My thesis for compiling without instructions

a good idea. *With respect to an input program* refers to the input high-level program that is responsible for the customization conditions. That is, the invariant or static conditions applied during specialization are a function of the input program. Finally, *efficient compilation* refers to a non-naive compilation method that optimizes performance — not just a theoretical demonstration that compilation is possible. Consider that emitting a logic-level description of a processor, along with instructions, is a valid, but inefficient, compilation strategy.

Given this thesis statement, I use specialization theory to show what it means to compile without instructions and then exemplify techniques for specializing both basic and advanced architectural mechanisms. The following contributions elaborate further.

1.2.1 Summary of Contributions

In this dissertation, I make the following contributions:

- I develop a theory of evaluation modes that highlights the differences between traditional processors and gate reconfigurable architectures as a difference between compiling and interpreting at different abstraction levels. I formulate my thesis theoretically with specialization terminology.
- I present approaches to specializing the basic mechanisms of von Neumann Architectures. As a part of these approaches, I have developed, with others, a bitwidth specialization technique and have applied it to reconfigurable architectures. My approach also specializes the control flow of imperative languages.
- I present approaches to specializing the advanced mechanisms of distributed architectures. This approach integrates compiler passes for memory disambiguation and pointer analysis; it also includes a space-time scheduling strategy for tiled architectures. My new scheduling strategy, developed with others, merges the scheduling approaches from high-level synthesis with static routing techniques from parallel processing, digitizing both space and time at the logic level.
- I contribute an implementation of the ideas in the dissertation in a single compilation system, the *DeepC Silicon Compiler.* DeepC is capable of compiling C and FORTRAN programs into parallel hardware in ASIC (Application Specific Integrated Circuit) and FPGA technologies.
- I present the Deep Benchmark Suite and evaluate the results obtained when compiling this suite with DeepC. This suite consists of fourteen programs; seven that can use significant parallel hardware. My results include general metrics such as area, power, and latency, as well as FPGA-specific metrics such as minimum track width. My results also include a grand comparison of four alternate evaluation modes.

1.3 Motivation for Specialization

Why specialize? There is a single overriding reason for specializing the instruction-level architecture in gate reconfigurable systems: efficiency. Since a gate reconfigurable architecture is already programmable, adding another interpretative layer will slow down the system by about an order of magnitude. The system will also be larger, more expensive, and power hungry. The only way to avoid these overheads is to specialize architecture-level mechanisms.

This section continues motivating specialization in several ways. First, it demonstrates specialization of five sets of mechanisms commonly found in digital systems. Second, it gives examples of the spatial and temporal application of specialization to consumer systems. Finally, it addresses the counterarguments to my claim, some that can be summarily dismissed and others that remain a constant tension throughout this work.

1.3.1 Five Classes of Mechanisms to Specialize

This section previews five classes of architectural mechanisms that can be specialized: combinational functions, storage registers, control structures, memory mechanisms, and communication mechanisms. These are presented in detail in Chapter 3 and Chapter 4.

Combinational Functions Section 3.1 focuses on specialization of combinational functions. If certain inputs of a Boolean function are known ahead of evaluation, then those inputs can be substituted and the remaining logic simplified. Compile-time knowledge can be used to eliminate gates, resulting in smaller, faster, lower power circuits. Consider Figure 1-4. In (1), one of three inputs to an AND gate is determined at compile time. This gate is then specialized into a two-input gate. In (2), static analysis of data dependencies determines that one ADD is immediately followed by a second ADD. Hence, specializing forms a three-input adder.

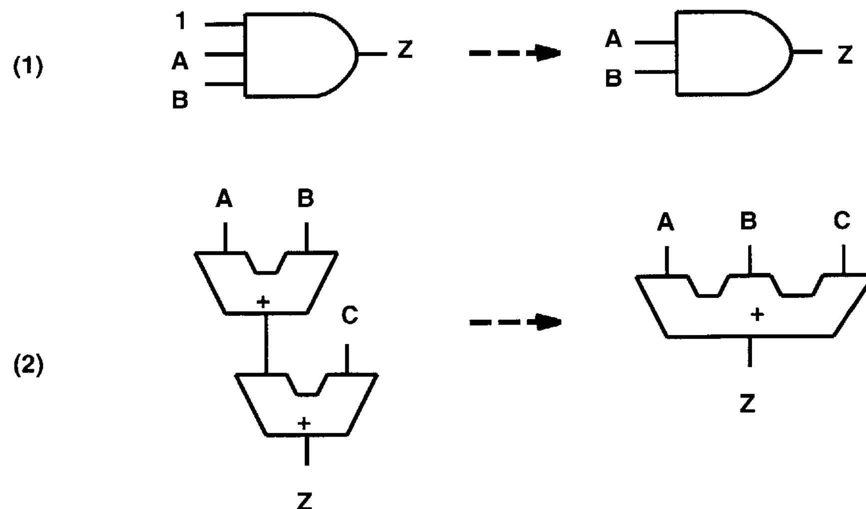


Figure 1-4: Example of combinational function specialization

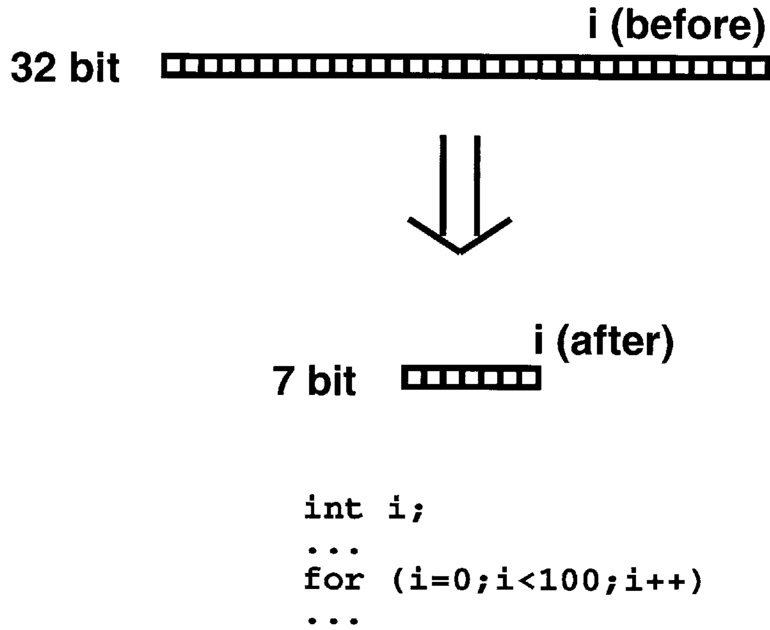


Figure 1-5: Example of storage register specialization

Storage Registers Section 3.2 focuses on specialization of storage registers. In many cases, a register can be converted into a wire. In other cases, register bitwdith can be reduced, as in Figure 1-5. The integer i, when declared in the C programming language, often has thirty-two bits by default. However, after analysis of i's use, the register can be specialized down to seven bits. In a general processor, all 32 bits are needed, even though a large number are unused.

Control Structures Section 3.3 focuses on specialization of control structures. The input program is used as static data to specializing von Neumann control flow mechanisms. To translate imperative control flow into a controller and datapath, a program counter register is introduced to determine the state of the machine. In this case, a branch instruction implies manipulation of the program counter beyond the default of advancing to the next state. The datapath performs the appropriate calculations on each clock cycle. Consider the loop in Figure 1-6. The control structures are specialized into state machines that orchestrate the combinational functions and storage registers in the body of the input program.

Memory Mechanisms Section 4.1 focuses on specialization of memory mechanisms. The goal is to decompose the default *main memory* into many small address spaces. Physically, these smaller spaces can occupy small local memories, allowing computation to be migrated close to data. Consider the example in Figure 1-7, in which storage cells are formed into a memory with a selecting multiplexer. If some data accesses can be disambiguated, this memory can be specialized into separate memory banks, each accessed only by local logic.

Communication Mechanisms Section 4.2 focuses on specialization of communication mechanisms, introducing *tiled architectures.* Each replicated, identical tile contains a processing core, data RAM, and a router that interconnects neighboring tiles. Tiling enables

```
for (i = 0; i< 100; i++) {
  for (j = 0; j < 100; j++) {
    [body1]
    [body2]
    [body3]
  }
}
```

```
...
case (pc)
  0: begin
     i = 0;
     pc = 1;
  end
  1: begin
     j = 0;
     pc = 2;
  end
  2: begin
     j = j + 1;
     [body1]
     pc = 3;
  end
  3: begin
     [body2]
     testj = (j < 100);
     pc = 4;
  end
  4: begin
     [body3]
     pc = testj ? 2 : 5;
  end
  5: begin
     i = i + 1;
     testi = (i < 100);
     pc = testi ? 1 : 6;
  end
endcase
...
```

Figure 1-6: Example of control structure specialization

A loop nest with a three-cycle body (left) and the resulting state machine in Verilog (right). In each state, a program counter (pc) is updated appropriately. The body of the program is scheduled across these states.

geometry management. Consider Figure 1-8, in which function units are conventionally arranged on a bus, with all-to-all connectivity. In the specialized arrangement, connections are statically scheduled onto a two dimensional topology that more closely matches physical constraints.

DeepC includes approaches for specializing all the above mechanisms. The following section overviews some example systems in which specialization techniques may be well suited.

1.3.2 The Big Picture: Specialization in General

This section demonstrates the benefits of specialization from a broader perspective. Skip this section if you are already comfortable with the concept of specialization. Recall that specialized systems, in comparison to unspecialized systems, are expected to be cheaper, faster, smaller and lower power. While in this work specialization is applied to a specific system layer (the architecture of a computer), in the big picture, specialization can be applied to all aspects of a system, from the subsystems in a chip to the devices in a network. The following examples demonstrate specialization in time and specialization in space of wireless systems.

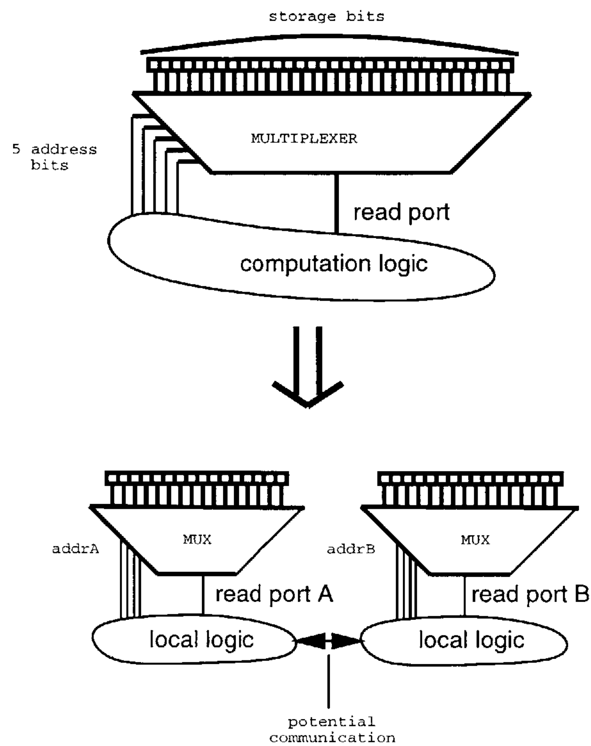


Figure 1-7: Example of memory mechanism specialization

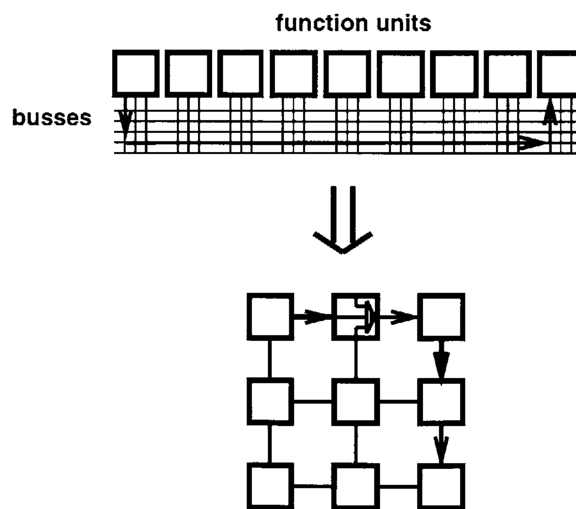


Figure 1-8: Example of communication mechanism specialization

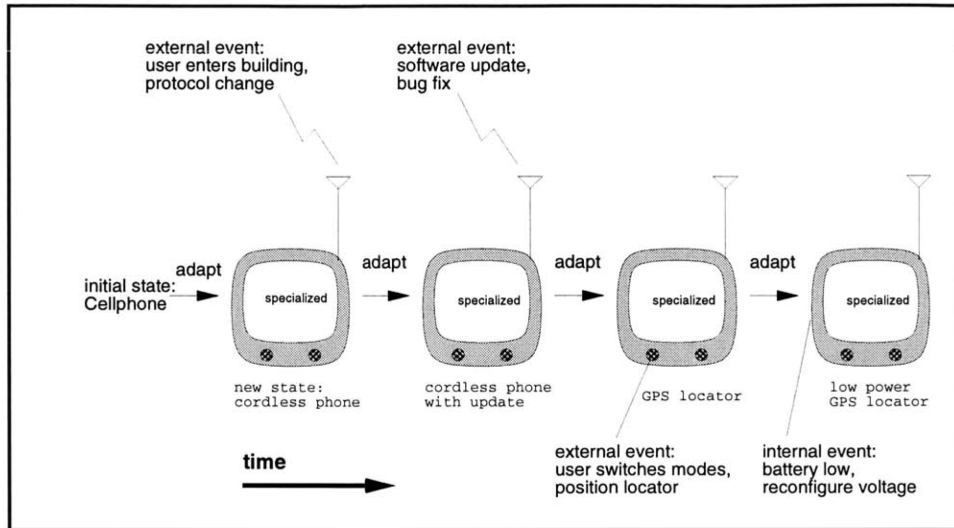


Figure 1-9: Specialization in time

Each state change is an adaptation to a new specialized machine. Briefly, the states are cellular phone, cordless phone, cordless phone with update, position locator, and low power position locator. Each state change is triggered by an internal or external event.

Specialization in time is similar to biological adaptation. The scenario in Figure 1-9 demonstrates a handheld device that evolves over time, from being a cellular phone, to being a low-power positioning system. Each major state change is stimulated by an external or internal event. The first triggering event occurs when an assumed user, holding the device, enters a building: the communication mode switches to that of a cordless phone. Given the new state, the device specializes its internal configuration to best communicate with wireless transceivers in the building. In the second adaptation, the device responds to a remote request to upload a software bug fix, perhaps improving the security protocol. Next, the user decides to locate his or her position by switching to position locator mode. With specialization, the circuitry previously encoding his or her voice is reconfigured to run position location software. Finally, the device detects an internal event (low battery), and reduces the voltage. In low voltage mode, a configuration that updates the position less frequently is loaded, resulting in yet another specialization of the device. At any given time in the scenario, the device is highly specialized. However, the device considered over a longer period is general. In contrast, consider the complex requirements of a single device trying to be all these at all times: multiple radio circuits, a fast *and* a low-power processor, and multiple custom chips for special functions. Moreover, the bug fix would require exchanging the entire system for new hardware! A system with adaptable components can thus benefit significantly from specialization.

Specialization in space naturally arises in collaborative systems. Figure 1-10 shows a collaborative bodynet scenario in which a woman is wearing three devices while fitness walking: an earphone/microphone, a heart monitor, and the handheld device from the previous scenario. A fourth device, a wireless router, is on a nearby telephone pole. Because

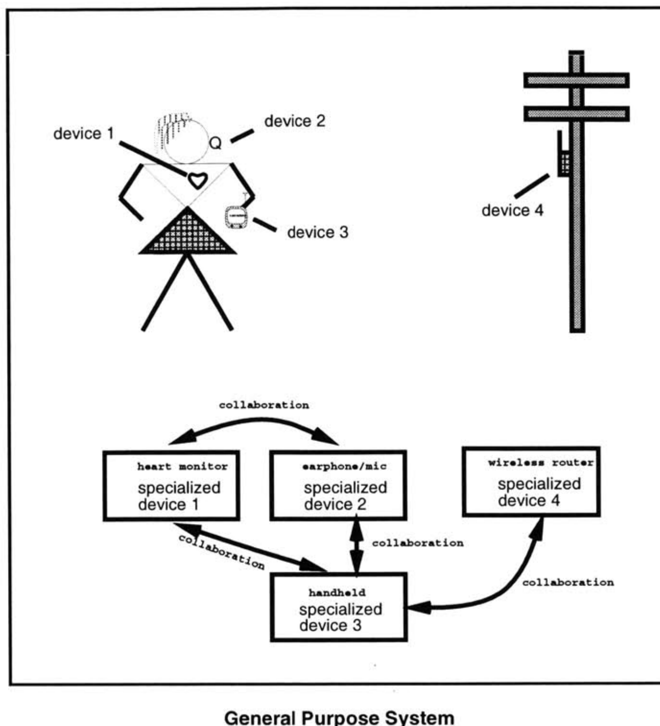


Figure 1-10: Specialization in space

Each device in space is specialized to a different function. Device 1 is a heart monitor; Device 2 is an earphone/microphone; Device 3 is a handheld PDA with a display and entry mechanism; Device 4 is a wireless router. These devices communicate over a local wireless network (bodynet), forming a more general wearable system.

these devices can collaborate, they are specialized. Only one device needs to be connected to the Internet in order to store a heart rate graph, generated during the exercise period, on her private website. The heart rate monitor can communicate with the earphone/microphone device to signal whenever she exceeds her target aerobic range. The handheld device can be used to enter a workout program and view results in real time. Each device in this scenario is highly specialized; however, the collaborative system comprising the devices is general. In contrast, consider the expense and bulkiness of a system in which each device is a general-purpose workstation. Even using popular handheld PDAs (personal digital assistants), the total system would still be overly cumbersome. A system with collaborative components is thus also an excellent candidate for specialization.

Relating these examples back to the topic of specializing architectural mechanisms, temporal and spatial specialization play a big role in high-level compilation to gate reconfigurable architectures. The compiler needs to adapt spatial regions of the architecture to evaluate different parts of an application collaboratively. As the mix of applications changes, the compiler needs to adapt, or re-specialize, the architecture over time. Architectural specialization enables application-level adaptation with minimal sacrifice in cost, performance, and power. Computing systems designed with these techniques can adapt to users rather than force users to adapt to the computer.

1.3.3 Counterarguments

This section presents thesis counterarguments as a list of frequently asked questions (a FAQ). You may wish to return to these questions, with more background, as they arise in your mind during the course of reading the following chapters.

1. **The applicability question:** Given the universality of processors, why use GRAs at all?
   Answer: First, this dissertation compares many modes of evaluation. Depending on the application and circumstances, any one mode may have characteristics superior to others. The common comparisons are cost, speed, size, and power, but non-technical factors may be just as important. Second, in Chapter 6, most of the benchmarks studied perform better on GRAs than processors. Thus, in many cases a GRA is a better match for the application at hand. But GRAs are not always better.

2. **The market-inertia question:** Given the business inertia of the "killer micro", and the billions invested in its development, won't the processor eventually obsolete alternatives?
   Answer: First, a substantial fraction of the billions invested has been invested in semiconductor fabrication, and this technology benefits approaches that compete with processors, including GRAs, as much as it does processors. Second, conventional processor microarchitects claim to be nearing the "end of the road" (Vikas Agarwal et. al [1]) given physical constraints, and these architects are adopting more configurable approaches (multimedia extensions in Pentium, ALU-chaining in Pentium 4, and static scheduling of function units in Itanium). Hence, the solutions to GRA compilation problems will likely apply to future processor architectures as well.

3. **The lack-of-coarse-grain-features question:** Aren't gate-level architecture hugely disadvantaged without coarse-grain function units and pipelined datapaths?
   Answer: FPGAs already embed coarse-grain functions such as multipliers without giving up gate-level programmability for the majority of the device. However, the advantage of the gate-level approach is that generic coarse-grain functions can be specialized to a particular use. A prime example is multiplication by a constant value. In this work, I show how to specialize common mechanisms found in coarser-grained architectures to overcome fine-grain disadvantages. In fact, the disadvantages are only important if specialization is not used – the contrapositive of my thesis.

4. **The applications-are-not-bit-level question:** But do applications really have many bit-level operations?
   Answer: First, architectural mechanisms are bit-level – consider combinational logic, state machines, registers, branch prediction bits, cache tags. Thus, specialization of these mechanisms results in many bit-level operations. Second, everyday applications do have many bit-level operations. Researchers at MIT, Berkeley, and Princeton (see Section 1.4.4) have applied static and dynamic analysis to discover bit-level operations in benchmarks such as the SPEC benchmark suite. In addition to narrow bitwidths, constant bit values can be discovered or predicted. Third, applications with bit-level parallelism are often coded with bit-vectors that retrofit traditional ISAs for bit-level computation.

5. **The virtualization question:** Aren't GRAs confined to running a single application? What about multithreading and process virtualization?
   Answer: This is a valid counterargument to replacing a workstation processor with a GRA in a multi-programmed, multi-user environment. There is active research in reconfigurable computing to overcome this problem (see Caspi et al. [38]), however there are also many applications that do not need virtualization. These applications will benefit from the high-level compilation techniques in this thesis.

6. **The appropriate-language question:** Are imperative languages like C an appropriate choice for hardware design?
   Answer: My view is that languages are for describing algorithms, not hardware. The language choice should be based on usability and human-centric aspects, not the eventual target

architecture. There are millions of C programs, thus a strong justification must be made for the use of an alternate language. Compilation technology must exist to support the constructs of C before languages like C++ and Java can be considered. In any case, this dissertation focuses on mechanisms rather than particular language idiosyncrasies, thus my case is for higher-level languages rather than for a particular language.

7. **The existing-industrial-compiler question:** Is industry already compiling C to gates?
Answer: There are several companies, introduced in the following related work section, building compilers from C to RTL-level hardware. Some techniques proposed here are also implemented in their schemes. Most of these companies are at a startup or research phase, with development efforts paralleling the work on which this dissertation rests. Companies rarely publish the full inner workings of their system for academic review or commit their approaches to the body of scholarly knowledge. Industry standards for generating gate-level hardware from a high-level are beginning to be considered, but the approach is not generally accepted or widely adopted by engineers.

8. **The unproven-research-passes question:** But doesn't the DeepC approach rely on new compilation techniques such as pointer analysis, bitwidth analysis, and static scheduling that are unproven in industry?
Answer: The national compiler infrastructure (including projects such as SUIF [140]) has been created to enable rapid sharing of compiler passes among researchers. Having access to new techniques as soon as they are developed is an advantage, not a disadvantage. Most of these techniques are used by multiple research projects and are quickly proven. But this is a valid counterargument – problems in new techniques have been inherited by DeepC. An example problem is the excess unrolling caused by one of the parallelizing techniques leveraged — Section 6.6.1 studies this problem in depth.

9. **The large-program question:** But can large programs, for example databases, be compiled to gate-level state machines? Won't the resulting state machine be prohibitively large?
Answer: For many small programs such as filters, software radios, software routers, and encryption algorithms this question is not an issue. Existing FPGA applications, expressed at a high-level, are small programs. Large programs can be handled in several ways. First, even though a program may be long, the inner loop is usually much smaller (maybe ten percent or less), allowing softcore processors or simple instruction ROMs to encode the non-critical program paths. Second, a hybrid system can continue to leverage traditional processors for unwieldy procedures while retaining the GRAs speed and power benefits for inner loops. Finally, dynamic reprogrammability, or a successful virtualization technique, can allow circuits to be cached and replaced over time to accommodate larger or multiple critical regions. To partially address these issues, this dissertation explores the effect of state machine size on clock speed — the proposed specialization techniques help by reducing the total states and the logic needed in each state.

10. **The slow-compiler question:** Aren't CAD tools too slow to be used as compilers for computing?
Answer: Because of the number of abstraction layers crossed and the optimization efforts expended when compiling to the gate-level, compile time is longer than when compiling to the instruction-level – hours instead of minutes. In a sense, a portion of the work spent synthesizing or manually designing a processor must be repeated by the compiler for each application. But this cost can be amortized across many application users. Moreover, gate-level compilation is parallelizable — regions can be synthesized and compiled in a divide-and-conquer fashion. Finally, this is a counterargument for FPGAs, not the approach in this work: faster and incremental compilation is an open problem in CAD for FPGAs. In fact, this dissertation speeds compilation by applying specializations at higher level and by partitioning the design into manageable *tiles*. Section 6.6.2 demonstrates that routers reduce the minimum FPGA track width and reduce the workstation memory needed for place and route.

11. **The specialization-versus-parallelization question:** Aren't the benefits from specialization just benefits from instruction-level parallelization?
    Answer: Specialization does often exposes parallelism, and parallel structures can often be specialized. However, specialization and parallelization are two separate concerns. As much as possible, Chapter 6 separates specialization gains and parallelization gains. The important point of this work is that the maximum performance for GRAs cannot be obtained without architectural specialization. Parallelization alone is not enough.

12. **The *reductio-ad-absurdum* question:** Your thesis is that specialization is the key to compilation. Is this a tautology?
    Answer: Without discipline, the compilation possibilities for crossing the abstraction gap from high-level languages to gates are overwhelming. But the architectural mechanisms of instruction-level processors have been studied for many decades and are well understood. Thus, specializing these mechanisms rather than trying to build a compiler from scratch is a smart strategy. The particular relation between these architectural mechanism and compilation to GRAs is intricate, and previous research has not identified or explained rigorously the important principles.

13. **The better-approach question:** Aren't there better approaches to architecture generation, maybe term rewriting systems [73], for example?
    Answer: There are many possible formalisms for manipulating architectural mechanisms, and alternate sets of mechanisms may be chosen (for example queues); however, in the end these general mechanisms must be specialized to the application at hand to overcome the interpretative overheads that result from their generality. For example, Marinescu and Rinard [97] convert conceptually unbounded queues to synchronous, pipelined circuits, with finite queues, as a function of specified applications. The advantages of DeepC's approach are that it accepts *familiar* imperative programs and targets *familiar* FPGA devices. Moreover, the intervening specializations are of *familiar* architectural mechanisms. The following related work section addresses many other complementary and competing approaches.

1.4 Previous Work

Besides culminating my own work, the contributions in this dissertation draw upon work in several disciplines of computer engineering. These disciplines are themselves interrelated and overlapping, including computer-aided design (CAD), computer architecture, and compiler and language implementation. The next section discusses how earlier work laid the foundation for this work.

1.4.1 My Previous Work

Logic emulation has been the most significant application of reconfigurable computing. A logic emulator enables hardware designers to functionally verify complex systems at a clock speed four to six orders of magnitude faster than software simulators. In previous work, developed with others, I invented a new technique for constructing FPGA-based logic emulators based on a technology called "Virtual Wires" [12]. Virtual Wires overcome pin-limitations in multi-FPGA systems by intelligently pipelining and multiplexing inter-FPGA signals. This technology has been commercialized as the VirtuaLogic Emulator (presently sold by IKOS Systems) and is the basis for the emulation system described in Section 5.3 and Appendix B. When programming at the gate-level, as a hardware designer, this technology enables one to treat a collection of FPGAs as a single, gigantic FPGA.

Dynamic Computation Structure [9] and Raw Benchmark Suite [10] work, developed with others, uses this emulation system to evaluate some general computing benchmarks,

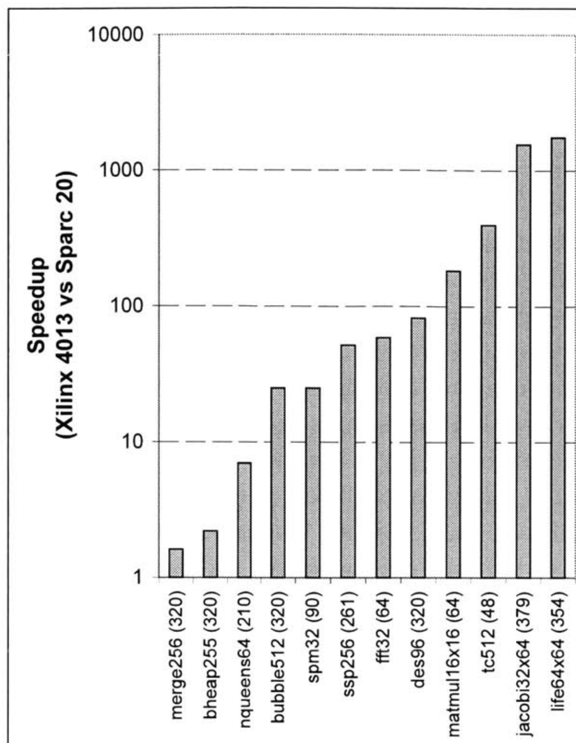


Figure 1-11: Summary of previous Raw Benchmark results

The parenthesis after each case contain the FPGA count — the part used is Xilinx XC4013. The comparison is versus a $50MHz$ SparcStation 20 (about the same vintage technology). Specific details for each case are in an earlier paper [10].

obtaining large speedups (Figure 1-11) when compared to a workstation. This comparison is between FPGAs and a workstation of the same vintage (circa 1996); however, both technologies have advanced exponentially, according to Moore's Law, since then. FPGA speedup ranges from about 2× to over 1000× faster than the workstation across benchmarks including mergesort, binary heap, nqueens, shortest path, FFT, DES, matrix multiply, transitive closure, jacobi, and the game of life. A subset of these benchmarks is also compiled with DeepC in Chapter 6.

To generate the Raw Benchmark Suite results, my colleagues and I leveraged Virtual Wires, in conjunction with behavioral synthesis (Section 1.4.2), to treat an array of FPGAs as a machine-independent computing fabric. This treatment enables application-specific, architecture-independent hardware design. However, this system implements a *generator*, not a compiler. Another major limitation is that design logic is not multiplexed onto the system — hardware area grows in proportion to problem size. This approach also does not support applications with significant memory requirements as it does not support embedded memory blocks in FPGAs. DeepC overcomes these limitations, and raises the design abstraction from Verilog to C.

The Raw Benchmark Suite motivated both the DeepC project and its sibling, the Raw project [136]. While DeepC automates all the transformations performed on these benchmarks, the compiler for Raw only automates the highest-level transformations. The Raw project eliminates the low-level transforms by upgrading the architecture to a coarse-grain,

multiprocessor-based system. While an improved architecture is a laudable goal, unless such improvements are widely adopted by the majority of FPGA vendors and new devices supplant the existing base of gate-reconfigurable architectures, the high-level compilation problem for FPGAs will remain important and unsolved. The Raw project is valuable for gate-reconfigurable computing in that the high-level transforms can be leveraged. For example, I have developed, with others, a space-time static scheduling algorithm for DeepC that is derived from a simpler Raw scheduler (Lee et al. [89]).

In [11], I discuss the first proof-of-concept version of DeepC. This work briefly mentions the application to gate-reconfigurable architectures and focuses on the problem of parallelizing programs into custom hardware. The major contribution of that work, which extends to this dissertation, is an approach for applying automatic parallelization technology for multiprocessors to the problems of silicon compilation.

Work with Stephenson et al. [127] extends DeepC 1.0 with a bitwidth analysis facility (Bitwise). Results show significant power, area, and latency savings. Those results are improved and included here with DeepC Version 2.0.

This dissertation is the culmination of all the previous work mentioned here. DeepC 2.0, the major system in this thesis, is a complete system integrating the concepts of several generation of previous systems: Virtual Wires, The Raw Benchmarks Suite, The Raw Compiler, DeepC 1.0, and the Bitwise Compiler. Evaluating DeepC indirectly evaluates the concepts in these systems as well. The next section continues with discussion of the external related work that motivated the DeepC system.

1.4.2 Computer-Aided Design (CAD)

A product of the Electronic Design Automation (EDA) industry, CAD tools automate the design of electronic systems. One of the principle goals in this area has always been to raise the abstraction level for designers. The highest abstraction level attained in CAD is the behavioral level. This section focuses on non-specific behavioral synthesis tools (for both FPGAs and ASICs), on high-level synthesis tools tailored for FPGAs, and on how both relate to DeepC.

Behavioral Synthesis

The goal of behavioral synthesis, or high-level synthesis, is to transform an algorithmic description into a digital system. A debated aspect of this goal is whether transforming the programming languages of the software world is a good idea (see Gupta et al. [67]). The focus of the debate is on which programming languages would make a good choice and whether architecture design, the task of hardware designers, can be automated.

The major advantage of behavioral synthesis over the more commonly used RTL synthesis is that behavioral synthesizers are capable of scheduling, or assigning, operations to clock cycles. However, both an RTL synthesizer and a behavioral synthesizer can perform resource binding (assignment of operations to physical function units). DeepC leverage some aspects of a behavioral compiler found in the Synopsys Behavioral Compiler [129]. However, because DeepC statically orchestrates the schedules of multiple communication state machines, it did not need behavioral scheduling.

Recent work in behavioral synthesis is focused on moving the abstraction level up to C, C++, or even Java [71]. In particular, the SpC pointer synthesis in Semeria et al. [121] is related to DeepC. While DeepC disambiguates pointer references and leverages pointer

analysis to maximize parallelism and to analyze bitwidths, SpC uses pointer analysis to determine the best encoding of resolved pointers. SpC leverages techniques from the CAD problems of state encoding and state minimization. Section 4.1.1 discusses these issues in more detail.

Earlier work in behavioral synthesis includes the optimization of address generation in Schmit et al. [117], which describes addressing schemes both similar to and more sophisticated than those in Section 3.1.2. Another early work, Filo et al. [54], studies interface optimization along with the scheduling of inter-process communication. A common goal with my work is to reduce synchronization by statically resolving communication schedules.

High-Level Synthesis to FPGAs

Reconfigurable computing experts have adopted high-level synthesis for FPGAs. Splash [59] and PAM [135] were the first substantial reconfigurable computers; systems like Teramac [4] followed later. As part of the Splash project, a team led by Maya Gokhale ported data-parallel C [100] to the Splash reconfigurable architecture. This effort was one of the first to compile programs, rather than design hardware, for a reconfigurable architecture. Although data-parallel C extended the language to handle bit-level operations and systolic communication, the host managed all control flow. Hardware compilation only operated within basic blocks of parallel instructions. This approach was ported to National Semiconductors processor/FPGA chip based on the CLAy architecture [61].

Programmable Active Memory (PAM) [135], designed at Compaq Paris Research Lab, connect to a host processor through memory-mapped I/O. The programming model treats the reconfigurable logic as a memory capable of performing computation. The design of the configuration for each PAM application is specified in a C-syntax hardware description language, but in this case, C was not used as a high-level programming language.

Beyond large reconfigurable computers, compiler projects included PRISM [124], which took functions derived from a subset of C and compiled them into an FPGA. The PRISM-I subset included IF-THEN-ELSE as well as for loops of fixed count; however, all code in functions assigned to hardware is scheduled to a single clock cycle. Hence, the PRISM-I system was limited to creating configurable *instructions*. The later PRISM-II subset added variable length FOR loops, WHILE, DO-WHILE, SWITCH-CASE, BREAK, and CONTINUE. Like PRISM, Transmogrifier C of Galloway et. al [57] is an elementary (4000 lines-of-code) compiler for a tiny subset of C (no multiplies, divides, pointers, arrays, structures, or recursion), extended with variable bitwidth types. Trasmogrifier's semantics assume a clock edge only at the beginning of a while loop or function call. In another compiler project, Peterson et al. [110] describe a partitioning and scheduling-based compiler targeted to multi-FPGA systems. Following function block partitioning, this system simultaneously schedules both intra-FPGA function unit and inter-FPGA communication using topology-sensitive resource pools. Other early projects include compilation of Ruby [66] - a language of functions and relations, and compilation of vectorizable loops in Modula-2 for reconfigurable architectures [138]. In several systems, high-level languages are used to specify the construction, and often placement, of hardware modules without invoking high-level algorithms to synthesize state machines and schedule operations. Example system using this approach, often termed module generation, include GAMA [32], Cynlib [44], SystemC [130], and JHDL [17]. Many favorite programming and specification languages have been claimed suitable for hardware description; consider the proposals for using ADA [50], Smalltalk-80 blocks [112], Haskell [21], Matlab [13], Java bytecode [35] and TRS [73] for hardware de-

sign. Space prohibits a detailed description of each proposed systems; in summary, they each contribute to high-level compilation, but over-emphasize the importance of the input language.

Research in hardware-software co-synthesis [68] has always been relevant for FPGA compiler research. This relevance is because FPGA systems commonly include a host computer or attach an FPGA to a processor as a function unit. For recent work, consider the CORDS project of Dick et al. [49], which targets distributed embedded systems that contain reconfigurable devices. CORDS relies on task graph scheduling and an evolutionary algorithm (CORDS breeds architectures!) to explore the high-level architectural design space for embedded application. This system also supports dynamic reconfiguration, in which the FPGA bitstream is re-loaded during the execution of a single application, even given real-time constraints. Both hardware/software partitioning and dynamic reconfiguration are orthogonal yet complementary to work in DeepC.

Recent projects building complete systems for compiling high-level languages into hardware are in industry: HP's PICO project [118], Synopsys's Nimble project [91], Celoxica's Handel-C [39], and C-Level Design's C2Verilog [126]. At the time of this writing, the PICO and Nimble projects are at the research stage while Celoxica and C-Level Design have working products that embody many concepts that support my thesis. For practical reasons, these systems often impose needless constraints on the high-level language. These constraints burden the programmer. For example, in Handel-C every statement executes in one clock cycle; the addition of a "par" statement forces the programmer to specify which statements are executed concurrently.

Improvements Over Previous CAD Tools

I improve on previous high-level synthesis work in several ways. Previous Virtual Wires work improved on the Splash and PAM systems by eliminating the domain-specific interconnect-topology between FPGAs; DeepC adopts and extends this virtualization. For example, the Splash system was designed to function as a SIMD array, not as a general-purpose system. Next, reconfigurable systems often rely on a host computer, or attached processor, to perform operations not considered "suitable" for hardware implementation. DeepC instead compiles entire programs into hardware and generates scheduled state machines. DeepC integrates high-level parallelizing transformations, space-time scheduling, architecture specialization, and logic synthesis. Rather than extending the language to allow the programmer to specify bit-level functionality or parallelism, DeepC detects parallelism and bitwidths from ordinary C programs. Furthermore, the focus of DeepC is not on the C language, but instead on the high-level features of C — pointers, arrays, control flow, and data types. DeepC is distinguished by its specialization of advanced architectural mechanisms, including the distributed memory and tiled communication mechanisms that are featured in the Raw architecture. Finally, the evaluation in this dissertation is more comprehensive than any previous work, comparing to both processor and custom alternatives, and analyzing the sensitivity of each optimization rather than only presenting apples-to-oranges speedup comparisons with a workstation.

1.4.3 Computer Architecture

Although DeepC is a compiler, it specializes architectural mechanisms. Thus, there is related work in computer architecture as well. In fact, many computer architectures support

specialization of some architectural mechanisms. The following discussion of these mechanisms is divided into three sections — advanced features added to FPGAs, FPGA-inspired features in new architectures, and exposed mechanisms in other architectures.

Advanced Architectural Mechanisms in FPGAs

DeepC specializes two advanced architectural mechanisms previously studied by FPGA researchers: distributed memory and pipelined interconnect. Distributed memory has been migrated into the FPGA architecture and the resulting memory/logic interface has been studied by Wilton et al. [142]. High-level compiler support for memory blocks has also been studied by Gokhale et al. [60].

Hardware-based pipelining has been proposed but not added to existing FPGAs. Singh et al. [123] recently proposed pipelined interconnect in the context of a new FPGA with wave-steered logic. Tsu et al., in HSRA [133], describe a LUT-based FPGA that contains registers embedded with communication switchboxes. Marshall et al. [98] describe the CHESS architecture with coarser-grained functional units and pipelined wiring. In contrast to DeepC's approach, which supports cycle-by-cycle communication reconfiguration, these architectures support only fixed communication patterns between processing elements. The proposed FPGA that is most similar to DeepC's communication synthesis approach is the Time-Switched FPGA (TSFPGA), described by Dehon [48]. TSFPGA supports time-switched communication between groups of LUTs. Unlike DeepC's high-level approach, compilation to TSFPGA has not been extended beyond the gate level. Moreover, DeepC can leverage hardware support if available — Section 6.6.2 studies this possibility.

Architectural Mechanisms in FPGA-Inspired Architectures

Results such as those in Figure 1-11 have inspired computer architects to propose new FPGA-like architectures that attempt to capture these gains while minimizing programming difficulties. In addition to Raw [136], these architectures include Matrix [103], RaPiD [64], GARP [70], PipeRench [63], and Smart Memories [96]. To the extent that these new architectures expose lower-level logic and wiring details to the compiler, architectural mechanisms such as busses, pipelines, and new instructions can be specialized to the application without requiring full compiler support for gate-level reconfigurability. Furthermore, low power reconfigurable architecture [137, 146] and energy-exposed architecture [8] take advantage of this exposure to reduce power and energy consumption. Finally, exposed mechanisms can improve performance in reconfigurable/intelligent memory systems [109, 105, 81] and configurable memory controllers [36]. Of the above architectures, both PipeRench and Garp have developed compilation systems to automate construction of architectural mechanism in software.

PipeRench's DIL compiler [28] applies the same basic optimizations as DeepC to compile programs to the PipeRench architecture, a pipelined, reconfigurable fabric. This compiler is not targeted to commercial FPGAs, thus direct comparison with existing reconfigurable approaches is difficult; instead, application speedup versus a $300MHz$ UltraSparc is reported. Assuming a hypothetical $100MHz$ chip with 496 (31 stripes of width 16) 8-bit processing elements, performance ranging from $15\times$ to $1925\times$ is reported with an impressive 8 second compile time. Like DeepC, the PipeRench compiler also includes a form of bitwidth analysis (bit value analysis [29]) as well as common sub-expression elimination, dead code elimination, algebraic simplification, register reduction, and interconnect simplification.

Like the earlier PRISC project [113], The GARP project advocates extending a processor with reconfigurable instructions. Compiler work related to this project includes work on instruction-level parallelism (Callahan et al. [34]) and software pipelining (Callahan et al. [33]). The software pipelining work, although targeted to GARP and not conventional FPGAs, significantly extends previous pipelining work for reconfigurable architectures and goes beyond the affine loop unrolling in this work. However, this system does not support distributed architectures. Interestingly, the GARP work has been leveraged in the industrial Nimble project mentioned in Section 1.4.2.

Exposed Architectural Mechanisms in Other Architectures

This section briefly mentions three classes of architectural mechanisms found in other architectures not influenced by FPGAs. First, the Transport-Triggered Architecture (TTA) [42] exposes the data-movement inside a VLIW to more precise software control. Specialization of this architecture as a synthesis technique has also been proposed (Corporaal et al. [43]). Like other VLIW research, TTA research has considered partitioned architectures, but has focused on designs with large busses rather than distributed architectures with spatial locality.

Second, existing multimedia extensions for subword parallelism (Lee [88]), as well as proposed exploitation of superword parallelism (Larsen et al. [84]), modify conventional architectures by enabling compilers and assembly-code writers to pack multiple shorter-width instructions into a single wide (or super-wide) instruction. For example, packing four 8-bit additions into a single 32-bit ALU instruction treats the architecture as if it has four datapaths. FPGAs with special carry-chains support the most general form of this concept, allowing the single-bit carry computations to be composed into a 32-bit adder on one clock cycle, four 8-bit adders on the next, and something altogether different on the next.

Third, research labs [55] and startups [131] have developed tools to synthesize "custom-fit", or configurable (but not necessarily reconfigurable), architectures with instruction sets optimized for a given application or domain. The added instructions are exposed to the compilation system, which must also be customized. Besides instruction addition and deletion, systems such as Extensa [131] also modify high-level structures such as the processor cache. These or similar systems can be targeted to FPGAs or other reconfigurable devices; however, they still interlard an ISA, albeit customized, between the application and the already programmable hardware. ISA customization is only able to capture a portion of the specialization possible with complete ISA elimination.

1.4.4 Compiler and Language Implementation

DeepC leverages two deep analysis packages that have previous related work in the compiler and language implementation field: bitwidth analysis, and pointer analysis. Also, research on partial evaluation is related to DeepC's specialization techniques. The next section discusses these three topics.

Bitwidth Analysis

Empirical studies of narrow bitwidths workloads (Brooks et al. [27] and Caspi [37]) foreshadow my findings that a wide range of applications, particularly multimedia applications, will benefit from bitwidth reduction. Although DeepC does not attempt to capture the dynamic opportunities for bitwidth specialization, in many cases static analysis can find all

or a significant fraction of the savings. Other static bitwidth work includes Scott Ananian's data flow techniques for Java [5] (his published method of propagating data-ranges is less precise for discovering bitwidths, although additional work is underway), Rahul Razdan's technique for PRISC [113] (except for loop induction variables, his analysis does not handle loop-carried expressions well), and the approach in Budiu et al. [29] (mentioned earlier). Bondalapati and Prasanna's work on dynamic precision management [22] extends the static approach by dynamically reconfiguring the architecture as a result of runtime variations in bitwidths. Although DeepC does not evaluate dynamic reconfiguration alternatives, such a dynamic approach is compatible with and would complement the other specializations applied. Finally, private communication with Preston Briggs [26], working on the compiler for Tera Computer (now Cray), reveals that bitwidth optimization has been investigated in supercomputer compilers. Briggs proposed to use bitwidth optimizations to perform bitwise dead code elimination, bitwise constant propagation, propagation of sign extension bits, and avoidance of type conversions.

Memory Disambiguation

The compiler techniques for memory disambiguation [51], and more recently pointer analysis [141], have been studied in many contexts, including vector processors [87, 7], systolic processors [6, 23], multiprocessors [3], DSP processors [116], VLIW processors [95], FPGAs [60], and hardware synthesis [121]. Although memory bank disambiguators for vector processors must resolve whether addresses reference the same bank, they do not need to further make static the access to a particular bank. Previous systolic compilers mapped programs into coarse-grain static streams rather than tackling ILP-level memory parallelism like my approach of leveraging MAPS (Barua et al. [14]). Similarly, previous multiprocessor work focused on creating coarse-grain threads. Memory disambiguation for DSPs and VLIWs have generally not supported programs with arbitrary pointers, although these techniques could leverage DeepC's pointer analysis and equivalence class unification algorithms (Section 4.1.1). Although Gokhale's FPGA compiler [60] maps arrays to memory banks, it does not further synthesize pipelined communication schedules to avoid long wires. Finally, for hardware synthesis, the work of Semeria et al. [121] (already mentioned) leverages a pointer analysis algorithm similar to DeepC's, although the complementary focus is on pointer encoding and representation rather than large-scale scheduling of parallelism.

Partial Evaluation

Partial evaluation[80] is a form of program specialization in which the portion of the program's instructions that depend on known data are evaluated at compile time. But the instructions that depend on dynamic data become a residual program, to be evaluated later, when the remaining data becomes available[2]. Constant propagation is a simple example of partial evaluation. An example of a full application is described in Berlin et al. [19]. They partially evaluated the static data corresponding to a solar system simulation with differential equations. The residual program was then statically scheduled onto a multiprocessor. A more general partial evaluator is Tempo [41]. Tempo can be applied to any C program.

[2]Partial evaluation and specialization are often used synonymously, though some researchers claim specialization to be more general.

Partial evaluation is known for eliminating interpretation layers and has been used to make operating systems more efficient. The Synthesis Kernel [99] uses partial evaluation to eliminate synchronization and increase I/O performance. In "Hey, You Got Your Compiler in My Operating System!", Howell and Montague [75] conjecture that partial evaluation can be extended to eliminate the Application Binary Interface (ABI). The ABI separates the compiler and operating system. In comparison, DeepC eliminates a similar interface: the Instruction Set Architecture (ISA). With the advent of gate reconfigurable architectures, the ISA has become the abstraction layer that systems can no longer afford to interpret.

1.4.5 Summary

In summary, substantial work in CAD, architecture, compilers, and languages has addressed architectural specialization. This work has been converging on a complete solution for compiling high-level languages down to the gate level. However, many previous solutions have been ad hoc and unconnected. Part of the problem lies in the interposed boundaries between fields; part of the problem lies in research that has been too focused on an application, language, or technology. To overcome these problems, an overriding goal of the DeepC project has been to integrate, extend, and formalize this related work into a complete system, crossing as many abstraction boundaries as needed.

1.5 Guide to Dissertation

This section serves as a guide to the remaining chapters of this dissertation.

- **Problem and Analysis:** Chapter 2 unfolds and then re-folds the possible interpretation and compilation alternatives for evaluating high-level programs. Emerging from this re-folding is a new taxonomy of evaluation modes — four of these modes are explicitly addressed in later results. My reason for exploring these alternatives is to free the reader's mind from the instruction-set thinkspeak of traditional architecture teachings. Further analysis leads to my specialization hypothesis: high-level programs can be compiled to gate-reconfigurable architectures by *specializing* traditional architecture mechanisms with respect to the input program.

- **Basic Approach:** The next two chapters address this problem of specializing traditional architectural mechanisms. Chapter 3 starts gently with specialization of combinational functions and registers. Next, the concept of bitwidth reduction is introduced. Bitwidth reduction is one of the best specialization techniques and pervades this dissertation. This chapter ends with a basic transform for partially-evaluating the control flow of a von Neumann Architecture with respect to a program.

- **Advanced Approach:** Chapter 4 describes specialization of advanced architectural features, with a focus on distributed architectures. The approaches are divided into two sections: specialization of memory mechanisms, dominated by the problem of disambiguation, and specialization of communication mechanisms, dominated by the problem of space-time scheduling. Overcoming the physical limits imposed when leveraging large silicon area requires implementation of both sets of approaches.

- **Implementation:** Chapter 5 describes the prototype system built to test my hypothesis. This system embodies the approaches discussed in Chapter 3 and Chapter 4. This chapter describes the compiler passes, hardware targets, and DeepC's simulation and verification environment. The chapter focuses on the phases of the compiler, including a Traditional Frontend Phase, a Parallelization Phase, a Space-Time Scheduling Phase, a Machine Specialization Phase, and a CAD Tool Phase.

- **Results:** Chapter 6 discusses use of the prototype system to compile and simulate new benchmarks called the *Deep Benchmark Suite.* Specializing basic mechanisms is very effective. The new system is not only efficient, but outperforms the traditional approach for most cases. Basic sensitivity analysis uncovers the specializations that are beneficial. Advanced results are also good, with parallel speedup on the benchmarks that could be parallelized. However, these results were not as good as desired. Thus, an advanced sensitivity section studies improvements.

- **Conclusions and Appendices:** Chapter 7 summarizes the dissertation and makes prospects for future work. The implications of DeepC for the free hardware community are discussed, and I challenge the compiler community to continue this work, to compile to gates. Several appendices follow. Appendix A presents the full eight evaluation modes discussed in the analysis. Appendix B describes a logic emulation system, with host interface, and results using DeepC to target this real hardware. Appendices C and D contain tables of data for basic and advanced results. Finally, Appendix E concludes with VPR place and route data and FPGA layouts.

Chapter 2

Problem and Analysis

Try to imagine an analogous evaluator for electrical circuits. This would be a circuit that takes as input a signal encoding the plans for some other circuit, such as a filter. Given this input, the circuit evaluator would then behave like a filter with the same description. Such a universal electrical circuit is almost unimaginably complex.

— *Abelson and Sussman on metacircular evaluators,*
Structure and Interpretation of Computer Programs,
second edition (1996), page 386.

This chapter qualitatively compares and contrasts the programming of two universal architectures: von Neumann [30], or instruction-programmable, architectures and gate-reconfigurable architectures (GRAs). This comparison leads to a theoretical analysis that develops a language-centric model of program evaluation. This model characterizes architectural differences with a new taxonomy of eight different evaluation modes (four of these are studied in detail). Further analysis leads to an important result: the technology needed for programming GRAs at a high-level is a compiler from an intermediate, machine-level language down to low-level logic. Furthermore, there is a direct relationship between this new compilation step and the mechanisms of computer architecture. This relationship reveals the GRA compiler's task — to *specialize* the architectural mechanisms already understood.

The next section draws insight from the old problem of compiling high-level programs to microprocessors. Given this insight, Section 2.2 analyzes the problem now encountered when compiling to GRAs. This comparison leads to the theory of evaluation modes in Section 2.3. It also leads to a special relationship between the old and the new problem in Section 2.4. Section 2.5 summarizes these findings.

2.1 The Old Problem: Compiling to Processors

Consider evaluation of the section of a high-level C program for multiplying two integer matrices (Figure 2-1). In earlier eras of computer science — during the 1970s — sophisticated hardware closed the semantic gap between programming languages and architectures; a processor could be expected to evaluate these instructions almost line-by-line. However, since the advent of RISC architectures, system builders close this gap with sophisticated compilation technology that translates the program into machine code. RISC systems stress compilation rather than interpretive mechanisms, allowing hardware to be dedicated to performance, not semantics.

```
for (i = 0; i < 32; i++) {

  for (j = 0; j < 32; j++) {
    dest[i][j] = 0;
  }

  for (j = 0; j < 32; j++) {
    for (k = 0; k < 32; k++)
      dest[i][k] += srcA[i][j] * srcB[j][k];
  }
}
```

Figure 2-1: Matrix multiply inner loop

Figure 2-2 contains a snippet of assembly for the inner loop of the multiply, compiled to a SPARC [128] architecture. The assembly instructions include move (`mov`), add (`add`), shift left logical (`sll`), load (`ld`), store (`st`), compare (`cmp`), branch if less than or equal to (`ble`), and a procedure call (`call`). Implementing these operations at the physical transistor level requires a hardware abstraction.

As an example hardware abstraction, consider the illustration in Figure 2-3 of the basic organization of a standard five stage pipeline [72]. This pedagogical architecture demonstrates direct hardware support for the machine level instructions in the matrix multiply example: there is an ALU that can perform additions, comparisons, and shifts; there is logic to determine whether the machine should branch and to set the program counter; there is a data memory for loads and stores[1]. We can conclude that *today's processors close the abstraction gap between machine-level opcode and physical transistors.*

2.2 The New Problem: Compiling to Gate Reconfigurable Architectures

Now consider the mechanisms in a GRA, typified by the Field Programmable Gate Arrays (FPGAs) available from Xilinx. An FPGA is an array of configurable logic blocks (CLBs) interconnected by programmable switches (Figure 2-4). In the figure, single lines connect adjacent switches while double and long lines connect to more distant neighbors. Figure 2-5 shows the CLB for the Xilinx XC4000 series FPGA [143]. Each CLB includes several look-up tables (LUTs) and single-bit registers. The remaining logic consists of control and multiplexers. This logic also functions to configure other features inside the CLB, for example the S/R Control (set/reset control) and the carry-chain (not shown).

GRAs are a promising technology. The number of programmable features in the largest devices is approaching one million. Just as the move from CISC to RISC enabled simpler

[1] An astute reader at this point may have noticed that SPARC instructions can not be executed on a DLX processor, even though the two architectures contain similar mechanisms — ah, the problems of binary compatibility!

```
.LL28:
mov %l7,%o0
add %l3,%i0,%l0
add %l5,%i0,%o1
add %i0,1,%i0
sll %o1,2,%o1
ld [%l4+%o1],%o1
call .umul,0
sll %l0,2,%l0
ld [%l6+%l0],%o1
cmp %i0,31
add %o1,%o0,%o1
ble .LL28
st %o1,[%l6+%l0]
```

Figure 2-2: SPARC assembly for inner loop of integer matrix multiply

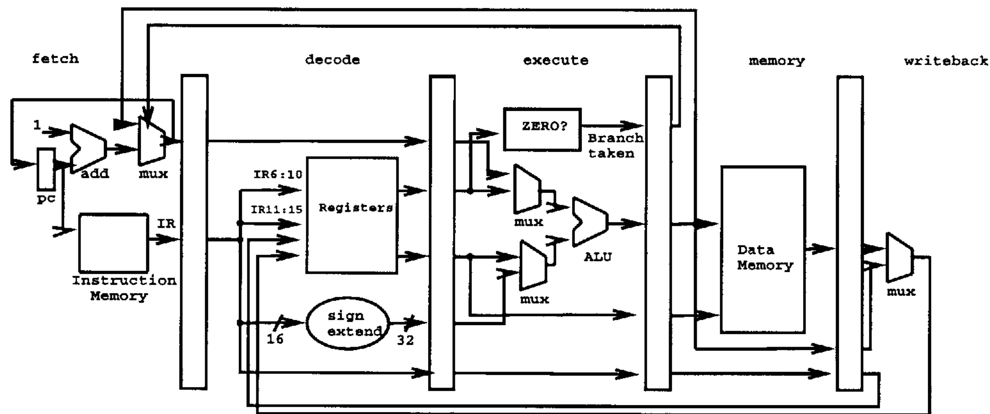


Figure 2-3: DLX pipeline

The first stage includes a program counter (PC) and logic to update the program counter. The program counter indexes into an instruction memory, whose output is stored in the instruction register (IR). If a branch is taken, the PC is loaded from the previous instruction's ALU output. Otherwise the PC is incremented to the next instruction. In the second stage, separated from the first stage by pipeline registers, a multi-ported register file is addressed by the IR. Two output ports generate values for the following execute stage. In the execute stage, an arithmetic and logic unit (ALU) performs binary operations on values from the register file, the previous program counter, or immediate data. In the final stage, values are written to and read from the data memory. See [72] for further details.

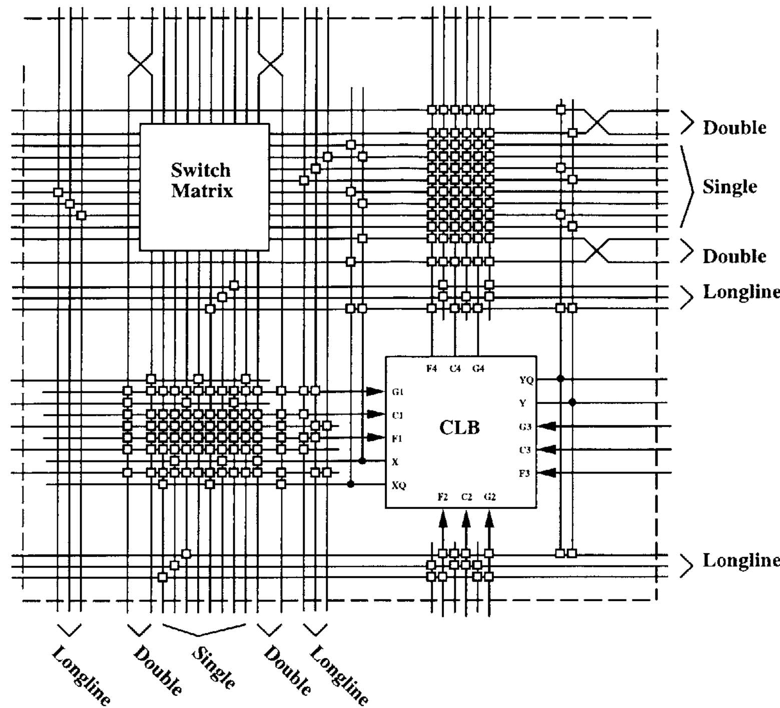


Figure 2-4: Xilinx XC4000 interconnect

Configurable Logic Blocks (CLBs) are interleaved with switch matrices, connected by single, double, and long lines. The internals of the CLB are shown in Figure 2-5.

architectural primitives, gate-level features are even simpler and are easier to design. These features expose parallelism and spatial locality at a fine grain, enabling high-performance, low-power designs. In some ways, GRAs return to a pre-von-Neumann, pre-stored-program era of computing, in which programs were built into special-purpose computer at a low level with wires, patch cables, and basic circuits. In other ways, programmatic configurability of GRAs retains the stored-program features of von-Neumann architectures, but without the resulting sequential bottlenecks.

Like RISC architectures, GRAs are very closely associated with, if not inseparable from, their compilation tools. However, in comparison to processor compilers, GRA tools operate at a low level. The usual input is a netlist or RTL-level design. Input sequential state machines must be mapped into Boolean functions of four inputs and single-bit registers. These "functions of four" and registers are further compiled into a *configuration* for a particular GRA device. The configuration of a GRA is its program, a bitstream loaded when the device boots. These bits determine the LUT function; they also control internal multiplexers. Thus, although a GRA supports logical structure, it does not include support for machine instructions. We thus conclude that *a gate reconfigurable architecture only closes the abstraction gap between logic-level gates and physical transistors.*

Herein lies the problem of this dissertation. Compiling high-level programs to GRAs requires development of a software infrastructure that closes the abstraction gap between machine instructions and logic gates. This solution must be compatible with existing compilation approaches for translating high-level programs to machine-level intermediate forms.

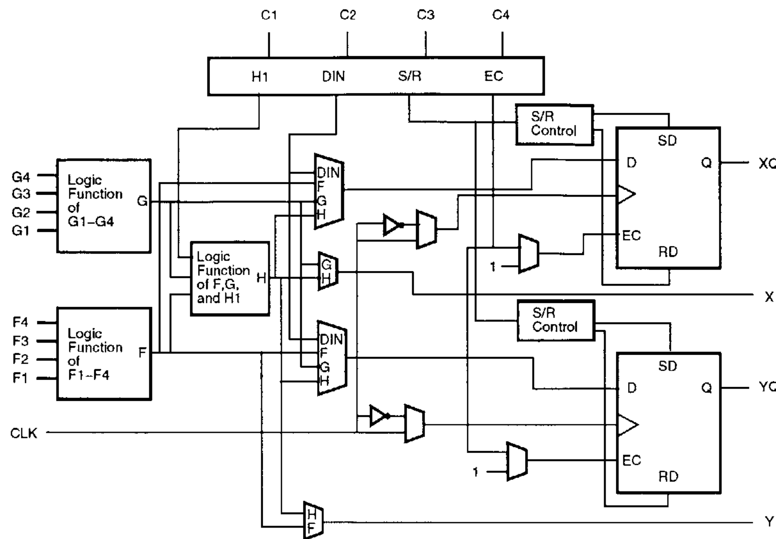


Figure 2-5: Xilinx XC4000 CLB

Two functions of four, one function of three, and two single bit registers make up the Xilinx XC4000 CLB. Additional multiplexers and set/reset control (S/R control) permit a range of gate-level configurations.

It must also be compatible with existing computer-aided design tools used to generate the GRA bitstream. These tools must be integrated for an automatic solution.

Although some functions (adders, multipliers, block memories) have been added to new generations of GRAs, these are efforts to close the abstraction gap with the hardware of the GRA. As this work will show, this gap can be closed with software. So, should computer engineers resist the rush to move all the features of von Neumann architecture into the fabric of GRAs? Or alternatively, are there advantages, significant advantages to having software control over architectural mechanisms? Before beginning to address these questions, the next section further develops the concept of "closing the gap with software".

2.3 Theory of Evaluation Modes

This section contains an intuitive, language-centric analysis of what it means to evaluate a program on a GRA rather than evaluating it on an instruction-programmable processor. This analysis develops four different evaluation modes that correspond to historically familiar approaches to computing. These modes are important because they distinguish high-level compilation to GRAs from other approaches (other approaches including processors, custom chips, and softcore processors). These modes also reveal a general compiler design strategy: they show the importance of specializing architectural mechanisms.

To get started, consider four levels of abstraction: High-level (H), machine-level (M), logic-level (L), and the physical-level (P). H corresponds to the high-level imperative languages (C, for example) to be evaluated — each instruction or statement corresponds to several instructions in machine language; M corresponds to the architectural mechanisms required to evaluate machine code — machine language is commonly generated from a high-

Mode	Example	level	high ($H \rightarrow M$)	machine ($M \rightarrow L$)	logic ($L \rightarrow P$)
Mode 0	Custom ASIC	0	compile	compile	compile
Mode I	FPGA	1.L	compile	compile	interpret
Mode II	RISC Processor	1.M	compile	interpret	compile
Mode III	Processor-on-FPGA	2.ML	compile	interpret	interpret

Figure 2-6: Basic evaluation modes

Modes correspond to different implementations of a high-level application. *Mode 0* is a system compiled at all levels, for example a custom ASIC designed to evaluate a single application. But a processor designed in an ASIC is not *Mode 0* — the processor is a machine-level interpreter! *Mode I* is a system in which only the logic-level is interpreted, for example an application implemented on an FPGA. *Mode II* is a system with a processor and a high-level compiler. Note that in *Mode II* the processor must be implemented in a custom process. If the processor is "soft", for example a processor implemented on a GRA, the system is termed *Mode III*.

level language by a compiler; L corresponds to the logic and state machine level — this level is customary in the design of hardware systems; P corresponds to the lowest physical level specified prior to fabrication — usually a rectangle-level layout such as the Caltech Intermediate Format (CIF).

Three translations are needed in order to span the abstraction gap between H and P. Each of the three translations can be performed with either a compiler-based approach or an interpreter-based approach, resulting in eight possible modes of evaluation. At this point, I restrict further discussion to the four evaluation modes that include high-level compilation; an overview of all eight modes is included in Appendix A for completeness. The results in this dissertation compare these four modes (Table 2-6). The "level" column lists the number of levels of interpretation followed by the letters H, M, or L to denote which levels are interpreted. Neither the exact boundary between abstraction levels nor the number of abstraction levels is overly significant; there is a continuum of system mechanisms between the mathematical high level and the physical low level. However, the model chosen matches previous technology generations and best explains the differences between gate reconfigurable modes and other alternatives while emphasizing compilation. See Dehon [48] for an alternate model.

Before continuing, consider a broader definition of compilation. For a particular level, compilation software may not exist. Compilation must then be performed *manually*. Compilation is not a common term for manual design, although mathematically a human designer and a compiler solve the same translation problems. For example, an RTL description of a processor can be synthesized, placed, and routed on an ASIC. Alternately, three hundred engineers can implement the processor rectangle-by-rectangle. The labor intensity of manual design relegates it to scenarios in which the design can be reused across a large number of instances. Manual compilation at high levels is rare.

The translation alternatives lead to the four evaluation modes. Figure 2-7 graphically represents the modes as possible compiler and interpreter nestings. Each *domino* represents either a compiler (horizontal) or an interpreter (vertical). These are similar to the T-diagrams first introduced in 1961 by Bratman [24], with the exception that they ignore the

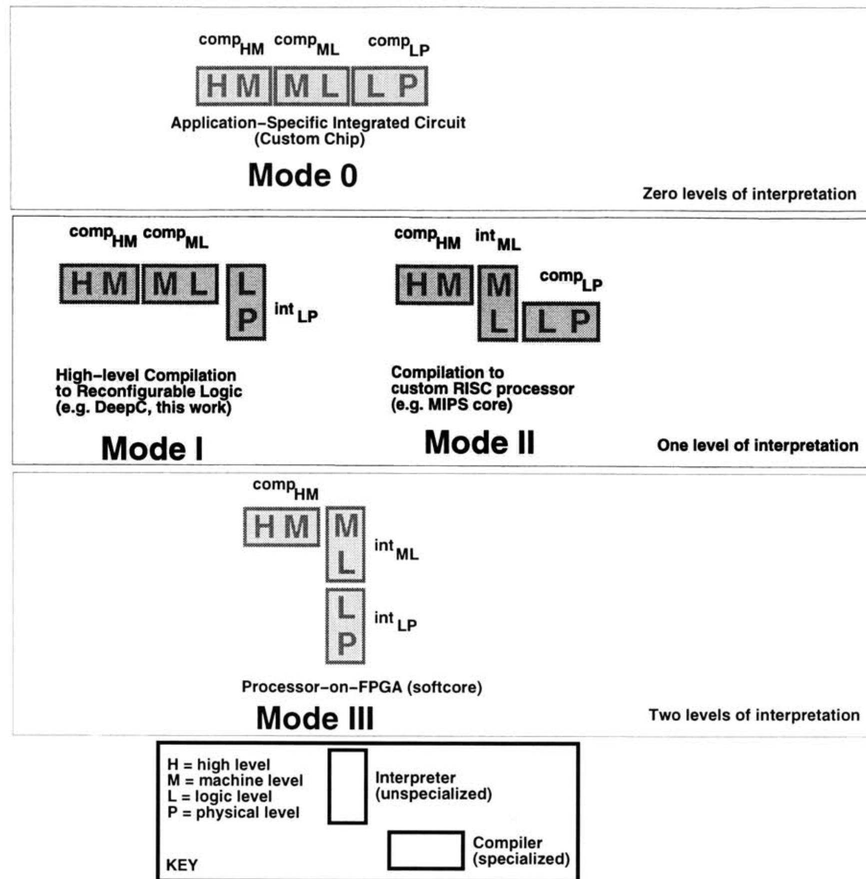


Figure 2-7: Evaluation dominos

implementation language of the compilers and focus on the nesting. This visualization is not picky about legal placement when making turns — a more consistent approach would be to overlap identical letters.

Consider the modes grouped by number of interpretation levels. A system with zero levels of interpretation, totally compiled, is superior to interpreted systems in area, speed, and power; however these advantages come at the sacrifice of reprogrammability. Thus, evaluation *Mode I* and *Mode II*, the most efficient fully-programmable modes, are the primary focus of this dissertation. These level one modes are termed *singly-interpreted*. Both FPGAs and microprocessors have a single level of interpretation. Thus, despite conventional beliefs, FPGAs are not more customized than microprocessors, only customized at a different level of abstraction. Evaluation *Mode III*, processor-on-GRA or softcore processor, is interesting in that a new compilation strategy is not needed: a user can design a processor (machine-level interpreter), run it through standard CAD tools, and compile high-level programs with conventional compilers. Furthermore, in the context of GRAs, a performance comparison of the advantages of *Mode I* over *Mode III* isolates the benefits of machine-level specialization.

The next paragraphs further elaborate the evaluation modes to be studied, including denotational semantics for each mode. To support this discussion, Figure 2-8 reviews standard

Let $[\![p]\!]$ denote the meaning of program p and $[\![\bullet]\!]_L$ denote the semantic function of language L. Furthermore, define the result r of evaluating program p with input data d in language H as:
$[\![p]\!]_H\ d\ =\ r$
This section now proceeds with the standard definitions of interpretation, compilation, and specialization [80]. An interpreter is a program that takes as input another program and data to produce a result. Formally, the direct evaluation of program p in language H is related to the program's interpretation with an M interpreter written in language M, denoted $I_{H\rightarrow M}$, assuming all languages are Turing general, as follows:
$[\![p]\!]_H\ d\ =\ [\![I_{H\rightarrow M}]\!]_M\ < p, d >$
A compiler is a program that transforms an input program written in one language into a program in a second language. As before, the direct evaluation of program p in language H can be related to the compiled program's evaluation with compiler $C_{H\rightarrow M}$ as follows: $[\![p]\!]_H\ d\ =\ [\![[\![C_{H\rightarrow M}]\!]\ p\ ]\!]_M\ d$
Compilers and interpreters can be stacked to multiple levels. The following example demonstrates nestings:

- Nested interpreters: $[\![p]\!]_H\ d\ =\ [\![I_{H\rightarrow M}]\!]_M\ < p, d >\ =\ [\![I_{M\rightarrow L}]\!]_L\ < I_{H\rightarrow M}, < p, d >>$
- Nested compilers: $[\![p]\!]_H\ d\ =\ [\![[\![C_{H\rightarrow M}]\!]\ p]\!]_M\ d\ =\ [\![[\![C_{M\rightarrow L}]\!]\ (\ [\![C_{H\rightarrow M}]\!]\ p\ )\ ]\!]_L\ d$
- Compiling H to an M interpreter in L: $[\![p]\!]_H\ d\ =\ [\![[\![C_{H\rightarrow M}]\!]\ p]\!]_M\ d\ =\ [\![I_{M\rightarrow L}]\!]_L\ < ([\![C_{H\rightarrow M}]\!]\ p), d >$

Note that when the evaluation language is unimportant (as is the compiler's implementation language), the language subscript can be dropped. Finally, let us review *specialization* theory, derived from Kleene's S-M-N theorem. Given an input program p, static data s, and dynamic data d:
$[\![p]\!]_H\ < s, d >\ =\ [\![\ [\![mix]\!]\ < p, s >]\!]\ d$, where mix is a program specializer.
The residual program $r = [\![mix]\!]\ < p, s >$ is the result of specializing (sometimes called partially evaluating) program p with respect to static data s. A classic result is that *specialization of an interpreter is equivalent to compilation.* This result is a motivator for this dissertation. Observe that:
$[\![mix]\!]\ < I_{M\rightarrow L}, p >\ =\ [\![C_{M\rightarrow L}]\!]\ p.$
In other words, a compiler from a machine-level language to a logic-level language is equivalent to a specialization of a machine-level interpreter with respect to some input machine-level program.

Figure 2-8: Review of denotational semantics

denotational semantics (hopefully familiar from your programming languages course).

Mode 0 A custom chip can be designed to evaluate a particular high-level behavior. The high-level behavior is customized into a machine-level behavior, usually represented in Register Transfer Language (RTL), which is then further customized to logic. This logic is in turn used to generate a physical layout. Specialization may be performed automatically, manually, or in some combination. This mode is the most efficient because there are zero levels of interpretation. However, a custom chip designed in this manner is not reprogrammable — it can only evaluate one high-level behavior.

The semantics for *Mode 0* are: $[\![\ [\![C_{L\rightarrow P}]\!]\ ([\![C_{M\rightarrow L}]\!]\ ([\![C_{H\rightarrow M}]\!]\ p)\ )\ ]\!]_P\ d$, meaning: first compile program p from language H to M, and then from language M to L, and then from language L to P. The result is a physical realization of p in hardware that is capable of processing data d.

Mode I The gate reconfigurable approach is the primary focus of this dissertation. A high-level language is compiled into machine-level mechanisms that are further specialized to the logic-level. This logic, unlike *Mode 0*, is not further specialized, but instead evaluates on programmable logic. Thus, this mode has one level of interpretation and is less efficient than *Mode 0*. Yet, with this level of interpretation the system is fully programmable and thus *universal* — different high-level behaviors can be evaluated on the same physical hardware.

The semantics for *Mode I* are: $[\![I_{L\rightarrow P}]\!]_P < [\![C_{M\rightarrow L}]\!]\ ([\![C_{H\rightarrow M}]\!]\ p), d >$,

meaning: compile the program p from language H to M and from language M to L. Then, interpret language L, along with data d, on a pre-fabricated $I_{L \to P}$ machine.

Mode II Traditional processors offer the most conventional form of computing. Languages are compiled to the instruction set architecture of a processor. The processor is in effect an interpreter for machine-level programs. The processor is implemented in logic that is further specialized, manually or automatically, to the physical-level. Like *Mode I*, *Mode II* is also universal because of the machine level interpretation layer.

The semantics for *Mode II* are: $[\![\; [\![C_{L \to P}]\!] \; I_{M \to L}]\!]_P \; < ([\![C_{H \to M}]\!] \; p), d >$, meaning: compile program p from language H to M and interpret it on a machine-level interpreter $I_{M \to L}$. This machine-level interpreter is written in language L and pre-fabricated by compiling language L to P. Note that this last compilation step can be done manually.

Mode III Softcore processors consist of a combination of the interpreter in *Mode I* with the interpreter in *Mode II*. Although the high-level language is compiled to machine instructions, neither is the machine further specialized to logic nor is the logic further specialized to the P. Thus, these systems have two levels of interpretation. FPGA vendors have released softcore processors into the marketplace. With softcore processors, new high-level programs can be evaluated without re-specializing to FPGA logic. However, with double the interpretation, one can expect roughly double the overhead in comparison to *Mode I* or *Mode II*.

The semantics for *Mode III* are: $[\![I_{L \to P}]\!]_P < I_{M \to L}, < ([\![C_{H \to M}]\!] \; p), d >>$, meaning: compile program p from language H to M and interpret the compiled program, along with its data d, on a machine-level interpreter $I_{M \to L}$. This interpreter is written in language L and is itself interpreted by a logic-level interpreter $I_{L \to P}$.

In summary, there are four evaluation alternatives that include an initial compilation from a high-level language to a machine-level language. These alternatives differ in whether they compiler or interpret the lower level languages. Because there are two translation steps, there are four permutations of compilers and interpreters. Exactly one interpretation level is needed for both universality and efficiency. Both GRAs (*Mode I*) and traditional processors (*Mode II*) share this property. The next section continues by demonstrating a relationship between the two modes.

2.4 A Special Relationship

The section shows that the goal of compilation to GRAs should be to specialize the architectural mechanisms present in von Neumann processors. First, substitute the specialization equation from Figure 2-8:
$[\![C_{M \to L}]\!] \; p \; = \; [\![mix]\!] \; < I_{M \to L}, p >$
(recall that mix is a program specializer) into the *Mode I* equation:
$[\![p]\!]_H \; d \; = \; [\![I_{L \to P}]\!]_P \; < \; (\; [\![C_{M \to L}]\!] \; ([\![C_{H \to M}]\!] \; p) \;), d >$
to yield:
$[\![p]\!]_H \; d \; = \; [\![I_{L \to P}]\!]_P \; < \; (\; [\![mix]\!] \; < I_{M \to L}, \; ([\![C_{H \to M}]\!] \; p) \; > \;), d >$.
This equation means: given an input program p, written in a high-level language H, first apply high-level compilation technology ($C_{H \to M}$) to generate a machine level program. This resulting machine level program is the input to another evaluation stage that is itself a compiler. This compiler is constructed from the specialization (mix) of the mechanisms of

machine-level interpretation[2]. The result of applying this second compiler to the machine-level program is a logic-level program. This logic-level program, along with the input data d, can be interpreted by $I_{L \to P}$ to generate the desired result.

The previous equation shows that, given a good program specializer, and a description of a machine architecture it can understand, nothing else would be left to do; ideally *mix*, a general program specializer, can remove all interpretation overhead! Although well-specified processors are available, program specializers are not yet mature enough for this general application. Thus, my compiler, DeepC, is a result of the manual construction of a system that performs this specialization.

2.5 Summary of Result

This chapter has made a theoretical case for my thesis that specialization of architectural mechanisms with respect to an input program is the key to efficient compilation of high-level programs to gate-reconfigurable architectures. The benefits predicted include:

1. smaller, cheaper, more cost-effective computing,
2. faster, lower latency computing,
3. computing with lower power, lower energy, and lower energy-delay.

This theory does not predict that these benefits will outweigh the overheads of gate-level interpretation. According to the theory, the relative overheads of interpretation at the gate-level versus instruction-level will determine the better approach. Using the DeepC implementation in Chapter 5, the empirical results in Chapter 6 include evaluations of this comparison for select benchmarks. But first, the following two chapters offer specialization exemplars for important architectural mechanisms.

[2]Recall that this theory specializes the architecture with respect to the input program, not the input program with respect to a static component of its data. Further program specialization is compatible with my approach — there are public domain C specializers that can be added to the front of the DeepC compiler flow.

Chapter 3

Specialization of Basic Mechanisms

If compilation is to achieve substantial efficiency gains over interpretation, the structure of the target language must be tractable enough that the compilation algorithms can be designed with a reasonable amount of effort. In other words, good algorithms for system design must be available in the target language. A target language that is loaded with exceptions, global interactions, and limiting cases is more difficult to compile to. This partly explains our use of interpretation for relatively difficult-to-compile-to target languages as electronic circuits and logic diagrams."

– *Ward and Halstead on Interpretation versus Compilation,*
from Computation Structures (1990), page 276.

Replacing an interpretative layer with a compilation layer is a major challenge. However, the previous chapter concluded that specialization of architectural mechanisms is the key for efficient program evaluation on gate reconfigurable architectures (GRAs). Thus, significant support is needed to prove this claim. The next two chapters provide concrete support by developing a set of approaches for specializing individual mechanisms. Specialization of the most basic mechanisms, described in this chapter, has a large impact on all areas of system performance. Specialization of advanced mechanisms, in the sequel, helps maintain performance advantages under difficult technological constraints, such as signal propagation limits on long wires, and help take advantage of opportunities created by parallelization technology.

Recall from the previous chapter that an interpretation layer adds overhead. When considering *Mode II* (traditional processor) versus *Mode 0* (custom device), specialization of architecture mechanisms leads to performance gains as interpretation overheads ($I_{M \rightarrow L}$) are *removed.* When comparing *Mode 0* to *Mode I* (GRA), performance is lost: logic-level interpretation overhead ($I_{L \rightarrow P}$) is *added.* Thus, when comparing *Mode I* versus *Mode II,* the gains of architectural specialization are offset by the losses of gate-level interpretation. In other words, without architecture specialization, there will be no gains to offset interpretation overheads, resulting in a non-competitive approach for GRAs. Finally, besides absolute performance, choice of device may be determined by many practical business and engineering factors, for example availability, reliability, brand, industrial partnerships, and even personal preference. If the overheads cannot be totally offset, specialization is still essential to increase the marketplace competitiveness of the GRA approach.

I continue with the mechanisms discussed in this chapter, grouped into three sets of basic architectural mechanisms:

- Combinational Functions,
- Storage Registers,
- Control Structures.

Each section presents the basic approach, several examples, and concludes by discussing ease of implementation.

3.1 Specializing Combinational Functions

3.1.1 Approach

Combinational functions are the most basic mechanisms that can be specialized. The essence of a combinational function is a continual evaluation of some Boolean function, usually once per clock period in a synchronous system. If certain inputs of a Boolean function are known ahead of evaluation, then those inputs can be substituted into the logic and the new equations can be simplified or otherwise reduced. The basic idea is to use compile-time knowledge to eliminate gates. The resulting logic can be implemented with smaller, faster, lower-power circuits.

At the lowest level, all architectural features are ultimately composed of combinational functions, or *gates*. Thus, a reductionist view is that all specialization can take place at this level. However, in practice it is better to specialize at the highest level possible, both to improve compilation time and to avoid the need to re-discover high-level information. Still, many opportunities for combinational specialization remain low-level. For example, folding constants into circuitry can only be performed at the gate level.

Logic Functions

Figure 3-1 summarizes some basic logic simplifications that can be applied to the standard AND, OR, and MUX (multiplexer) gates used in computer design. Inputs that are labeled 0 or 1 are discovered at compile time, allowing the gates on the left to be specialized into the gates on the right. From top to bottom: the first gate on the left is a NOT gate, the next two gates are AND gates, and then two OR gates, and then a three-input AND gate, and finally a two and three input MUX. In each case, the specialized gate on the right is simpler, or even trivial, when compared to the gate on the left. For example, the three input AND simplifies to a two input AND.

The specialization approach complements customary digital logic optimizations. It exposes new simplifications and reductions that arise only after folding information from the input program into lower-level circuitry. Because it can leverage mature optimizations from synthesis, it is a powerful compilation technique.

Arithmetic Functions

Arithmetic functions such as addition (ADD), subtraction (SUB), and multiplication (MUL) are often constructed as combinational functions (as opposed to sequential functions) even though they are more complex than ANDs and ORs. Recall from the DLX example, in

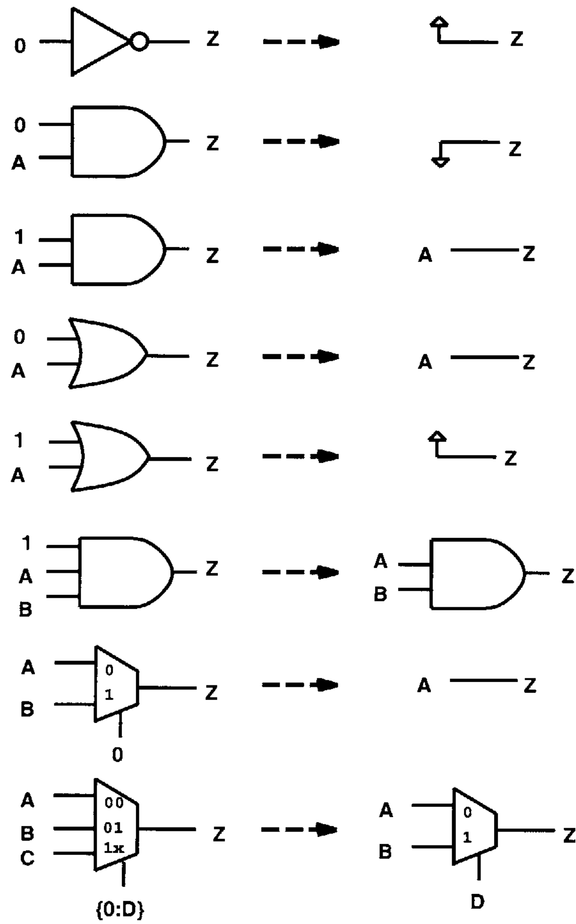


Figure 3-1: Specialization of combinational logic

Inverters, AND gates, OR gates, and multiplexers can be specialized into simpler logic if some inputs are known at compile time. The pullups and pulldowns shown represent logical ones and zeros, respectively.

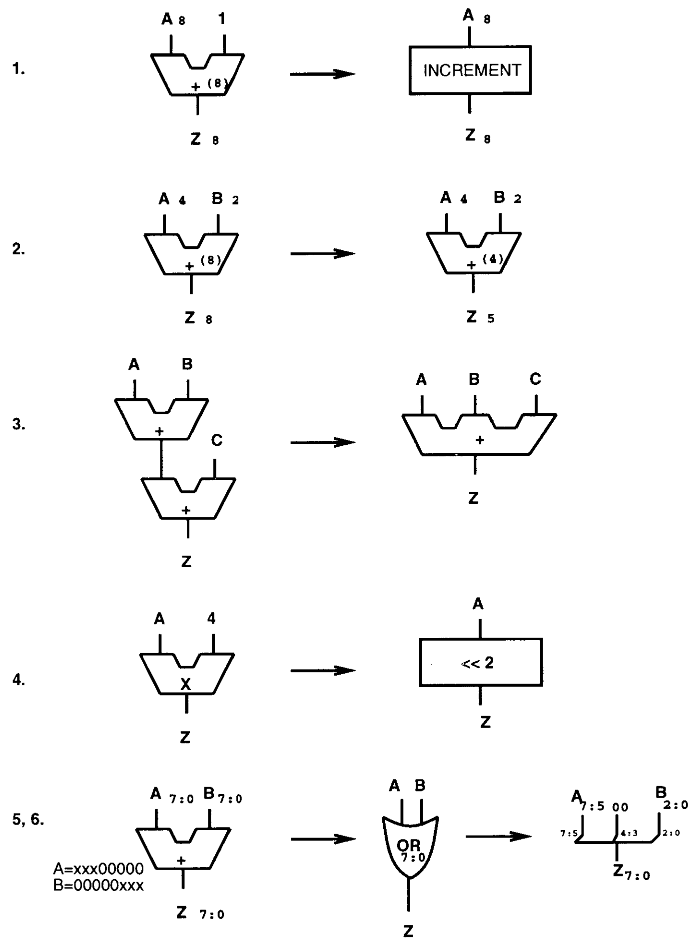


Figure 3-2: Specialization of combinational adders

The subscripts denote the bitwidth, or range of bits, for each variable.

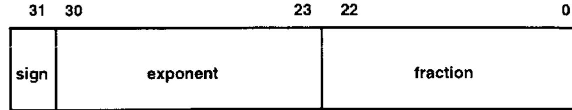


Figure 3-3: IEEE 32-bit floating point format

Chapter 2, that arithmetic functions are used for internal calculations such as incrementing the program counter and calculating addresses. As a result, fixed function bitwidths are exposed to the assembly-level programmer and compiler back-end. For example, In C, the *char* type is 8 bits, *short* is 16 bits, and *int* is 32 or 64 bits, depending on the architecture.

With a multi-bit function, finding the precise *bitwidth* of operands and datapaths is an important analysis. A more general theory for analyzing and optimizing bitwidths is developed in Section 3.2.1, after registers are introduced. At this point, I give several examples of my approach for specializing functions operating on vectors of bits. Figure 3-2 shows the following cases:

- If one of the inputs is known to be a constant, then the function can be specialized with respect to that constant (1.).
- If the bitwidth of an operator is less than the standard bitwidth, the upper bits of the function can be eliminated (2.).
- Two or more such functions may be *chained* or otherwise combined into a special function (3.).
- Under certain conditions arithmetic operators can be strength reduced to simpler operators: MUL to SHIFT (4.), ADD to OR (5.), OR to bit concatenate (6.).

Signed Integers

Consider current hardware description languages. VHDL supports user-defined data types, but Verilog does not even support signed integers. When the sign can be determined at compile-time, signed integers can be specialized into unsigned integers. In DeepC, this specialization is implemented in the range-propagator, part of bitwidth analysis. When a signed representation is needed, the code generator modifies comparison statements, which are always unsigned in Verilog, to handle the sign bit. The correct modification is to `XOR` the sign bits with the result of any comparison. The default properties of two's complement numbers take care of other operations on signed numbers.

Floating Point

Many scientific programs use floating-point arithmetic. Thus, architecture support for floating point is quite common, especially support for the IEEE standard [77]. Numbers in this format are composed of three fields: a sign bit, an exponent, and a fraction (Figure 3-3). The exponent is a power of two and the fraction stores the floating point mantissa.

Although the main thrust of this research has not included floating point and my results are focused on multimedia and integer applications, I have explored two approaches for specializing floating point. In the first approach, floating-point functions are manually designed and instantiated at the logic level. A fixed number of such floating-point

functions (ADD, MUL, DIV) are included in the architecture. Then, during scheduling, floating-point operations are mapped onto these functions. Specialization occurs around the floating-point operations (loads and stores, multiplexers, control), but the computation within the float point function units is *lifted* to runtime. That is, the computation inside the function call is not unfolded or partially evaluated at compile time. DeepC 2.0 includes scheduling support for this first approach but it does not yet have an interface to a library of well-designed floating-point functions. However, well-designed floating-point functions, optimized for FPGAs, have been studied by Fagin et al. [53], Ligon et al. [94] and Shirazi et al. [122], and several free designs are available on the Internet.

In the second approach, the floating-point functions are described as library calls in the high level language (C). The approach first inlines these library calls into the program, dismantling floating-point functions into their constituent integer and logical operations. In other words, each floating-point operation in the program has been replaced with a set of integer and bit level micro-operations. My technique is similar to Dally's proposal [46] – these resulting micro-operations are optimized and scheduled. Because the constituent exponent and mantissa micro-operations are exposed, a compiler can exploit the parallelism inside individual floating-point operations and also between different floating-point operations. In DeepC 2.0, a prototype of this approach has been tried but not fully explored.

The second approach allows more specialization, and thus would be expected to yield better results. However, the code explosion resulting from inlining such complex operators has so far prohibited exploiting this approach. Backend CAD tools cannot yet handle the complexity of the resulting state machines. Finally, manual optimizations applicable in the lifted approach cannot be applied in the second approach.

3.1.2 Examples

The following examples further clarify combinational specialization approaches.

Eliminating a logical OR

```
if(delta >= step) {
   delta |= 1;
}
```

Figure 3-4: Example of eliminating a logical OR

The code in Figure 3-4 is a part of the ADPCM benchmark. The lowest bit of the variable delta is set to 1 when delta is greater than or equal to the variable step. Consider just the assignment for now, Section 3.3 will consider the conditional. At a certain point in the evaluation of the program, the lowest order bit of delta should be set to 1. Assuming delta is stored in a register, what was a multi-bit OR in a traditional processor ALU is reduced to an assignment of 1 to a single register bit.

Simplifying a Comparison

Consider the comparison of the variable *sum* to the value 3 in the following expression from the newlife benchmark:

```
(sum==3) | (data1[i][j] & (sum==2))
```

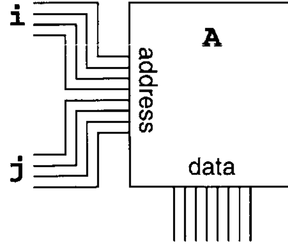


Figure 3-5: Example of address specialization

The address calculation for the array reference `A[i][j]` can be entirely eliminated. Instead, the values of `i` and `j` are placed on the corresponding address wires.

This example is only considering the ($sum == 3$) portion of the expression, not the entire expression. Comparison of sum (a three-bit number) to the number three would usually require a subtraction, or at minimum an XOR in an unspecialized processor. In this case, the logic need only check that bits 0 and 1 are TRUE and bit 2 is false, as computed by the expression: $sum_0 \wedge sum_1 \wedge \overline{sum_2}$. Note that a similar approach can be used for the ($sum == 2$) comparison, after which the entire expression can be optimized at the logic level.

ADD/SUB function

In a generic ALU, a standard function unit of RISC processors, the subtract operation is implemented in two's complement by inverting (applying NOT to) the term being subtracted. Because this inversion is conditioned on whether the operation is a subtract, an XOR gate is needed. With specialization, this gate is eliminated in the ADD case and reduced to NOT in the SUB case.

Strength Reducing Address Generation Equations

This longer example explains strength reduction within an address calculation. Consider what happens when referencing a 32-bit, two dimensional array, like $A[i][j]$. The naive code for this address calculation is
$A + (i * YDIM + j) * 4.$
Note that the constant 4 adjusts 32-bit addresses for byte-addressable memory. When $YDIM$ is a power of 2, the multiplication operations are usually realized as shifts to produce:
$A + (i << log_2(YDIM) + j) << 2.$
In an unspecialized architecture, this is the minimal calculation to reference a two dimensional array. Combinational specialization, on the other hand, can do much better. Assume that the array can be fully disambiguated from other arrays and variables. (Memory disambiguation is described in Section 4.1.) Thus, the A array has its own private memory space. Every reference will lie at an offset from location 0 of the memory for array A, so the first addition operation can be eliminated. Furthermore, the memory for A can be built with the same width as the data word, so that the final multiplication by 4 can be eliminated, leaving $i << log_2(YDIM) + j$. Next, the `add` operation can be transformed into `OR` operations. First, calculate the maximum width of j. Because $j < YDIM$, the maximum value of j never consumes more than $log_2(YDIM)$ bits, so the bits of j and the bits of

$i << log_2(YDIM)$ never overlap. The addition in this expression can be optimized away to produce $i << log_2(YDIM) \mid j$. Finally, because the OR of anything with 0 produces that value, the OR gates can be eliminated and replaced with wires. The result is that the two values, i and j, can be concatenated to form the final address. This final transformation is in Figure 3-5. If only some conditions for strength reduction hold, then only a portion of these optimizations are applied.

Floating Point Division

Consider dividing a floating-point number with a constant that can be written as a factor of two, *e.g.*, $y = x/2.0$. Executing this division operation would take many cycles on a traditional floating-point execution unit.

However, by deconstructing the division into the micro-operations on its exponent and mantissa, and then generating specialized hardware for each, performance is increased by an order of magnitude. The micro-operation to perform such a division is to subtract 1 from the exponent of x. The time to execute the floating-point division operation reduces to the latency of a single fixed-point subtraction.

3.1.3 Ease of Implementation

Specialization of Boolean logic is commonly studied in logic or digital design. Large circuits can be re-optimized with tools such as ESPRESSO [25]. In practice, the target technology should be taken into account to minimize multiple objectives such as area, delay, and power. These objectives are common in commercial synthesis tools. Combinational specialization can be implemented with the following three steps:

1. identify static values from the input program,
2. substitute static values into architectural gates,
3. optimize with commercial tools.

In this way, logic optimization techniques developed for circuit synthesis, now a mature field, can perform the majority of the work for specializing combinational functions.

The disadvantage of relying on commercial synthesis tools is that they are often slow. They are tuned for hardware design, where long compilation time is more acceptable than in the software world. Thus, a good implementation lightens the synthesis burden by finding optimizations at as a high a level possible. For example, many arithmetic and bitwidth reductions discovered at the logic level are easily detected during earlier compiler phases. As another example, the address generation specialization described in the previous section can be implemented during array dismantling.

3.2 Specializing Storage Registers

3.2.1 Approach

There are two primary register optimizations:

1. register-to-wire conversion
2. bitwidth reduction

I do not consider optimizations at the register file level because of the availability of large quantities of individual registers in today's FPGAs. As an example, a register file can be partly specialized into smaller groups of registers with shared ports [79]. Likewise, register spilling is another mechanism that can be specialized.

Wire Discovery

When a value is consumed in the same cycle that it is produced, any registering can be eliminated altogether and replaced with a wire.

To explain specialization of registers to wires, I introduce sequencing. This concept will be covered more generally in Section 3.3. The basic idea of sequencing is that some operations will be scheduled to occur at different times during program executing. For example, consider the expression $A = B[i]; C = A + D$. The first assignment accesses an element of the array B and stores the result in the variable A. There is a dependence of the second assignment on the first (through the variable A). Because of this dependence, the addition cannot be scheduled earlier than the load. If the addition is scheduled one or more clock cycles after the load, then the value for the variable A must be stored in a register. However, with specialization, these two operations can be *chained* in the same clock period. In this case, if the value A is not used elsewhere, then the register can be replaced with a wire. This register-to-wire replacement policy applies to the case when all potential producers and consumers of an intermediate variable are scheduled in the same clock cycle.

Bitwidth Reduction

Just as the bitwidth of combinational functions can be specialized, so can the bitwidth of a register. I have mentioned bitwidth reduction in previous sections. Now, I take the opportunity to further elaborate those concepts and formally present a bitwidth optimization approach. In particular, datawidths of program variables and the operations performed on them can be used in both forward and backwards propagation of data-ranges. These data-ranges ultimately determine the minimum register bitwidths. Elimination of logic in combinational functions can be derived during logic synthesis once registers are appropriately reduced. Because FPGAs are register rich, the savings from eliminating registers comes from elimination of the associated logic.

Bitwidth Analysis Algorithm

Although a compiler could expect the programmer to annotate bitwidth information by specifying more precise data types [85] (e.g., int3, int13), a more elegant and satisfactory approach is to have the compiler determine the bitwidth of variables and operators. In addition to storage registers, a compiler can also use bitwidth analysis to determine function unit and memory widths.

The goal of bitwidth analysis is to analyze each static instruction in a program to determine the narrowest return type that retains program correctness. This information can in turn be used to find the minimum number of bits needed to represent each program operand. Library calls, I/O routines, and loops make static bitwidth analysis challenging. Because of these constructs, the analysis may have to make conservative assumptions about an operand's bitwidth. Nevertheless, with static analysis, bitwidth information can be inferred.

My approach to bitwidth analysis is to propagate data-ranges both forward and backward over a program's control flow graph (Stephenson et al. [127]). A data-range is a single connected subrange of the integers from a lower bound to an upper bound (e.g., [1..100] or [-50..50]). Thus, a data-range keeps track of a variable's lower and upper bounds. Because only a single range is used to represent all possible values for a variable, this representation does not permit the elimination of low-order bits. However, it does allow precise operation on arithmetic expressions. Technically, this representation maps bitwidth analysis to the more general *value range propagation problem.* Value range propagation is used in value prediction, branch prediction, constant propagation, procedure cloning, and program verification [108, 115].

Figure 3-6 shows a subset of the transfer functions for propagation. The forward propagated values in the figure are subscripted with a down arrow ($\downarrow$), and the backward propagated values with an up arrow ($\uparrow$). In general, the transfer functions take one or two data-ranges as input and return a single data-range. Intermediate results on the left are inputs to the transfer functions on the right.

Consider the transfer functions in Figure 3-6 in more detailed. The variables in the figure are subscripted with the direction in which they are computed. The transfer function in (a) adds two data-ranges, and (b) subtracts two data-ranges. Both of these functions assume saturating semantics that confine the resulting range to be between the bounds of its type. The AND-masking operation for signed data-types in (c) returns a data-range corresponding to the smaller of its two inputs. This operation uses the *bitwidth* function, which returns the bitwidth needed to represent the data-range. The typecasting operation in (d) confines the resulting range to be in the range of the smaller data-type. Because variables are initialized to the largest range that can be represented by their types, ranges are propagated seamlessly, even through type conversion. The function in (e) is applied when it is known that a value is in a specified range. For instance, this rule is applied to limit the data-range of a variable that is indexing into a static array. Note that rule (e) is not directionally dependent. Rule (f) is applied at merge points, and rule (g) is applied at locations where control-flow splits. In rule (g), x^b corresponds to an occurrence of x^a such that $x^a < y$. This information can be used to refine the range of x^b from the outcome of the branch test, $x^a < y$. As a final extension (not shown in the figure), note that rule (d) is not directionally dependent if b's value is dead after the assignment.

Forward Propagation

Initially, all intermediate variables (after renaming) are initialized to the maximum range allowable for their type. Informally, forward propagation traverses an SSA graph [45] in breadth-first order, applying the transfer functions for forward propagation. Because there is one unique assignment for each variable in SSA form, bitwidth analysis can restrict a variable's data-range if the result of its assignment is less than the maximum data-range of its type.

Backward Propagation

Forward propagation allows identification of a significant number of unused bits, sometimes achieving near optimal results. However, further minimization can be performed with *backward propagation.* For example, when data-ranges step outside of known array bounds, this fact can be back-propagated to reduced the data-range of instructions that have used its

(a)	$b_\downarrow = \langle b_l, b_h \rangle$ $c_\downarrow = \langle c_l, c_h \rangle$ $a_\uparrow = \langle a_l, a_h \rangle$	$\downarrow$ a = b + c $\downarrow$	$b_\uparrow = b_\downarrow \sqcap \langle a_l - c_h, a_h - c_l \rangle$ $c_\uparrow = c_\downarrow \sqcap \langle a_l - b_h, a_h - b_l \rangle$ $a_\downarrow = a_\uparrow \sqcap \langle b_l + c_l, b_h + c_h \rangle$
(b)	$b_\downarrow = \langle b_l, b_h \rangle$ $c_\downarrow = \langle c_l, c_h \rangle$ $a_\uparrow = \langle a_l, a_h \rangle$	$\downarrow$ a = b - c $\downarrow$	$b_\uparrow = b_\downarrow \sqcap \langle a_l + c_l, a_h + c_h \rangle$ $c_\uparrow = c_\downarrow \sqcap \langle a_l + b_l, a_h + b_h \rangle$ $a_\downarrow = a_\uparrow \sqcap \langle b_l - c_h, b_h - c_l \rangle$
(c)	$b_\downarrow = \langle b_l, b_h \rangle$ $c_\downarrow = \langle c_l, c_h \rangle$ $a_\uparrow = \langle a_l, a_h \rangle$	$\downarrow$ a = b & c $\downarrow$	$b_\uparrow = b_\downarrow$ $c_\uparrow = c_\downarrow$ $a_\downarrow = a_\uparrow \sqcap \left\langle -2^{n-1}, 2^{n-1} - 1 \right\rangle$, where $n = min(bitwidth(b_\downarrow), bitwidth(c_\downarrow))$
(d)	$b = \langle b_l, b_h \rangle$ $a = \langle a_l, a_h \rangle$	$\downarrow$ a = b $\downarrow$	 $a = a \sqcap b$
(e)	$x = \langle a_l, a_h \rangle$	$\downarrow$ $\{x_l \leq$ x $\leq x_h\}$ $\downarrow$	$x = x \sqcap \langle x_l, x_h \rangle$
(f)	${x^b}_\downarrow = \langle b_l, b_h \rangle$ ${x^c}_\downarrow = \langle c_l, c_h \rangle$ ${x^a}_\uparrow = \langle a_l, a_h \rangle$	x^b $\searrow$ $\swarrow$ x^c $x^a = \phi(x^b, x^c)$ $\downarrow$	${x^b}_\uparrow = {x^b}_\downarrow \sqcap {x^a}_\uparrow$ ${x^c}_\uparrow = {x^c}_\downarrow \sqcap {x^a}_\uparrow$ ${x^a}_\downarrow = {x^a}_\uparrow \sqcap ({x^b}_\downarrow \sqcup {x^c}_\downarrow)$
(g)	${x^a}_\downarrow = \langle a_l, a_h \rangle$ $y_\downarrow = \langle y_l, y_h \rangle$ ${x^b}_\uparrow = \langle b_l, b_h \rangle$ ${x^c}_\uparrow = \langle c_l, c_h \rangle$	x^a $\downarrow$ $x^a < y$ $\swarrow$ $\searrow$ x^b x^c	${x^a}_\uparrow = {x^a}_\downarrow \sqcap ({x^b}_\uparrow \sqcup {x^c}_\uparrow)$ ${x^b}_\downarrow = {x^a}_\downarrow \sqcap {x^b}_\uparrow \sqcap \langle a_l, y_h - 1 \rangle$ ${x^c}_\downarrow = {x^a}_\downarrow \sqcap {x^c}_\uparrow \sqcap \langle y_l, a_h \rangle$

Figure 3-6: A subset of transfer functions for bi-directional data-range propagation

deprecated value to compute their results. The back propagation algorithm begins at the node where the boundary violation is found and propagates the reduced data-range in a reverse breadth-first order, using the transfer functions for backward propagation. Back propagation halts either when the graph's entry node is reached, or when a fixed point is reached. (Note that a fixed point is reached when the algorithm finds a solution that cannot be refined further.) Forward propagation then resumes.

Sequence Solver

Along with data-range propagation, a solver can be applied to find closed-form solutions to loop-carried expressions using the techniques introduced by Gerlek et al. [58]. These techniques allow identification and classification of mutually dependent instructions called *sequences*. Figure 3-9 in Section 3.2.2 further elaborates this technique.

3.2.2 Examples

The following examples further clarify register specialization approaches.

Turning A Register into a Wire

Consider the code in Figure 3-7. Assume that the first two additions are "chained" in the same cycle and the value a is not consumed elsewhere. In this case the value for a should be carried on a wire (during that same cycle) and not stored in a register. Assume that the subtraction is scheduled in a later cycle. The value for e must be stored in a register[1].

```
a = b + 1; // scheduled on clock cycle 1
e = a + b; // scheduled on clock cycle 1
f = c - e; // scheduled on clock cycle 2
```

Figure 3-7: Example of turning a register into a wire

General Bitwidth Analysis Example

The C code fragment in Figure 3-8, an excerpt of the *adpcm* benchmark in the Deep Benchmark Suite (Chapter 6), is typical of important multimedia applications. Each line of code in the figure is annotated with a line number for the following discussion. The excerpt contains several structures, such as arrays and conditional statements, that provide bitwidth information. The bounds of an array can be used to set an index variable's maximum bitwidth. Other analyzable structures include AND-masks, divides, right shifts, type promotions, and Boolean operations.

Assume the precise value of `delta`, referenced in lines (1), (7), and (9), is not known. Because it is used as an index variable in line (1), its value is confined by the base and bounds of `indexTable`[2]. By restricting `delta`'s range of values, its bitwidth can be reduced.

[1]Under certain circumstances e could also remain on a wire. However, this case would violate the synchronous design principle that all values settle by the end of the clock period. In order to explore this case, a researcher would need to re-implement or modify backend tools such as timing analyzers.

[2]The analysis assumes that the program being analyzed is error free. If the program exhibits bound violations, arithmetic underflow, or arithmetic overflow, changing operand bitwidths may alter its functionality.

```
(1)  index += indexTable[delta];
(2)  if ( index < 0 ) index = 0;
(3)  if ( index > 88 ) index = 88;
(4)  step = stepsizeTable[index];
(5)
(6)  if ( bufferstep ) {
(7)    outputbuffer = (delta << 4) & 0xf0;
(8)  } else {
(9)    *outp++ = (delta & 0x0f) |
(10)             (outputbuffer & 0xf0);
(11) }
(12) bufferstep = !bufferstep;
```

Figure 3-8: Sample C code illustrating bitwidth analysis

This code fragment was taken from the loop of `adpcm_coder` in the adpcm multimedia benchmark.

```
addr = 0;
even = 0;
line = 0;
for (word = 0; word < 64; word++) {
  addr = addr + 4;
  even = !even;
  line = addr & 0x1c;
}
```

Figure 3-9: Example for bitwidth sequence detection

Similarly, the code on lines (2) and (3) ensures that **index** is restricted to be between 0 and 88. The AND-mask on line (7) ensures that **outputbuffer** is no greater than `0xf0`. Bitwidth analysis can also determine that the assignment to ***outp** on line (9) is no greater than `0xff` (`0x0f | 0xf0`). Finally, **bufferstep** is either *true* or *false* after the assignment on line (12) because it results from the Boolean *not* (`!`) operation.

Bitwidth Sequence Example

This example demonstrates sequence detection, an advanced optimization of bitwidth specialization. A sequence is a mutually dependent group of instructions that form a strongly connected component in a program's dependence graph. Figure 3-9 is an example loop with a tripcount range of $< 0, 63 >$. Knowing the tripcount allows analysis of sequences in the loop. Consider the calculation of *addr*. Bitwidth analysis can determine its range, $< 0, 255 >$, symbolically from the tripcount and the increment value. Note that the sequence includes the $addr = addr + 4$ increment instruction, but it does not include the initialization, $addr = 0$, since the initialization does not depend on any other instructions (it is loading a constant) and is therefore not part of a strongly connected component.

As mentioned, any strongly connected components in the body of a loop can be used to form a sequence dependence graph. In this example the sequence is a *linear sequence.*

In general, this result is more conservative than finding the precise range for each variable, but it works well in practice [127].

Unlike linear sequences, not all sequences are readily identifiable. In such cases, iterating over the sequence will reach a fixed point. For example, consider the calculation of *even* in Figure 3-9. A fixed point of $< 0, 1 >$ can be reached after only two iterations. Not surprisingly, sequences that contain Boolean operations, AND-masks, left-shifts, or divides – all common in multimedia kernels – can quickly reach a fixed-point. For cases when a fixed-point is not reached quickly, see my prior publication with Stephenson and Amarasinghe [127], which also gives more details of this example.

3.2.3 Ease of Implementation

Wire inferencing is a common optimization performed during RTL synthesis (Section 5.2.5). But register sharing (assigning more than one virtual register to a single physical register) can hide wires. Thus, wire inferencing should come earlier in the compiler flow (between scheduling and register allocation). One strategy is to annotate the registers that will become wires so that they will not participate in register allocated. Because wires do not consume resources explicitly, there is no reason to assume a bound on the wires used in a given clock cycle.

Deriving bitwidths from data-range propagation is well understood. However, analyzing loops and identifying common sequences can uncover many more reductions. At first, DeepC performed limited bitwidth analysis for address calculations. However, DeepC 2.0 uses a new set of compiler passes, called Bitwise [127].

3.3 Specializing Control Structures

3.3.1 Approach

A gate-level architecture does not have hardware support for control flow. There is no fetch-decode-execute sequence as in a von Neumann processor. However, the input sequential programs to be compiled are written in imperative languages. Efficient compilation of these languages to gates requires the ability to specialize their control flow.

How can this control flow be specialized? Consider the RTL equations for branching in a processor (Figure 3-10). Now consider the standard finite state machine (FSM) controller and datapath that can be created by logic synthesis tools (Figure 3-11). The controller consists of a state register and combinational logic that computes the next state as a function of 1) the previous state and 2) a condition from the datapath. In any given clock cycle, the values of the state register, as well as other datapath registers, are updated as a combinational function of the register state. All signal propagation must complete during the clock period (unless the circuit is wave-pipelined [31]). Without loss of generality, assume a global synchronous clock to each register. On a given clock cycle in the datapath, the output of a set of registers is routed through a set of multiplexers to function units and then back to registers. This register-mux-logic-mux-register structure is a standard in logic synthesis tools such as the Synopsys Behavioral Compiler [129].

My approach uses the input program as data for specializing von Neumann control flow mechanisms (Figure 3-10) into a controller and datapath (Figure 3-11). One way to translate imperative control flow into a controller and datapath is to have a program counter register that determines the state of the machine. In this case, a branch instruction implies

```
IF(OPCODE==JMP)
  PC=BRANCH_TARGET
ELSE
  PC=PC+1

IF(OPCODE==CMP)
  CONDITION = (R1 > IMMEDIATE)

IF(OPCODE==BGT)
  IF(CONDITION)
     PC=BRANCH_TARGET
  ELSE
     PC=PC+1
ELSE
   PC=PC+1
```

Figure 3-10: Simplified control flow of a von Neumann Architecture

Among other instructions, the opcode may be a jump (JMP), a compare (CMP), or a conditional branch, such as branch-if-greater-than (BGT). The program counter (PC) is by default incremented by one unless these instructions otherwise modify it.

manipulation of the program counter beyond the default of advancing to the next state. The datapath is then responsible for performing the appropriate calculations on each clock cycle.

To automate this translation, I take advantage of a well known folk theorem that all programs can be written as while programs [69] (or see Kozen's example [82] for a more recent discussion of this classic theorem). The structure of my while program consists of a case statement with a single large while loop. This structure is in Figure 3-12. In Verilog, an *always @(posedge Clock)* statement is executed indefinitely, once per clock cycle. RTL synthesis (Section 5.2.5), translates this code into a finite state machine. After reset, the state machine will advance to the following state. In any given state, the ellipses represent combinational work that is performed. An example of a looping operation on the program counter (a back edge in the control flow) is in state 4. An IF-THEN operation is in state 5. A state machine in this form is Turing general.

Two alternative ways to update the program counter include incrementing the PC and using zero-hot encoding (Figure 3-14). (Note that zero-hot encoding is an extension to one-hot encoding with a initial state vector of zero; for example: {000, 001, 010, 100}.) For small programs, I did not find an advantage from either alternative. As a heuristic, the current compiler uses an incrementer for procedures larger than 100 states. Zero-hot encoding was not effective because control signals usually are recombined when controlling resources shared across multiple states.

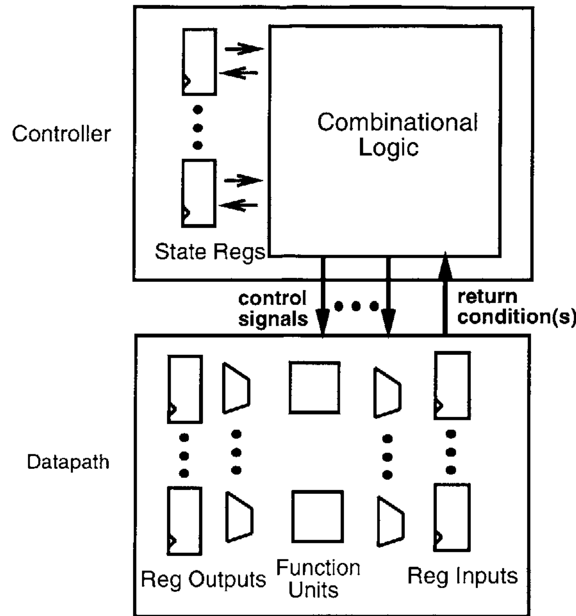


Figure 3-11: Controller and datapath

A synthesized design has two parts: a controller and a datapath. The controller includes state registers, combinational logic to compute the next state, and output control signals to the datapath. Return conditions connect from the datapath back to the controller, for data-dependent state transitions. The datapath contains function units sandwiched between register outputs and register inputs. Note that a single physical register contributes one input and one output. Multiplexers control the routing of registers to function units on each cycle of a global synchronous clock.

Predication: Converting IFs to Multiplexers

In some cases, an IF-THEN or IF-THEN-ELSE control structure can be efficiently represented as a multiplexer rather than state transitions. It must be reasonable to execute the body of the structure in a single clock cycle. In general, the traditional "if-conversion" compiler transformation, by predicating individual instructions, always satisfy this case. In DeepC 2.0, the default IF body must be manually sized to avoid resource conflicts and/or clock cycle "bloat" with this specialization. My approach is to emit Verilog IF-THEN or IF-THEN-ELSE code that is to be executed in a single state.

Basic Blocks

Although a general state machine can jump to any state from any other state, when the state machine is produced by specialization of a high level program, often straightline sets of states contain only *pc* increments in all but the last state. These correspond to basic blocks. Ideally, a basic block will take only one cycle to execute. However, both timing and resource constraints will lengthen basic block beyond one cycle. Timing constraints result from program dependencies and the evaluation time of operations in the target technology. Resource constraints arise when two or more operations share the same physical resource

```
always @(posedge Clock)
begin
  if (Reset) begin
    pc = 0;      // reset to initial state
  end
  else begin
    case (pc)
      0: begin
         ...
         pc = 1; // example of straight line control
      end
      1: begin
         ...
         pc = 2;
      end
      2: begin
         ...
         pc = 3;
      end
      3: begin
         ...
         pc = 4;
      end
      4: begin
         ...
         test1 = ...
         pc = (test1) ? 1: 5;   // example of LOOP type control
      end
      5: begin
         ...
         test2 = ...
         pc = (test2) ? 6 : 7;  // example of IF-THEN type control
      end
      6: begin
         ...
         pc = 7;
      end
      7: begin
         ...
         pc = 7;
         Finish = 1; // identify final state
      end
    endcase
  end
end
```

Figure 3-12: Structure of generic while program (in Verilog)

```
always @(posedge Clock)
begin
  pc = pc + 1;
  if (Reset) begin
    pc = 0;
  end
  else begin
    case (pc)
      0: begin
         ...
      end
      1: begin
         ...
      end
      2: begin
         ...
      end
      3: begin
         ...
      end
      4: begin
         ...
         test1 = ...
         if (test1) pc = 1;
      end
      5: begin
         ...
         test2 = ...
         if (!test2) pc = 7;
      end
      6: begin
         ...
      end
      7: begin
         ...
         pc = 7;
         Finish = 1;
      ...
```

Figure 3-13: Incrementer PC

```
always @(posedge Clock)
begin
  pc = pc << 1;
  if (Reset) begin
    pc = 0;
  end
  else begin
    case (pc)
      0: begin
         ...
      end
      1: begin
         ...
      end
      2: begin
         ...
      end
      4: begin
         ...
      end
      8: begin
         ...
         test1 = ...
         if (test1) pc = 1;
      end
      16: begin
         ...
         test2 = ...
         if (!test2) pc = 64;
      end
      32: begin
         ...
      end
      64: begin
         ...
         pc = 64;
         Finish = 1;
      ...
```

Figure 3-14: Zero-hot PC

and thus must be evaluated in non-overlapping time intervals.

Instruction Scheduling

I now introduce my approach to instruction scheduling in basic blocks. Scheduling is common in both the compiler and microprogramming fields as well as in architectural synthesis. My approach includes chained operations, multi-cycle operations, and pipelined function units. Chained operations are dependent operations that are executed combinationally in a single clock cycle. A multi-cycle operation is scheduled in one clock cycle, but does not complete until a later clock cycle. A pipelined function concurrently supports more than one multi-cycle operation with the constraint that only one operation can start at a time.

The goal of scheduling is to determine the precise clock cycle, $cycle_i$, to start each operation, i. Scheduling determines the concurrency needed in the resulting hardware; therefore, it is valuable to include resource constraints for memories and expensive functions such as multiplication.

I use the partial order relation $\prec$ to denote the precedence constraints on operations. Thus, $i \prec j$ denotes that operation i must complete before operation j begins. Precedence is derived from potential data dependencies. The predecessors of i in the data dependencies graph (DDG), a direct acyclic graph $G = (V, E)$, where vertices denote operations and edges denote data dependencies, is the set $pred_i \equiv \{j \in I \mid j \prec i\}$. All operations in $pred_i$ must be scheduled before i.

I consider both pipelined and unpipelined multi-cycle operations. For example, in modern processors a multiplier is usually pipelined, while a divider is not. By separating latency from throughput, either case can be modeled as well as the continuum between the two. Specifically, I use two parameters: latency and cost. The latency of operation v_i, denoted by l_i, is the minimum number of clock cycles after operation v_i is initiated before a dependent operation can be initiated. If $v_i \prec v_j$, then $\sigma_i + l_i < \sigma_j$, where σ_i is the completion time of operation v_i. The occupancy cost of operation v_i, denoted by c_i, is the number of clock cycles that operation v_i will occupy its resource. There are several possibilities:

- If $c_i = l_i$ then the operation is unpipelined.
- If $c_i = 1$ then the operation is fully pipelined.
- If $c_i < l_i$ then the operation is partially pipelined.

In the general case, an operation may consume more than one resource and c_i is augmented with a resource vector $res_{i,\tau}$, $\tau \in [0, c_i)$. This vector includes the set of resources required for i at each cycle, with offset $\tau = [0, c_i)$, during its execution. Space-time resource vectors come in handy in Section 4.2.1, which extends this approach to schedule communication as well as computation instructions.

Next, consider sub-cycle latencies. I assume that resources can only be re-used across clock cycles. Thus, $c_i \geq 1$. However, an operation may have a latency of less than one clock cycle. If $l_i < 1$, then the operation can be *chained* with other operations in the same cycle. In this context, chaining refers to the scheduling of an operation in the same clock cycle as the operations predecessor. Chaining is common in high-level synthesis and the Pentium 4 has an ALU that can support chain-like operations (although a faster clock is used as well). In order to chain, the combined latency of the chained operations, which may be a tree, not just a linear sequence, must be less than one clock period.

Chaining and bitwidth reduction complicate sub-cycle latency calculations. Chaining is further complicated because sub-cycle latencies can be a function of both the data and the preceding operation. For example, an addition followed by another addition would be faster than an addition followed by a store. Although a bit-by-bit timing analysis can be carried out (Park et al. [107]), I instead define latency as a function of both parent and child: l_{ij}. With bitwidth reduction, generic 32-bit or 64-bit operations will be replaced with smaller, faster operations. Thus, l_{ij} must be adjusted to match the bitwidth of each operation.

I solve the scheduling problem with a forward, cycle-driven, critical-path-based, list scheduler [102]. In contrast to backward scheduling, forward scheduling assigns the roots of the precedence graph in the earliest possible time slots and moves forward through the graph, scheduling all operations that are *ready*. As opposed to operation-driven scheduling, cycle-driven scheduling greedily solves each cycle before moving on the next cycle. Finally, basing the scheduler's selection heuristic on the critical-path avoids congesting resources with non-critical work. A critical-path schedule is the same as a height/depth/level-priority schedule that uses the length of the longest path from an operation to the leaves of the precedence graph as a priority function. These approaches are well known in the scheduling community.

The specification of my algorithm is in Figure 3-15. The given inputs have been discussed in the preceding few paragraphs. Briefly, a *ReadyList* is kept updated with unscheduled operations. Operations are sorted by a priority function, *CriticalSelect*, which returns the instructions with the longest chain of successors. An operation can be scheduled to a resource only when that resource is available. Operations may begin and end in the middle of the cycle, with the worst latency for preceding instructions determining the start time. However, except for multi-cycle operations, instructions must complete by the end of the cycle. Once an instruction is scheduled, it is removed from the set of unscheduled instructions I, and the availability of the consumed resource is decremented. Finally, *ReadyList* is updated and the algorithm continues until all operations are scheduled. My implementation of this algorithm is further described in Section 5.2.3.

3.3.2 Examples

The following examples further clarify control structure specialization approaches.

Loop Nest Example

Consider a loop nest with a body that takes three cycles to execute (Figure 3-16). Also consider the resulting state transitions (Figure 3-17). Because there is no fetch-decode-execute cycle, the update of the program counter is always a direct calculation of the previous state. In the new state, the circuit has the information it needs to compute the appropriate control signals without an instruction fetch or decode. Furthermore, the resulting state machine contains the artifacts predicted by the specialization theory of Jones et al. [80]: the control flow of the residual program (the output) matches the control flow of the static specialization data (the input program) while the operations match the mechanisms in the algorithm being specialized (the von Neumann architecture). These artifacts provide evidence that supports my thesis.

Given:

I : set of unscheduled instructions
$pred_i$: precedence set for each instruction i
c_i : cost of instruction i
l_{ij} : latency for instruction i preceded by j
$\forall l_{ij} > 1; l_{ij} = \lfloor l_{ij} \rfloor$: no non-integer delays greater than one
l_i : worst latency for instruction i preceded by any instruction
$res_{i,\tau}$: resource vector for instruction i at time offset $\tau \in [0, c_i)$
R : set of resources
n_r : number of resources of type r
T : set of cycles
σ_i : (intermediate) completion time, not necessarily integer, for instruction i
$avail_{r,t}$: (intermediate) availability of resource r at time t

Produce:

$\forall\ i \in I,\ cycle_i$: output schedule time $cycle_i$ for each instruction i

```
Procedure Schedule {
    t ← 0
    S ← ∅
    ∀ i ∈ I; σ_i ← ∞
    ∀ r ∈ R, t ∈ T; avail_{r,t} ← n_r
    ReadyList ← {i ∈ I | pred_i = ∅}
    while (I ≠ ∅) {
        i = CriticalSelect(ReadyList) // select most critical instruction
        cycle_i = t
        σ_i = max(t + l_i, max_{j∈pred_i} σ_j + l_ij) // fraction ending time
        I = I \ i
        τ ∈ [0, c_i), r ∈ res_{i,τ} : avail_{r,t+τ} − −
        if (I ≠ ∅) {
            do {
                ReadyList ← {i ∈ I | ∀ j ∈ pred_i, σ_j + min(l_ij, 1) < t + 1;
                                    ∀τ ∈ c_i, ∀ r ∈ res_{i,τ}, avail_{r,(t+τ)} > 0}
                if (ReadyList = ∅) t + +
            } until (ReadyList ≠ ∅)
        }
    }
```

Figure 3-15: Forward, cycle-driven, critical-path-based list scheduler

Supported features includes fractional delays instructions consuming multiple resources across multiple cycles. *CriticalSelect* contains the priority heuristic, in this case returning the instruction with the longest chain of successors.

```
for (i = 0; i < 64; i++) {
  for (j = 0; j < 64; j++) {
    [body1]
    [body2]
    [body3]
  }
}
```

Figure 3-16: Loop nest with three cycle body

```
   ...
0: begin
   i = 0;
   pc = 1;
end
1: begin
   j = 0;
   pc = 2;
end
2: begin
   j = j + 1;
   [body1]
   pc = 3;
end
3: begin
   [body2]
   testj = (j < 64);
   pc = 4;
end
4: begin
   [body3]
   pc = testj ? 2 : 5;
end
5: begin
   i = i + 1;
   testi = (i < 64);
   pc = test ? 1 : 6;
   ...
```

Figure 3-17: Transitions for loop nest with three cycle body

```
if (i == 0)
 a = 1;
else
 b = 1;
```

Figure 3-18: IF-THEN-ELSE construct

```
   ...
0: begin
   test = (i==0);
   pc = test ? 1 : 2;
end
1: begin
   a = 1;
   pc = 3;
end
2: begin
   b = 1;
   pc = 3;
end
3: begin
   ...
```

Figure 3-19: State transitions for IF-THEN-ELSE construct

If-then-else Example

Consider an IF-THEN-ELSE construct (Figure 3-18) and the resulting state transitions (Figure 3-19). This example is clear, but what happens in the unspecialized case? The difference between setting the *pc* to 1 versus 2 only changes a single register bit, while in the unspecialized case a large program counter would be updated, an instruction memory or cache accessed and loaded into an instruction register, that instruction register would be decoded, and only then would the machine "know" to conditionally change its *pc*.

Predication Example

Consider an if-then-else construct as in Figure 3-20. The control flow can be eliminated with predication. The resulting Verilog is in Figure 3-21. Because synthesis is required to complete both the test and the assignment in clock cycle 0, the resulting digital logic includes a multiplexer.

Next, consider an if-then-else construct in Figure 3-22. In this case, more than one multiplexer is needed. However, embedding of the IF statement in Verilog can leverage RTL synthesis to generate predicating multiplexers (Figure 3-22).

Procedure Call Example

I briefly describe an example procedure call. Small, non-recursive procedures can be inlined. When procedures are not inlined, a callsite must pass return information to the procedure. Consider Figure 3-24 and the resulting Verilog (without inlining) in Figure 3-25. Rather

```
if (i == 0)
   a = b;
else
   a = c;
```

Figure 3-20: Predication example: IF-THEN-ELSE construct

```
   ...
0: begin
   test = (i==0);
   a = test ? b : c;
   pc = 1;
   ...
```

Figure 3-21: Predication example: Verilog after eliminating control flow

```
if (i == 0)
   a = b;
else
   c = d;
```

Figure 3-22: Another predication example

```
   ...
0: begin
   test = (i==0);
   if (test) begin
     a = b;
   end
   else begin
     c = d;
   end
   pc = 1;
   ...
```

Figure 3-23: Another predication example: resulting Verilog

```
main {
...
if (i == 0)
 d = mult(a,b);
else
 d = mult(a,c);
...
}
mult(in1,in2,out) {
out = in1*in2;
}
```

Figure 3-24: Procedure call example

than storing the program counter state, the compiler can optimize for the two callsites present, using only one dynamic bit to resolve the return state. The callsite state pointers are specialized into the pc calculation in state 5.

Besides just encoding the return state, support for procedure calls, especially recursive ones, requires saving and restoring other registers and managing a call stack. Although this support is beyond the scope of DeepC 2.0, a casual reading of assembly-code for RISC processors with general-purpose registers (MIPS for example) is enough to see that compilers already specializes procedure calls — calls are dismantled into register, memory, and branch instructions. Specialization to logic additionally requires dismantling the remaining jump-and-link instruction (an optimized instruction combining a register move and a branch).

3.3.3 Ease of Implementation

The basic concept of sequencing straight-line code was introduced as a scheduling problem; however, during implementation the scheduling interactions are more complex. Critical path timing calculations must be extended to fractional cycles. Generation of specialized Verilog control flow from program control flow is not difficult once the expectations of the RTL synthesize tool being used are well characterized. Although general control flow can be treated with this approach, restructuring compiler passes, which find loops and IF-THEN-ELSEs from GOTOs, are valuable. Although partial unrolling of loops is easy and advantageous, the more-difficult-to-implement strategy of loop pipelining is known to produce more compact circuits.

The controlling state machine totally replaces the program counter, instruction register, and instruction memory in a basic RISC architecture. Unrestricted specialization of a large program will lead to inefficiencies that can only be overcome by compressing the unimportant states back into a ROM-type structure. This problem can be avoided with configuration selection and configuration sequencing algorithms [92]. These algorithms select important loops and use caching strategies, with only a portion of the program loaded at any given time, to support longer programs. Alternately, known techniques for placing large decoders into FPGA block RAMs may solve this problem by indirectly re-compressing over-specialized instructions [2].

```
always @(posedge Clock)
begin
  if (Reset) begin
    pc = 0;      // reset to initial state
  end
  else begin
    case (pc)
      0: begin
         ...
         test = ...
         pc = (test) ? 1 : 3; // branch
      end
      1: begin
         in1 = a;
         in2 = b;
         callsite = 0;
         pc = 5; // first call site
      end
      2: begin
         d = out;
         pc = 6; // unconditional jump
      end
      3: begin
         in1 = a;
         in2 = c;
         callsite = 1;
         pc = 5; // second call site
      end
      4: begin
         d = out;
         pc = 6; // unconditional jump
      end
      5: begin   // multiply sub-procedure
         out = in1 * in2;
         pc = callsite ? 2 : 4; // return
      end
      6: begin
         ...
         pc = 6;
         Finish = 1; // identify final state
      end
    endcase
  end
end
```

Figure 3-25: While Program with procedure call (in Verilog).

3.4 Summary

This chapter introduced approaches for specializing basic architectural mechanisms: combinational functions, registers, and control structures. These approaches included several new techniques. For combinational functions, which conventional logic synthesis can handle at the low level, specialization of language-specific logic, such as address generation, is introduced earlier in the compiler flow. For registers, this chapter introduced general techniques for discovering wires and for performing bitwidth analyses on data flows and sequences in loops. For control structures, this chapter introduced specializing to a Verilog while loop that can evaluate any program. This chapter also discussed support for predication and procedures calls. These constructs are the building blocks of stored program computers. Thus, they must be specialized for gate reconfigurable architectures to be competitive with other evaluation modes. The following chapter continues with specialization of more advanced mechanisms: memory and communication.

Chapter 4

Specialization of Advanced Mechanisms

Scaling of the technology to higher densities is producing effects that may be clarified by analogy with events in civil architecture. Decades ago, standard bricks, two-by-fours, and standard plumbing were used as common basic building-blocks. Nevertheless, architects and builders still explored a great range of architectural variations at the top level of the time: the building of an individual home. Today, due to enormous complexities of large cities, many architects and planners have moved on to tackle the larger issues of city and regional planning. The basic blocks have become the housing tract, the business district, and the transportation network. While we may regret the passing of an older style and its traditions, there is no turning back the forces of change.

In present LSI, where we can put many circuits on a chip, we are like the earlier builder. While we no longer tend to explore and locally optimize at the circuit level (the level of bricks and two-by-fours), we still explore a great range of variation at the level of the individual computer system. In future VLSI, where we may put many processors on a chip, architects will, like the city planners, be more interested in how to interconnect, program, and control the flow of information and energy among the components of the overall system. They will move on to explore a wider range of issues and alternatives at that level, rather than occupy themselves with the detailed internal structure, design, and coding of each individual stored program machine within a system.

— *Mead and Conway reflecting on the classical stored-program machine,*
from their seminal VLSI textbook:
Introduction to VLSI Systems (1980), page 216.

The basic architectural mechanisms of the previous chapter concerned small regions of hardware, limited to combinational logic, registers, and a single finite state machine. Semiconductor technology now offers a wealth, or some say an excess, of hardware. So, this chapter turns to specialization of *advanced* architectural mechanisms, with a focus on specializing mechanisms for distributed architectures. This focus is motivated by Mead and Conway's correct projection: architects are now more concerned with interconnecting distant regions of a chip than with new designs for basic functions[1].

[1]However, in some sense the quote goes against my thesis because Mead and Conway did not forecast the role of high-level compilation in specializing both the basic building blocks and the interconnect. Designers can occupy themselves with system-level issues while the compiler manages the details.

Specialization of basic mechanisms (Chapter 3) either eliminates logic or dismantles features expected by the language, but not supported in the architecture. In contrast, specialization of advanced mechanisms is not required by language semantics, but instead is needed for high-performance, given physical constraints. Sequential programs can be mapped to parallel architectures by unrolling loops across space, allowing application of dense hardware, with billions of transistors, to a single problem. While unrolling exposes parallelism, the resulting distributed programs expose physical constraints such as wire delay and power limitations.

This chapter divides these issues into approaches for specializing *memory mechanisms* and *communication mechanisms*. Memory and communication mechanisms are interdependent; for example, consider the decomposition of a large memory structure into smaller memories and send/receive messages. Furthermore, because these mechanisms are not completely understood, their specialization is even more challenging.

The advanced mechanisms in this section borrow heavily from the mechanisms of the Raw Project (www.cag.lcs.mit.edu/raw). When possible, they leverage and reuse Raw approaches directly. Some Rawcc compiler approaches are extended or otherwise generalized. Some parts of the Raw hardware — namely the static router and the concept of tiling — are moved to software. I make these distinctions clearer when discussing DeepC implementation details in Chapter 5. In any case, the following approaches are indebted to other Raw Project members and their ideas.

4.1 Specializing Memory Mechanisms

4.1.1 Approach

The conventional memory model of imperative, von Neumann-style languages is a single, giant address space. Even in large parallel systems, there is a strong bias to support a shared address space for all memory. The goal of memory mechanism specialization is to decompose this giant address space, often called the *main memory*, into many small address spaces. Physically, these smaller spaces can occupy small local memories, allowing computation to be migrated much closer to its data than in traditional systems. Specialization of memory, by distinguishing which memory location an address points to at compile time, also exposes parallelism in the input algorithm; disambiguated loads and stores can be granted unhindered access to memory resources, with no potential for conflict.

I motivate this approach with an example. Consider Figure 4-1 (a), a 32x1 memory implemented as a single multiplexer. Only the read port is shown, followed by some computation logic. Functionally, this multiplexer can be thought of as a tree of many smaller multiplexers. We can divide the multiplexer into two levels, as in Figure 4-1 (b), creating two sets of storage bits. A small 2-1 multiplexer, controlled by one of the address bits, chooses between the two sets. Finally, in Figure 4-1 (c), we can expose the inputs to the 2-1 multiplexer and separate the computation logic into two smaller functions. If we know statically which set of storage bits are referenced by a memory access, then that access and its associated logic can be assigned to the computation logic close to those bits. Note that the computation partitions may need to communicate if there are dependencies between other non-memory functions in the computation. Specialization of this resulting communication will be addressed in Section 4.2.

Besides partitioning the main memory and specializing bank multiplexers, my approach also includes specialization of address generation logic as introduced in an example in Sec-

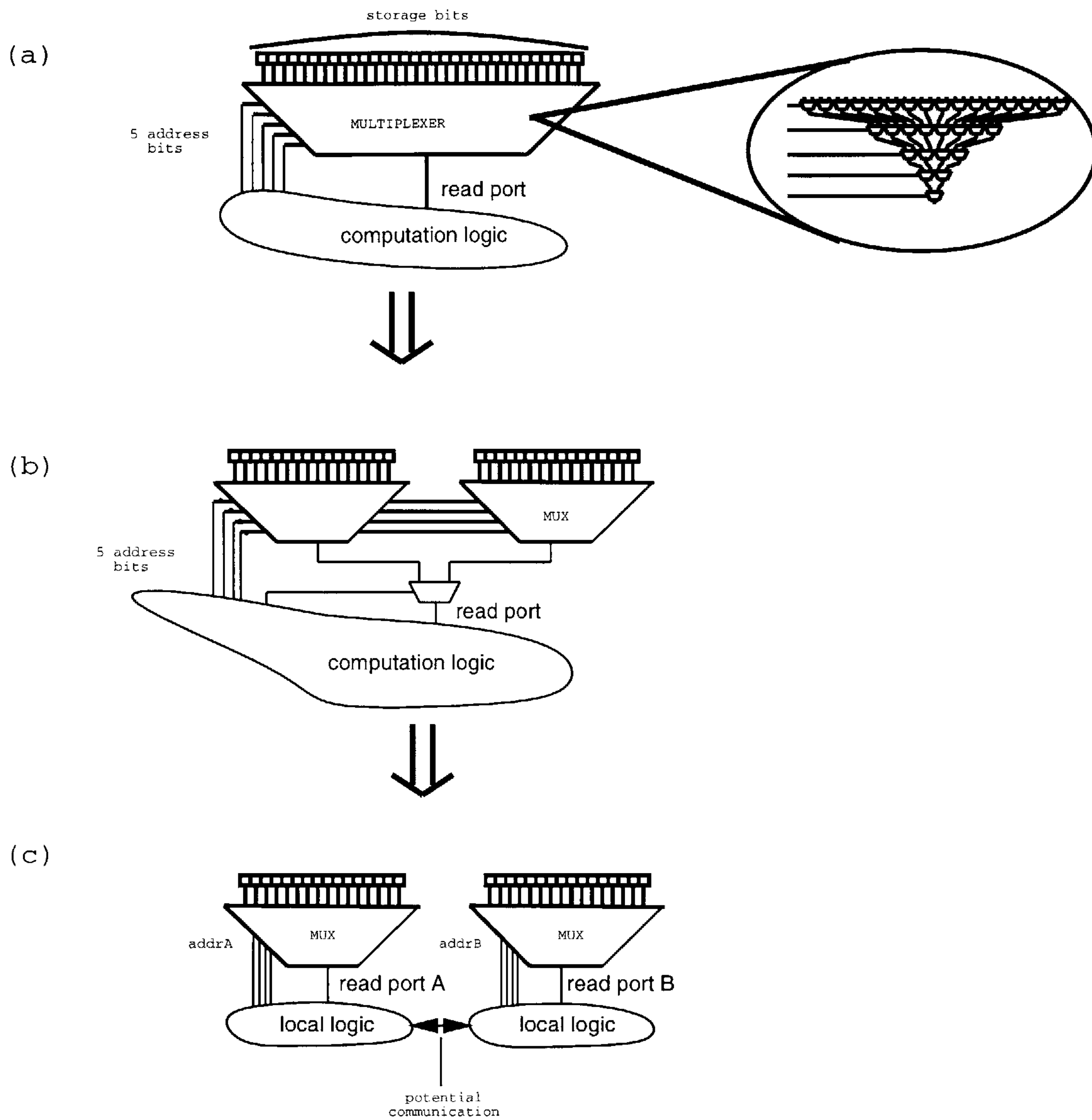


Figure 4-1: Memory specialization example

(a) A 32x1 memory implemented as a single MUX. (b) A 32x1 memory implemented as two 16x1 memories and a 2-1 multiplexer. (c) Two 16x1 memories with separate ports and locally assigned computation.

tion 3.1.2. Specialization of more advanced memory structures, beyond the approaches here, include software specialization of caching structures (Moritz et al. [104]), and compiler-directed adaptation of cache line size (Gupta et al. [134]). The positive results in these papers support my thesis by demonstrating the feasibility of specializing other advanced memory mechanisms.

Although code generation algorithms to create smaller memories from larger memories are not difficult, algorithms to determine which memories can be decomposed are challenging. Programs with pointers and large arrays require special analysis to disambiguate loads and stores at compile time. Luckily, memory disambiguation algorithms such as pointer analysis are available from the broader compiler community. The following sections explain how I integrate these analyses into my approach. They also demonstrate how to extend bitwidth analysis techniques to handle pointers and arrays.

Pointer Analysis

Without pointer analysis, a single pointer in the input program will thwart memory specialization. This is true because the data structures in C-like high-level languages share a single address space. Fortunately, disambiguation of pointers in sequential programs is a well-understood problem with available compiler passes that can be reused. To disambiguate targets of loads and stores, I use Rinard and Rugina's SPAN [114], a flow-sensitive, context-sensitive interprocedural pointer analysis package. *Flow sensitive* analysis takes the statement ordering into account, providing more precise results than flow-insensitive algorithms. *Context sensitive* analysis produces a different result for each different calling context of a procedure. All memory references are tagged with *location set* information. Location set information is also determined for every pointer in the program. A pointer's *location set list* records the abstract data objects that it can reference. Pointer analysis can determine the location set a pointer *may* or *must* reference. SPAN also distinguishes between *reference location sets*, and *modify location sets*: a reference location set is a location set annotation that occurs on the right hand side of an expression (such as a read), whereas a modify location set occurs on the left hand side of an expression (such as a write).

Equivalence Class Unification

Given the results of pointer analysis, my approach is to apply a static promotion technique called Equivalence Class Unification (ECU), coined by Barua et al. [16]. ECU collects memory objects into groups called equivalence classes. The goal is to form classes where the class of every memory reference can be determined statically, at compile time. While the equivalence class must be statically determinable, the exact object to be selected in a class may be selected dynamically, at runtime.

ECU works by first determining *alias equivalence classes* — groups of pointers related through their location set lists[2]. Pointers in the same alias equivalence class can alias to the same object; pointers in different equivalence classes can never reference the same object.

Unification takes place when all objects referenced by the same alias equivalence class are formed into a memory. In the worst case, for example if a pointer aliases every memory object, unification will create a single memory. In practice, multiple classes are found,

[2]More formally, alias equivalence classes are the connected components of a graph whose nodes are pointers and whose edges are between nodes whose location set lists have at least one common element.

resulting in many smaller memories. Then, all pointers to an alias class will reference a single small memory associated with that class.

Pointer Representation In the ECU approach, pointer information is encoded in the memory and port *name* that is referenced and in the *address* within that memory. If more than one array is stored in the same RAM (because of aliasing), then the address will have a *base* and *offset* in the traditional style of computer addressing modes. In the static case, the port name is evaluated at compile time, forming wires and eliminating multiplexers. That is, disambiguated references do not need selection logic to determine which memory they should access. When a base address is needed, if that base address is known in a given state, this address becomes an immediate value that can be encoded in the controller state machine rather than in a general register. Beyond this specialization, the address itself can be bitwidth reduced.

Partial Unification ECU is an all or nothing approach. If a single reference ever has the opportunity to access two memory blocks, then those blocks must be located in the same memory. This works well for arrays in some applications. However, there are many instances where partial unification is more desirable. When objects are partially unified, there will exist dynamic references that must be resolved at runtime. These references are resolved with multiplexers. A portion of the address controls the multiplexer in the same manner as the decoders in a conventional RAM. The asymmetry here is in sharp contrast to the more regular structures created by full ECU and the modulo unrolling approach in Section 4.1.1.

Example Figure 4-3 shows a short piece of code with two arrays, C and D. The points-to-set of the pointer q is $< C, D >$; the points-to-set of the pointers p is $< C >$. With ECU, C and D will be in the same equivalence class and assigned to the same memory. With partial unification, the arrays can be decomposed. Figure 4-2 shows the read ports after decomposition. The pointer p has a dedicated port to C while q selects either C or D as a function of its most significant bit (MSB), a common encoding. A control multiplexer is inferred on C's address port. However, the control multiplexer on D's address port has been specialized away.

More complex multiplexer formations are possible. Hardware synthesis research is beginning to consider similar issues, including pointer encoding (Semeria et al. [120]), and synthesis of structures to support dynamic memory allocation(Semeria et al. [119]). This work supports my thesis claim by demonstrating that good algorithms for advanced memory specialization either already exist, or can be designed from existing CAD approaches. For example, the pointer encoding algorithm cited is like an earlier approach for state minimization.

Modulo Unrolling

A major limitation of equivalence class unification is that arrays are treated as single objects that belong to a single alias equivalence class. Mapping an entire array to a single memory sequentializes accesses to that array and destroys the parallelism found in many loops. Therefore, my approach is to leverage another advanced strategy, called *modulo unrolling* (Barua et al. [15]), to distribute arrays across different memories. This approach is similar to the bank allocation strategies for VLIWs, originally proposed by Ellis [51]).

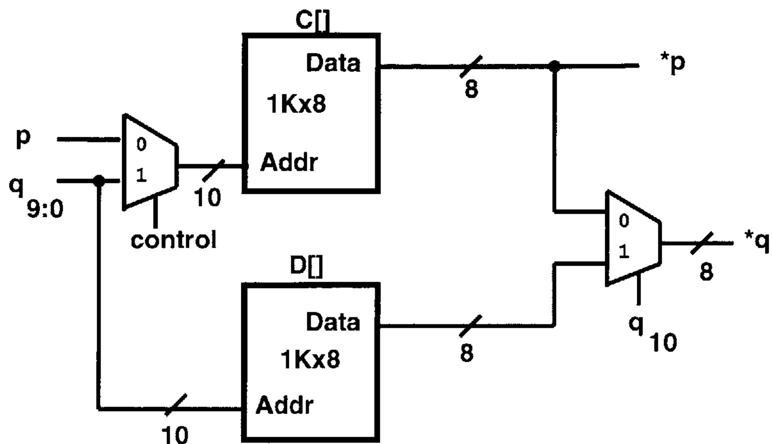


Figure 4-2: Logic for partial unification example

```
int C[1024];
int D[1024];
...
p = C;
if(foo) {
  q = C;
else
  q = D;
}
for(i=0; i<1024; i++) {
  *q = i;
  *p = 0;
  p++;
  q++;
}
...
```

Figure 4-3: Partial unification example

There are three cases: (a) when p references C, no output multiplexer is needed, (b) when q references D, no input multiplexer is needed, but in (c) when q references C, both multiplexers are required. Note that p never references D.

Modulo unrolling can be applied to array accesses whose indices are affine functions of enclosing loop induction variables. First, arrays are partitioned into smaller memories through *low-order interleaving*. In this scheme, consecutive elements of the array are interleaved in a round-robin manner across different memory banks. Given this data layout, modulo unrolling can perform static promotion on the array accesses in certain loops.

Modulo unrolling first identifies *affine* array accesses inside loop-nests. An unroll factor is computed for each loop. This factor is determined for each affine array access in the loop from its enclosing loop-nest. The symbolic derivations of the minimum unroll factors and the code generation techniques needed are described further in Barua et al. [15]. Given the unroll factors for each affine access, the final unroll factor for any loop is computed with a least common multiple *lcm* function. This *lcm* function often leads to an undesirable side effect of modulo unrolling: code explosion. Section 6.6.1 has more to say about how this problem affects synthesized structures.

Memory and Pointer Bitwidth Analysis

Just as the width of a register can be minimized, so can the width of an entire array or other structure. To perform this advanced specialization, the bitwidth analysis of Section 3.2.1 is extended to treat arrays and pointers as scalars. When treating an array as a scalar, if an array is modified, bitwidth analysis must insert a new SSA ϕ-function to merge to array's old data-range with the new data-range. A drawback of this approach is that a ϕ-function is needed for every array assignment. Every element in an array will have the same size — in this approach, this size becomes the memory width for the resulting embedded RAMs. The following example demonstrates how to extend SSA to treat de-referenced pointers in exactly the same manner as scalars.

Example: Consider the following C memory instruction, assuming that `p0` is a pointer that can point to variable `a0` or `b0`, and that `q0` is a pointer that can only point to variable `b0`:

```
*p0 = *q0 + 1
```

The location set that the instruction may modify is $\{a0, b0\}$, and the location set that the instruction must reference is $\{b0\}$. Because there is only one variable in the instruction's reference location set, it *must* reference `b0`. Because there are two variables in the modify location set, either `a0` or `b0` *may* be modified.

To keep the SSA guarantee that there is one unique assignment associated with each variable in the instruction's modify location set, `a0` and `b0` must be renamed. Furthermore, because either variable may be modified, a ϕ-function has to be inserted for each variable in the modify location set to merge the previous version of the variable with the renamed version:

$\{\texttt{a1}, \texttt{b1}\} = \{\texttt{b0}\} + 1$

$\texttt{a2} = \phi(\texttt{a0}, \texttt{a1})$

$\texttt{b2} = \phi(\texttt{b0}, \texttt{b1})$

If the modify location set has only one element then the element *must* be modified, and a ϕ-function is not inserted.

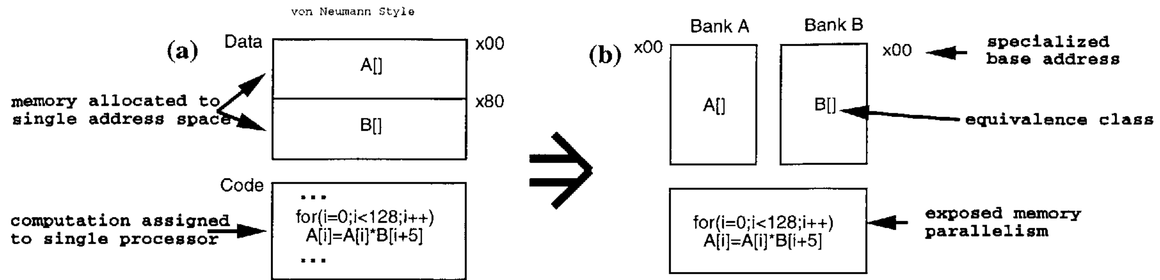


Figure 4-4: Equivalence class unification
(a) initial program; (b) after equivalence class unification;

4.1.2 Examples

Equivalence class unification

Figure 4-4 introduces a running example that illustrates the steps for advanced specialization of memory. This same example will continued in Section 4.2. Figure 4-4(a) shows the initial code and data fed to the compiler. The code contains a *for* loop with affine references to two arrays, A and B. Initially, the two data arrays are mapped to the same monolithic memory bank. Figure 4-4(b) shows the results of equivalence class unification. Because no static reference in the program can address both A and B, pointer analysis determines that A and B are in different alias equivalence classes. This analysis allows the two arrays to be mapped to different memories while ensuring that each memory reference only addresses a single memory. In the figure, this mapping results in a specialized base address (zero) for each array. Also, this technique has exposed memory parallelism between the two equivalence classes.

Modulo Unrolling

Figure 4-5(c) shows the result of four-way low-order interleaving and Figure 4-5 (d) shows the following output of the unrolling phases of modulo unrolling. Low order interleaving splits each array into four sub-arrays whose sizes are a quarter of the original. Modulo unrolling uses symbolic formulas to predict that the unroll factor for static disambiguation is four. In this figure, it is apparent that each reference always goes to one memory. Specifically, A[i], A[i+1], A[i+2] and A[i+3] access sub-arrays 0, 1, 2 and 3; B[i+5], B[i+6], B[i+7] and B[i+8] access sub-arrays 1, 2, 3 and 0. In the figure, the array references have been converted to reference the partitioned sub-arrays, with appropriately updated indices. Thus bank selection is now determined statically, eliminating the bank multiplexers formed after unrolling (c).

4.1.3 Ease of Implementation

Pointer analysis and equivalence class analysis are available as research compiler passes in the MIT compiler infrastructure. However, leveraging these passes into a more complex compiler flow is challenging. Bitwidth analysis of memories and pointers is more difficult than data-range propagation for scalars. Supporting arrays and pointers requires extensions to standard SSA representation.

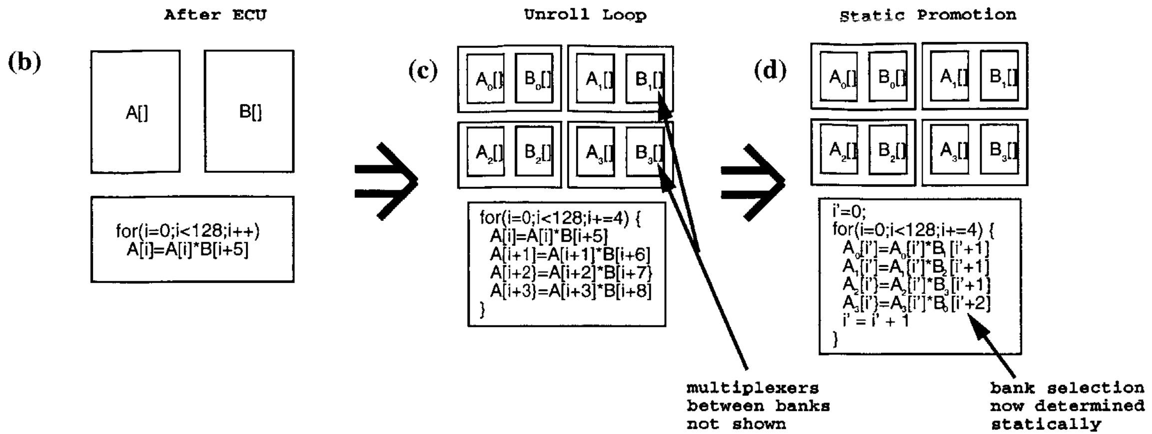


Figure 4-5: Modulo unrolling in two steps
(b) after equivalence class unification; (c) after modulo unrolling step 1; (d) after modulo unrolling step 2. The index for array B now is assigned a new base offset, $\lfloor \frac{oldbase}{4} \rfloor$, a result of static promotion.

Of these techniques, only modulo unrolling can expose large amounts of parallelism. However, the decision to leverage modulo unrolling leads to considerable backend headaches because of code explosion. Code explosion can translate into large and thus slow state machines that can ultimately defeat the point of parallelization. For example, in Figures 6-17 — every 10× increase in states leads to 3× reduction in clock frequency! Attempting to handle the most general loops yields an approach that unrolls too much and creates low-order interleaving that destroys locality — low-order interleaving splatters physical memory space with objects that are otherwise close together in program memory space. For example, the second and third element of an array, may be spread across different memory banks. Thus, more work is needed to develop an approach for practical integration of modulo unrolling.

One architectural technique to integrate with modulo unrolling is *pipelining*. Pipelining in software is a well-known compiler optimization that exposes parallelism with minimum unrolling. Other researchers, including Petkov et al. [111] and Callahan et al. [33], have begun applying and improving pipelining techniques for hardware synthesis and for reconfigurable architectures. Their work demonstrates that the overheads of unrolling can be kept in check. Their work also demonstrates that pipelined architectural mechanisms can be specialized, supporting my thesis.

In synthesis, memory is not as standardized as registers and logic, so work is needed to generate the appropriate memory primitives for a target technology. Furthermore, low-level module generation work is needed to optimize the size of the memories generated and the size of the memory blocks in gate reconfigurable architectures. These blocks must be wired together to form larger memories. Although tedious, this work is well defined.

4.2 Specializing Communication Mechanisms

My approach to specializing communication mechanisms is to specialize tiled architectures. Because tiled architectures are new, this section departs slightly from the format of my

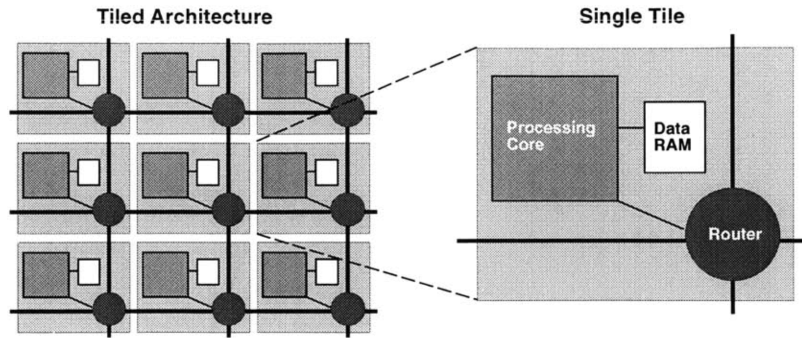


Figure 4-6: Tiled architecture

A tiled architecture comprises a replicated set of identical tiles. Each tile contains a processing core, data RAM, and a router that interconnects neighboring tiles. Both the processing core and the router can evaluate in any of the Modes introduced in Chapter 2. Although a single memory is shown in each tile, this memory may be composed of many smaller memories.

previous approach sections. The following paragraphs first introduces tiled architectures, before Section 4.2.1 describing how to specialize them.

Introduction to Tiled Architectures

Unlike the more familiar computer architecture mechanisms described thus far, the architectural mechanisms specialized here are less traditional and are from parallel processing. The goal is to specialize architectures that consist of a set of regularly connected *tiles*. A tile is a region of hardware in which communication is cheap (local) and between which communication is expensive (global). Inside each tile is a machine that communicates with machines in other tiles through a pipelined, point-to-point network.

Spatial tiling helps us separate the *micro-architectural* design issues from the *macro-architectural* issues. Until now, I have focused on micro-architectural design issues, defining how computation takes place in a tile-sized region of hardware. Macro-architectural design issues focus on how communication takes place between tiles, independent of how computation is executed in a single tile.

Tiling and tile sizes are motivated by two sets of constraints. First, computation and memory density constraints are imposed by the current technology feature size. In the limit, mass-producing atomic-sized features is difficult. Second, physical interconnect constraints are imposed by wire delays. In the limit, communicating near light-speed over short distances is difficult.

Given density constraints, the tiling approach allocates the area in each tile to three parts: a processing unit for computation, a data memory for storage, and a router. These tiles are arranged in an array, as in Figure 4-6. Their regular nature simplifies device layout, scalability, and verification.

Given interconnect constraints, tiling introduces a point-to-point communication network between neighboring tiles. This work is restricted to *static* communication networks. A network is static if the space-time location of both the sender and the receiver of a message

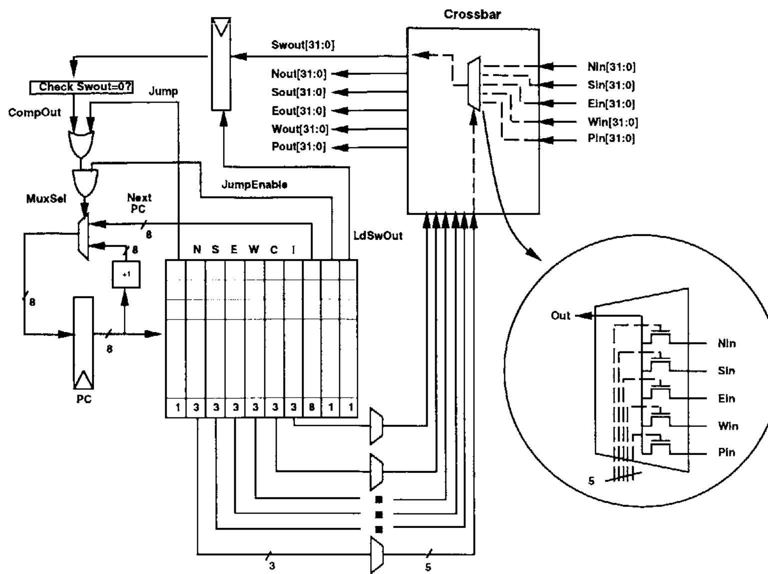


Figure 4-7: Example of a static router

A state table controls multiplexers in a crossbar. This table determines the connectivity of north, south, east, west, and processor ports as a function of an indexing program counter. A branching mechanism supports dynamic control flow. This example is from Liang et al. [93].

can be determined at compile time. Although not considered here, the general approach of architecture specialization can also be applied to dynamic messages. For example, dynamic messages are partially specialized in the *turnstiles* of [14]. For unresolved dynamic messages, either busses or dynamic routers need to be instantiated in the resulting gate-level logic.

An example architecture for a static router is in Figure 4-7. The central component of the switch is a crossbar between north, south, east, west, and a processor port. This crossbar can change connectivity on every clock cycle as a function of a lookup table. The table can be viewed as microcode for a tiny processor in the switch, with its own PC and control logic.

Alternatives for Router Specialization

In Chapter 3, I demonstrated how to specialize traditional architecture features to gate-level logic. This logic replaces the processing core in the tiled architectures. For specializing the router, there are three choices:

1. Specialize static routing instructions into the microcode of the router. This specialization is equivalent to the specialization performed in architectures like Raw [136].

2. Specialize the routing engine into custom logic. The basic specializations are performed on the routing engine, but long distances routes will continue to pass through intermediate tiles and there will be no long wires.

3. Specialize the routers into global crossbars — point-to-point connections between all

communicating tiles. In this case, the routers are specialized all the way down to individual wires. Although these wires can be pipelined, the default allows long wires, as in a traditional FPGA.

All the above options are available in my implementation: DeepC can emit a general routing engine with instructions or DeepC can generate specialized logic for either the router or the crossbar communication structure.

4.2.1 Approach

My approach for specializing tiled architectures consists of three parts: global binding, space-time scheduling, and local binding. Global binding, performed by the global placement algorithms, results in each operation being localized to a region: a single tile. During scheduling, local constraints are observed (for example, a maximum of two multiplies per cycle), but local resource assignments are left unspecified. Local binding is performed by later RTL synthesis algorithms. The following sections describe each in detail.

Global Binding

Section 4.1 described my approach to memory specialization and demonstrated how to decompose program data structures into separate memories. As a part of communication specialization, the location of computation must be determined. Computation should be assigned close to the most appropriate memory — I refer to this process as *global binding.*

My approach is to assign load and store instructions that access a memory to the tile containing that memory. The remaining computation can be assigned to any tile; its actual assignment minimizes two factors: communication between tiles and the latency of the critical path.

My approach for assigning computation to memory tiles leverages the non-uniform resource architecture (NURA) algorithms developed for Raw [89]. Instruction-level parallelism within a basic block is orchestrated across multiple tiles. This work is in turn an extension of the MIT Virtual Wires Project [12] work, in which circuit-level, or *combinational,* parallelism within a clock cycle is orchestrated across multiple tiles.

The NURA approach to global binding consists of three steps: clustering, merging, and placement. *Clustering* groups together instructions that have no parallelism and that, given communication cost, can not be assigned to different tiles profitably. *Merging* combines clusters until there is one cluster, or *virtual tiles* per physical tile. *Placement* maps virtual tiles onto physical tiles while minimizing communication cost and observing interconnect constraints. Section 5.2.3 describes my exact implementation of these steps as part of the *Global Partitioning, Global Memory Placement,* and *Global Placement* compiler passes.

Space-Time Scheduling

DeepC's space-time scheduler is an extension to the scheduling algorithm introduced in Section 3.3.1. Recall that this approach uses a forward list scheduler operating on a precedence graph, and each operation i has predecessors $pred_i$, a cost c_i, and latency l_i. The algorithm for extending this approach, from the approach in Lee et al. [89], is as follows:

1. Perform global binding, as described in the previous section, such that every operation is localized to a tile.

2. Replace inter-tile communication, due to precedences between operations in different locations, with a sequence of route operations.

3. Globally bind route instructions to the appropriate router, dismantling all global precedences.

4. List schedule all tiles simultaneously, observing local resource constraints and precedence constraints.

Because all operations are bound to a specific tile prior to scheduling, the list scheduling algorithm in Section 3.3.1 does not need to be modified. This design decision is not the best — in Virtual Wires scheduling [12], shortest path routing is used as an inner loop of the list scheduler. However, this approach leverages the same algorithm as Raw — both because the code was available for modification and for accurate comparisons. One limitation/feature of the Raw algorithm is that it does not support/require storage in the network — channel resources are allocated along an entire path before the send operation is initiated. Another limitation of the Raw routing algorithm is that all routes are dimension-ordered — the X dimension is completely routed and then the Y dimension. Although dimension ordering can eliminate certain deadlock situations in dynamic routing, static routing does not need ordered dimensions. DeepC inherits these limitations from Raw, although the general cases are not difficult to implement[3]. There is no other reason for using dimension-ordered routing.

Static Routing Example Although evaluating instructions in a distributed system eliminates long memory communication latencies in comparison to monolithic processor design, new inter-tile communication paths are required for correct program execution. The data dependencies between instructions assigned to each tile will require communication, or *route* instructions. Figure 4-8 (a) illustrates the generation of route operations from the precedence graph for a short loop that is fully unrolled. The array A has four elements. These elements are low-order interleaved across four processors, one element per processor. In the now flattened loop, each array element is assigned the value of variable X, however X can only be loaded from the first tile. Hence, route instructions are needed to communicate the value of X to neighboring processors. The values are transmitted through the processor's respective routers. However, processor three does not have a direct communication channel with processor zero. Thus, a through-route, or *hop*, must be created. In this embedding, the route operations have predecessors and successors in only neighboring tiles. Thus, as long as resource constraints are observes, the list scheduler in Section 3.3.1 can be used without other modifications.

Handshaking and Asynchronous Global Branching In my approach, static routing may be cycle-counted or handshaken. When routing is cycle-counted, no synchronization is needed between tiles; a send on cycle 14 is always received on cycle 15 (with a delay of one in this case). In contrast, handshaken routing includes request (from the sender to the receiver) and acknowledge (from the receiver to the sender) signals that can stall the state machine on either tile. Handshaking is needed for dynamic events that are not

[3]Because Raw's static network is fast in comparison to the processor and the compiler code quality, network contention was not a problem and these limitations were not important in the context of Raw research.

```
for(i=0;i<4;i++) A[i]=X;
```

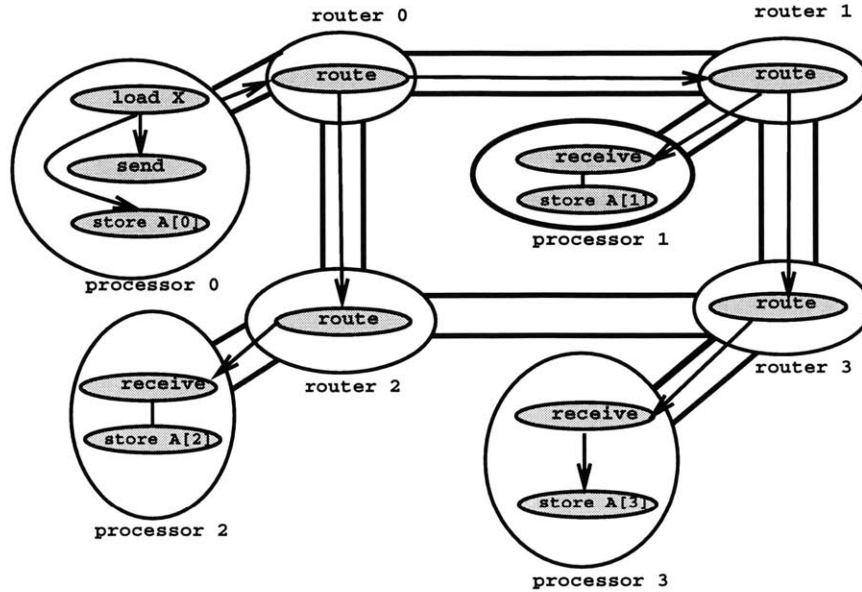


Figure 4-8: Example of router operations

Variable X is stored on Tile 0 and variable A is low-order-interleaved across all four-tiles. The shaded operations are embedded into the topology graph of a four tile architecture with routers.

explicitly accounted for by the compiler (for example a cache miss). Cycle counting is a specialization of handshaking, possible when the compiler can determine the clock cycles on which communication will take place; thus this work focus on the cycle-counted option.

My approach also uses static messages to control flow between basic blocks, explicitly orchestrating *asynchronous global branching*, a concept first described in Lee et al. [89]. The branch condition is computed on one tile and then statically routed to the other tiles, turning control dependencies into data dependencies. With cycle counting, this mechanism becomes a synchronous branch with a prefetching broadcast of the branch condition, allowing a predetermined overlap of computation with branch condition transmittal.

Local Binding

During scheduling, I have relaxed the resource allocation problem for both registers and combinational logic, while retaining the constraints for memory accesses and inter-tile communication channels. Furthermore, my approach supports resource bounds on some computationally expensive functions, such as multiplication, by limiting the maximum concurrency per tile during scheduling. The computation schedule for a given program state will determine the resource needed in each tile. These resources, both registers and functions, may be shared between states by resource allocation. Allocation of these remaining resources is performed during RTL synthesis, in the CAD tool phase of DeepC (Section 5.2.5). My approach to parallelizing sequential code is limited by memory and communication cost, thus constraints for logic and register bits are not necessary. That is, logic and register

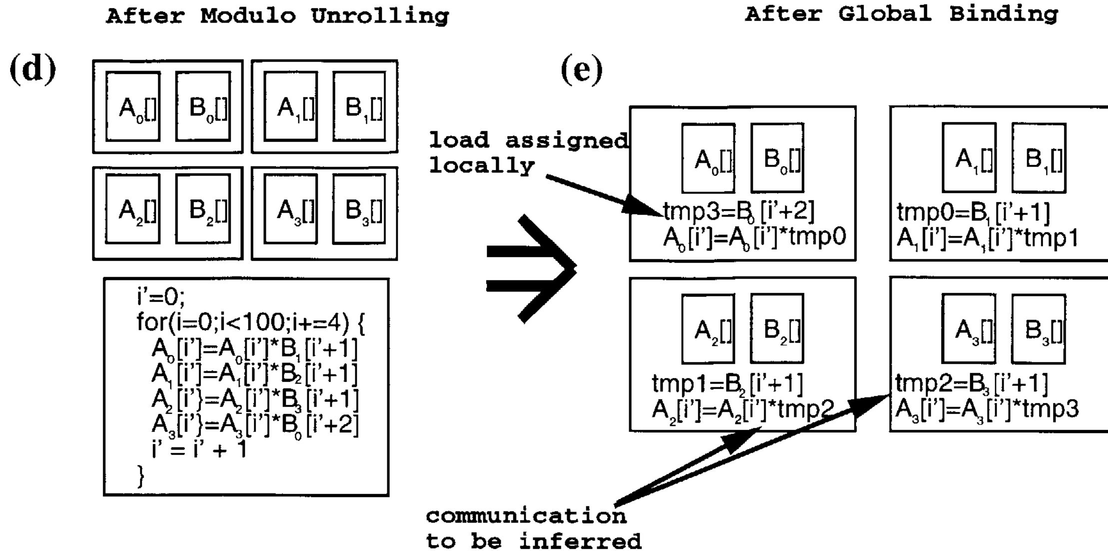


Figure 4-9: Global binding example

This figure continues the running example, showing the transformed code in (d) after modulo unrolling, and in (e) after global binding;

areas are only limited by the amount of parallelism exposed during unrolling.

4.2.2 Examples

Global Binding

Figure 4-9(d) continues the end-to-end example with the results of global binding. As I described in the previous section, each tile contains two memories, one for each array. Computation is assigned to small memories. In the figure, each tile has been assigned a subset of the original instructions. In Figure 4-9(e) the data dependences between tiles are shown as temporary variables introduced in the code. For example, *tmp0* needs to be communicated between the upper-left tile and the upper-right tile.

Scheduling

Figure 4-10(f) shows the code in the running example after communication scheduling. At this point, all operations are assigned to specific clock cycles, labeled to the left of each statement. Data and control dependencies are communicated by send and receive instructions added to each state machine. These instructions multiplex communications across the inter-tile channels. Finally, memory addresses are dereferenced, resulting in loads and stores to local memory ports.

Local Binding

Figure 4-11 completes my example with the last steps of (g) state machine synthesis and (h) RTL generation. These transforms are from the approach described in Section 3.3. Only tile zero is shown. The local binding step generates custom logic, converting machine-level

Figure 4-10: Space-time scheduling example

This figure continues the running example, showing the transformed code in (e) after global binding, and in (f) after space-time scheduling;

instructions to pipeline registers and multiplexers. These registers and multiplexers are in turn controlled by bits from the tile's state machine.

Several features are noticeable in these final transformations. The operations scheduled to each clock cycle form a complex instruction. In (g), state 5 also shows the branch targets (dropped in the earlier figures) for the loop. The branch is implemented with a conditional assignment of the program counter, as described in Section 3.3.

In (h), more features are dismantled. Loads and stores become accesses to the address and data ports of RAMs. Sends and receives, including through-routes, become accesses to the communication channels between tiles. The RTL also shows other features discussed in Chapter 3, including the chaining of two instructions, and values stored on both wires and registers.

4.2.3 Ease of Implementation

Static routing is easy and fun. DeepC's static routing implementation is an extension of a simpler list scheduler coded for the Raw project. Because routing instructions are bound globally before scheduling, the scheduling algorithm is not further complicated. During implementation, interactions between scheduling and register allocation should be studied. However, this phase ordering problem arises in traditional compilation and is not new here. In this approach to communication specialization, scheduling is limited to basic block, simplifying the overall problem. However, to hide message latencies, long-term solutions should include pipelining across both the temporal control flow and the spatial communication network. Latency hiding also makes it desirable to extend these specialization techniques to multi-threaded architectural mechanisms.

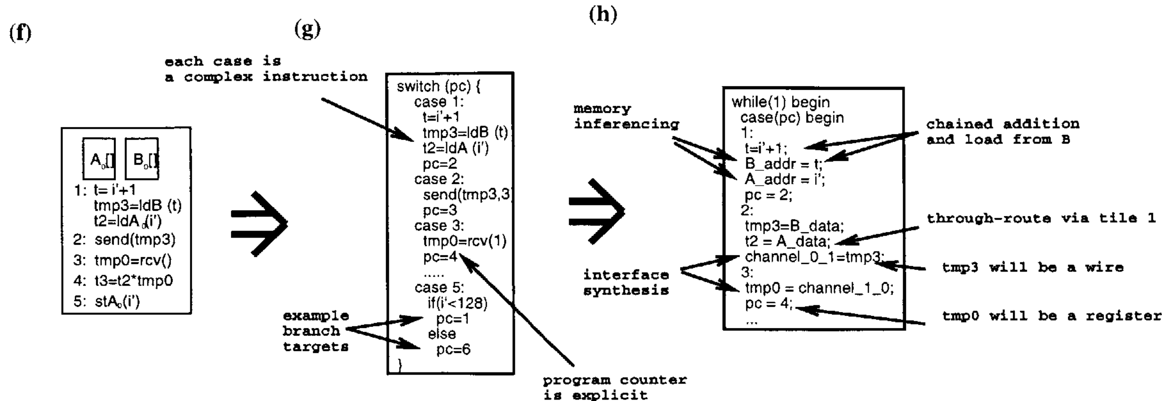


Figure 4-11: Local binding example

This figure continues the running example, showing the transformed code in (f) after space-time scheduling, in (g) after state machine synthesis, and in (h) after RTL generation.

I have not treated cases where dynamic routing is needed. One approach is to implement the pseudo-static methods of Raw (Barua et al. [15]). Another approach, common in VLIW machines, is to have a slow "back door" to all of memory (Ellis [51]). This earlier works demonstrates that dynamic events are not a counterexample to the claims in this thesis, although dynamism can limit specialization opportunities.

4.3 Summary

This chapter has explained my approach to specializing advanced architectural mechanisms, including memory and communication mechanisms. Specialization of the mechanisms commonly found in distributed architectures is necessary to achieve the highest performance while meeting physical constraints. The introduced ideas underlying memory specialization are based on transforms for unrolling, partitioning, decomposing, disambiguating, and otherwise separating memory elements into spatially locatable parts. The introduced techniques for communication specialization use tiled architectures, replacing traditionally global features with a tessellation of computation, memory, and routing structures. Both of these approaches borrowed heavily from compiler and architecture work in the Raw Project. The following chapter continues with a description of my implementation of the specialization approaches discussed thus far.

Chapter 5

DeepC Compiler Implementation

At first blush compiling high level languages for VLIWs might appear to be an impossible task, given that they are programmed at such a fine-grained level. But in fact the Bulldog compiler isn't that much different from a traditional optimizing compiler.

— John Ellis on the implemention of Bulldog, from his Yale PhD, Bulldog: A Compiler for VLIW Architectures (1988)

Gate reconfigurable architectures are significantly finer grained than traditional processor architectures, even than VLIWs, so compilation to gates requires substantial and more diverse technologies. While the frontend compiler passes are similar to traditional optimizing compilers, the middle passes are dominated by parallelization techniques, and the backend passes are akin to hardware synthesis tools. Furthermore, construction of a complete system is needed to demonstrate the techniques for architectural specialization introduced in the previous two chapters; an implementation strongly supports my thesis (that specialization of traditional architectural mechanisms with respect to an input program is the key to efficient compilation of high-level programs to gate-reconfigurable architectures). Therefore, this chapter describes my implementation: the *DeepC Silicon Compiler*. DeepC is a prototype research compiler that is capable of translating sequential applications, written in either C or FORTRAN, into a hardware netlist. This chapter also includes DeepC's environment: the target platforms, simulators, and verification approaches needed for a complete system.

5.1 Compiler Lineage and Use

DeepC comes from a rich line of research compilers. Figure 5-1 shows the genealogy of DeepC. All members of the DeepC development effort were also members of the Raw Project. As a result, DeepC reuses many passes of Rawcc, the main compiler for the Raw Microprocessor. Likewise, because the Raw processor consists of an array of MIPS-like processors, Rawcc extends Harvard's MachSUIF compiler [125]. Both Rawcc and DeepC also use the SPAN pointer analysis package, also developed at MIT. DeepC furthermore uses the Bitwise Compiler, developed at MIT, and leverages the VeriSUIF Compiler [56], constructed at Stanford. All these compilers are built upon the SUIF infrastructure.

Version 1.0 of DeepC was released to others in academia on September 10, 1999. Researchers at MIT, Princeton, and the University of Massachusetts have used DeepC for reconfigurable computing and system-on-a-chip (SOC) research. This was the first release

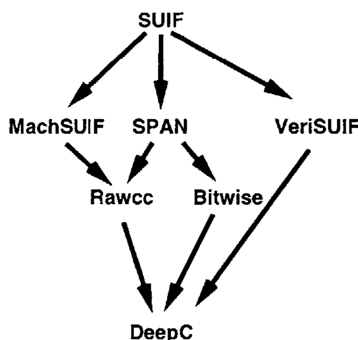


Figure 5-1: DeepC compiler lineage

of a parallelizing compiler from high-level languages to gate-level reconfigurable hardware. Previous compilers were only able to transform small C expressions or were only able to map a sequential program into a sequential state machine. The results in this dissertation correspond to DeepC 2.0.

5.2 Overview of Compiler Flow

The compiler generates a specialized parallel architecture for every application. To make this translation feasible, the system incorporates the latest code optimization and parallelization techniques as well as modern logic synthesis tools.

Figure 5-2 divides the DeepC compiler flow into two paths, basic and advanced. The basic path specializes the mechanisms described in Chapter 3. The basic path includes three major phases: a *Traditional Frontend Phase*, a *Machine Specialization Phase*, and a *CAD Tool Phase*. The frontend performs traditional compiler optimizations and analysis. Pointer analysis and bitwidth analysis, used for both basic and advanced specialization, are shown as separate packages following the Frontend Phase. The Machine Specialization Phase includes finite state machine generation, wire identification, register allocation, and other passes that produce RTL Verilog. The CAD Tool Phase includes commercial CAD tools that translate from RTL Verilog to the final logic-level hardware description.

The phases in the advanced path are required to specialize the mechanisms described in Chapter 4. The advanced path includes two phases for automatic parallelization: the *Parallelization Phase* and the *Space-Time Scheduling Phase*. The advanced path is invoked with the compiler parameter *-nprocs*. The following sections describe each of the five phases.

5.2.1 Traditional Frontend Phase

The first phase of DeepC is the Traditional Frontend Phase. Lexical and syntax analysis, followed by parsing, produce an intermediate form with symbol tables and abstract syntax trees. FORTRAN programs are transformed to C while retaining relevant array information. Optimizations include constant propagation, loop invariant code motion, dead-code elimination, and strength reduction. Following the Traditional Frontend Phase are two new analyses: Pointer Analysis and Bitwidth Analysis, described next.

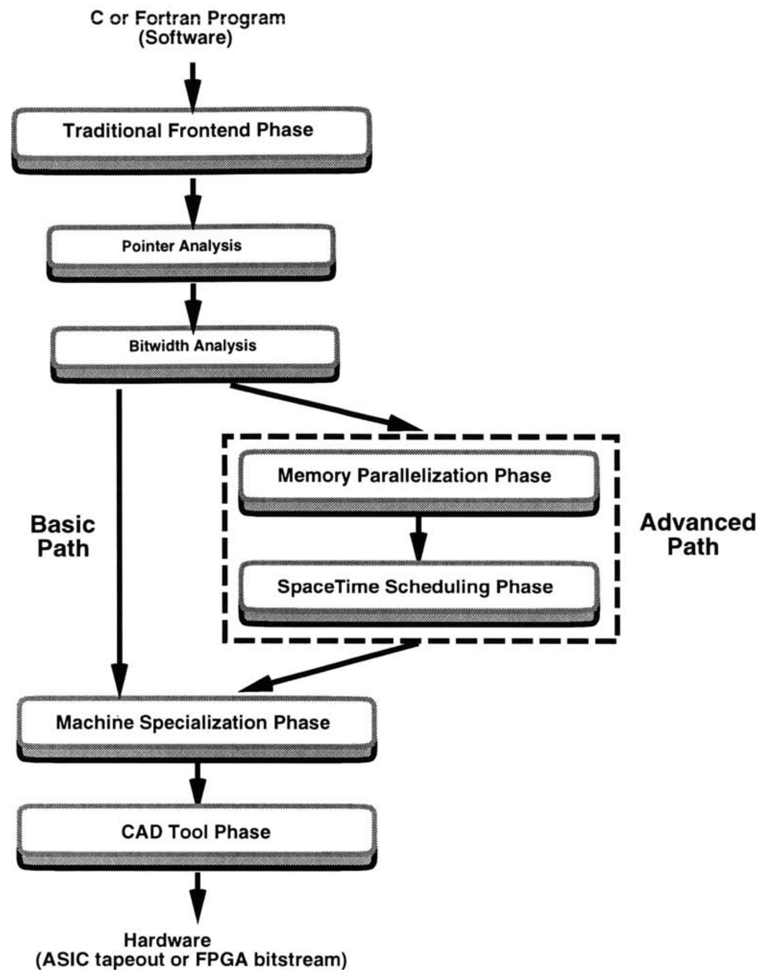


Figure 5-2: Overview of DeepC compiler flow

Pointer Analysis DeepC performs pointer analysis as early as possible. Pointer analysis is used by MAPS, Bitwise, and in DeepC memory formation. DeepC uses SPAN, a pointer analysis package developed at MIT by Rinard and Rugina [114]. SPAN can determine the sets of variables — commonly referred to as *location sets* — a pointer *may* or *must* reference. The analysis package tags all memory references with location set information.

Bitwidth Analysis I helped Mark Stephenson develop the Bitwise Compiler [127] as a part of DeepC. This set of passes performs Bitwidth Analysis. The goal of Bitwidth Analysis is to analyze each static instruction in a program to determine the narrowest return type that retains program correctness. This information can in turn be used to find the minimum number of bits needed to represent each program operand. The Bitwidth Analysis approach is described in detail in Section 3.2.1.

Because instructions are generated during parallelization, DeepC 2.0 performs Bitwidth Analysis after Modulo Unrolling. However, the Bitwidth Analysis pass can be executed in several locations along the advanced specialization path. In general, Bitwidth Analysis should be executed multiple times if intermediate passes do not propagate bitwidth

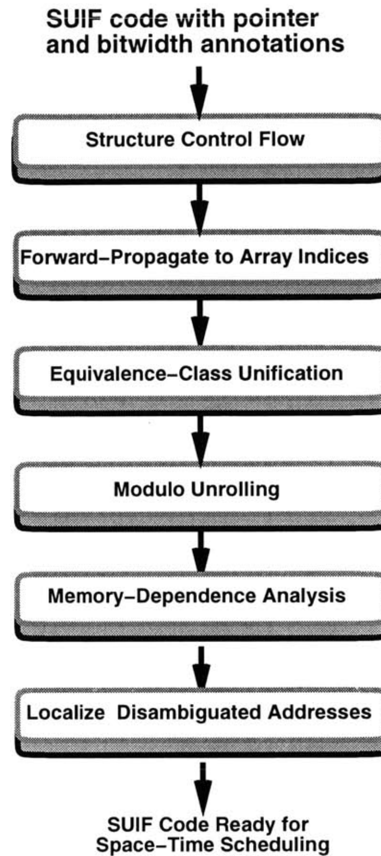


Figure 5-3: Steps of Parallelization Phase

information or naively create unoptimized temporaries.

5.2.2 Parallelization Phase

The Parallelization Phase (Figure 5-3) leverages Rawcc technology developed for MAPS [14]. This technology, in combination with the following Space-Time Scheduling Phase, creates parallel threads from a sequential program. These parallel threads communicate statically when possible. Rawcc's support for dynamic communication is not included in DeepC 2.0. The following paragraphs continue with the important steps of the Parallelization Phase.

Control Flow Structuring This task is performed with several standard SUIF passes. Structured control flow is built from unstructured control flow using the program *structure*. If possible, **while** loops are formed. If an induction variable can be recognized, while loops are transformed into **for** loops with the SUIF pass *porky*. This transformation is important because the later Modulo Unrolling pass is restricted to *for* loops in DeepC 2.0.

Forward-Propagate to Array Indices In preparation for later MAPS passes, DeepC invokes a standard compiler pass called forward-propagation. This task moves forward

certain local variable calculations. In particular, Modulo Unrolling benefits from cases where index expressions of array references can be replaced by affine functions.

Equivalence-Class Unification Equivalence Class Unification (ECU) is one of the core memory disambiguations performed by MAPS. Barua [15] summarizes ECU as follows:

> First, the results of pointer analysis are used to construct a bi-partite graph. Next, connected components of the graph are found; the connected components form an equivalence class. Finally, each equivalence class is mapped to a unique virtual bank.

DeepC uses ECU to disambiguate arrays into equivalence classes. However, rather than assigning these classes to virtual banks, DeepC groups the arrays in a class into a single large memory. For FPGAs, when a memory is larger than one internal block RAM, several block RAMs are composed with multiplexers.

Modulo Unrolling This step groups a series of MAPS passes that modulo unroll loops. The steps, described in detail in Barua's Phd thesis [14], are:

1. Detect affine accesses
2. Reshape arrays
3. Compute unroll factor
4. Unroll loops producing mod-region assertions
5. Strip-mine distributed arrays
6. Strength-reduce integer remainder (mod) and divide (div) operators
7. Infer mod values using mod-region assertions

After Modulo Unrolling, multiple small memories have been created from initial arrays. Modulo Unrolling allows these memories to be stored in banks that can then be accessed in parallel, without conflicts. In the common case, the analyzable arrays in each unrolled loop will be transformed into n arrays, where n is the number of tiles.

For DeepC, an undesirable side effect of Modulo Unrolling is that loops are sometimes unrolled prodigiously. Although some unrolling is good, the depth of unrolling needed to expose static array accesses can be large.

Memory-Dependence Analysis DeepC uses the general Memory-Dependence Analysis pass in MAPS. This pass introduces dependence edges between all pairs of memory references that can access the same location. Dependence edges are computed using pointer analysis and further refined using array-index analysis. Although MAPS enforces dependence in different ways, in DeepC only static-static memory dependencies are supported. Static-static dependencies are enforced by explicit serialization on the disambiguating tile.

Localize Disambiguated Accesses This pass replaces global accesses disambiguated by Modulo Unrolling with local addresses. The local addresses are derived from the corresponding array address computation by dropping the last dimension. Because the address is disambiguated, the last dimension contains the constant bank number, which is then annotated on the memory reference. This annotation is used during data placement in the Space-Time Scheduling Phase.

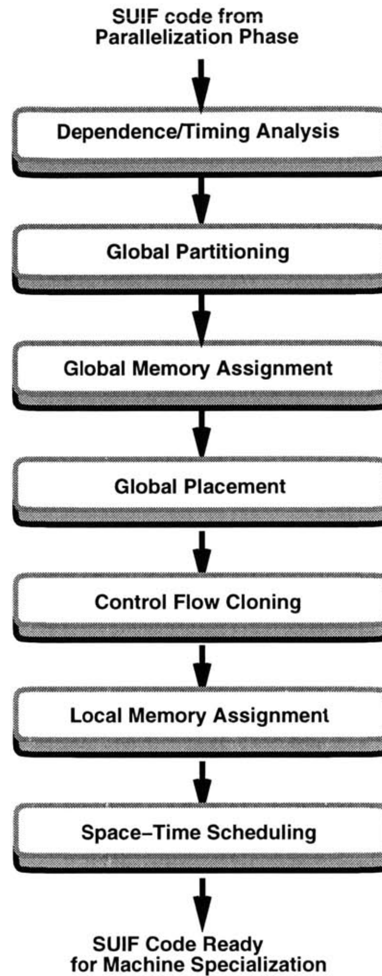


Figure 5-4: Steps of Space-Time Scheduling Phase

5.2.3 Space-Time Scheduling Phase

The Space-Time Scheduling Phase is responsible for partitioning and scheduling instructions into communicating physical tiles. At the end of the Space-Time Scheduling Phase every instruction has been assigned to a tile and time-slot. Communication between tiles is likewise constrained. Figure 5-4 shows the steps of the Space-Time Scheduling Phase. Before the first step, variables are renamed to SSA form. The following paragraphs discuss each step in turn.

Dependence/Timing Analysis In Rawcc, Dependence Analysis determines the data dependence graph, or precedence graph, for the instructions in each basic block. In the same pass, Rawcc computes the time to execute each instruction. DeepC extends this analysis to include fractional cycle delays. In some cases, the input to an instruction is a constant, and the delay can be reduced. Likewise, address calculation that is expected to be optimized away can be assigned a negligible cost. DeepC 2.0 uses the timing data in Table 5.1. This estimated data was tuned during prototyping of the compiler. These

Suif Instruction	General (cycles)	Has constant (cycles)
add	0.7	0.4
and	0.1	0.05
asr	0.2	0.0
cpy	0.0	0.0
cvt	0.0	0.0
ior	0.1	0.05
ldc	0.0	0.0
lod	1.0	1.0
lsl	0.2	0.0
lsr	0.2	0.0
mul	0.9	0.9
neg	0.7	0.4
not	0.05	na
rot	0.2	0.0
seq	0.2	0.2
sl	0.7	0.4
sle	0.7	0.4
sne	0.4	0.2
str	0.0	na
sub	0.7	0.4
xor	0.1	0.05
predicate	0.9	na

Table 5.1: Heuristically determined timing estimates

The first column contains the SUIF opcode for the instruction. The second column shows the delays used in the general case while the last column shows the delay estimated when one input is a constant. The estimated number of clock cycles allocated for each SUIF instruction is used during scheduling to determine the assignment of instructions to control states.

estimates are heuristics, they are not exact — high estimates will results in a slower clock period; low estimates will result in extra clock cycles.

Global Partitioning This pass groups all non-memory instructions into virtual tiles. Two opposing goals complicate this pass: minimizing communication and maximizing parallelism. DeepC included an unmodified version of the Rawcc partitioning pass (from Rawcc's space-time scheduler), which employs a multiprocessor task-scheduling algorithm. In Rawcc, partitioning is performed by two phases: clustering and merging.

Clustering assumes non-zero communication cost but infinite resources. Instructions are grouped if they have no parallelism that is exploitable given communication overheads. Grouping is performed by Dominant Sequent Clustering [145], a greedy technique minimizing the estimated completion time. The algorithm visits instructions in topological order. At each step, it selects, from a list of candidates, the instruction on the longest path. It then checks whether merging that instruction into the cluster of a parent instruction will improve performance. The algorithm completes when all nodes have been visited exactly once.

Merging combines clusters until the number of clusters equals the target number of tiles, assuming a crossbar-like interconnect. Rawcc's algorithm uses load balancing and communication minimizing heuristics as follows. The compiler initializes an empty partition for each tile and then visits clusters in decreasing order of size. A cluster is merged into

the partition with the most communication between the clusters in that partition and the visited cluster. However, if merging would increase the size of a partition to more than twenty percent of the average partition size, the *smallest* partition is chosen.

Global Memory Assignment This pass assigns the home location of all scalars and arrays. Arrays that have been Modulo Unrolled are distributed across tiles. As much as possible, scalars are assigned to tiles near where they are most frequently accessed. This step is unmodified from the Rawcc space-time scheduler.

Global Placement At this point in the compiler flow, instructions and data have been formed into tiles. However, these tiles are virtual and not yet assigned, or "placed" on physical tiles. This phase uses simulated annealing to minimize communication and thus reduces routing requirements. This step is unmodified from the Rawcc space-time scheduler, with the exception that the input timing analysis is calculated differently, as described in the earlier paragraph on dependence/timing analysis.

Control Flow Cloning With the exception of predicated instructions, every tile executes the same control flow. While ordinary instructions are assigned to particular tiles, control flow instructions must by copied to each tile. This step is part of the Rawcc space-time scheduler, with a few unimportant modifications for DeepC.

Because of control flow cloning, each branch instruction becomes a global branch instruction. For most tiles, the branch condition is not computed locally and adds to the global communication for the basic block. By default, DeepC 2.0 uses a synchronous global branch in which every tile branches on the same cycle. If handshaking synthesis is turned on (*-fhand*), DeepC generates asynchronous global branches identical to those of Rawcc.

Note that some control flow is not cloned. DeepC includes a modification of Rawcc's control localization (macro formation) technique for IF-conversion. During scheduling, these short code sequences are treated as a single instruction.

Local Memory Assignment Local memory assignment is a new pass. Memory assignment determines the final mapping of program arrays into on-chip memories. In the completely disambiguated case, for example with no pointers, no two arrays will share the same memory. However, if access to one or more arrays is ambiguous, then these arrays will be grouped into one memory. Memory assignment in DeepC 2.0 uses the following algorithm. The first array is given an offset A_0 of zero. The nth array is given an offset of $A_n = A_{n-1} + size_{n-1}$. The width of the memory is the maximum data width of all arrays assigned to the same physical memory.

Space-Time Instruction Scheduler and Static Router For each basic block in the program, DeepC applies a list scheduler to schedule both instructions and static routes. This scheduler is forward, cycle-driven, and critical-path-based (see Section 3.3.1). In contrast to the Rawcc scheduler, DeepC's scheduler supports concurrent instructions in the same tile. Instructions scheduled in the same cycle may be executed in parallel (VLIW-style) or sequentially (chained-style). To support chaining, instructions are scheduled to the thousandth of a cycle. This accuracy is only an estimate — the exact schedule, even the clock period, is not determined until synthesis. When chaining, DeepC does not permit operations to be scheduled across clock boundaries.

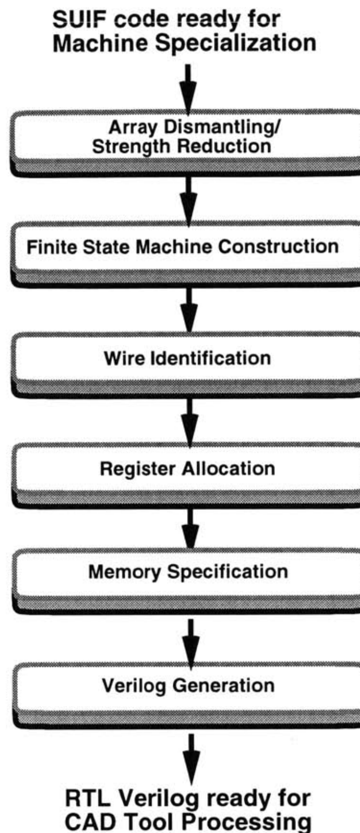


Figure 5-5: Steps of Machine Specialization Phase

In list scheduling, a priority queue maintains a list of available items to be scheduled. Because routing paths are determined before scheduling, the resulting communication instructions can be included in the list, permitting usage of a single list. The priority scheme uses the critical path length from an instruction to the end of the basic block. The Rawcc metric of *average fertility*, defined as the number of descendants of a node, is not used.

This scheduler fits into an overall approach of global binding, space-time scheduling, and local binding. Global binding, performed by the global placement algorithms described in the previous steps, results in each operation being localized to a particular region (a tile). During scheduling, local constraints are observed (for example, a maximum of two multiplies per cycle), but local resource assignments are left unspecified. Local binding is performed during later RTL synthesis by downstream CAD tools.

5.2.4 Machine Specialization Phase

The Machine Specialization Phase (Figure 5-5) is responsible for generating a hardware implementation from communication and computation schedules. First, the system generates a finite state machine that controls the cycle-by-cycle operation of all the memories, registers, and functional units that make up a tile. Then, the finite state machine is synthesized into a technology-independent RTL specification of the final circuit.

Programs have several properties when they enter the Machine Specialization Phase. On

the advanced path, previous phases have mapped data to memories, extracted concurrency, and generated parallel threads of instructions for each tile. The basic path is a degenerate of this case — the memories are not partitioned and all concurrency is restricted to a single tile. An exception is that Equivalence Class Unification (ECU) is permitted on the basic path. A tile also has scheduled evaluation at this point: each memory access, communication operation, and computation instruction has been mapped to a specific cycle. Finally, the space-time scheduler has mapped the computation and communication to a specific location in the memory array. Thus, at the beginning of the Machine Specialization Phase, the program, in SUIF format, has the following properties:

1. Spatiality: Every operation executes on exactly one tile.
2. Temporality: Operations are scheduled to time steps in each basic block.
3. Causality: Operation order preserves dependencies.
4. Monosemy: All loads and stores unambiguously reference a single memory port.

Given the previous program properties, the Machine Specialization Phase is responsible for synthesizing the following structures to Verilog:

1. branch/jump instructions
2. load/store instructions with base plus offset addressing mode
3. pointer dereferences
4. floating point operations
5. signed integers
6. Special *Send* and *Recv* instructions that statically denote all inter-tile communication

In order to compile to a gate-level executable, these features must be constructed in software. The following paragraphs explain the steps for doing so.

Array Dismantling and Address Strength Reduction DeepC dismantle arrays with Porky, a SUIF pass developed at Stanford that performs assorted code transformations. DeepC modifies *Porky's* address calculation logic to take account of gate-level targets. Specifically, when dismantling arrays, this new Porky performs the strength reduction introduced in Section 3.1.2.

Finite State Machine Construction Finite state machine (FSM) construction generates code for the approach discussed in Section 3.3. This step takes the threads of scheduled instructions and produces a state machine for each thread. This state machine contains a state register, called *babb_pc*, which serves a function similar to the program counter in a processor. Each state of the FSM contains the work to be done in a single cycle of the resulting hardware. That is, each state of the FSM contains a set of directed-acyclic graphs (DAGs) of dependent operations. The FSM generator turns these DAGs in each state into combinational logic. Any dependence between operations in the same state will result in wires between the dependence-related functions. For any data values that are *live* between states (produced in one state and consumed in another), registers will be inferred during later RTL synthesis.

Wire Identification This step identifies scalar variables that are generated and consumed in the same clock cycle. These variables should not be allocated to registers because later RTL synthesis will be able to map them to wires. Wires are identified by walking the instruction tree and finding variables that have a lifetime shorter than one clock cycle.

Register Allocation Register sharing is performed with a register allocator. DeepC uses the MachSUIF [125] register allocator, *raga*, and a target machine with hundreds of registers. This is not the ideal register allocator for DeepC, but it was easy to construct. Because the allocator is not aware of bitwidth-reduced registers, some sharing may waste bits. After register allocation, the bitwidth of each physical register is computed as the maximum bitwidth of the virtual registers assigned to it. DeepC does not include a code generator for register spills, although the hooks for spilling are in place from MachSUIF.

Memory Specification For Xilinx targets, DeepC declares special modules to instantiate block RAMs. A single memory may need more than one block RAM. For emulation, this pass generates the *vmw.mem* file used to describe design memories.

Verilog generation This pass takes SUIF as input and generates RTL Verilog. This pass was created by modifying the VeriSUIF code generator. This approach maps SUIF instructions to Verilog instructions whenever possible, allowing compiler transforms to be made in SUIF, not VeriSUIF. A few optimizations, such as final address computation and some bitwidth related calculations are performed in this phase for practical reasons.

5.2.5 CAD Tool Phase

This phase uses standard CAD tools to translate RTL Verilog into FPGA bitstreams. Given RTL Verilog, many compatible tools can be applied. Figure 5-6 shows the tool flow supported by DeepC. The center flow corresponds to the tool flow for FPGA compilation. An RTL Verilog design is synthesized by Synopsys and then processed by FPGA vendor tools to produce the final bitstream. The figure also shows two alternate flows. When targeting an ASIC process, a different technology library is needed for synthesis. The output of Synopsys, when using an ASIC technology library, is a netlist ready to be processed by ASIC vendor tools. The other alternative is multi-FPGA logic emulation, in which the Virtual-Wires synthesis algorithm [12] partitions and schedules the design across multiple FPGAs. The output is a collection of FPGA files that can be processed by the FPGA tools. The next paragraphs review the CAD tool steps for a single FPGA.

Behavioral/RTL Synthesis The general theory of behavioral synthesis includes many of the algorithmic optimizations performed by higher-level steps of the DeepC compiler. DeepC takes advantage of the fact that industrial behavioral synthesizers, namely the Synopsys Behavioral Compiler (BC), already perform a subset of these optimizations. In particular, DeepC's compiler flow invokes the resource sharing and carry-save arithmetic (CSA) features of Synopsys. DeepC also uses Synopsys Design Compiler's RTL synthesis approach to infer registers and control logic from a cycle-by-cycle description of state machines.

While DeepC leverages some behavioral/RTL features, it does not use all the behavioral features currently available. Operation scheduling and memory inferencing are performed by new DeepC passes rather than by invoking similar algorithms in the CAD tools. For

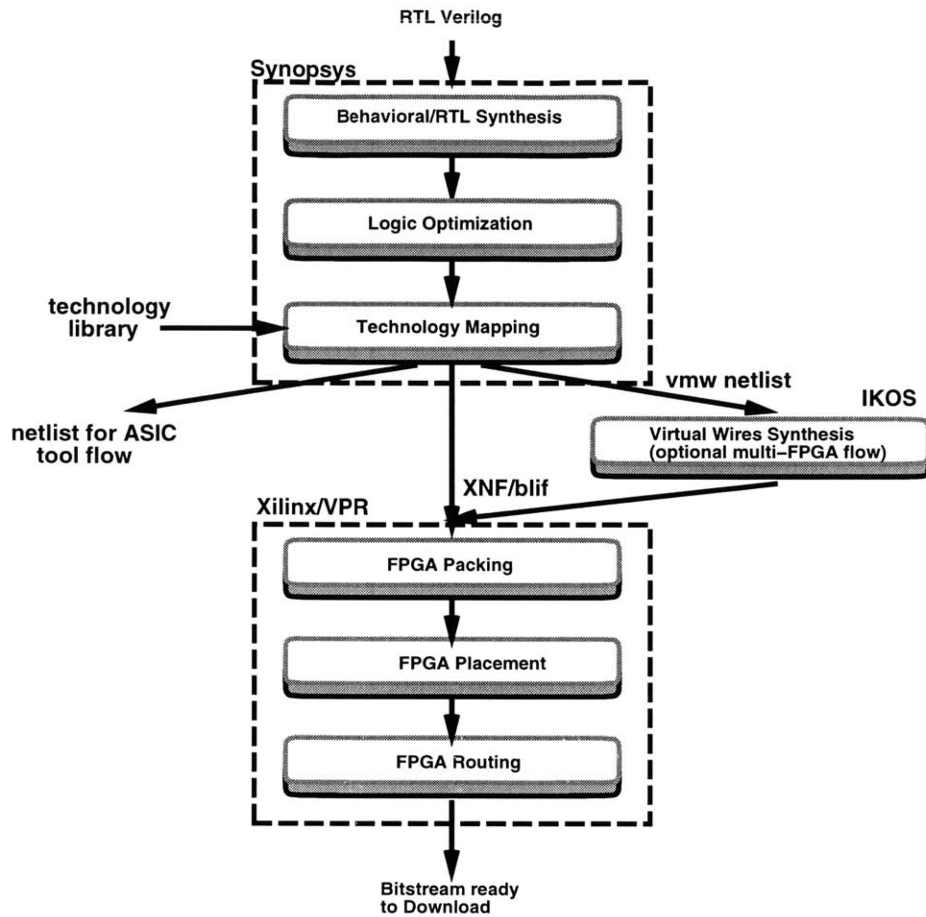


Figure 5-6: Steps of CAD Tool Phase

operation scheduling, the need to synthesize distributed architectures required a new spatially aware scheduler. Also, memory inferencing was not suitable mature in BC at the time DeepC was developed. A final problem to note is that the user interface of behavioral and RTL compilers is targeted at manual design, where a developer has the opportunity to fix small problems and work around quirky behaviors in the tool. This problem would be overcome if tool vendors exported a useful API (application programmer interface) for developers of higher-level compilers.

Logic Optimization Synthesis and optimization of circuits at the logic level determines the microscopic structure of a circuit. Steps include flattening multi-level logic to two-level logic, performing Boolean factorization and reduction, and reducing area by removing redundant logic. Additional timing optimizations reduce the critical path while minimizing the resulting area increase. See [102] for more on this subject.

Technology Mapping A technology mapper takes optimized Boolean logic and maps it to an input target library. All logic functions must be implemented by specific cells in the target library. For FPGAs, the mapping problem is simplified to a covering problem — the

input logic is covered with functions of four inputs (or five inputs). An example covering algorithm for FPGAs is Flowmap [40]. In practice, fast carry chains and vendor macros for multiplication and RAM complicate this step.

FPGA Packing The FPGA packer groups basic FPGA elements, such as LUTs and registers, into a cluster of logic blocks. Unless the technology mapper is aware of all FPGA constraints, this step is needed to match input logic to the internal structure of the FPGA.

FPGA Placement Placement determines the specific location of logic gates or clusters in the pre-fabricated array of LUTs on the FPGA. The goal of placement is to improve routability by placing communicating gates close to one another. After an initial placement, pairs of blocks are swapped in a search to reduce an overall cost function, such as total wire length. A common algorithm to enhance placement is iterative simulated annealing, in which some higher-cost permutations are accepted to avoid local minima.

FPGA Routing An FPGA Router assigns the connections between logic blocks to exact routing segments and switches in the FPGA. In contrast to ASIC routing, FPGA routing is constrained by a fixed routing grid. Maze-routing algorithms [86] are known to work well for FPGAs, although path-based algorithms [132] are faster. A timing driven router can be used to assign critical nets to faster routing resources.

5.3 Target Hardware Technologies

The previous section presented a complete path for compiling a high-level program written in the C language to a logic-level program. This section overviews the physical platforms DeepC targets.

In order to target a platform, the compiler needs a technology library for that platform. The library is specified with the compiler flag *-technology* and is also an input to technology mapping in the CAD phase of the compiler (see Figure 5-6). The DeepC 2.0 library options are: ibm (IBM SA-27E ASIC), virtex (Xilinx Virtex), vpr (Toronto academic FPGA target), and vmw (IKOS VirtuaLogic Emulator). The following paragraphs overview of each of these targets.

ASIC Target: IBM SA-27E A target library from IBM is used to synthesis to the SA-27E process. Unlike the following Xilinx case, the resulting RTL cannot be compiled into hardware with only a touch of a button; work is needed to place pads, generate test vectors, and perform detailed timing, power, and noise analysis. Although DeepC generates area, timing, and power estimates for this library, these results do not reflect the complete design of a chip. The results reported in this dissertation reflect the post-synthesis estimates determined by the Synopsys Design Compiler given the IBM technology library.

FPGA Target: Xilinx Virtex DeepC supports several Xilinx libraries, including the libraries for Virtex [144]. The resulting gate-level netlist *can* be compiled into hardware with the touch of a button. DeepC supports the block RAMs in this technology.

Simulation Approach	Detail and Accuracy	Speed	Run time
Workstation Execution (300MHZ)	Functional, no timing	Very Fast	1
Parallel Simulation (RawSim)	Functional	Slow	5K
Profiling (HaltSUIF)	Cycle-accurate	Fast	5
RTL Simulation - pre-synthesis	Cycle-accurate	Medium	1K
RTL Sim. - post-synthesis	Timing accurate	Very Slow	50K
RTL Sim. - power estimation	Cycle-accurate	Medium	1K
Logic Emulation (5MHZ)	Cycle-accurate	Very Fast	1.5

Table 5.2: Simulators

Academic FPGA Target: VPR DeepC also supports Toronto's Versatile Place and Route (VPR) tool [20]. V-TPACK and VPR are packing, placement, and routing tools for FPGAs. These tools are downloadable for academic use. After synthesis (using the Xilinx 5200 library for Synopsys), the output netlist is converted to BLIF format for use with VPR. The FPGA model is described in the architecture file virtex.arch (see Appendix E).

Commercial Emulation Target: IKOS VirtuaLogic DeepC supports IKOS's Virtual Logic Emulator, as pictured in Figure 5-7, with the top board partly removed. Figure 5-8 shows a newer generation emulator. The programmable hardware in a logic emulator is at the logic or gate-level. Emulators have a massive number of gates from the combination of many Field Programmable Gate Arrays into a single system. In the VirtuaLogic system, an automated resynthesis process is used to allow the array of FPGAs to be treated as a single, giant FPGA. Section B.2 describes host interfaces to emulation systems, when used for computing.

5.4 Simulation and Verification Environment

Although peripheral to the main compiler implementation, it would be a mistake to discuss compilation to gates without adequately addressing simulation and verification. DeepC supports several alternatives, summarized in Table 5.2. First, because DeepC compiles from a high-level language, the design can be compiled and executed on a workstation. Second, parallelization can be verified on a multiprocessor simulator. Third, DeepC includes profile-based simulation using a modified version of Harvard's HaltSUIF profiler (part of MachSUIF [125]). Unlike the general multiprocessor simulation, this technique is cycle accurate when communication patterns are static.

Tight integration with CAD tools enables the next set of alternatives. Post-synthesis RTL simulation provides cycle accuracy that can handle data-dependent control flow. However, CAD tools can only determine cycle time and gate area *after* RTL synthesis. Note that synthesis may be time consuming, but is not a function of program runtime. Thus gate area and cycle time can be obtained for a constant cost. DeepC also includes a post-RTL synthesis methodology that quickly estimates dynamic power consumption. Finally, logic emulation can be used for high-speed gate-level verification.

Table 5.2 also shows the runtimes for each of these approaches simulating a four tile, 32x32 Matrix Multiply benchmark. All cases are normalized to the fastest simulator, the workstation execution, which simulates all four tiles at an effective 40M cycles/second. As

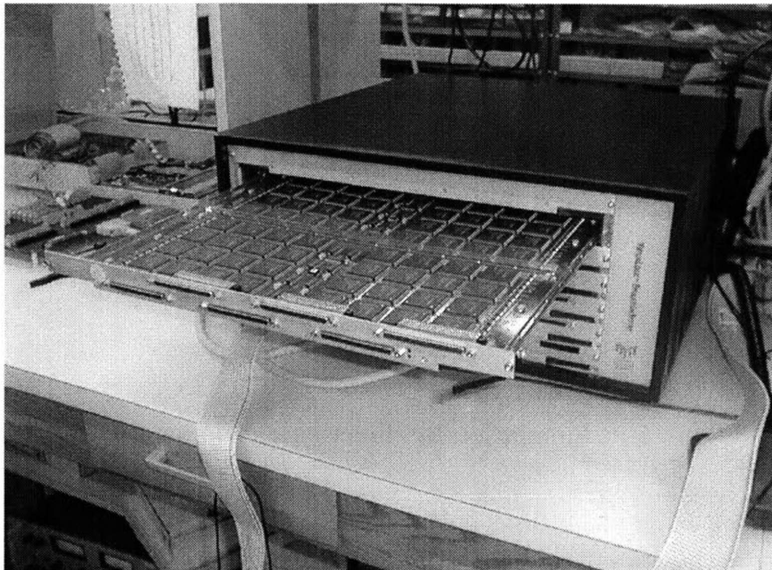


Figure 5-7: VirtuaLogic Emulator

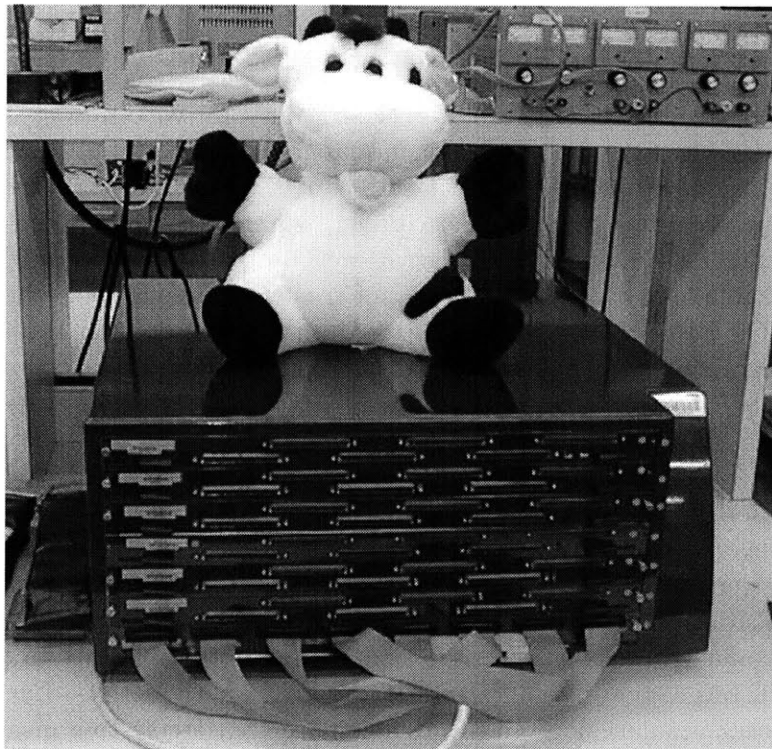


Figure 5-8: A newer generation VirtuaLogic Emulator

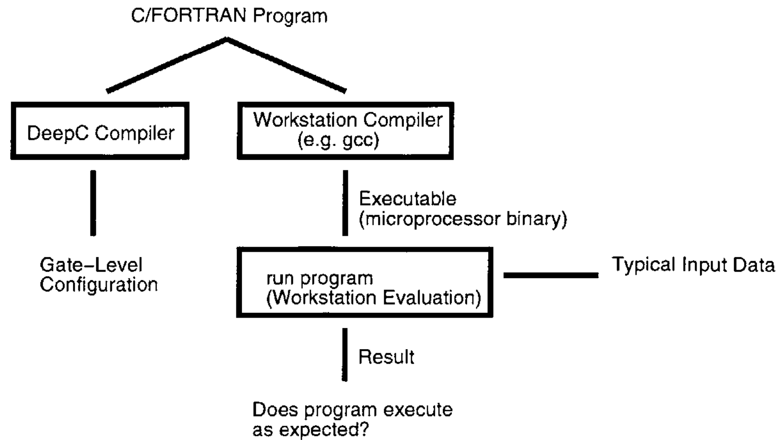


Figure 5-9: Simulation by direct execution on a workstation

accuracy is increased, the runtime is lengthened. An exception is logic emulation, which is cycle accurate and very fast due to the use of multiple FPGAs running at MHz speeds. In fact, as the number of tiles is increased, logic emulation exceeds the speed of a single workstation because of its internal parallelism. For example, emulation is 3× faster for sixteen tiles.

5.4.1 Direct Execution on Workstation

The easiest way to test a C program is to run it on your workstation! Figure 5-9 depicts this scenario. In the figure, the exact same program can be input to DeepC or gcc (the GNU C compiler, developed by Richard Stallman and other free software hackers).

If the input program computes the correct answer on a workstation, then the output hardware should also compute the correct answer. But what if the compiler has a bug? For non-reconfigurable hardware, designers do not trust the compilation/synthesis tool suite to be bug free. The cost of a re-spin could exceed a million of dollars. But for reconfigurable hardware, a re-spin is comparatively trivial. Just recompile and reload the bitstream. Thus, when targeting FPGAs, many designers may be content with software-only verification without further simulation. Because compilation for FPGA is significantly longer than compilation for processors, direct execution on a workstation provides a fast way to verify program functionality.

A major drawback to software only verification is that while the user can verify functionality, only very simple timing analysis is possible. In real-time and embedded systems, accurate verification of timing conditions is important. Even if timing conditions are relaxed, normal development is usually iterative. A developer not only wants to know whether the code is correct, but also wants to know certain performance metrics. Each of the following sections increases simulation detail to provide more accurate system measurements.

5.4.2 Parallel Simulation with RawSim

One of the first options beyond testing program correctness is to test how amenable the program is to parallelization. This approach is useful when the programmer intends to

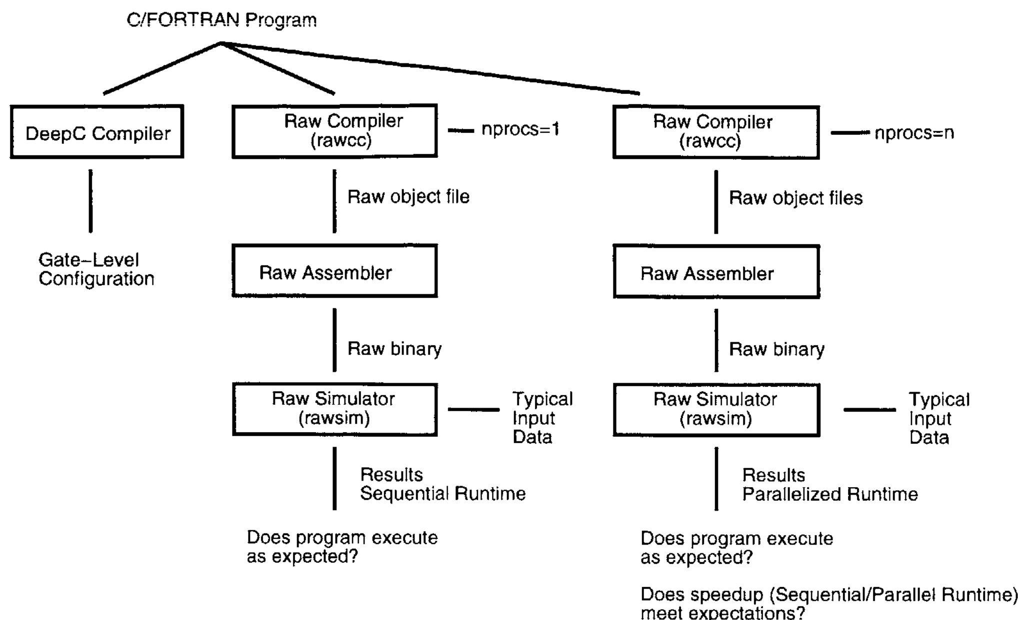


Figure 5-10: Simulation with the Raw Simulator

compile to multiple tiles. The programmer may have made a mistake in algorithm design, or more likely failed to understand the limitations of the parallelizer. Because DeepC is integrated with the Raw Compiler, it can target a Raw Machine with a similar number of tiles. The result is then run on a Raw Simulator (or perhaps a Raw chip!). Figure 5-10 depicts this approach. A DeepC compiler flag (*-t raw*) generates Raw code rather than RTL. In summary, the basic premise of this approach is that parallelization of the user program can be analyzed quickly, before the more complicated task of specializing and parallelizing.

5.4.3 Compiled-code Simulation with HaltSUIF

The next approach verifies compilation up to the instruction-scheduling phase. The idea is to feed the scheduled execution time of each basic block to a compiled-code profiler. Using a modified version of Harvard's HaltSUIF profiler, DeepC annotates each basic block with its static length. Then during profiling, dynamic runtime is accumulated with the following equation:

$$Runtime = \sum_{bb \in program\ basic\ blocks} length_{bb} \times frequency_{bb},$$

where $length_{bb}$ is the number of clock cycles (states) needed to execute basic block bb. The variable $frequency_{bb}$ is the number of times the block is executed dynamically, given some typical input data.

Figure 5-11 shows the flow for HaltSUIF simulation. Compilation interrupts parallelization to generate a matching sequential program. Then, the run length of each basic block, after parallelization and space-time scheduling, is determined. Given these lengths and following the HaltSUIF profiling methodology, runtime statistics can be generated.

A caveat to this approach is that HaltSUIF cannot execute a multi-threaded version of

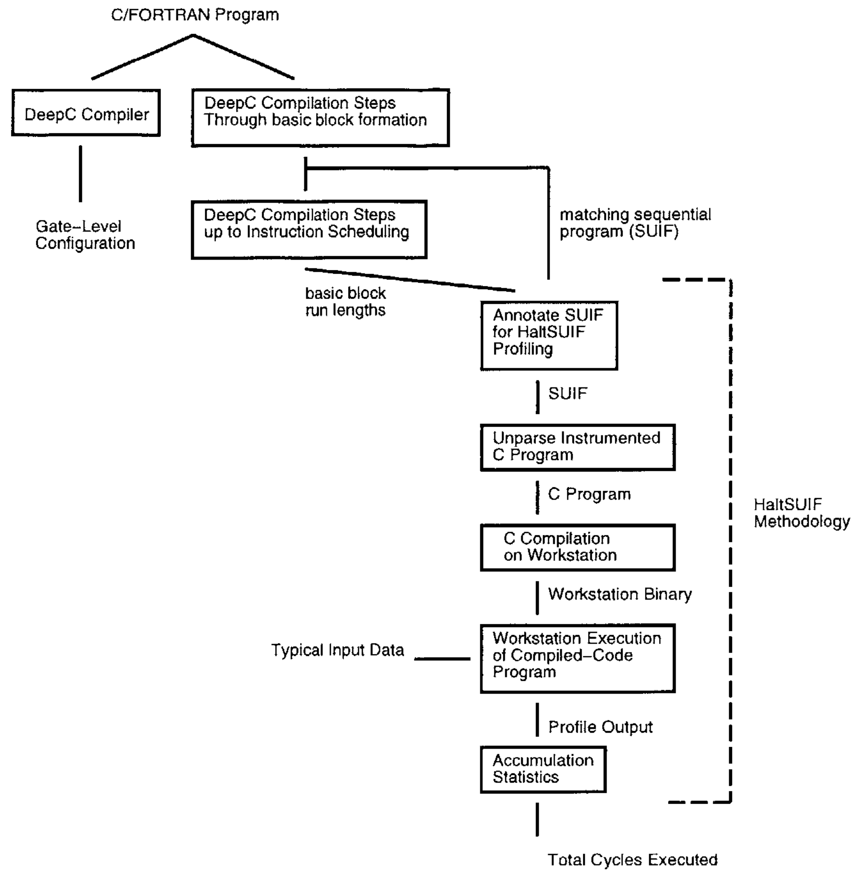


Figure 5-11: Simulation with HaltSUIF

the program. The program, annotated with the basic block schedule lengths, is unparallelized, so this approach only works for programs with data-independent control flow.

5.4.4 RTL Simulation with Verilog Simulator

Verilog simulation can be performed at several downstream locations. This section focuses on pre-synthesis, post-synthesis simulation, and use of simulation for power estimation. As in the previous cases, a DeepC compiler flag invokes one of these modes.

Pre-Synthesis / Behavioral

Figure 5-12 depicts the overall compilation flow for pre- and post-synthesis simulation. Just before RTL synthesis, the DeepC compiler outputs Behavioral Verilog. This generated Verilog is lower level than that accepted by Synopsys's Behavioral Compiler. In contrast to structural Verilog, the code has many unmapped parts, such as adders and shifters. Pre-synthesis verification is fast and cycle-accurate. That is, the total number of simulation clock ticks will equal the total program runtime, in clock ticks. However, this approach does not determine information such as gate area, clock speed, and power.

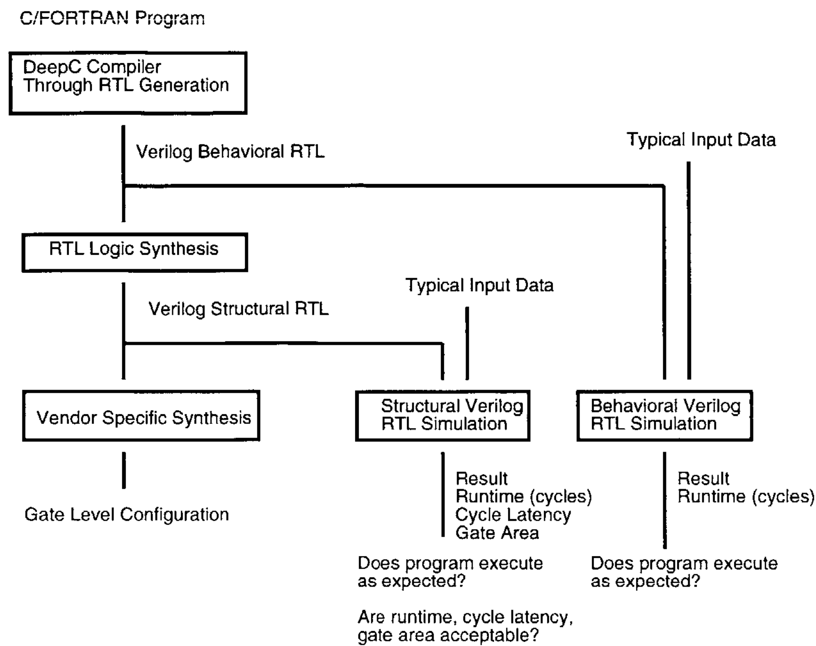


Figure 5-12: RTL simulation

Post-Synthesis / Structural

The gate-level Verilog generated by synthesis can be simulated. Post-synthesis simulation is much slower than pre-synthesis simulation. This scenario is depicted in Figure 5-12, with post-synthesis output fed to a simulator. Besides functional verification after synthesis (and thus catching any synthesis introduced bugs), the post-synthesis results include runtime, clock cycle latency, and gate area. If these results are not acceptable, the user will need to modify the input program.

Dynamic Power Estimation

One final use of Verilog simulation is dynamic power estimation. Dynamic power estimation, as the name suggests, dynamically estimates the power consumed by hardware. In contrast, static power estimation, because it does not execute the program, cannot accurately predict power consumption.

Simulating gate-level hardware gives a dynamic estimate as follows. First, an analysis pass annotates the RTL Verilog to record how many times each register switches. Second, behavioral simulation, performed with an input dataset, determines the number of times each register switches. Third, a zero-delay simulation determines the switching probabilities of all signals. Finally, the total dynamic power is calculated with the following equation:

$$DynamicPower = \frac{1}{ClockFrequency} \sum_{gate \in all\ gates} (Energy_{gate} \times Prob_{gate}),$$

where $Energy_{gate}$ is the switching energy of a gate and $Prob_{gate}$ is the switching probability resulting from the simulation.

Figure 5-13 depicts the CAD flow to perform dynamic power estimation. The flow is

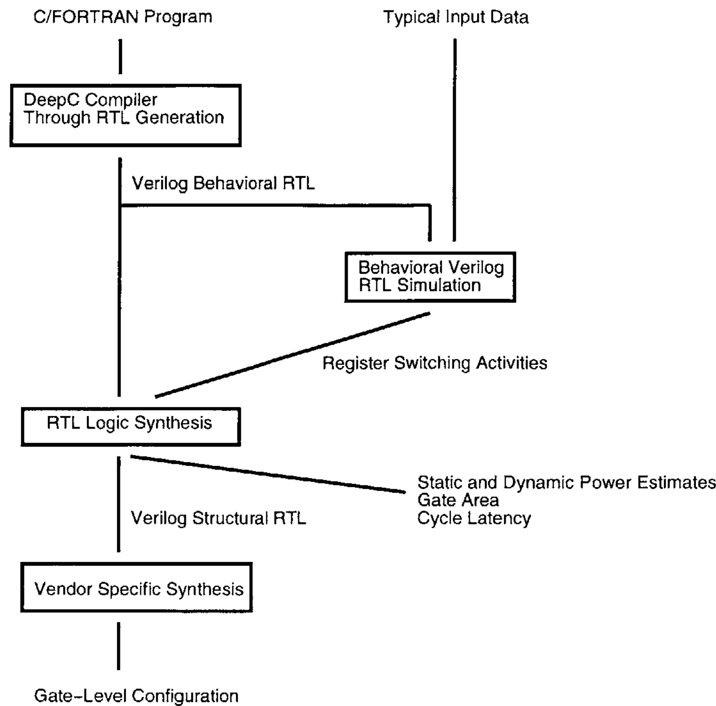


Figure 5-13: RTL power estimation

the same as that of manual hardware designers and is common in industry. The resulting estimations are more accurate than static power estimation, but still limited by the accuracy of the technology library and by the design compilers ability to estimate interconnect costs. DeepC has a compiler flag to invoke power estimation; this flag requires that the target synthesis libraries contains power information.

5.4.5 Verification with a Logic Emulation

DeepC can target a logic emulator (see [12] for tool flow) for high-speed, gate-level verification. Note that the methodology introduced in Section 5.3, where a logic emulator is described as a DeepC target, is very similar to traditional verification approaches. Because the details of logic emulation are closing related to previous FPGA computing research, Appendix B provides additional details on my early integration of DeepC with logic emulation.

5.5 Summary

This chapter has described the components of the DeepC compiler, including a Traditional Frontend Phase, a Machine Specialization Phase, and a CAD Tool Phase. An advanced path includes a Parallelization Phase and a Space-Time Scheduling Phase. These passes leverage compilers from academia and CAD tools from industry. Many new passes, predominately in the Machine Specialization and SpaceTime Scheduling Phase, were developed exclusively for DeepC. Finally, this chapter explained the hardware and simulation platforms needed.

Chapter 6

Results

With future silicon budgets approaching 100M transistors, we need to consider integration of other platform components (e.g. Memory controller, graphics). We need to consider special purpose logic, programmable logic, and separately programmable engines. But all have very complex/costly software issues.

— *Fred Pollack, Intel Fellow, Director of Microprocessor Research Lab, Micro32 Keynote speech slides, November 1999.*

IBM and Xilinx recently announced a partnership to create a new generation of devices for use in communication, storage, and consumer applications. Under the agreement, we are working together to embed IBM PowerPC processor cores in Xilinx VirtexTM-II FPGAs.

— *Ann Duft, Manager of North American PR, Xilinx, February 2001.*

The industrial movement to merge gate reconfigurability and instruction programmability is underway. Undoubtedly their merger will not be perfectly balanced — one paradigm will take the leading role. The chance for programmable logic to take the lead, or at least play more than a peripheral extra, is slim unless its compiler is competitive. Therefore, I include extensive results to support my thesis that specialization of traditional architectural mechanisms with respect to an input program is the key to efficient compilation of high-level programs to gate-reconfigurable architectures.

These results provide coverage of many practical issues that arise when building a compilation system of this complexity. Interactions between compiler phases have been notoriously problematic — a complete system of this size, combining specialization, parallelization, and logic synthesis, requires demonstration of its viability.

I begin by delineating experimental apparatus and presenting a new benchmark suite. The chapter focus is then on basic and advanced results, along with sensitivity analysis of important specializations.

6.1 Experimental Setup

The experimental setup includes the software tools that constitute the DeepC Compiler implementation described in Chapter 5, along with other tools. Table 6.1 lists the major compiler tools and version numbers. These tools are products of collaboration across several compiler and computer architecture groups at MIT. Bitwise [127] is a set of compiler

Company/Group	Program	Description	Version	Date	Ref.
MIT (Deep Project)	deepc	DeepC Silicon Compiler	2.0.1	May 2001	this work
MIT (Deep Project)	dbench	Deep Benchmark Suite	2.0.1	May 2001	this work
MIT (Deep Project)	bitwise	Bitwidth Analysis	2.0.1	Apr 2000	[127]
MIT (Raw Project)	rawcc	Raw Compiler	3 beta 8	Apr 2000	[136]
MIT (Rinard Group)	SPAN	Pointer Analysis	1 beta 5	Apr 2000	[114]
Stanford/MIT	SUIF	Compiler Infrastructure	1.2/mit/0.b13	Apr 2000	[140]

Table 6.1: Description of DeepC experimental components

Company/Group	Program/Tool	Description	Version	Date
MIT (Raw Project)	RawSim	Raw Simulator	version 0 beta 4	Apr 2000
MIT (Raw Project)	rbinutils	Raw Utilities	version 2 release 0	Apr 2000
Cadence	Affirma NC Simulator	Verilog Simulator	v2.1.(p2)	Nov 1998
Chronologic	VCS	Verilog Simulator	version 5.2	Dec 2000
GNU	gcc	C++ compiler	version 2.95.2	Oct 1999
Synopsys	Design Compiler	Synthesis	1999.10	Sep 1999
Toronto	VPR	Academic FPGA Tool Flow	version 4.30	Mar 2000
Xilinx	ngdbuild, par, etc	FPGA Tool Flow	v3.2.0.5i D.24	Oct 2000
IKOS/VMW	vlc,synopsys library	VirtuaLogic Compiler	version 2.0.12	Oct 1998

Table 6.2: Other experimental tools

analyses implemented by Mark Stephenson and tightly coupled into DeepC. Bitwise is generally applicable, a set of standard SUIF passes, although it has been incorporated only in the DeepC system. DeepC also leverages Radu Rugina's pointer analysis package (SPAN), a package developed in Martin Rinard's group and in wide use for compiler research. Finally, DeepC invokes many Rawcc compiler passes. Rawcc passes include Rajeev Barua's static memory analysis passes in MAPS [14], and Walter Lee's scheduling and partitioning algorithms [89]. These passes are based on a version of the Stanford SUIF compiler infrastructure maintained as a part of the MIT Raw project.

Auxiliary tools are in Table 6.2. These tools include compilers, synthesizers, simulators, emulators, and the requisite computer workstations (not shown in the table). The Raw simulator and utilities enable comparison of FPGAs and embedded processors. The Verilog simulators permit gathering of cycle counts for programs compiled to the gate level as well as performing traditional verification. The Gnu C compiler and a 300 MHz Sun UltraSPARCII generate workstation cycle counts and run times. Synopsys synthesizes to several targets, including IBM SA-27E ASIC, Xilinx FPGA, and an academic model for VPR. The IKOS software and a donated VirtuaLogic Emulator serve as a reconfigurable platform and a verification platform. These or similar tools are needed in order to reproduce my results.

Version 2.0

These results were generated with DeepC version 2.0. Previous results published in [11] were generated with DeepC Version 1.0. Version 2.0 is more robust, with many extensions, including bitwidth analysis, code generation for multiple tiles, network overclocking, multiple network and memory ports, and support for Xilinx Virtex parts and block RAMs.

Benchmark	Type / Source / LOC	run time	Primary (Sec) Array size	Data Width	Description
adpcm	Multimedia / Mediabench / 216	1.5M	2407	8	Speech compression
bubblesort	Dense Mat. / Nasa7:Spec92 / 80	1.6M	512	32	Bubble Sort
convolve	Multimedia / Mediabench / 94	61K	16(256)	16	Convolution
histogram	Multimedia / Mediabench / 139	81K	64x256	8	Image histogram
intmatmul-v1	Dense Matrix / Rawbench / 100	429K	32x32(32x32)	16	Integer Matrix Mult.
intmatmul-v2	Dense Matrix / Deepbench / 109	429K	16x128(16x16)	16	Integer Matrix Mult.
intfir	Multimedia / Mediabench / 80	7.1M	32x32	16	Integer FIR Filter
jacobi	Dense Matrix / Rawbench / 95	317K	64x64	8	Jacobi Relaxation
life-v1	Dense Matrix / Rawbench / 167	567K	32x32	1	Life
life-v2	Dense Matrix / Deepbench / 129	269K	32x32	1	Life (v2)
median	Multimedia / Mediabench / 105	723K	16x16	32	Median filter
mpegcorr	Multimedia / UC Berkeley / 153	14K	32x32	16	MPEG-1 Encoder
parity-v1	Reduction / Deepbench / 74	112K	128(1024)	32	Parity / checksum
parity-v2	Reduction / Deepbench / 74	240K	128(1024)	32	Parity / checksum (v2)
pmatch-v1	Reduction / Deepbench / 83	1.0M	128(1024)	32	Pattern Matcher
pmatch-v2	Reduction / Deepbench / 89	1.0M	128(1024)	32	Pattern Matcher (v2)
sor	Dense Mat. / Greenwald / 86	62.0K	64x64	32	Successive Over Relax.

Table 6.3: DeepC Benchmark Suite

Column runtime shows the run times, in clock cycles, for uniprocessor code generated by GCC 2.95.2 -O5 on an UltraSPARC IIi. LOC is lines of code, including comments. Data width is for the main arrays. Several benchmarks have two versions with slightly different source code.

6.2 The Deep Benchmark Suite

These benchmarks are released as the Deep Benchmark Suite version 2.0. Basic benchmark characteristics are in Table 6.3. Each benchmark is a self-initializing C program. Some benchmarks are from the Raw Benchmark Suite [10]. Others benchmarks are simplified versions of the University of Toronto (UT) Mediabench benchmarks (`adpcm`, `convolve`, `histogram`, `intfir`, `median`). I have also added three programs: `parity`, `pmatch`, and `sor`.

The following paragraphs describe the benchmark programs in more detail. An asterisk identifies the benchmarks used in the advanced results: `intmatmul`, `jacobi`, `mpegcorr`, `life-v2`, `pmatch`, and `sor`. These cases are parallelizable.

ADPCM (`adpcm`) Adaptive differential pulse code modulation is a family of speech compression and decompression algorithms. This implementation is from 16-bit linear PCM samples and compresses them 4:1 into 4-bit samples. This coder is from the IMA Compatibility Project, Version 1.0, 7-Jul-92. I have manually removed *structs* to avoid supporting them in DeepC's prototype frontend.

Bubblesort (`bubblesort`) The `bubblesort` benchmark is the familiar sorting algorithm from introductory computer science textbooks. This algorithm has an $O(n^2)$ time complexity. This version is hand-coded in a manner that is amenable to predication. Note that for this benchmark the best source code for logic synthesis significantly penalizes the performance of the processor case. I avoid this problem by using alternate source code, that best matches processors, for processor results.

Convolve (`convolve`) The `convolve` benchmark is a typical media benchmark that computes the convolution of two input vectors at each step in time. For a length M input vector and a length N input vector, the output is a vector of length M+N-1.

Histogram (`histogram`) The `histogram` benchmark is from the Mediabench suite. The program enhances a 256-gray-level, 64x64 pixel image by applying global histogram equalization. The program uses routines and algorithms found in [52].

Integer FIR (`intfir`) Finite input response filter is a media benchmark that multiplies an input stream by a tap filter. This version multiplies integers — the more standard FIRs work with complex reals.

Integer Matrix Multiply* (`intmatmul-v1`) This benchmark multiplies two integer matrices. It uses a simple algorithm with a triply nested loop.

Integer Matrix Multiply Version 2* (`intmatmul-v2`) In this version, array indices have two dimensions instead of one, $a[i][j]$ instead of $a[i * JSIZE + j]$.

Jacobi* (`jacobi`) Jacobi relaxation is an iterative algorithm. Given a set of boundary conditions, it finds discrete solutions to differential equations of the form $\nabla^2\mathcal{A} + \mathcal{B} = 0$. Each step of the algorithm re-computes points in a grid, assigning each point the average of the values of its nearest neighbors. This version is integer.

Life (`life-v1`) Conway's Game of Life program [18] is represented on a two-dimensional array of cells, each cell being alive or dead at any given time. The program begins with an initial configuration for the cells and obeys the following set of rules: a living cell remains alive if it has exactly two or three living neighbors, otherwise it dies; a dead cell becomes alive if it has exactly three living neighbors, otherwise it stays dead.

Life Version 2* (`life-v2`) This re-written version of `life` uses Boolean algebra instead of control flow in the inner loop.

Median Filter (`median`) Median filter is a common multimedia benchmark. Given a 2-D input array, it produces a 2-D output array where each output element is the median of its nine neighbors in the input array.

MPEG Kernel* (`mpegcorr`) The MPEG standard includes encoders and decoders for video bitstream playback. This benchmark is an implementation of the correlator in the MPEG-1 Software Video Encoder. This version is a kernel, not a full MPEG implementation.

Parity* (`parity-v1`) The `parity` benchmark computes the parity of 32-bit elements and accumulates the result into a checksum. An inner loop contains a parity calculation equation.

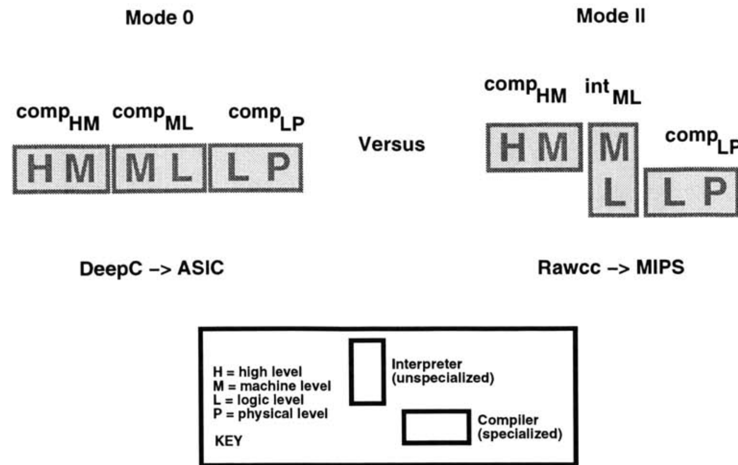


Figure 6-1: *Mode 0* versus *Mode II*

Parity Version 2* (`parity-v2`) In this version, the iteration space is transposed to expose low-order parallelism. It also fixes a bug in the previous version that uses an OR instead of XOR (previous version results were not re-compiled because of time limitation). The transposition, which DeepC does not perform automatically, is needed because the parallelizer unrolls inner loops.

Pattern Match* (`pmatch-v1`) Given a string of length M and a pattern of length N, Pattern Match determines if the string contains the pattern. A nested inner loop compares characters of the pattern with characters of the string.

Pattern Match Versions 2* (`pmatch-v2`) In this version the iteration space is transposed to expose parallelism in the inner loop.

Successive Over Relaxation* (`sor`) The source code for the `sor` benchmark, a well-known five point stencil relaxation, is borrowed from Greenwald's Master's Thesis [65]. This benchmark is similar to `jacobi`.

6.3 Basic Results

Recall the evaluation modes introduced in Chapter 2 and summarized in Figures 2-6 and 2-7. The modes studied in basic results include: *Mode 0* (for example, ASICs), *Mode I* (for example, FPGAs), and *Mode II* (for example, RISC processors). In order to support my thesis, this section applies specialization to the architectural mechanisms commonly found in a processor (*Mode II*). Without any other changes, this specialization yields results for the uninterpreted *Mode 0* (Section 6.3.1). Combining this specialization with gate reconfigurable FPGA targets ($I_{L \to P}$) gives *Mode I* results (Section 6.3.2). Data tables for these results are included in separate appendices, which the following text references.

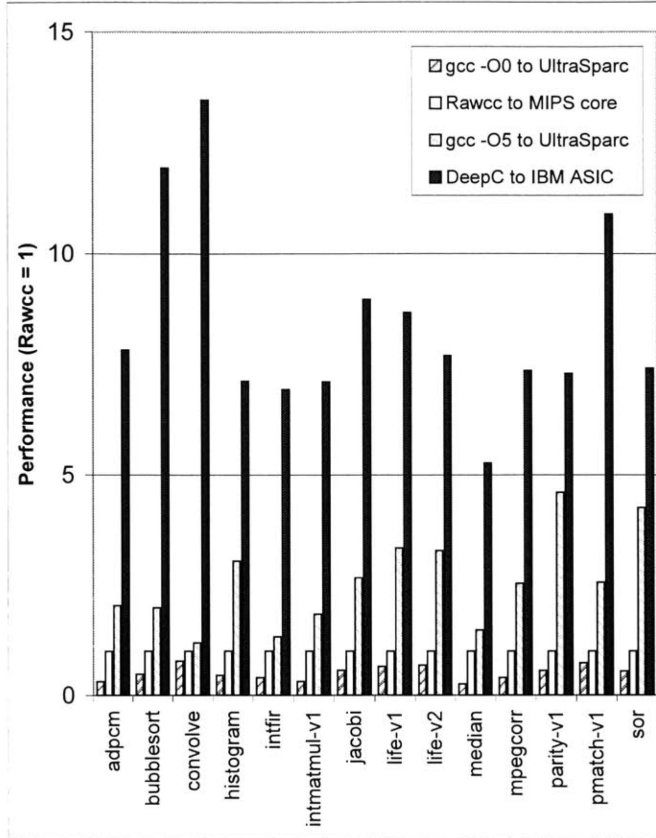


Figure 6-2: Runtime comparisons between DeepC, Raw, and an UltraSPARC

The data for this graph is in Appendix Table C.1. The graph is normalized to the Rawcc data. Performance is the inverse of the number of clock cycles needed to evaluation each benchmark. Target clock speed is 300 MHz.

6.3.1 *Mode 0* Basic Results

Recall from Chapter 2 that *Mode 0* does not have interpretation layers. DeepC transforms *Mode II* evaluation into *Mode 0* evaluation by specializing the machine level interpreter ($I_{M \to L}$). DeepC can generate results by targeting Verilog RTL and then synthesizing to the same IBM ASIC target library used for Raw. Notice from the dominos in Figure 6-1 that this comparison isolates the difference between compilation and interpretation at the machine-level, with the fewest changes elsewhere.

RTL Verilog Target: Pre-Synthesis

Figure 6-2 shows the runtime, in clock cycles, after compiling the Deep Benchmark Suite to RTL Verilog. The figure compares these results to Rawcc targeting a 300 MHz MIPS processor. For more comparisons, the figure also presents the cycle counts on a 300 MHz UltraSPARC IIi. For the UltraSPARC, I obtained data using gcc's -O0 flag (unoptimized) and the -O5 flag (highly optimized). (Recall that gcc is the GNU C Compiler developed by Richard Stallman and others.) Because the benchmarks are short, I repeatedly executed each for one second.

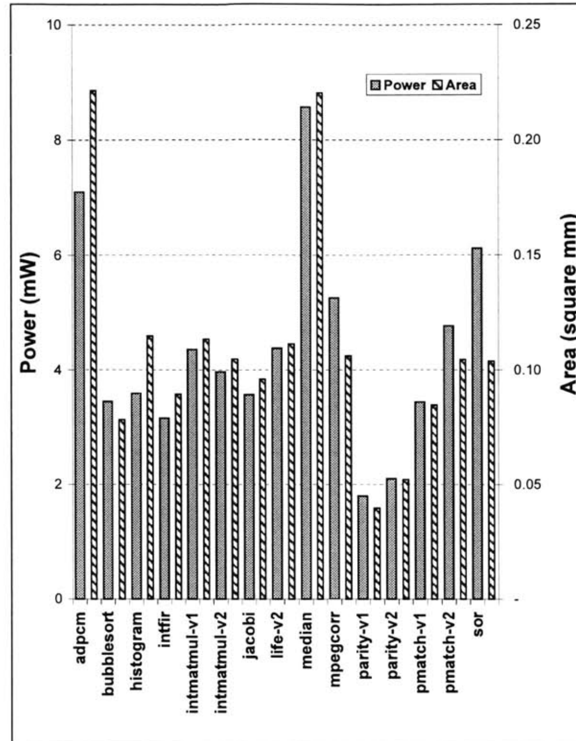


Figure 6-3: IBM SA-27E ASIC non-memory power and area

Results are obtained after Synopsys synthesis to IBM SA-27E ASIC technology. Area numbers shown assume a cell size of 3.762 um^2 with 50 *percent* routability. Area and power estimates do not include RAM cost or additional overhead for pads and periphery circuitry. A clock rate of 300 MHz or more is achieved. Power is estimated at 300 MHz. Appendix Table C.2 lists the complete data.

Being a research compiler, the Rawcc compiler is missing some traditional frontend compiler optimizations. The Raw cycle counts thus fall between the optimized and the unoptimized gcc cycle counts. Notice that the differences between Rawcc and gcc are reduced for `convolve`, which is already unrolled and thus easier to optimize.

Along with incorporation of advanced compiler optimizations, the UltraSPARC processor and gcc combination also has advanced architectural features such as superscalar scheduling. These techniques are neither implemented in the Raw processor nor specialized by DeepC. Thus, at least a portion of the difference between the Rawcc and the optimized UltraSPARC cycle counts should be transferable to a commercial DeepC implementation. In addition, some differences will not transfer because some gains result from DeepC speeding up under-optimized intermediate code.

Given the above caveats, specialization decreases cycle count by 5× to 13× versus the interpretive MIPS core, resulting in a similar performance gain. Even when conservatively comparing to the commercial UltraSPARC case, specialization averages a gain of 3.7×. When clock frequencies are the same, these gains are absolute.

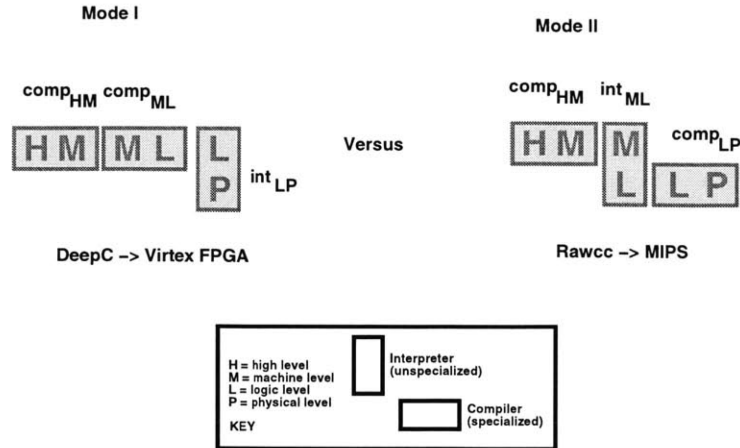


Figure 6-4: *Mode I* versus *Mode II*

ASIC Synthesis Target: IBM SA-27E Process

Figure 6-3 graphs the power and area reported by Synopsys after synthesizing to the IBM SA-27E technology library. The area numbers are manually converted from generic "cells" to physical numbers using the technology constants in the IBM Cell Library Guide [76]. More accurate data could be obtained by applying downstream physical design tools proprietary to IBM. I used this technology because it was available at MIT in conjunction with Raw, allowing comparison of evaluation modes across constant silicon technology. At 300 MHz, in the IBM process, benchmark power varied from less than 2 mW to over 8 mW. Area varied from under 0.05 mm^2 to just over 0.20 mm^2. Most of the benchmarks were under 0.10 mm^2 and consumed power in the $3 - 4$ mW range. These results do not include I/O and RAM power and area.

6.3.2 *Mode I* Basic Results

As expected, *Mode 0* results are excellent when compared to *Mode II* because there is one less interpretation layer. This section adds back an interpretation layer, but at the logic level rather than the machine level, forming *Mode I*. Comparing the *Mode I* and *Mode II* dominos in Figure 6-4 reveals this interpreter-exchange relationship.

Mode I logic-level interpretation is supported on traditional FPGAs. The next two sections present both commercial and academic FPGA results. Results for the IKOS logic emulator are in Appendix B.

Commercial FPGA Target: Place and Route to Xilinx Virtex-E-8

To generate *Mode I* FPGA experiments, I used Synopsys to synthesize the RTL for each benchmark to Xilinx Virtex-E-8. Note that 8 refers to the speed grade of the FPGA. DeepC contains scripts to automate synthesis. Synthesis data is in Appendix Table C.3.

Speedup versus a MIPS core, along with lookup table (LUT) and register area, is in Figure 6-5. The average performance gain is 2.4×. Although a few of the benchmarks are larger than one thousand LUTs, most are small, consuming only a few hundred LUTs. The average combinational area consumed is 700 LUTs and the average register usage is 300 flip-flops.

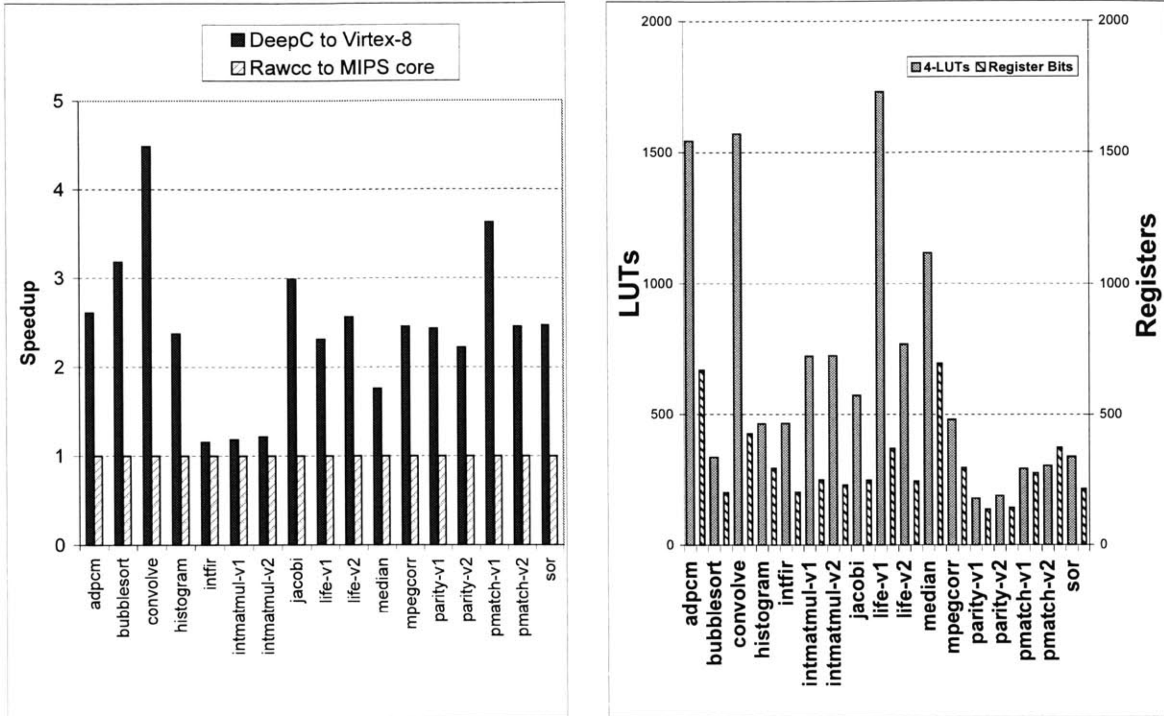


Figure 6-5: Speedup, LUT and register area for Virtex-E-8

Clock speeds ranged from 50–100 MHz and are reported along with other data in Appendix Table C.4. Performance for `intfir`, `intmatmul`, and `intmatmul-v2` is half speed because a synthetic multiplier is used instead of faster vendor soft macros.

The clock speed of several benchmarks is less than the target 100 MHz for a couple of reasons. First, the `bubblesort` and `life` clock speeds were limited by the multiplexers created from unpipelined predication instruction. Without these multiplexers, or with pipelined ones, a target clock of 100 MHz is possible. Also, `intfir` and `intmatmul` were limited by a synthetic multiplier, which reduced performance by 2×. Activation of the Xilinx XBLOX multiplier can overcome this limitation (DeepC does not do this). Moreover, newer FPGAs have built in hard multipliers. I verified that the rest of the circuit would run at speed by substituting adders for the multipliers. However, to be conservative, the results here are based on the clock speeds I obtained.

Figure 6-6 shows the absolute and relative power reported by the Xilinx Power Estimator, Version 1.5. The right graph shows relative power consumption sorted by percent of power consumed by RAMs. Data for these results is in Appendix Table D.8 and Table C.6. Table C.5 contains the data, including switching activity, used to generate this result. This comparison assumes a MIPS core consuming 1 Watt at 300 MHz. Quiescent power is a function of the FPGA device. The remaining cumulative power bars are for LUTs, register bits, and block RAMs.

For SOR, I hand partitioned the RAMs generated by DeepC to minimize the total number of RAM enables. This improvement, labeled sor-mem4, would fit into DeepC's Machine Specialization Phase. It reduces RAM power from $414mW$ to $182mW$. There are likely similar improvements that save even more power.

Figure 6-7 shows the energy and the energy-delay product versus a MIPS core. The

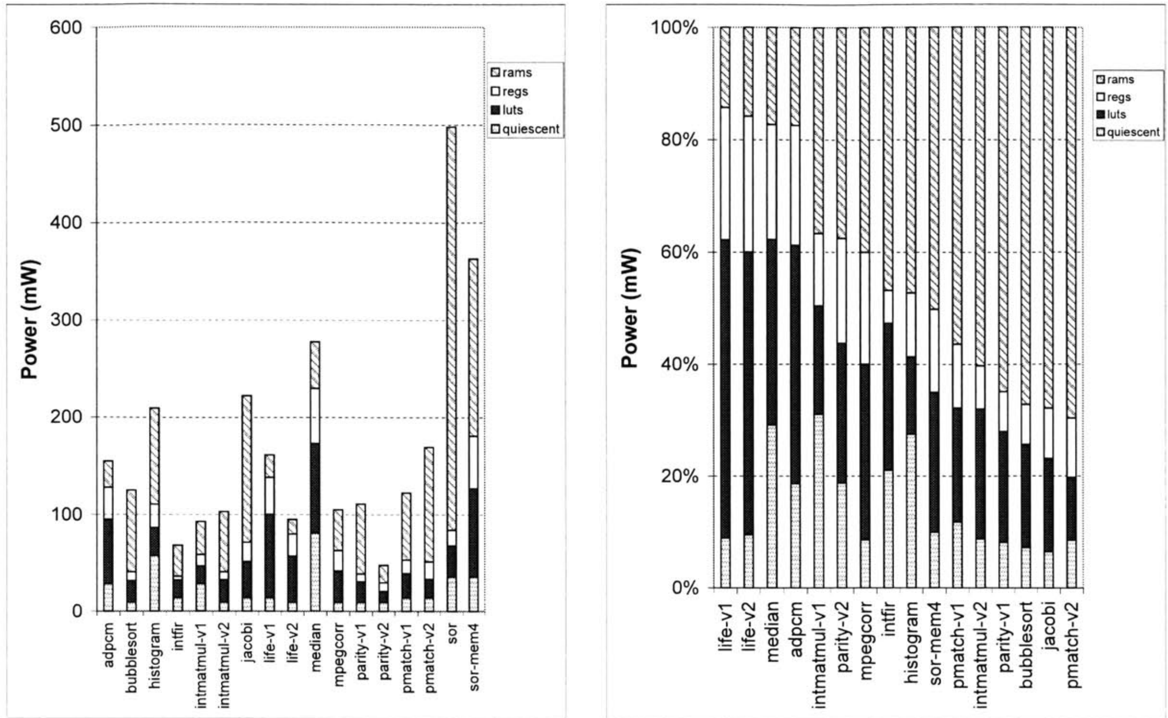


Figure 6-6: Absolute and relative power reported by the Xilinx Power Estimator 1.5

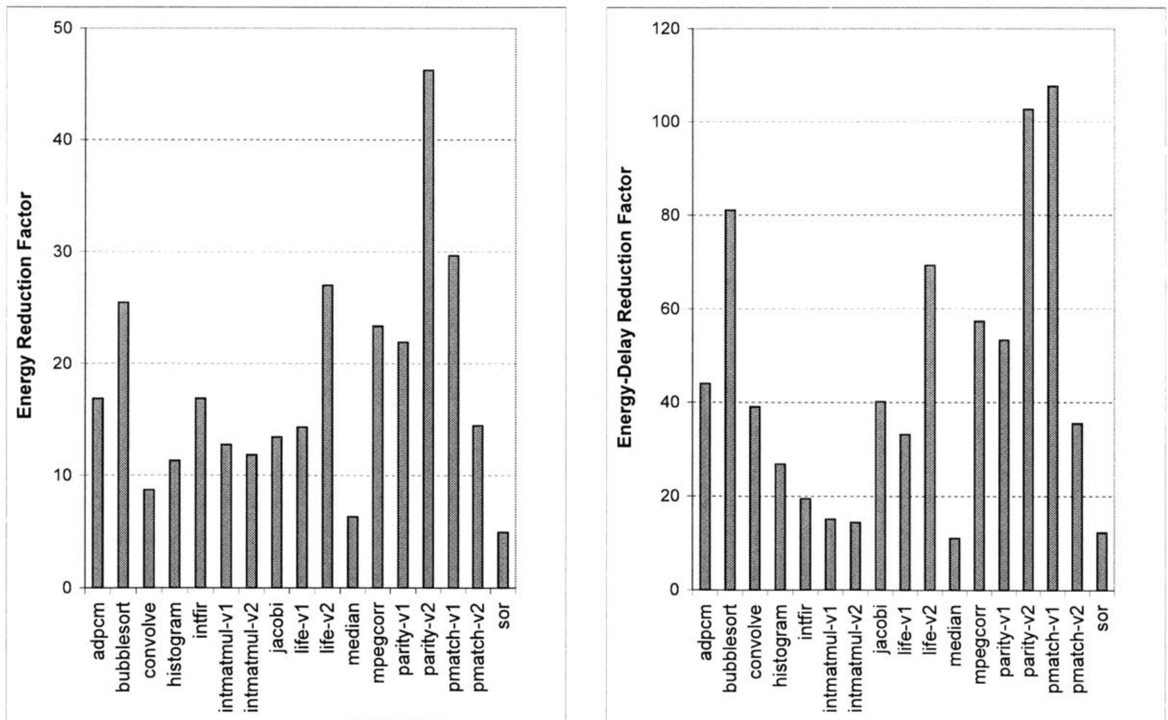


Figure 6-7: Energy and energy-delay comparison

DeepC specializing to Virtex-E-8 (*Mode I*) versus a MIPS (*Mode II*).

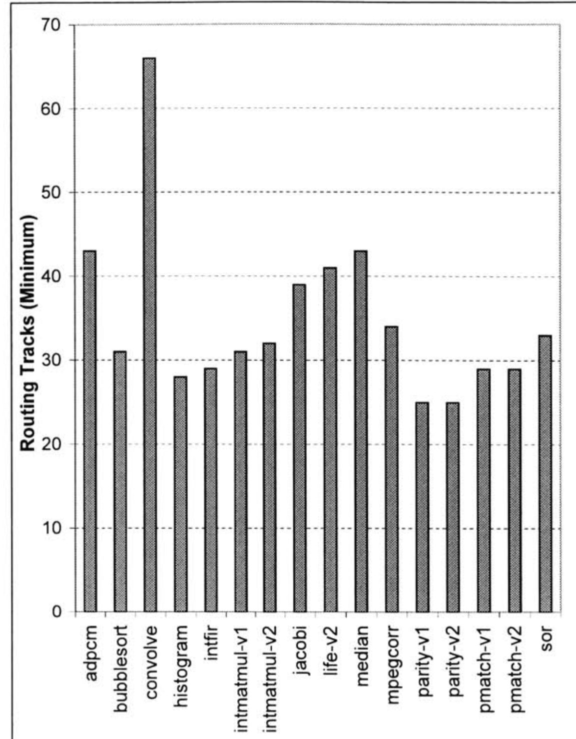


Figure 6-8: Minimum track width to place and route each benchmark

reduction factors for energy range from 5× to nearly 50× in favor of *Mode I*. The Energy-delay reduction is even more dramatic, ranging from $10 - 100\times$ better for it Mode I.

Academic FPGA Target: Experiments with VPR

In this section, DeepC's interface to VPR (Section 5.3) is configured to find, with a binary search, the minimum number of tracks needed to route the design. Other FPGA parameters are set to match the Xilinx Virtex part — see Appendix E.

Most FPGA silicon area is devoted to routing, so the number of routing tracks and the number of transistors needed for routing provides informative metrics of specialization efficiency (beyond LUT counts). This version of VPR does not support embedded RAMs, so generating this result required making all memories external to the FPGA. To make design memories external, I formed a new design from all the non-memory circuitry and left only connections to external memories. Along with memory connections, all other I/Os are connected to the periphery of the layout. Section 6.5 uses this methodology to compare partitioned versus unpartitioned designs for parallelized benchmarks.

Figure 6-8 shows results for minimum routing tracks. The minimum routing track results range from 25 to 66 tracks, low compared to recent FPGA track capacities. A full suite of VPR results is in Appendix E. This suite includes statistics for the number of blocks, nets, global nets, average input pins per LUT, average net length, total wire length, routing area per LUT, critical paths, and total tracks and segments.

DeepC includes a modified version of VPR's X-Windows interface. This version gener-

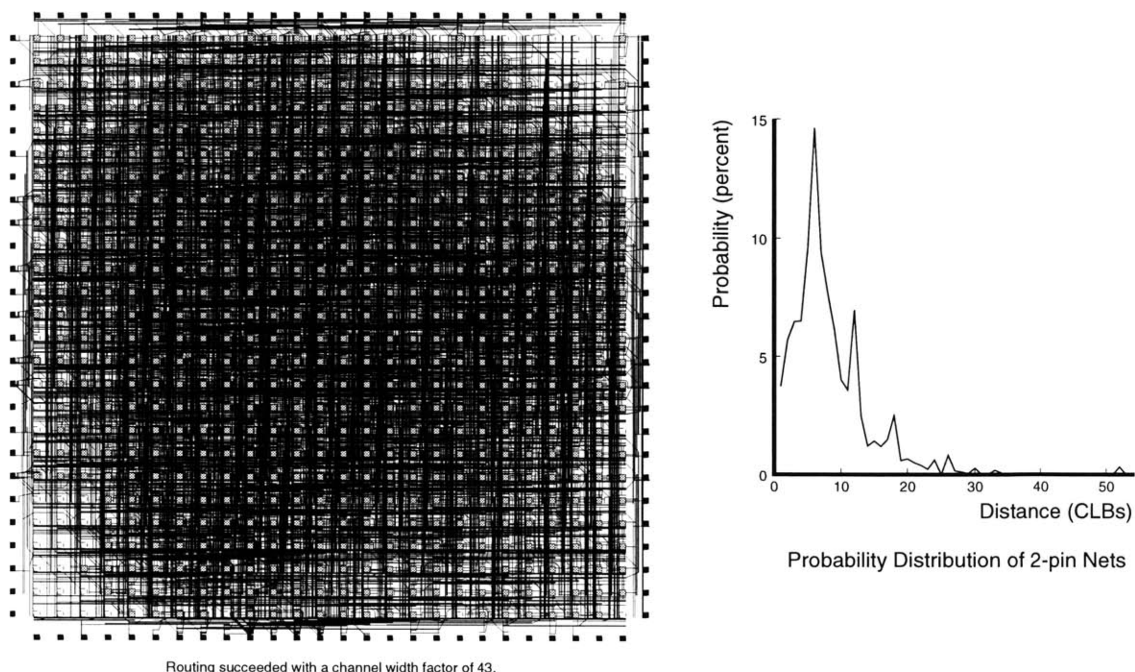


Figure 6-9: adpcm layout and wire length distribution

ates a postscript output of the resulting layout. Figure 6-9 shows a layout of `adpcm` generated from the place and route with VPR along with its associated wire length distribution. The distribution of 2-pin net lengths is a good metric for routability. This data is generated with DeepC's default *-O5* compiler flag. In the layout, each small square represents a CLB containing four LUTs and four registers. Shaded squares contain active logic. The density of interconnecting wires should give the reader an intuitive feel for the compilation effort required. Non-orthogonal lines represent turns through the routing switches, which are not otherwise in the figure. Layouts and wire-length distributions for the remaining benchmarks are included in Appendix E.

6.4 Basic Sensitivity Analysis

In order to provide more supporting evidence for my thesis, this section studies the sensitivity of basic results to the specialization of particular mechanisms. Before examining post-synthesis effects, this section first studies the components of *cycle count* speedup. Varying DeepC compiler flags demonstrates that memory disambiguation and control predication yield worthy increases. Extra unrolling benefits all benchmarks, although some gains will be offset by a post-synthesis decrease in clock speeds.

Simulation of DeepC-generated RTL is introduced in Section 5.2.5 as a means of verifying the correctness of the compiled input program. A successful simulation predicts the program execution time in units of clock cycles. The number of clock cycles is determined by the number of iterations in each inner loop multiplied by the length of the basic blocks in the respective loop. In DeepC, the length of the loop basic block is determined statically, during instruction scheduling (Section 3.3.1). Among other things, the schedule length is sensitive to the following three DeepC feature flags:

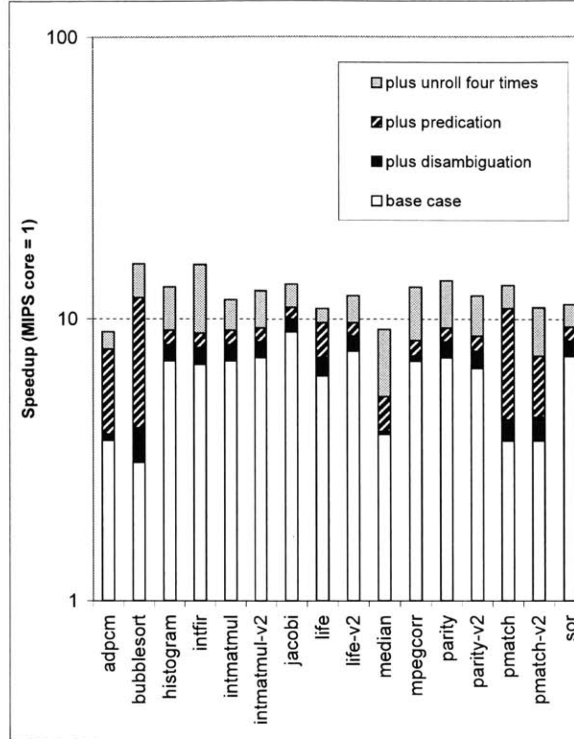


Figure 6-10: Components of cycle count speedup

A base case averages over 6× speedup versus the MIPS core simulated by RawSim. Specialization of memory (ecmap) and control structure (predication, unroll four times) increases gains to 12×. Appendix Table C.7 contains the cumulative data presented here.

- *-fecmap* and *-fno-ecmap* turn on and off equivalence class mapping, a form of memory disambiguation described in Section 4.1.1 as equivalence class unification. When ecmap is turned off, all memory is assigned to a single address space, generating one large RAM structure. When ecmap is turned on, different arrays are assigned to different local memories when references can be disambiguated.

- *-fmacro* and *-fno-macro* turn on and off macro formation code, the predicate generator in DeepC. When macro formation is turned on, some conditional control structures are converted to multiplexer logic, as described in Section 3.3.1.

- *-min-unrollx 4* unrolls the inner loop an extra four times. For basic results, the default unroll factor is one, so the result is 4× unrolling. This flag leverages the unroll routine used in the advanced modulo unroll phase, described in Section 4.1.1.

Varying these flags exposes the components of cycle count speedup (Figure 6-10). As before, all speedups are with respect to the MIPS core simulated by RawSim. Note that the figure is on a logarithmic scale to permit proper comparison of multiplicative optimizations (otherwise the top components on the bar chart would appear artificially enlarged). The first component, the base case, includes traditional compiler optimizations and all DeepC defaults except *-fecmap* and *-fmacro*. The default case keeps loops rolled. The remaining components result from cumulatively turning on *-fecmap*, *-fmacro*, and *-min-unrollx 4*.

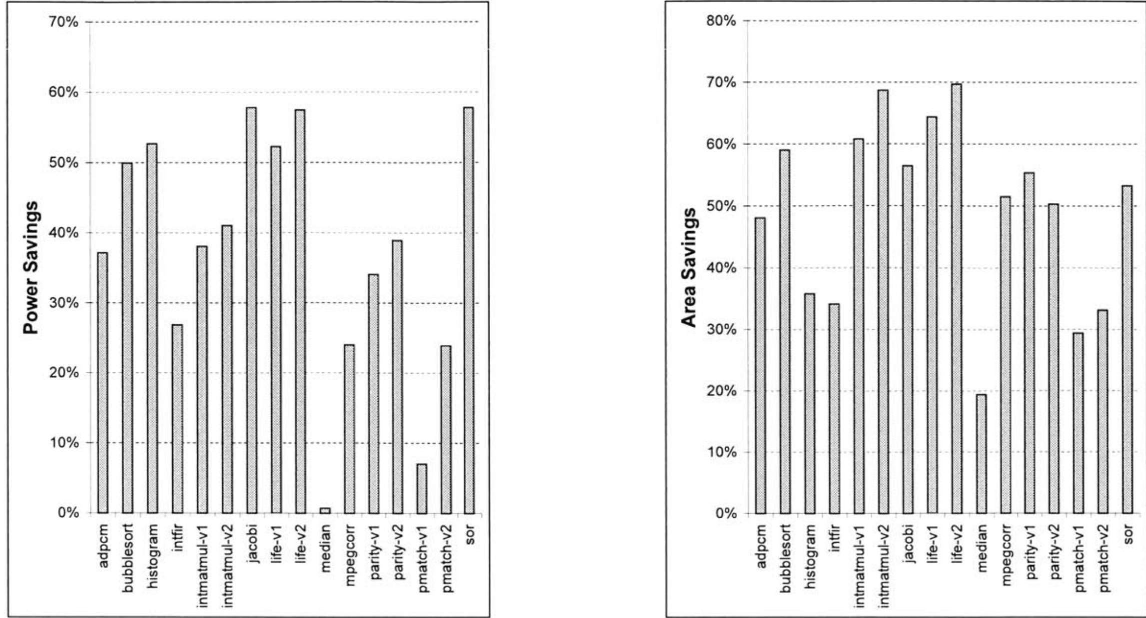


Figure 6-11: Benefits of bitwidth reduction in *Mode 0*

These comparisons are derived from data in Appendix Tables C.2 and C.8.

The component breakdown is quite revealing. The cumulative average speedup is 12×, more than double the base case speedup of 6×. Disambiguation with equivalence class unification has a small benefit for five of the thirteen benchmarks. As expected, this benefit only occurs when memory references are on the critical path. Likewise, predication can only help if there are potential IF-conversions in the inner loops of the program, true for only five benchmarks. For example, `bubblesort` has a convertible inner IF that permits a 2.9× speedup from predication. In addition, the more complex `adpcm` gains 2× from predication of several inner IFs.

Unrolling has the most consistent gain, increasing performance by 40 percent, on average, across all benchmarks. Unrolling has the most applicable benefits because it increases the size of basic blocks and amortize branch overhead for inner loops. However, because unrolling increases the total number of states, post-synthesis clock speed may be penalized (Section 6.6.1 studies this problem). Unrolling also generates more hardware and increases synthesis time.

These cycle count speedups are roughly the same for *Mode 0* and for *Mode I*. However, cycle count is only one component of total performance. Therefore, the next section continues by studying how individual specializations affect synthesis in both modes.

6.4.1 *Mode 0* Basic Sensitivity

This section presents a single *Mode 0* sensitivity result: sensitivity to bitwdith reduction. It discusses the effect of bitwidth reduction on power, area, and clock speed. These results demonstrate superior performance after bitwidth reduction.

Bitwidth Reduction is Necessary

Effective bitwidth reduction is an important contributor to efficient machine-level compilation. Both this section and the sequel present evidence to this point. This study re-synthesizes the designs reported in Section 6.3.1 with bitwidth optimizations turned off (*-fno-bit*). Figure 6-11 shows the savings of the original results versus this less optimized version. Savings include both power (37 percent on average) and area (49 percent on average). Power and area savings are correlated, but not exactly. Without bitwidth optimization, one third of the results (`bubblesort`, `intfir`, `jacobi`, `life-v1`, and `sor`) failed to meet the target clock speed of 300 MHz. DeepC achieved or exceeded this speed on all the original results. Specializing the bitwidths of architectural features is a wonderful optimization — it improves all aspects of system performance without any offsetting overheads.

6.4.2 *Mode I* Basic Sensitivity

This section shows that *Mode I* synthesis results, like *Mode 0*, are sensitive to bitwidth reduction. Studies find that turning off two multiplexing optimizations — register allocation and resource allocation — permits specializations that improve performance but sacrifice area. Cumulatively, bitwidth reduction and register allocation save registers, but have a mixed effect on speed. A final data set shows that increased synthesis effort increases the chances of meeting target clock speed, although perhaps with an area penalty.

The results in this section are generated with FPGA vendor (Xilinx) tools. The details are contained in the following tables: Table 6.4 (LUT count), Table 6.5 (register count), and Table 6.6 (clock speed). The DeepC flags varied are: *-fbit*, *-freg*, *-resource none*, *-fmacro*, and *-fsynopt*. The following prose discusses the important points.

Bitwidth Reduction is Good

Bitwidth reduction is a good optimization on all accounts, as *Mode 0* results also showed. Bitwidth reduction greatly reduces synthesis time and workstation memory requirements, as they are in proportion to total hardware size. One benchmark, `intmatmul-v2`, failed to compile on a 1GB RAM workstation without this specialization. On average, clock speeds were penalized 26 percent when benchmarks are compiled without bitwidth reduction. LUT area increased 146 percent and registers increased 12 percent. Register usage increased 43 percent with register allocation also turned off.

Register Allocation and Resource Sharing Reduce Performance

Register allocation and resource sharing are two optimizations that multiplex hardware resources. Recall from Section 5.2.3 that DeepC's register allocator uses MachSUIF's *raga* pass. Resource sharing is done during RTL synthesis with the Synopsys CAD tool.

Turning off register allocation increases the number of registers, but usually reduces multiplexers and thus LUTs. Without register allocation, every intermediate variable is assigned to its own FPGA register. Likewise, turning off resource allocation reduces the sharing of function units generated during RTL synthesis.

When not multiplexed, the logic surrounding these registers and resources can be further specialized. In the tables here (Tables 6.4, 6.5, and 6.6), the *-O6* optimization level has register allocation and resource sharing turned off, while *-O5* has both on. Turning off register allocation increases clock speed 4 percent and turning off resource sharing increases

Benchmark	-O6 LUTs	-O5 LUTs	noreg LUTs	noshare LUTs	nosynopt LUTs	nobit LUTs	nobit-noreg LUTs
adpcm	814	823	751	758	839	1413	1115
bubblesort	351	337	324	344	340	550	550
histogram	443	463	392	409	471	862	741
jacobi	461	601	494	466	608	1153	984
life-v1	867	1721	810	863	1718	3862	1834
life-v2	572	784	555	582	757	2202	1579
median	742	1034	641	650	1050	1201	835
mpegcorr	425	416	351	373	429	917	908
parity-v1	166	161	145	143	164	300	300
parity-v2	197	189	158	169	193	355	355
pmatch-v1	258	254	213	233	258	447	447
pmatch-v2	308	254	213	233	258	447	447
sor	269	405	333	365	404	754	746

Table 6.4: Sensitivity of LUT area to specialization flags

Benchmark	-O6 Regs	-O5 Regs	noreg Regs	noshare Regs	nosynopt Regs	nobit Regs	nobit-noreg Regs
adpcm	576	361	575	575	361	611	759
bubblesort	221	201	221	221	201	247	247
histogram	377	306	391	391	306	367	402
jacobi	376	248	376	376	248	446	557
life-v1	893	368	897	897	368	794	1477
median	822	686	831	831	686	752	853
mpegcorr	397	281	397	397	281	496	577
life-v2	453	245	435	453	245	624	1062
parity-v1	158	137	158	158	137	241	241
parity-v2	172	153	171	171	153	264	274
pmatch-v1	254	300	310	310	300	311	311
pmatch-v2	337	300	310	310	300	311	311
sor	215	257	293	293	245	390	391

Table 6.5: Sensitivity of register bit area to specialization flags

Benchmark	-O6 MHz	-O5 MHz	noreg MHz	noshare MHz	nosynopt MHz	nobit MHz	nobit-noreg MHz
adpcm	105	53	66	100	67	53	57
bubblesort	83	79	63	78	70	87	84
histogram	98	81	95	101	102	67	77
intmatmul-v2	60	47	fail	52	46	fail	fail
jacobi	101	88	66	104	80	62	56
life-v1	79	55	72	86	57	37	39
median	90	96	92	100	76	71	79
mpegcorr	88	95	87	102	94	78	63
life-v2	97	73	65	83	81	65	42
parity-v1	97	104	101	92	99	104	104
parity-v2	110	104	110	107	106	72	104
pmatch-v1	98	93	100	106	105	69	80
pmatch-v2	103	101	104	90	94	82	86
sor	94	100	105	105	103	51	59

Table 6.6: Sensitivity of clock frequency to specializations

Target clock frequency is 100 MHz. *-O6*, is equivalent to *-O5 -fnoreg -fnoshare -fbram.* Block ram synthesis is enabled with *-fbram.* Only *-O6* includes *-fbram.*

clock speed 12 percent on average. The LUT count is reduced 34 percent and 25 percent, respectively. As expected, register count is increased — by an average of 40 percent in both cases. However, registers are not a critical resource and not worth sharing in most of the earlier end-to-end basic results. An exception is that for large programs there are pragmatic reasons for sharing — sharing eases the synthesis burden.

Faster Synthesis Reduces Performance

Logic synthesis, in the CAD phase, performs the lowest level specializations in DeepC. Therefore, if synthesis effort is reduced, there should be less specialization and lower performance. This study reduces Synopsys's synthesis effort from DeepC's default medium to low with the *-fnosynopt* flag. These results should be compared to the "-O5" column. The major effect of reducing Synopsys optimization effort is a reduction in clock speed for some benchmarks and a lower chance of meeting the target clock speed. However, some designs were faster with low optimization, leading to the conclusion that either the synthesis tools are unpredictable or perhaps the difference between low and medium is not significant. A longer synthesis time (1 week) on high effort would permit better optimization, however I did not collect this data given time limitations. The difficulties of specializing the entire design at this lowest level highlight the importance of specializing architectural features at levels where they are recognized rather than trying to uncover all optimizations during logic synthesis.

6.5 Advanced Results

This section considers specialization of advanced architectural mechanisms as described in Chapter 4. Advanced specialization can increase performance beyond what is reported in the previous basic results section. Most of the programs considered in basic results only occupy a tiny corner of silicon, leaving plenty of silicon area to leverage for higher performance. Although the parallelism needed to keep the silicon area of a large chip busy is not present in all applications, a handful of the Deep Benchmarks can be parallelized (`intmatmul`, `jacobi`, `mpegcorr`, `life-v2`, `pmatch`, and `sor`). These benchmarks can be parallelized with the loop-unrolling techniques in the Rawcc-derived frontend.

The general approach in this section is to vary DeepC's *-nprocs* flag. This flag determines how many spatial tiles are used to represent the computation. Recall that each tile contains its own finite state machine for computation. Experiments with router specialization use the $-froute$ flag. Recall that with routers a second state machine is added to each tile for communication. These approaches can be compared to a monolithic approach, with a unified state machine. The monolithic approach is invoked with the *-fnopart* flag, in which loops are unrolled and memories specialized, but tiles are not created. Note that for post-synthesis results, experimentation time is saved by only compiling a sample tile (tile zero) rather than synthesizing an entire parallel machine.

This section continues by demonstrating advanced results for these benchmarks with a focus on efficient scalability. It compares performance per area, performance per power, and performance per energy as parallelism is increased. As in the basic results, first *Mode 0* performance is treated, and then *Mode I*.

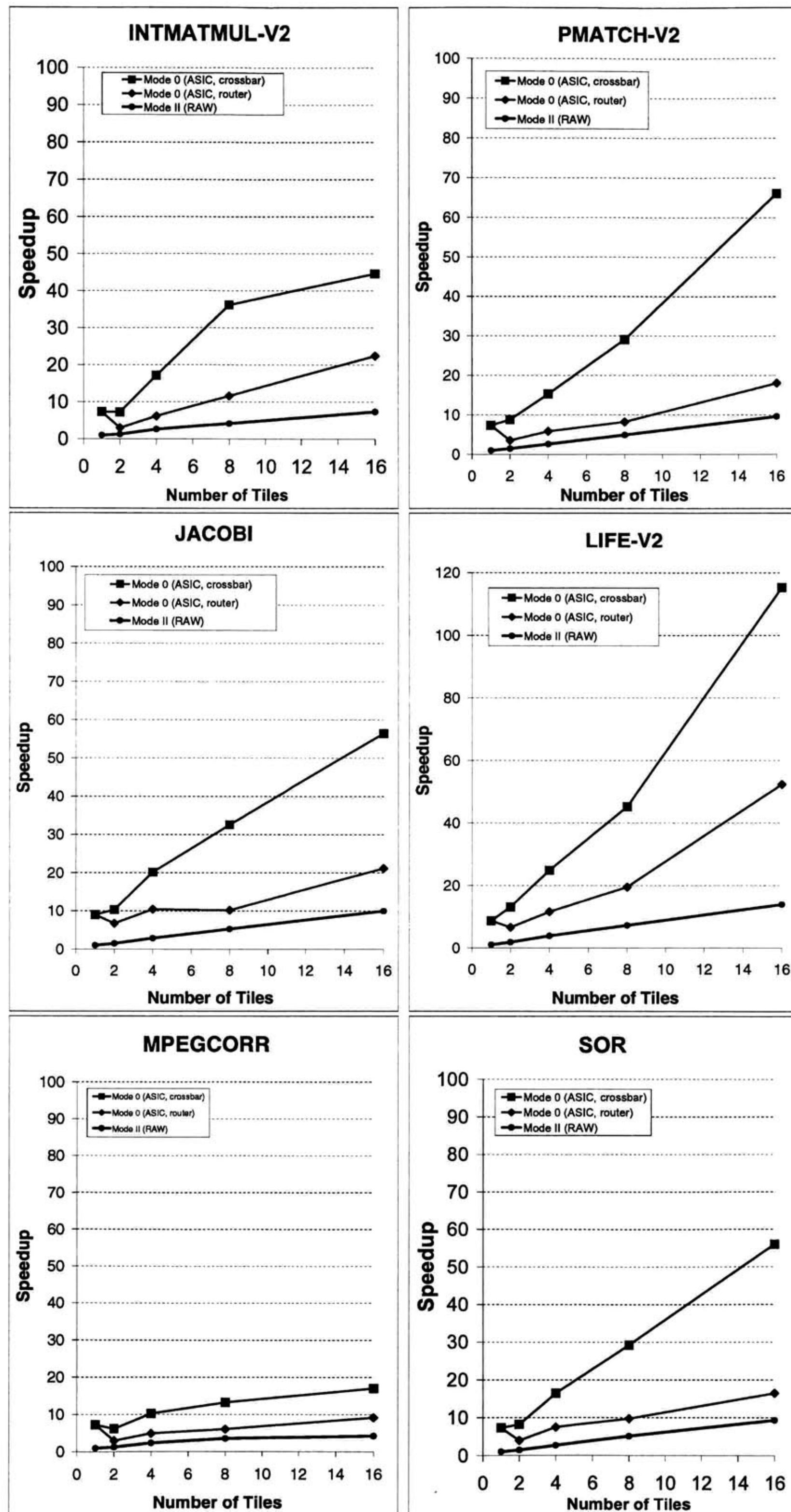


Figure 6-12: *Mode 0* performance normalized to one Raw tile.

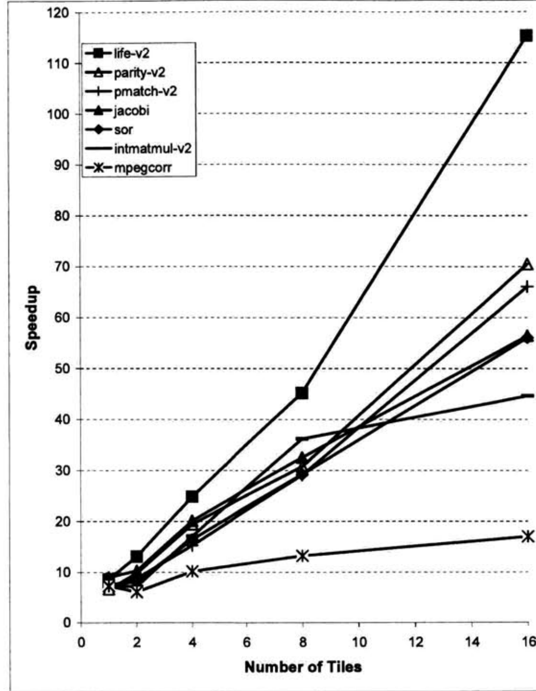


Figure 6-13: *Mode 0* speedup

Comparison is of a crossbar at 300 MHZ in IBM SA-27E process versus a single MIPS core. Table D.1 contains additional router data.

6.5.1 *Mode 0* Advanced Results

Detailed graphs in Figure 6-12 compare both the crossbar and router case across the advanced benchmarks. The same graph includes the Rawcc speedup curve. The initial specialization speedup gives the *Mode 0* cases a boost that Rawcc never overcomes. However, the router case looses much of this gain to poor parallelization, a problem that Section 6.6 will address. Note that equal tile count does not correspond to equal silicon area. Figure 6-13 displays the absolute speedup of the crossbar cases on the same graph for comparison.

Appendix Table D.2 and Table D.3 contain area and power results similar to the results in Figure 6-3. However, without running the ASIC place and route tools or estimating the size and power of the respective embedded RAMs, this data was difficult to decipher. The following section reports this class of data for the *Mode I* results, which are of more relevance to this dissertation.

6.5.2 *Mode I* Advanced Results

Once again, the *Mode 0* results have shown dramatic gains from specialization, this time in the context of large parallel structures. This section now turns to *Mode I*, to see if gate reconfigurable architectures can maintain their specialization gains alongside the parallelization gains.

The discussion in this section focuses on performance per area, performance per power, and performance per energy. The appendix contains the corresponding data: Table D.4 contains the clock frequencies achieved, Table D.5 and Table D.6 contain the number of LUTs

and registers, Table D.7 show average switching activity (as described in Section 5.4.4), and Table D.8 the estimated power.

Performance per Area

Figure 6-14 shows performance as a function of LUT area. Except for `intmatmul-v2`, which consumes exorbitant area because of excessive unrolling (producing logic ten times current FPGA sizes), each design will easily fit on an FPGA sold at the time of this publication. LUT area increases faster than performance in many benchmarks because of high parallelization overheads. An exception is `life-v2`, which exhibits super-linear speedup in area for the crossbar case.

Performance per Power

Figure 6-15 graphs performance as a function of power. I was not able to gather switching activity for the Xilinx mapping, so I use the register switching activity gathered during ASIC synthesis and applied it to the FPGA netlists, taking the average of the register activity as a rough approximation of the overall activity. Appendix Table D.7 contains the estimated switching activity. For the crossbar cases, the performance per power is close to linear. The router case does not scale — Section 6.6 will revisit this issue. Note that the area for the router case does not include the static routers themselves. In practice, this area is about 10–20 percent of the size of the main tile.

Performance per Energy

Because specialization both reduces power *and* increases performance, performance per energy graphs are steep, as shown in Figure 6-16. First, consider the crossbar cases. The first data point to the left is for the one tile case and the highest data point is for the 16-tile case. If both performance and hardware area scaled linearly with tile count, these curves should be vertical. However, the overall scalability is low in some cases, so these curves are not as steep as expected. This result is caused by inefficiencies that generate idle hardware that does not penalize total energy. The large cases, with a slow clock speed, magnify this problem. Some curves bend backwards as the parallelization overhead becomes amortized into total energy.

Next, consider the router cases. In several benchmarks (`intmatmul`, `mpegcorr`, and `pmatch`) a large parallelization startup overhead precludes any performance gain. Furthermore, energy consumption increases dramatically. Although parallelism is not effective in these cases, do not dismiss them — there are improvements, discussed in Section 6.6, that can overcome these problems. Because the crossbar case is not physically scalable and the router case can be improved, more accurate results will lie somewhere in the region between these two cases.

6.6 Advanced Sensitivity Analysis

Performance is sensitive to the advanced architectural mechanism specialized. The following sections discuss sensitivities to both memory and communication structures. These results support my thesis by revealing limitations in both the parallelization strategy and in the

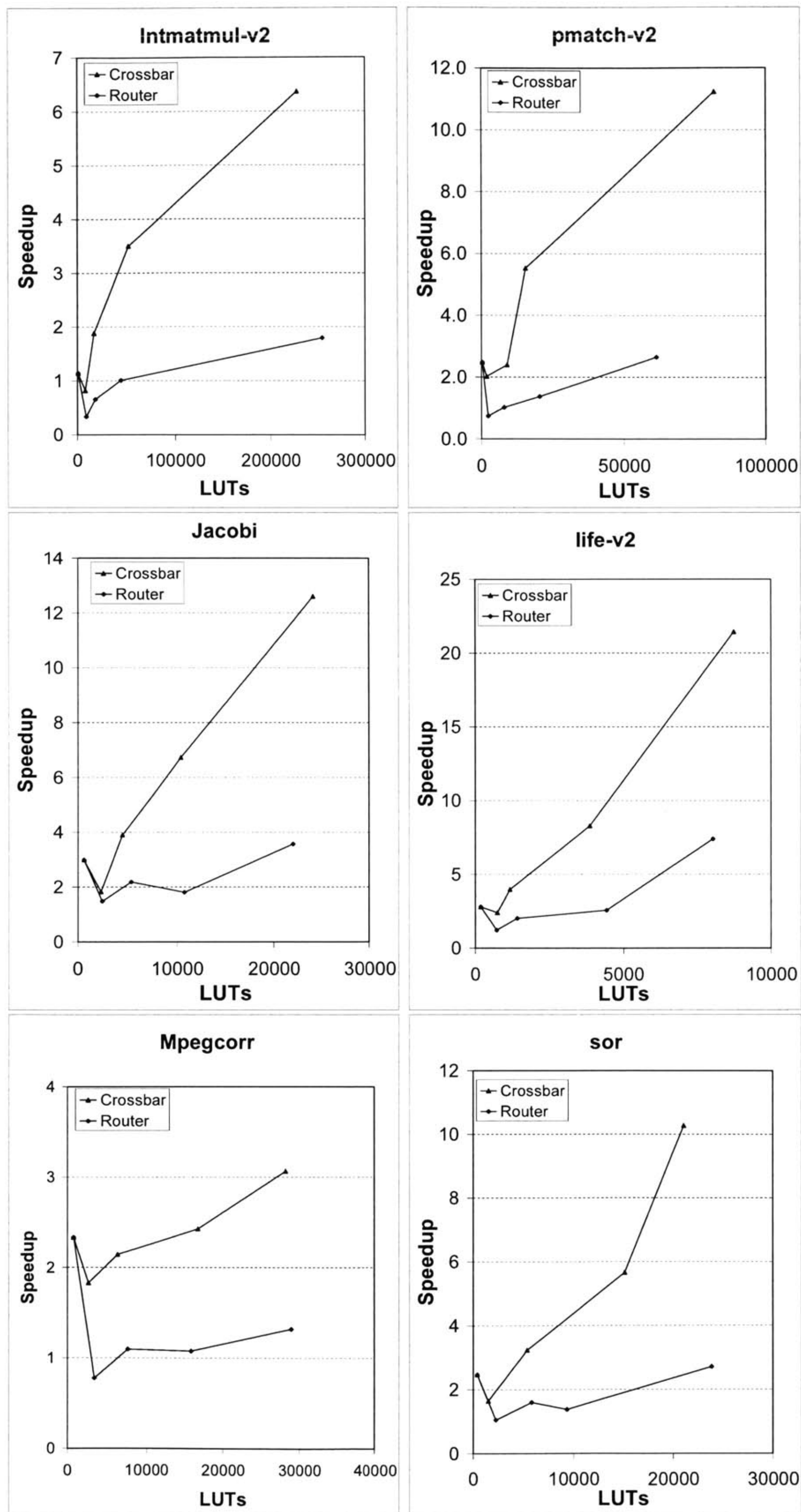


Figure 6-14: Performance as a function of LUT area.

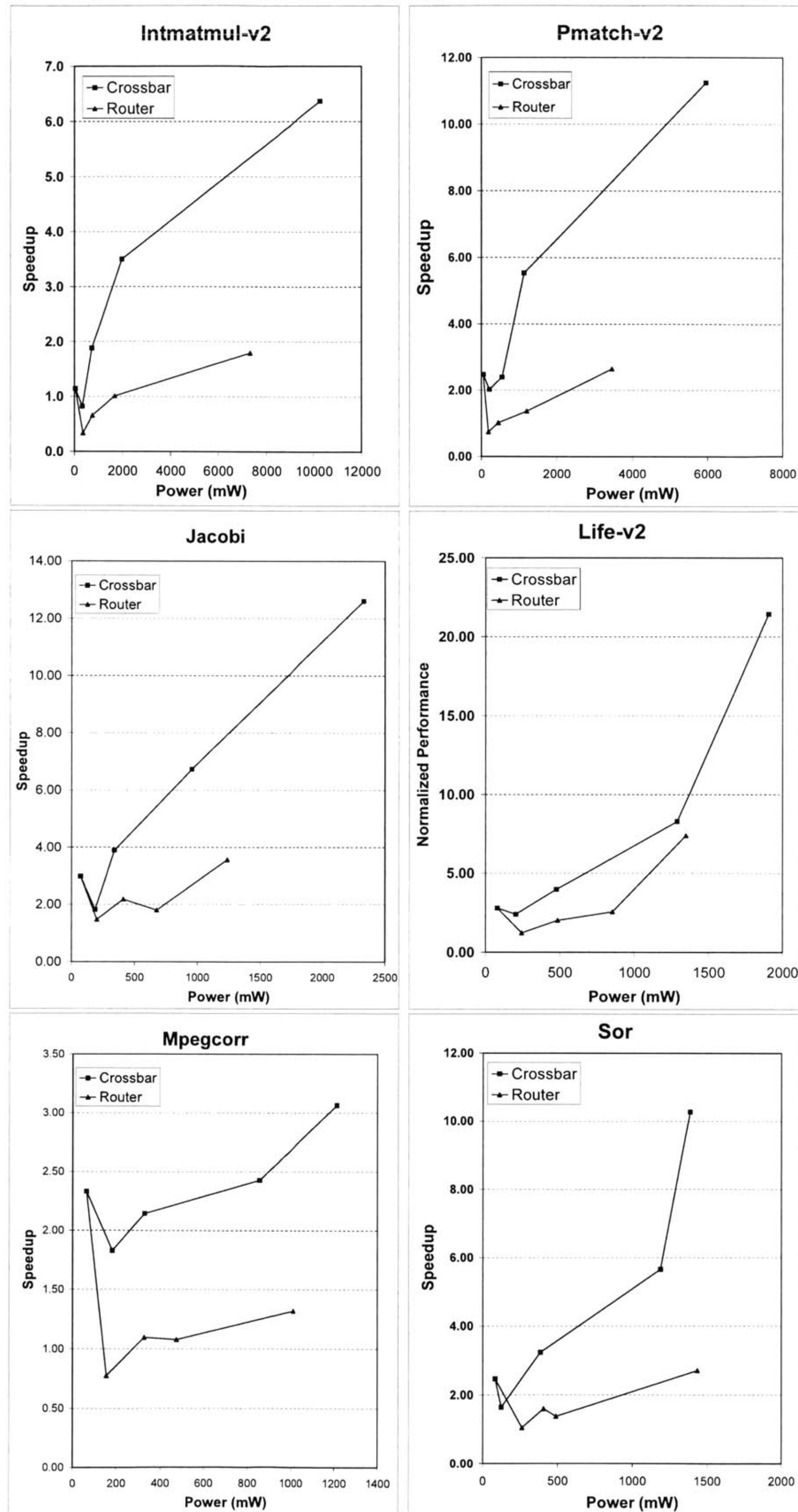


Figure 6-15: Performance as a function of power

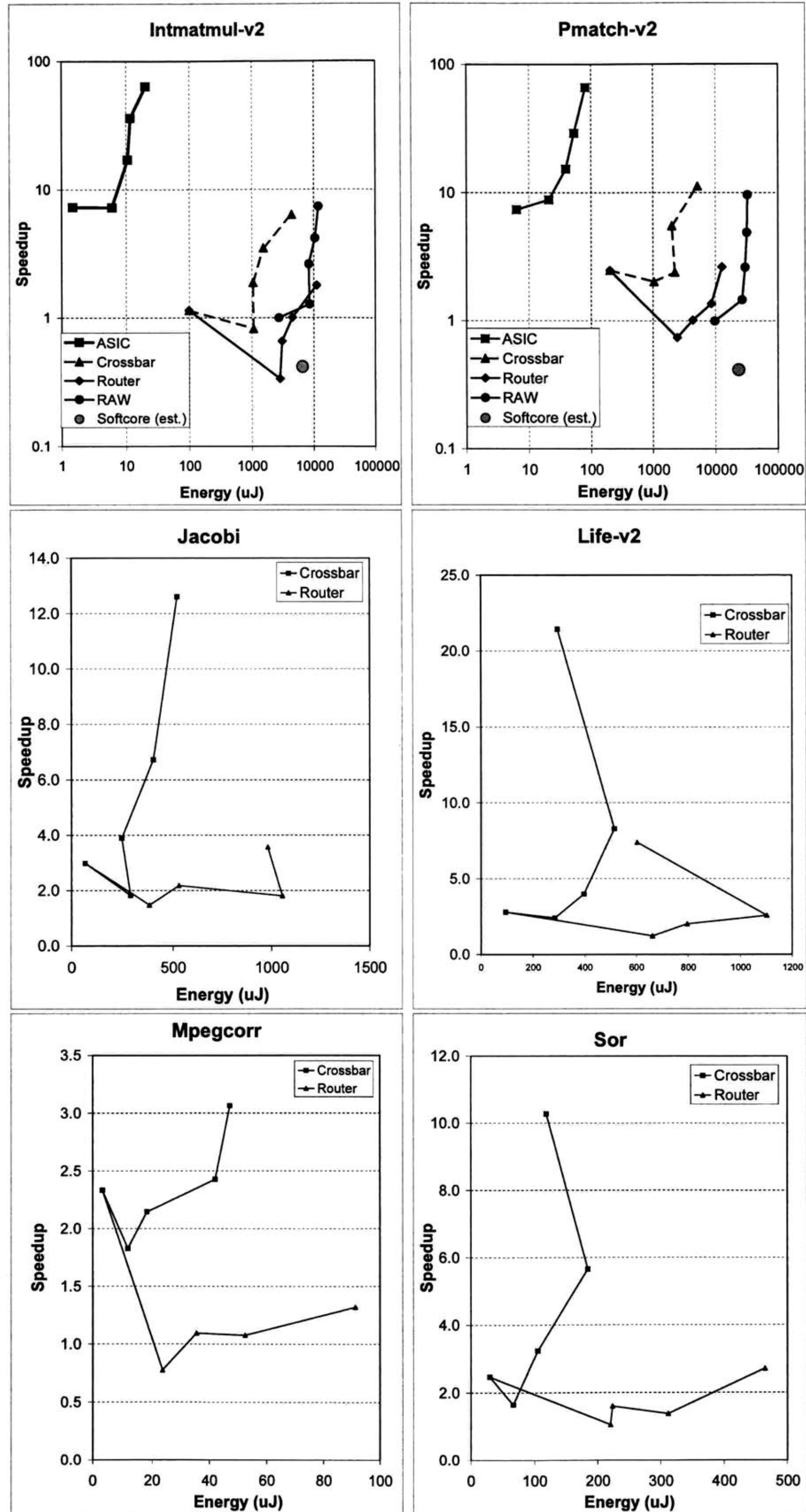


Figure 6-16: Performance as a function of total energy

communication structures that are specialized. These limitations, rather than the specialization technique, reduce the effectiveness of the advanced cases in the previous section. Furthermore, there is evidence that the limitations can be overcome.

6.6.1 Memory Sensitivity

The DeepC system did not generate perfect speedup when parallelizing applications that should be easy to parallelize. To understand why, this section studies the sensitivity of DeepC to memory parallelization. There are two main results:

- unrolling increases the total number of states and the number of states is correlated with a decrease in clock speed,
- low order interleaving produces square root speedup that can be overcome with known blocking transformations.

Another study, not further discussed here, is sensitivity to the number of ports per memory (DeepC flag *-mports*) — none of the parallel cases were sensitive to this parameter.

Clock Speed Slows with Increasing States

One problem with unrolling is that it increases the total number of states. As the frontend unrolls loops during parallelization, these extra states reduce clock speed as RTL synthesis adds extra multiplexers. This effect can be demonstrated by correlating the number of states with the clock speed (Figure 6-17). This clock slowdown is one reason the multi-tile cases do not meet the target clock of 100 MHz. Because other factors may limit clock speed, the correlation is most pronounced near the upper end of the clock speed achieved for a given number of states. The empirical trend line drawn in this area of the graph shows that the best clock periods are reduced by a factor of $3\times$ for every $10\times$ state space explosion. Other factors limit data points not lying near this trend line. For example, studying the output netlists reveals that large crossbar multiplexers limit the crossbar data points, not the finite state machine.

Besides minimizing unrolling, for example by software pipelining, improvements to RTL synthesis could also help alleviate this problem. It would help to encode states better and to predict the next state so that state calculation can be pipelined. This improvement would lead to specialization of the architectural mechanism of branch prediction. Alternately, the state machine can be decomposed into smaller state machines. Although the interaction of state machines synthesis and loop unrolling is a complex problem, there are many potential solutions.

Low-Order Interleaving Creates Long-Distance Messages

This section shows that the low-order interleaving strategy of the frontend parallelizer significantly inhibits performance. If this strategy works well for a sixteen tile Raw machine, should it also work for a sixteen tile specialized machine? The answer is no. The fact that a specialized machine takes fewer cycles to evaluation inner loops magnifies the overheads of transforms that do not preserve locality. The MAPS parallelization strategy, because it relies on low-order interleaving, produces long-distance messages that must cross the chip during the evaluation of a basic block. For example, consider the long message in Figure 6-18, a common occurrence in affine expressions such as $A[i] = A[i+1]$. Between certain

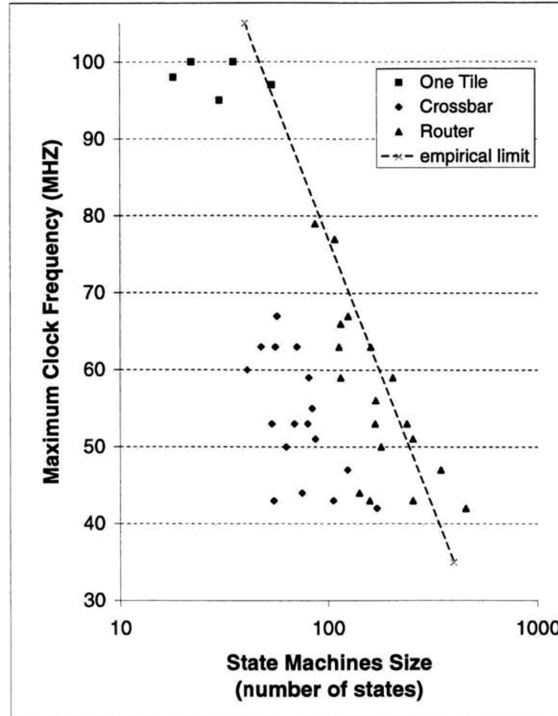


Figure 6-17: Clock frequency versus number of states

Data is in Table D.9. Clock period is reduced by roughly 3× for every 10× increase in state space size, as shown by the line market empirical limit. The trend line is a rough empirical observation of a limit imposed by the size of the state machine. Note that it only applies to the crossbar cases — the other cases were not limited.

"local" assignments, messages will need to cross the entire chip! Asymptotically, this limits speedup to roughly the number of tiles divided by the diameter of the network — a familiar square-root speedup. Instead of solving the long wire problem, the long wires have been replaced with long-distance messages with the exact same scalability problem[1].

Fortunately, there exist parallelization approaches for certain classes of loops that *can* preserve locality. A source code transformation can be used to "trick" the parallelizer into blocking rather than low-order interleaving. Figure 6-19 gives an example of this trick applied to the inner loop of `jacobi`. In the figure, $SIZEX$ and $SIZEY$ are the original sizes of the X and Y dimensions. Two new parameters, NX and NY, encode the blocking factors in the respective dimensions. Notice that the data arrays a and b have an added dimension in the transformed code. The indices, $i1$ and $j1$, are also dimensioned to enable local indices on each tile. Boundary condition code is needed to handle the edges of the block.

Figure 6-20 presents the results from blocking a sixteen tile `jacobi`. Although other inhibitors, such as transmittal of the branch condition, add overhead, blocking improves the total performance substantially. Thus, more traditional frontend parallelization approaches would interact better with specialization.

[1]This result also imply that the early Raw compilation strategies will not scale to machines with faster processor versus network speed or large diameter networks.

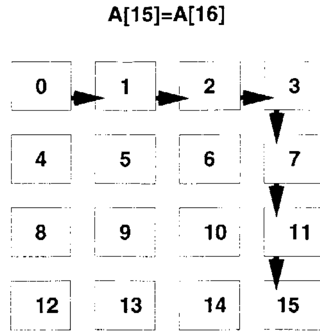


Figure 6-18: Long-distance messages

Messages are created by the expression $A[15] = A[16]$ when A is low-order interleaved across sixteen tiles. Even though elements fifteen and sixteen are adjacent, low-order leaving places them as far apart as possible.

```
BEFORE:

for (i=1;i<SIZEX-1;i++)
 for (j=1;j<SIZEY-1;j++)
   a[i][j]=
    (b[i-1][j
    +b[i+1][j]
    +b[i][j-1]
    +b[i][j+1])>>2;
```

```
AFTER:

for (i=1;i<SIZEX/NX-1;i++) {
 for (n=0;n<N;n++) {
  j1[n]=1;
  i1[n]=i;
 }
 for (j=1;j<SIZEY/NY-1;j++) {
  for (n=0;n<N;n++) {
   a[i1[n]][j1[n]][n]=
    (b[i1[n]-1][j1[n]][n]
    +b[i1[n]+1][j1[n]][n]
    +b[i1[n]][j1[n]-1][n]
    +b[i1[n]][j1[n]+1][n])>>2;
   j1[n]++;
  }
 }
}

/* copy boundary conditions */
for (i=1;i<SIZEX/NX-1;i++)
 for (n=0;n<N;n++)
  if(n % NX)
   a[i][0][n]=
    a[i][SIZEY/NY-2][n-1];
 for (n=0;n<N;n++)
  if ((n+1) % NX)
   a[i][SIZEY/NY-1][n]=
    a[i][1][n+1];
 for (n=0;n<N;n++)
  if(n>=NX)
   a[0][i][n]=
    a[SIZEX/NX-2][i][n-NX];
 for (n=0;n<N;n++)
  if(n<N-NX)
   a[SIZEX/NX-1][i][n]=
    a[1][i][n+NX];
```

Figure 6-19: Jacobi inner loop before and after blocking transformation

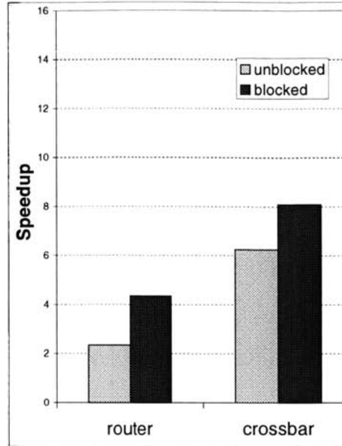


Figure 6-20: Improvements after blocking for sixteen-tile Jacobi

The crossbar case is reduced from 15K cycles to 12K, improving speedup from 6.3× to 8.1×. The router case is reduced from 40K cycles to 22K, improving speedup from 2.4× to 4.3×.

6.6.2 Communication Sensitivity

Efficient parallel machine synthesis requires specialization of communication mechanisms. The advanced results demonstrated great performance when specializing a partitioned design with crossbar-like communication structures. However, the results were less impressive for router specialization, especially when compared to the nice scalability of the router-based *Mode II* systems (Raw). Section 6.6.1 blames the increase in state size and subsequent decrease in clock speed for part of this problem. For example, at sixteen tiles the clock speed of the router cases average twenty percent slower than the crossbar cases, even though the crossbar communication structures were more complex because of large multiplexers. Yet, crossbars do not scale physically, and ultimately only routers can be used for large silicon areas.

This section continues with sensitivity studies that investigate the communication network. First, using VPR I compare the sensitivity of FPGA routing track count to communication structure. Beyond four tiles of parallelism, track resource needs become prohibitive without partitioning. In addition, the router case needs fewer tracks than the crossbar case. Second, increasing the number of communication ports per tile yields small improvements for some benchmarks, demonstrating bandwidth limitations due to network congestion. This section concludes by investigating the sensitivity of absolute performance results to the speed of the static network. By *overclocking* the network, the number of states can be reduced and the component of the critical path that passes through the network can be shortened, partly closing the gap between the router performance and the idealistic crossbar performance.

Routers Reduce Minimum Track Width

Minimum track width is a measure of the routability of a design when mapped to a target FPGA family. Although heuristics, such a leaving unused LUTs, allow use of a lower track width FPGA, in practice FPGAs are designed with more than the minimum number of

Benchmark	One Tile	Two Tiles nopart / xbar / router	Four Tiles nopart / xbar /router	Eight Tiles nopart / xbar /router
jacobi	39	63 / 50 / 51	79 / 51 / 53	105 / 59 / 59
mpegcorr	34	56 / 41 / 47	66 / 49 / 52	91 / 51 / 51
life-v2	41	57 / 57 / 59	92 / 58 / 63	* / 65 / 63
pmatch	29	55 / 55 / 55	83 / 73 / 71	* / 89 / 83
sor	33	54 / 51 / 53	* / 68 / 62	* / 67 / 56

Table 6.7: Minimum track width

These results are generated with the VPR tool performing a binary search on routing tracks. * denotes a case that failed to place and route in 1GB of memory.

tracks for the class of designs they support. Recall that in the basic results section, Figure 6-8 showed the benchmarks requiring between 25 and 66 routing tracks. This section continues in this vein and determines the sensitivity of tracks to the advanced communication specializations. It compares three communication structures. In the *nopart* case, memory structures are parallelized; yet no tiles are formed. The *crossbar* and *router* case should be familiar. The crossbar case may be optimistic because it does not consider the constraints of routing the global wires between tiles. The router case is realistic.

The findings in Table 6.7 show that the unpartitioned case begins to fail at four tiles and fails for most of the benchmarks at eight tiles. These cases fail because VPR could not route them in 1GB of memory, and likely they need more than 100 tracks. In addition, the crossbar case begins to underperform the router at eight tiles. This underperformance will only worsen with more tiles.

Partitioning is thus essential beyond a small amount of internal parallelism. Furthermore, a limited number of crossbar-like connections may be desirable to around eight tiles, but the router case is better at eight tiles and beyond. Given the importance of the router case, the next two sections investigate sensitivity to improvement in specialized communication mechanism.

Multiple Communication Ports Increase Performance

The performance of routers can be improved by adding more communication ports between each router and the main tile logic. Figure 6-21 shows the results for a select set of benchmarks. These results are generated with DeepC's *-cports 2* and *-cports 4* options. All cases show at least a small improvement, demonstrating that there are at least small routing bandwidth bottlenecks. In `life-v2` and especially `mpegcorr`, large gains are demonstrated with multiple ports, showing that serious bottlenecks exist for these cases. For comparison, Raw has two static networks, and each static network has two input ports from the router and one output port to the router. Therefore, specialization of architectures with multiple communication ports is a good idea. A benefit of the *Mode I* approach is that extra ports can be eliminated when they are not needed.

Overclocking the Static Network Increases Performance

Another technique for improving the performance of the router communication mechanism is to run the routing network at a faster clock speed than the main tile logic. There are four practical ways to running a faster network:

Benchmark	cport 1	cport 2	cport 4
intmatmul	54464	54016	54016
jacobi	40329	39073	39073
mpegcorr	3970	3186	3122
life-v2	19279	16995	16995
pmatch	289424	286902	286902
sor	16194	16186	16186

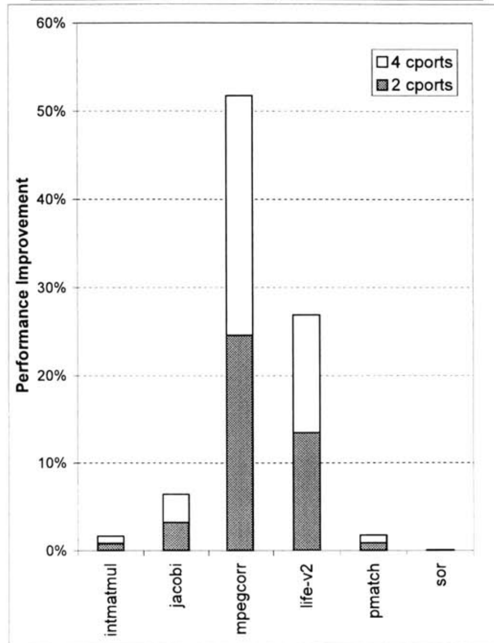


Figure 6-21: Performance improvements after increasing communication ports

Ports per tile is increased from one to two and four. The table reports the clock cycles for sixteen tile router cases; the graph is normalized to the single cport case.

- Run the router faster if it is less complex than the tile computation.
- Underclock the main tile to conserve power and reduce states.
- Handcraft fast softcore routers inside the FPGA.
- Embed fast hardcore routers in FPGAs.

Because FPGA clock speeds are slower and specialized tiles are smaller when compared to a tiled-machine like Raw, it should be possible to cross several tiles in a fast clock cycle before wire delay becomes a problem.

To see whether these approaches would help, DeepC has a feature that allows the routing network to run faster than the tile network, set with the *-netclock* flag. With this flag, the scheduler and state machine generators can allocate multiple network clocks to each main tile clock. For synchronization purposes, the network clock must be a multiple of the main tile clock (in general any rational clocking scheme will work). The main tile can only inject/receive messages to/from the network on a main tile clock boundary. Therefore, the main tile can only transfer data to the network at the rate of the main clock, but once in the network data can traverse the chip at the faster network clock speed.

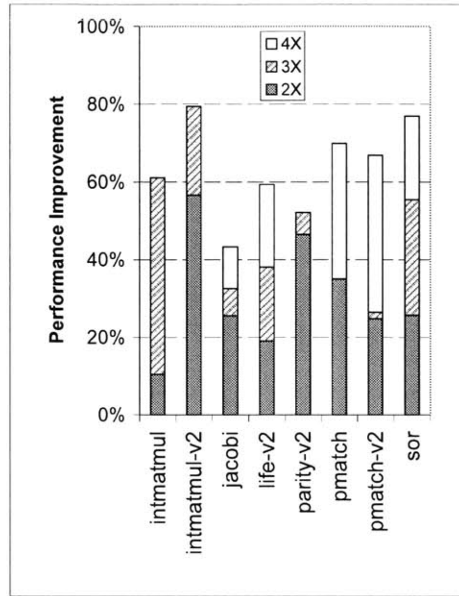


Figure 6-22: Improvements from overclocking the routing network

Figure 6-22 shows the resulting improvements for the 16-tile router case. A 2× faster clock, easily doable, yields substantial improvements for all cases (30 percent on average). A 3× and 4× faster clock also continues to yield improvements, to nearly 80 percent for some cases. Figure 6-23 shows the more detailed speedup graphs, comparing these results to both the crossbar and router results from earlier in this chapter. Note that it does not help to overclock more than the network diameter, so there are no data points for some small tile counts. Even if overclocking is unlimited, synchronization and pipeline startup overheads prohibit achieving full crossbar performance.

As the system scales to a much larger number of tiles, crossbar-like performance cannot be maintained — ultimately wire delays will dominate. Two areas of future research are needed to address these issues. First, circuit-level designers should develop the fastest possible routers and embed them into FPGA architectures. Because router speed is a large determinant of system speed, the architecture with the fastest routers may be the best architecture. Second, it is possible to work around the problem in the compiler by specializing more advanced multithreaded architectures and/or software pipelining the inner loops.

To illustrate this point, `jacobi` in Figure 6-23 contains a result labeled "router6x-u4" in which the router is overclocked substantially and the inner loop is unrolled four times. The extra unrolling permits more overlap of computation with communication, and thus comes close to achieving the ideal crossbar level of performance.

6.7 Grand Finale: Comparing All Modes

As a grand finale, this section compares the performance per area and performance per energy of all the modes encountered so far. It also includes a rough *Mode III* (softcore, or processor-on-GRA) data point based on recently released information from Xilinx.

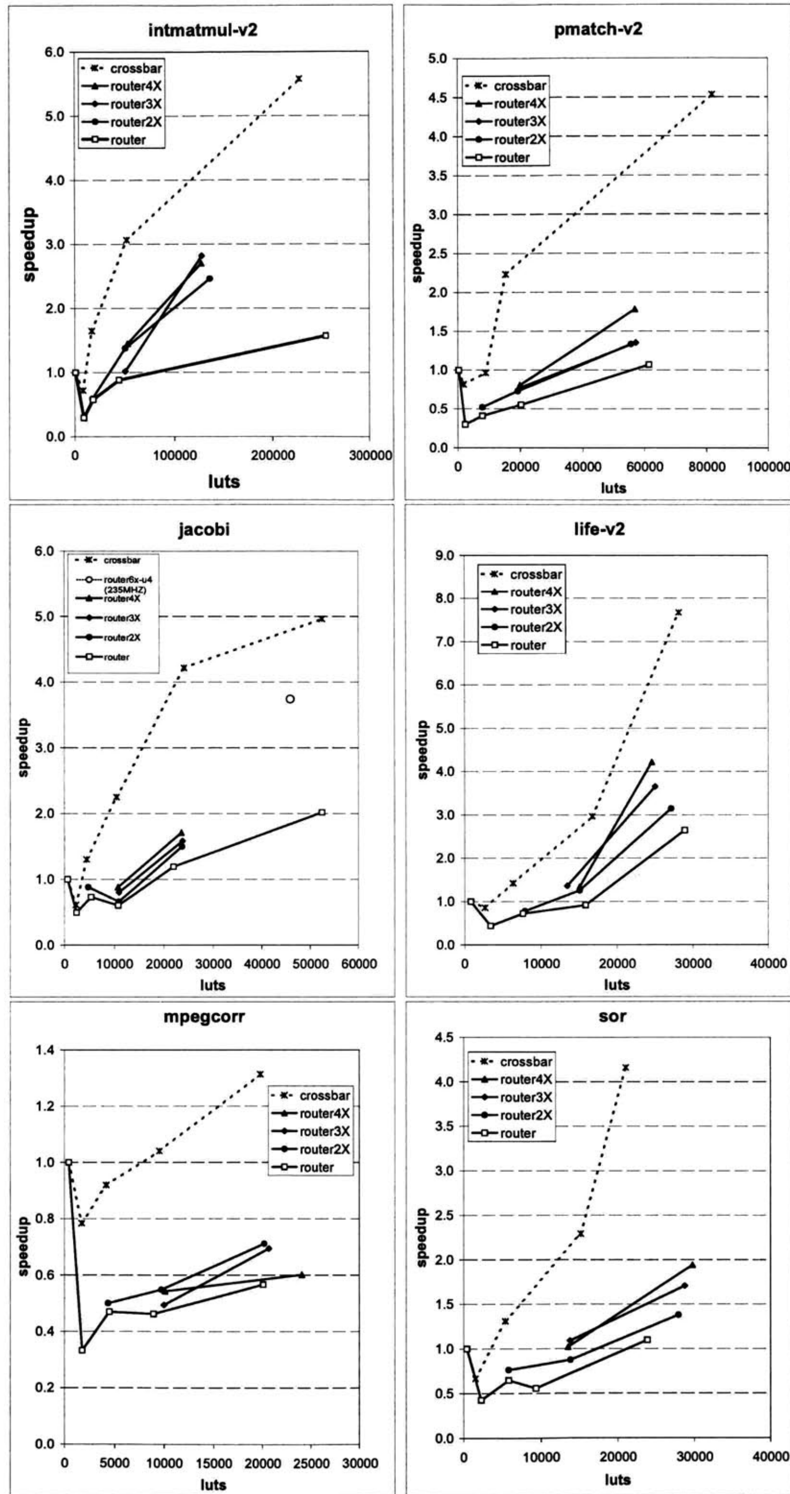


Figure 6-23: Sensitivity to variations in network speed

As the network speed is "overclocked" by 2×, 3×, and 4×, performance approaches crossbar-like performance. A point is added for `jacobi`, where it is unroll four extra times and the network is overclocked by 6× to get crossbar-like performance.

Parameter	Estimate
ASIC cell area	3.762 micron
LUT area	690K λ 2
λ	0.09 micron
Raw speed	300 MHz
MIPS core size	1500 LUT (equivalent)
MIPS core power	1 Watt @ 300MHZ
Raw tile size	3000 LUT (equivalent)
Raw power	2 Watts / tile
Softcore speed	125 MHz
Softcore speed	125 MHz
Softcore size	2000 LUT
Softcore tile size	4000 LUT (rough)
Softcore performance	75 MIPS

Table 6.8: Assumptions and estimates for important parameters

There are a few caveats to these results. First, Raw is a research architecture and the clock range, power, and performance are estimates. Second, the ASIC and FPGA case do not include RAM power. The ASIC case would be penalized more by inclusion of RAM and I/O power as the computation logic consumes much less energy than in the other cases. Furthermore, for the crossbar case, scalability may breakdown as soon as 8 or 16-tiles, so the projected performance may be artificially high. Tools to estimate power are immature and not entirely trustworthy. Finally, the softcore estimates are based on preliminary data from Xilinx, and they assume a MIPS/MHz ratio similar to Raw. Given these caveats, these results support my thesis by comparing all four evaluation modes introduced in Chapter 2 and by demonstrating an ability to efficiently specialize advanced architectural mechanisms.

Grand Comparison of Performance Per Area

Figure 6-24 compares performance per area for the Deep Benchmark Suite. Speedup is normalized to a base MIPS core and area is calculated in LUTs. Assumptions and estimates are in Table 6.8. The area for the ASIC and Raw case is translated into equivalent LUTs using the assumptions in Table 6.8. The softcore case is from press releases about Xilinx's Microblaze softcore (Xilinx claims 900 LUTs; these result assume 2000 LUTs to include routing area and for good measure). Tile sizes are twice the size of processor cores to account for router and multiprocessing overhead. Because calculation of memory area is difficult, these results do not include the RAM area.

The difference in performance between modes can be attributed to the number of interpretation levels. The *Mode 0* case (ASIC, no interpretation) is significantly smaller and higher performance than the other cases (*Mode I*,*Mode II*,*Mode III*). In general the crossbar *Mode I* case outperforms Raw (*Mode II*), although the router case underperforms Raw given the same area. The router 4× case (see Section 6.6.2) has similar performance to Raw as the clock speed of the networks are roughly the same! These results are so similar because *Mode I* and *Mode II* are both singly interpreted. Finally, there is the doubly interpreted softcore case — the worst performer because it is *Mode III*. Without the architectural specialization techniques in this thesis, *Mode I* would not be possible and all evaluation of high-level programs on FPGA would be in *Mode III*.

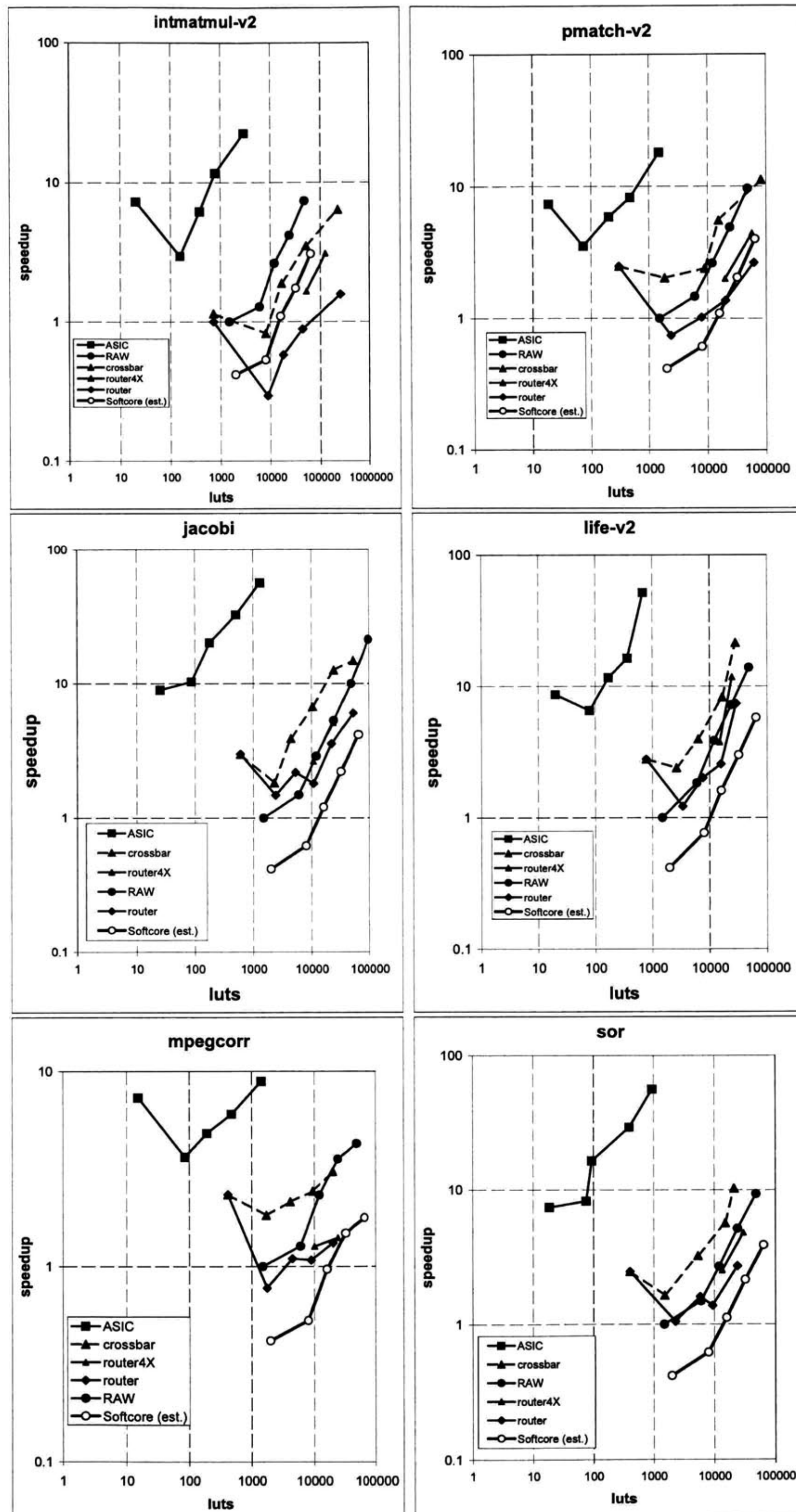


Figure 6-24: Grand comparison of performance as a function of non-memory area

Grand Comparison of Performance Per Energy

This section concludes with a grand comparison of performance as a function of energy for *Modes 0–III* (Figure 6-25). The absolute energy difference between modes is dramatic. The graphs span four orders of magnitude in energy and two orders of magnitude in performance. Except for the softcore case, the data points are generated by varying the number of tiles from one to sixteen. The leftmost data point is for the one tile case.

The assumptions for these results are in Table 6.8. Although the ASIC and Raw cases execute at 300 MHz, the crossbar and router FPGA cases are much slower (23–108 MHz). The ASIC and FPGA cases are optimistic by a factor as high as 2× in comparison to Raw as memory and pin power consumption are included for Raw only. Many low power optimizations can be applied to all cases; improving low-power architectural mechanisms is an active area of research.

As total energy input to the system is increased, it is desirable to increase performance. With perfect parallelism, increases in performance should not increase energy needs. Only the work resulting from parallelization overheads cost extra energy. For these benchmarks, the Raw results and the crossbar results scale well. In comparison, the router cases scale poorly. However, Section 6.6 has discussed the problem associated with these cases and proposed solutions. These results do not include any of those solutions. Therefore, better *Mode I* results will fall between the crossbar and router data.

It may be tempting to emphasize the differences between the FPGA and processor cases, especially when the FPGA case is one or more orders of magnitude better for some benchmarks. However, the relevant difference in support of my thesis is between the softcore processor and the left most FPGA data point (one tile). This difference best captures the benefits of architectural specialization. This difference shows that architectural specialization increases the energy-delay of the FPGA-based solution by three orders of magnitude — from much worse than traditional solutions to much better.

6.8 Summary

This chapter provides results to demonstrate my thesis. It shows that C programs can be efficiently compiled to gates. The absolute benefits of specializing architectural mechanisms were demonstrated by comparing *Mode 0* to *Mode II*. The benefits are large in metrics including area, delay, and power. This chapter measured the contributions of individual specialization techniques to this overall gain. Bitwidth reduction was the best specialization with no adverse side effects.

This chapter has demonstrated the relative performance of *Mode I* when compared to *Mode II* — FPGAs versus processors. For the benchmarks chosen, *Mode I* beat *Mode II* in area, delay and power metrics, although the thesis goal is to show that the two modes are comparable in contrast to the much better *Mode 0* and the much worse *Mode III*. An exception to the comparability claim is an impressive energy-delay boost for *Mode I* of one to two orders of magnitude. Finally, the advanced architectural mechanisms of Chapter 4 were successfully specialized, although unrolling presented some difficulties. A 3× slowdown for every 10× increase in state size was measured. This problem is exasperated when routers are synthesized, with low-order interleaving creating a long message problem. This chapter offered several solutions to overcome memory and communication bottlenecks: better scalability, blocking transforms, better network bandwidth with multiple ports, and better network speed with overclocking.

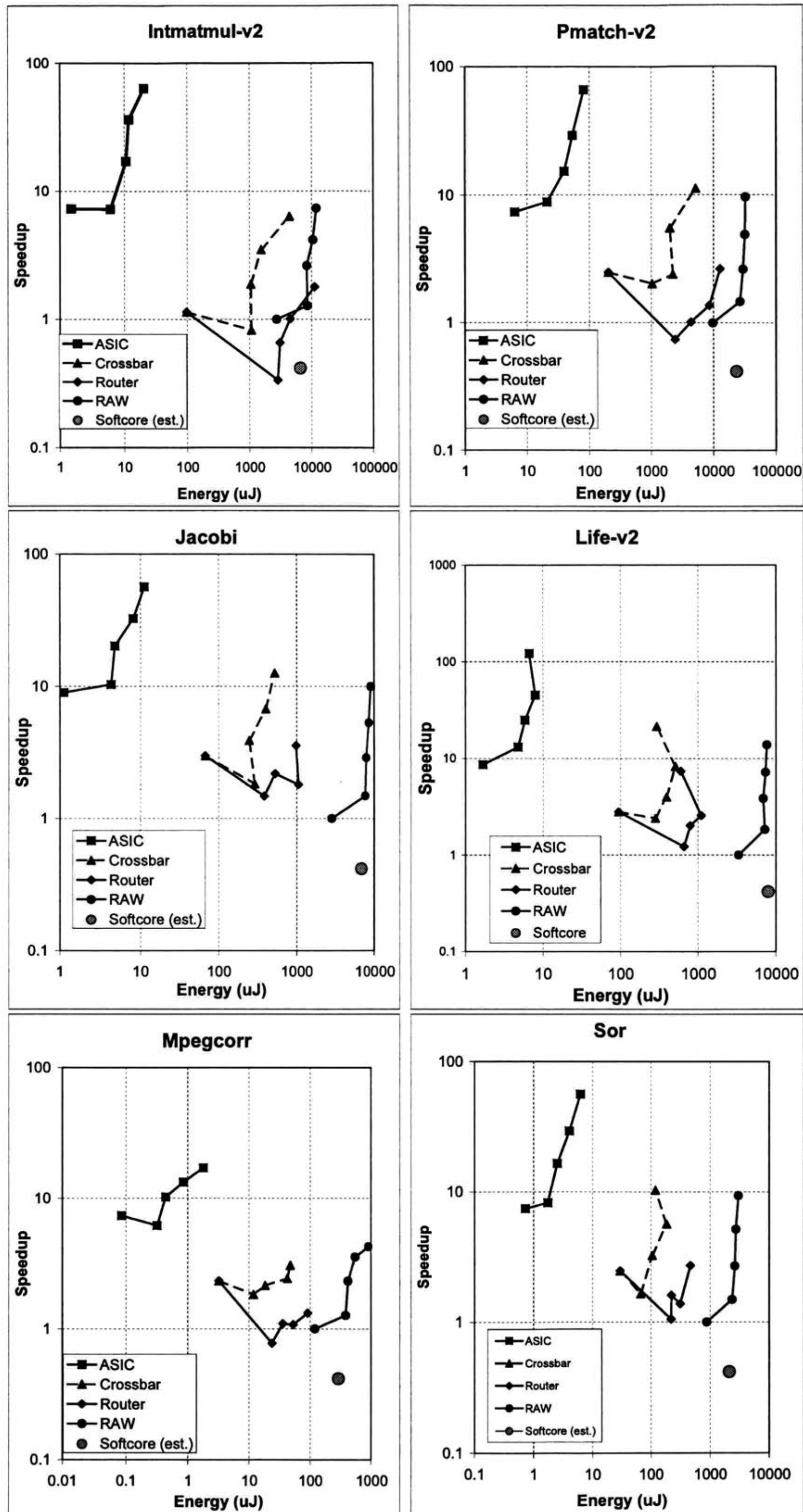


Figure 6-25: Grand comparison of performance as a function of non-memory energy

Chapter 7

Conclusions

Here I summarize the dissertation, restate my thesis, and make several extrapolations for future work. To close, I comment on how DeepC can advance the open hardware movement and I challenge the compiler community to compile to gates, if they dare.

7.1 Wrap Up

This dissertation has shown that specialization of traditional architectural mechanisms with respect to an input program is the key to efficient compilation of high-level programs to gate-reconfigurable architectures. Chapter 2, with an analysis of evaluation modes, has demonstrated that such specialization enables interpretation at the logic level to compete with instruction-level interpretation. Chapter 3 and Chapter 4 have shown how to specialize individual mechanisms. These chapters cover specialization of basic gates and wires through specialization of mechanisms found in advanced distributed architectures. With this understanding of how to specialize, Chapter 5 has described a working implementation of DeepC, a compilation system that embodies these concepts. Chapter 6 uses DeepC to map a new suite of benchmarks to modern FPGAs. For comparison, the same benchmarks are targeted to an uninterpreted substrate (a custom ASIC), to traditional processors, and to the Raw processor, an architecture that more traditionally embodies the advanced mechanisms of Chapter 4. This comparison demonstrates that compiling high-level programs to GRAs can improve all aspects of performance — area, latency, power, energy, and energy-delay — when compared to traditional approaches. These findings support my thesis that specialization of architectural mechanisms is the key to *efficient* compilation to gate-reconfigurable architectures. In addition, selectively turning off compiler features that specialize particular mechanisms demonstrates the benefits of each specialization technique. Finally, DeepC's current strategy for specializing parallel structures suffers from several limitations. I have shown these limitations and proposed solutions, including alternative parallelization strategies and techniques for overclocking the network.

7.2 Future Prospects

Having achieved efficient compilation to GRAs, I now make several extrapolations. First, because GRAs are fine grained, having smaller structures than traditional architectures, they are more fault-tolerant and reliable. Specialization strategies can work around tiny faults in the GRA with little sacrifice in area or performance. Second, their simplicity

enables GRAs to migrate to future fabrication technologies sooner than coarse-grain approaches. In many future technologies, such as cellular quantum dots, the physical nature of the machine requires a fine layer of interpretation. Programming this layer directly is more efficient that attempting to construct and program a von Neumann processor in the new substrate. Third, by eliminating complex state machines that are not programmable, GRAs will be less susceptible to hard-coded errors. In addition to problems like the year 2000 crisis and the shortsightedness of hard-wiring encryption algorithms that may become obsolete, constraints as basic as processor word length become legacies over time. As computers are embedded into the environment, long-term use will require extended serviceability without having to rip out silicon and replace old parts.

Beyond the unequivocal need for new compiler analyses, new reconfigurable architectures, and new high-level languages, the future of two other important areas is intertwined with gate reconfigurable technology. The first important area is verification. Elimination of the instruction set architecture (ISA) may seem to be a hindrance to verification, but going ISA-less is an advantage. Logic-level constructs are a better match for reasoning about programs. For example, Darko Marinov, in his excellent work on Credible Compilation, under advisor Martin Rinard, suggests:

> The presented framework can also support "compiling to logic" as done, for instance, in the DeepC compiler developed by Babb et al. [11]. This compiler targets FPGA-based systems and has a much cleaner code generation than a compiler that targets some specific instruction set. We believe that this makes it even easier to develop a credible code generation for DeepC.

Furthermore, anyone who has seen the errata list for a recent X86 architecture [78] will realize that modern processor implementations have grown so complex that companies can no longer guarantee that they are free from defects[1].

The second important area is *embedded operating systems*. For application to take full advantage of specialization and gate reconfigurability, operating systems must be capable of exposing this flexibility to the application level while retaining the ability to multiplex and protect hardware resources. Operating system reconfiguration has been studied at the software level and these studies may serve as the starting point for extending reconfigurability down to hardware architecture layers. This research space is otherwise largely unexplored; however, industry is not waiting — leading embedded operating system companies such as Wind River Systems (www.windriver.com) have announced partnerships with configurable device vendors to develop *platform* FPGAs.

[1]According to Intel, errata are design defects or errors. Errata may cause the processor's behavior to deviate from published specifications. Hardware and software designed to be used with any given stepping must assume that all errata documented for that stepping are present on all devices. For example consider errata 16 — IA-32: Code with FP instruction followed by integer instruction with interrupt pending may not execute correctly!

7.3 A New Philosophy for Architecture Design

> *Without the freedom to adapt software for your own needs, to help you neighbor by redistributing software, and build the community by adding new features, you get caught in a horrible proprietary tyranny and lose your morale and enthusiasm.*
>
> — *Richard Stallman opening Singapore Linux conference, March 10, 1999.*

Being able to adapt hardware to your own needs is just as important as being able to adapt software. With this perspective, my dissertation contributes to one of the most powerful visions of the computing community. The open hardware movement is underway (see fpgacpu.org, opencores.org, open-hardware.org, openhardware.net, openip.net, and the list goes on). However, these designs are specified at a low-level (usually RTL) and thus not accessible to the everyday software developer. DeepC's specialization techniques can be used to raise the abstraction to enable more would-be hardware modifiers to realize their collective dream. Second, DeepC moves the mechanisms of architecture from the hardware to the software domain. In the hardware domain, design of new architectural mechanisms has been restricted to an elite few. Even an engineer working for a leading processor manufacturer is unlikely to have any say in the design of that processor (unless that engineer happens to be the lead architect). But by moving these mechanisms to the software domain, many will be able to contribute to the advancement of computing machinery.

7.4 Parting Words to the Compiler Community

In parting, this dissertation has demonstrated techniques to automate the programmatic use of gate-reconfigurable architectures and has made the case for their advantages. Still, there remain enormous opportunities for further innovation in this area. Specialization techniques offer a method for codifying decades of knowledge about computer mechanisms into an automatic compilation system; this work only codifies a small subset of the known mechanisms. Moreover, new mechanisms appear every year. In many ways this is the ultimate compiler challenge, one that to date has been too formidable for much of the compiler community, who either prefer to stop at the instruction level or do not know that there is something beyond. I hope this work will inspire some of you to step outside this boundary, take the more encompassing view, and compile to gates!

Appendix A

Complete Evaluation Modes

In total, eight evaluation modes result from the permutation of compilation and interpretation at the high-level, machine-level, and logic-level. This chapter restates the first four modes, introduced in Chapter 2, and continues with the remaining high-level interpretation modes. These last four modes complete the range of possibilities and should stimulate further research.

Table A.1 lists all eight modes. The "level" column lists the number of interpretation levels followed by the letters H, M, or L to denote which levels are interpreted. Figure A-1 contains the evaluation dominos for the eight modes. Modes are grouped by level of interpretation.

Figures A-2 shows an alternative arrangement of the eight modes on the corners of a cube. In this cube, modes connected by an edge differ by a single interpret/compiler interchange. Each cube face corresponds to a common evaluation mode at one level. For example, the shaded face corresponds to the high level compilation modes, *Modes 0–III.* This figure can also be viewed as a lattice with *Mode 0* as top and *Mode VII* as bottom. The number of levels of interpretation in a given mode equals the distance in the lattice from the top to that mode.

Mode	Example	level	high ($H \rightarrow M$)	machine ($M \rightarrow L$)	logic ($L \rightarrow P$)
0	Custom ASIC	0	compile	compile	compile
I	FPGA Computing	1.L	compile	compile	interpret
II	RISC Processor	1.M	compile	interpret	compile
III	Softcore processor-on-FPGA	2.ML	compile	interpret	interpret
IV	PicoJava Chip, Scheme Chip	1.H	interpret	compile	compile
V	JavaVM-on-Processor	2.HM	interpret	interpret	compile
VI	JavaVM-specialized-to-FPGA	2.HL	interpret	compile	interpret
VII	JavaVM-Processor-FPGA	3.HML	interpret	interpret	interpret

Table A.1: Complete evaluation modes with examples

A.1 High-Level Compilation Modes

The following paragraphs restate the high-level compilation modes from Chapter 2, along with their denotational semantics. See Figure 2-8 for a review of denotational semantics.

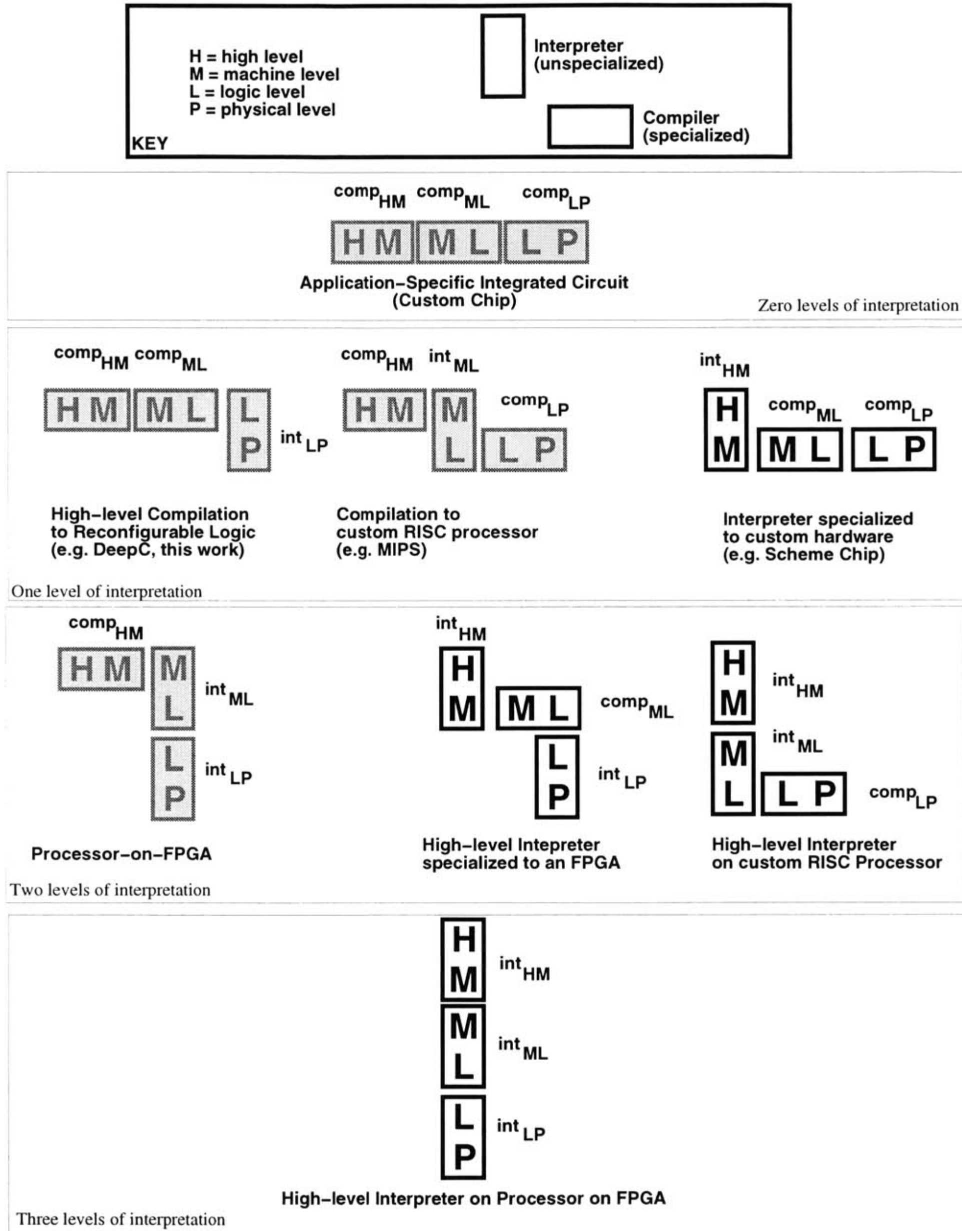


Figure A-1: Complete evaluation dominos

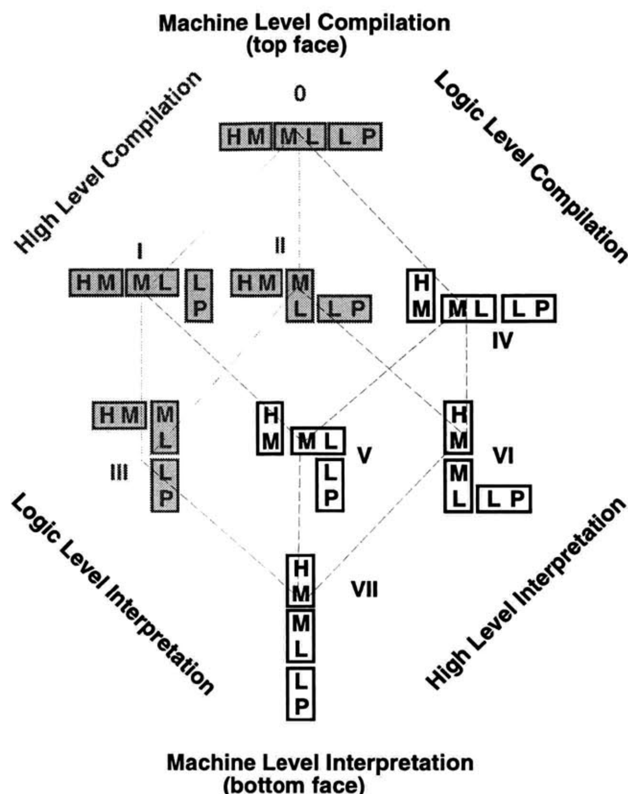


Figure A-2: Dominos arranged on the corners of a cube

Mode 0 A custom chip can be designed to evaluate a particular high-level behavior. The high-level behavior is customized into a machine-level behavior, usually represented in Register Transfer Language (RTL), which is then further customized to logic. This logic is in turn used to generate a physical layout. Specialization may be performed automatically, manually, or in some combination. This mode is the most efficient because there are zero levels of interpretation. However, a custom chip designed in this manner is not reprogrammable — it can only evaluate one high-level behavior.

The semantics for *Mode 0* are: $[\![\ [\![C_{L\rightarrow P}]\!]\ ([\![C_{M\rightarrow L}]\!]\ ([\![C_{H\rightarrow M}]\!]\ p)\ )\ ]\!]_P\ d$,
meaning first compile program p from language H to M, and then from language M to L, and then from language L to P. The result is a physical realization of p in hardware that is capable of processing data d.

Mode I The gate reconfigurable approach is the primary focus of this dissertation. A high-level language is compiled into machine-level mechanisms that are further specialized to the logic-level. This logic, unlike it Mode 0, is not further specialized, but instead evaluates on programmable logic. Thus, this mode has one level of interpretation and is less efficient than *Mode 0*. Yet, with this level of interpretation the system is fully programmable and thus *universal* — different high-level behaviors can be evaluated on the same physical hardware.

The semantics for *Mode I* are: $[\![I_{L\rightarrow P}]\!]_P < [\![C_{M\rightarrow L}]\!]\ ([\![C_{H\rightarrow M}]\!]\ p), d >$,
meaning compile the program p from language H to M and from language M to L. Then, interpret language L, along with data d, on a pre-fabricated $I_{L\rightarrow P}$ machine.

Mode II Traditional processors offer the most conventional form of computing. Languages are compiled to the instruction set architecture of a processor. The processor is in effect an interpreter for machine-level programs. The processor is implemented in logic that is further specialized, manually or automatically, to the physical-level. Like *Mode I*, *Mode II* is also universal because of the machine level interpretation layer.

The semantics for *Mode II* are: $[\![\ [\![C_{L\rightarrow P}]\!]\ I_{M\rightarrow L}]\!]_P\ <([\![C_{H\rightarrow M}]\!]\ p),d>$, meaning compile program p from language H to M and interpret it on a machine-level interpreter $I_{M\rightarrow L}$. This machine-level interpreter is written in language L and pre-fabricated by compiling language L to P. Note that this last compilation step can be done manually.

Mode III Softcore processors consist of a combination of the interpreter in *Mode I* with the interpreter in *Mode II*. Although the high-level language is compiled to machine instructions, neither is the machine further specialized to logic nor is the logic further specialized to the P. Thus these systems have two levels of interpretation. FPGA vendors have released softcore processors into the marketplace. With softcore processors, new high-level programs can be evaluated without re-specializing to FPGA logic. However, with double the interpretation one can expect roughly double the overhead in comparison to *Mode I* or *Mode II*.

The semantics for *Mode III* are: $[\![I_{L\rightarrow P}]\!]_P < I_{M\rightarrow L}, < ([\![C_{H\rightarrow M}]\!]\ p), d >>$, meaning compile program p from language H to M and interpret the compiled program, along with its data d, on a machine-level interpreter $I_{M\rightarrow L}$. This interpreter is written in language L and is itself interpreted by a logic-level interpreter $I_{L\rightarrow P}$.

A.2 High-Level Interpretation Modes

High-level interpretation modes, exhibited by architectures such as Sun's PicoJava chip [101] or the Scheme-79 chip [74], should not be discounted. These direct-execution modes relieve compiler burdens while allowing devices to adapt quickly to new software configurations — software that recently arrived over a slow network link, for example. In modern computer architecture teachings, direct-execution machines are considered "dead", and usually only mentioned historically or as an example of what not to do. This design space should not be discarded so lightly. The following paragraphs discuss each high-level interpreted mode in further detail.

Mode IV This mode is popular for evaluating interpreted languages. An early example of this mode is a Scheme Chip designed at MIT in 1978–79. A modern example is Sun's PicoJava chip, a hardware implementation of the Java Virtual Machine. Another example is the Basic Stamp [106], a popular Basic platform for robotics. The success of *Mode IV* lies in the ability to overcome the interpretation overheads at the highest level by specializing the machine level with respect to this high-level interpreter. This machine specialization results in only one interpretative layer and thus competitive performance with *Mode I* and *Mode II*[1]. There are many advantages to this mode when the high-level language is dynamic.

The semantics for *Mode IV* are: $[\![\ [\![C_{L\rightarrow P}]\!]\ ([\![C_{M\rightarrow L}]\!]\ I_{H\rightarrow M})]\!]_P\ < p, d >$, meaning interpret program p, in a high-level language H, with an $I_{H\rightarrow M}$ interpreter written in

[1]Whether or not there is only one interpreter if the machine is specialized into microcode is open for debate.

language M. This interpret is compile from language M to language L, and furthermore to the physical level (language P).

Mode V This mode consists of a high-level interpreter executing on a traditional processor — for example, a Scheme or Java interpreter running on a workstation. This mode is known to be slow. Advantages are portability — the same high-level interpret can execute on different machine-level processors — and minimal compile time. A new language may be interpreted until compiler support is available.

The semantics for *Mode V* are: $[\![\ [\![C_{L \to P}]\!]\ I_{M \to L}]\!] < I_{H \to M}, < p, d >>$, meaning interpret program p, in a high-level language H, with an $I_{H \to M}$ interpreter, written in language M. This interpret is interpreted in language L with an $I_{M \to L}$ machine, itself compiled to P.

Mode VI This mode consists of a high-level interpreter that is specialized to an FPGA. This is an uncommon mode, sort of the antithesis of the traditional *Mode II* approach. You may need to think for a few minutes to appreciate what it means to execute in this mode. I am not aware of commercial examples of this mode, although synthesis of the publicly available PicoJava architecture to an FPGA has likely been tried. This mode may be a promising area for future research, but it does carry a double interpretation penalty.

The semantics for *Mode VI* are: $[\![I_{L \to P}]\!]_P < ([\![C_{M \to L}]\!]\ I_{H \to M}), < p, d >>$, meaning interpret program p, in a high-level language H, with an $I_{H \to M}$ interpreter, written in language M, and compiled to language L. This compiled interpreter is further interpreted by an $I_{L \to P}$ machine (such as an FPGA).

Mode VII This mode is fully interpreted. Like *Mode V*, this Mode is interesting when compilation tools are immature for a new language or architectural technique. The three levels of interpretation overhead will multiplicatively inhibit performance, making this unlikely to be a desirable mode except for pedagogical purposes. However, a *Mode VII* system can be built without compilation technology.

The semantics for *Mode VII* are: $[\![I_{L \to P}]\!]_P < I_{M \to L}, < I_{H \to M}, < p, d >>>$, meaning interpret program p, in a high-level language H, with an $I_{H \to M}$ interpreter, written in language M. This interpret is interpreted in language L with an $I_{M \to L}$ machine, itself interpreted by an $I_{L \to P}$ machine. Thus this mode needs no compilation.

A.3 Hybrid Modes

Hybrid systems are formed when two or more evaluation modes are combined in the same system. For example, embedding hardcore processors in FPGA devices forms a Hybrid I-II system. Both Xilinx and Altera have introduced Hybrid I-II systems at the time of this writing. Hybrid systems have already been explored in research by many systems cited in the related work (Section 1.4), including NapaC, GARP, PRISC, and the originally proposed Raw machine. *Mode I-II* hybrids are also being combined with softcore processors, resulting in *Mode I-II-III* hybrids. Furthermore, other companies (such as adaptivesilicon.com) are selling FPGA cores to ASIC customers and FPGA manufacturers are customizing standard interfaces into their *Mode I* devices. With the ever increasing transistor densities expected in future devices, design engineers should be prepared to encounter all permutations of modes.

Appendix B

Emulation Hardware

Section 5.3 and Section 5.4.5 discussed use of the VirtuaLogic Emulation System as a DeepC target and for functional verification of designs compiled from a high level. Section 1.4.1 also discussed previous work where an emulation system was used as a reconfigurable computing platform. This appendix extends the previous discussions by describing further results using DeepC to compile to an emulator. It also includes descriptions of several prototype host interfaces used to turn an emulator into a reconfigurable computer.

In support of my thesis, I use DeepC to generate results similar to those presented in the Raw Benchmark Suite. In the Raw Benchmark Suite, results were generated with a combination of manual and automatic techniques. Here, besides being automatically generated, the results are improved and more scalable. Like in the Raw Benchmark Suite, the emulator and its host interfaces are working hardware systems, further proving the approaches of DeepC in the context of a real system.

B.1 DeepC Compilation Results and Functional Verification

Compilation to the VirtuaLogic system enables comparison with earlier Raw Benchmark Suite results, in which specialization of architectural mechanisms was performed manually. Table B.1 and Figure B-1 contain the gate counts reported by Synopsys when targeting the VMW technology library. Readers interested in studying these results further can compare to the gate counts reported in the Raw Benchmark Suite [10]. The gate counts here are larger; however, in contrast to the previous results, these tiles have been generated from a high-level language. Also, gate area does not grow (except logarithmically, as bitwidths increase) with increasing problem size. Only memory area grows proportional to problem size.

Jacobi has been downloaded to the emulator and verified to work. Figure B-2 shows the tool flow used. The left flow is the standard DeepC compiler flow. Following RTL generation, an RTL logic synthesizer and then FPGA vendor synthesis tools are run. The result is a gate-level configuration, an FPGA bitstream. The flow on the right shows verification with emulation. The target of logic synthesis is a special technology library for emulation. Following synthesis are the steps of Virtual Wires: Multi-FPGA partitioning, Virtual Wires Scheduling, and Virtual Wires Synthesis. This path synthesizes multiple FPGAs. The resulting bitstreams are downloaded to the emulator and targetless emulation is performed. Targetless emulation, also called simulation acceleration, uses one of the host interfaces described in section B.2.

Benchmark	Combinatonal (VMW Gates)	Sequential (VMW Gates)	Total Area (VMW Gates)
adpcm	8626	3192	11818
bubblesort	2668	1640	4308
convolve	13303	3584	16887
histogram	4116	2680	6976
intfir	3169	1712	4881
intmatmul-v1	4073	2088	6161
intmatmul-v2	3940	2956	5996
jacobi	3844	2120	5964
life-v1	12822	3504	16326
life-v2	4355	2120	6475
median	7316	5768	13084
mpegcorr	4027	2416	6443
parity-v1	1223	1176	2399
parity-v2	1337	1152	2489
pmatch-v1	2266	2440	4706
pmatch-v2	2972	3096	6068
sor	3672	2040	5712

Table B.1: VMW gate count

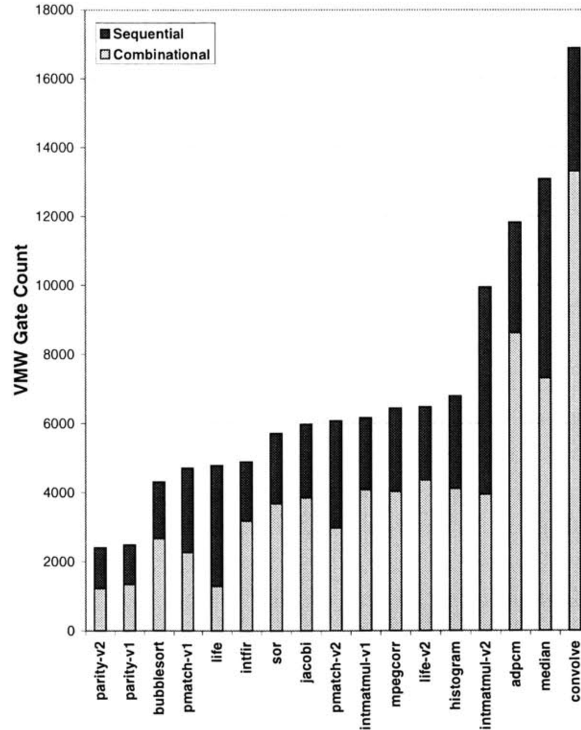


Figure B-1: Gatecount after synthesis to VirtuaLogic Emulator

Benchmarks are sorted by size to compare requirements across benchmark types. This metric is comparable with the VMW gates reported in [10] and [11].

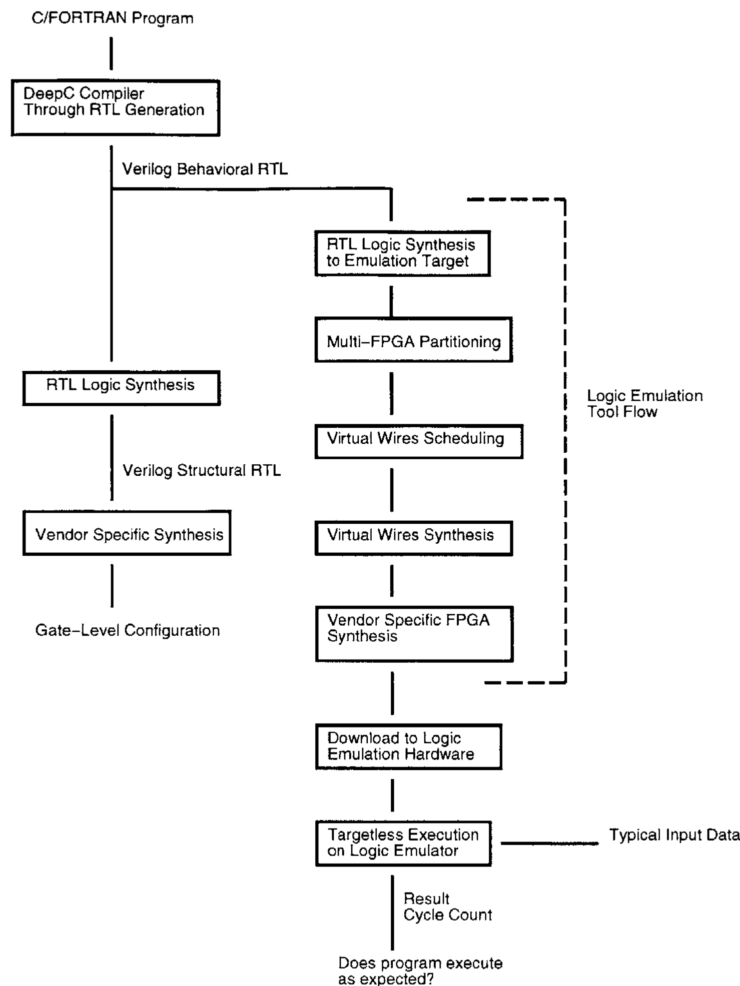


Figure B-2: Verification with a logic emulator

B.2 Host Interfaces

In this prototype system (Figure B-3), the emulator is extended with a host interface. A VirtuaLogic Emulator (VLE) from IKOS Systems is coupled with a host computer via an interface card. Not shown is a SCSI interface to the emulator for downloading configurations and controlling clock speed. This production VLE system consists of five arrays of 64 Xilinx 4013 FPGAs each. The FPGAs on each board are connected in nearest-neighbor meshes augmented by longer connections to more distant FPGAs. Boards are coupled with multiplexed I/Os. Each board has several hundred external I/Os, resulting in total external I/O connections of a few thousand.

When used as a logic emulator, the external I/O interface of the VLE is connected to the target system of the design under emulation. For reconfigurable computing, we have instead connected a portion of the external I/O to an interface card. This card serves as a transportation layer for communicating with device drivers in the host. The following section describes several generations of host interfaces.

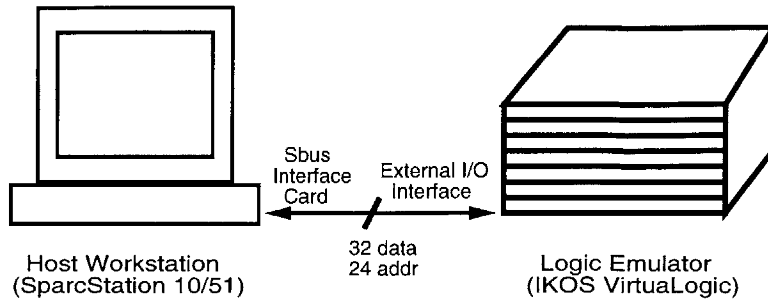


Figure B-3: Host interface to emulator

B.2.1 WILD-ONE Host Interface

The latest host interface is the WILD-ONE host interface, built with an Annapolis Micro Systems WILD-ONE programmable PCI card. (I helped Alex Kupperman implement this interface for his B.S. Thesis [83].) This card has a synchronous bus with 20 bits of address and 64 bits of data that may be read and written directly. Both DMA and memory-mapped I/O are supported. We have clocked this interface at emulation speeds of *1-2 MHz*.

This card installs in a personal computer (PC). The reconfigurability of the WILD-ONE card permits a flexible host interface design. The PCI card contains two Xilinx 4036 FPGAs that implement the host interface. Three 32-bit busses connect the FPGA pins to an extra connector on the card. We hired a consultant (Eric Bovell) to design a custom daughtercard to interface between this connector and the VirtuaLogic cable. The Annapolis system includes API drivers for the PC to talk to the card from a C program. Figure B-4 shows the WILD-ONE card with the daughtercard installed. Figure B-5 show the other side of the WILD-ONE card.

In contrast to previous interfaces, this design implements a synchronous interface with a common clock. Both the WILD-ONE FPGAs and the emulator sample a clock generated by the WILD-ONE card. Internal to the Annapolis card, a datapath is implemented to interconnect the PCI bus to a read/write interface. Although a fast clock runs inside both the emulator and the WILD-ONE (about 25MHZ), they are only synchronized at emulation speed.

Figure B-6 shows the components inside the host interface. This figure is from Alex's Thesis [83]. In this model, applications in the emulator access the host as if it were memory. On the PC side, writes to particular addresses are used as specific commands to read, write, reset, and otherwise control the emulator. In the figure, the memory-mapped module in the emulator, written in Verilog, is like a device driver. Upon compilation, this module is linked with the output from DeepC. The components inside the PC include the WILD-ONE FPGA bitstream and other software. This software includes the Annapolis driver, an interface library written by Alex, and a user C program (labeled "Emulated Device"). The emulated device is a shell for the portion of the user program that evaluates on the emulator.

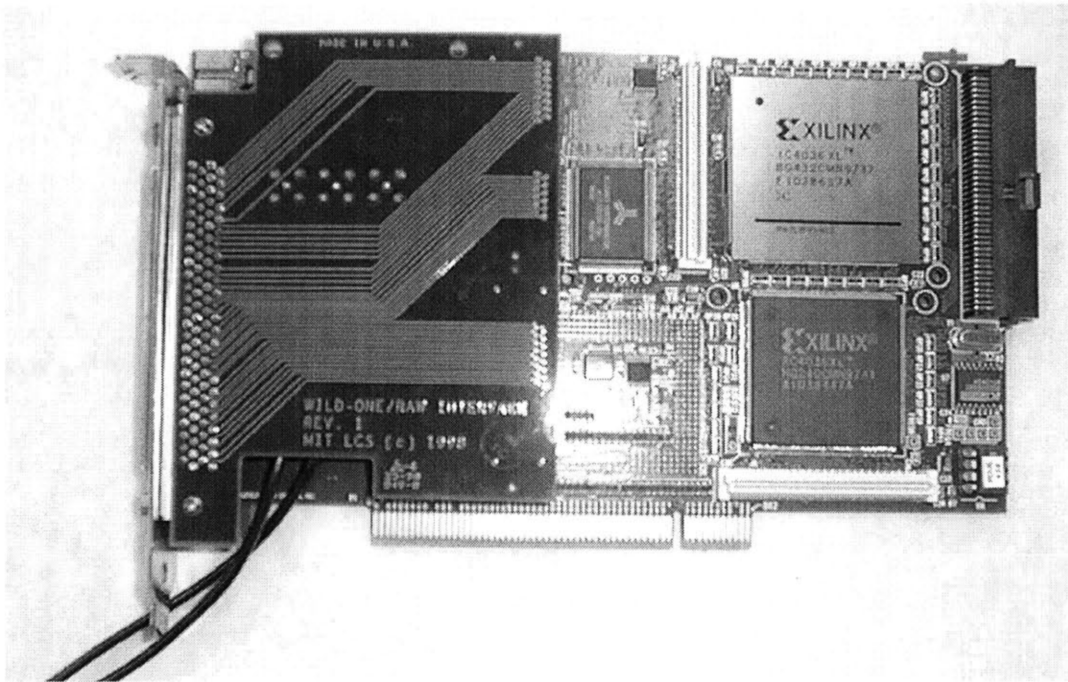


Figure B-4: Annapolis Micro Systems WILD-ONE with custom connector card

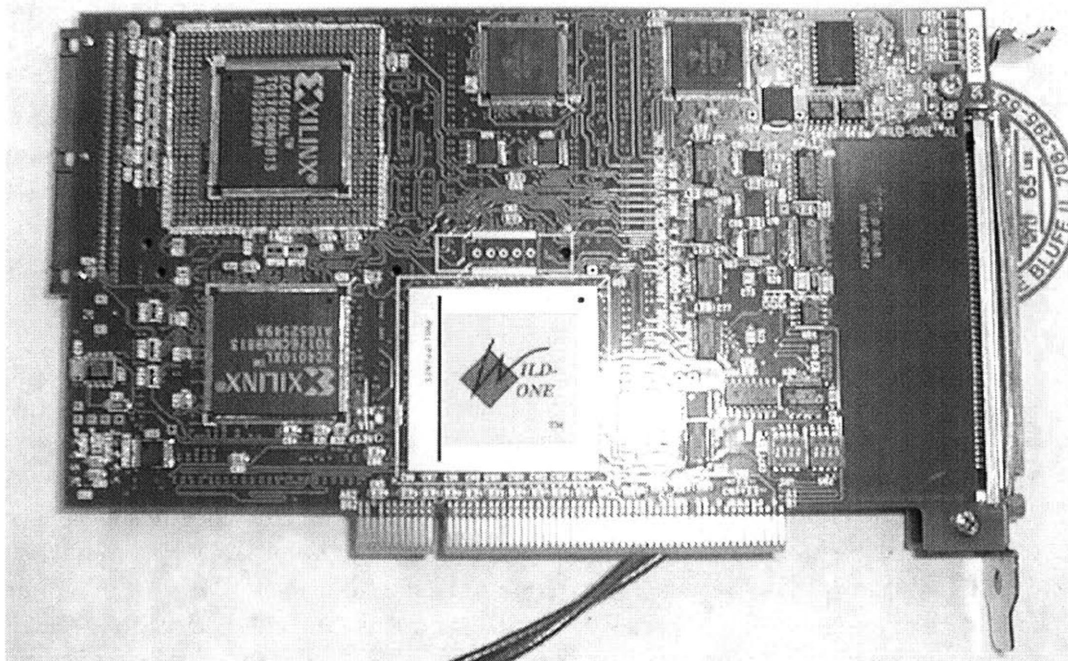


Figure B-5: Back of WILD-ONE

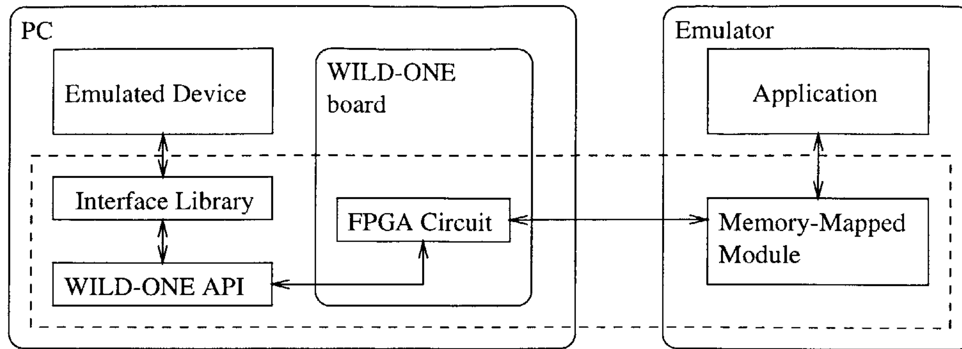


Figure B-6: Components of host interface

The interface is enclosed in dashed boxes and works across hardware and software.

B.2.2 SLIC Host Interface

Before development of the Wildfire host interface, I designed an interface with the Sun SLIC EB-1 SBus card [47] development card. The introduction from the SLIC documentation described the card as follows:

> *Dawn VME Products* SLIC EB-1 is a kit for prototyping hardware and software that offers a total solution for developing new SBus applications or porting existing applications from a different bus architecture. The kit centers around the Motorola MC92005 (a.k.a. SLIC), a complete SBus slave interface in a single chip. The MC92005's PBus is an asynchronous programmable private bus, whose signals and timing can be redefined dynamically, allowing it to interface to many commercially-available peripheral chips.

In this setup (Figure B-7), the external I/Os of an IKOS Hermes VirtuaLogic Emulation System are connected to the PBus interface of a SLIC card. This SLIC card is then installed in a host SPARC workstation. The SLIC card plus its device driver function as a transport layer. In the figure, the central board of connectors is the breakout board used to match of emulator I/Os with a target system. In this case, the target system is the connector on the interface card. A separate SCSI interface cable is connected to the emulator for downloading configurations and controlling clock speed.

When used as a logic emulator, the external I/O interface of the VLE is connected to the target system of the design under emulation. Just as in the Annapolis interface, some external I/Os are instead connected to an interface card. The SLIC card has an asynchronous bus with 24 bits of address and 32 bits of data. This bus can be read and written by memory-mapped I/O to the Sparc Sbus. I operated this interface at conservative rates of 0.25MHz for reads and 0.5MHz for write operations given a 1MHz emulation clock. This provided 1-2 Mbytes/sec rates for communication between the host CPU and the FPGAs of the emulator. This limited I/O rate allows one 32 bit read/write every 100/50 cycles of the 50MHZ host CPU.

Verilog code in files named interface.v and system.v implement the VLE side of the interface. Synopsys scripts interface.syn and system.syn control synthesis of this code into an IKOS-supplied target library. The IKOS files vmw.clk and vmw.pod describe the con-

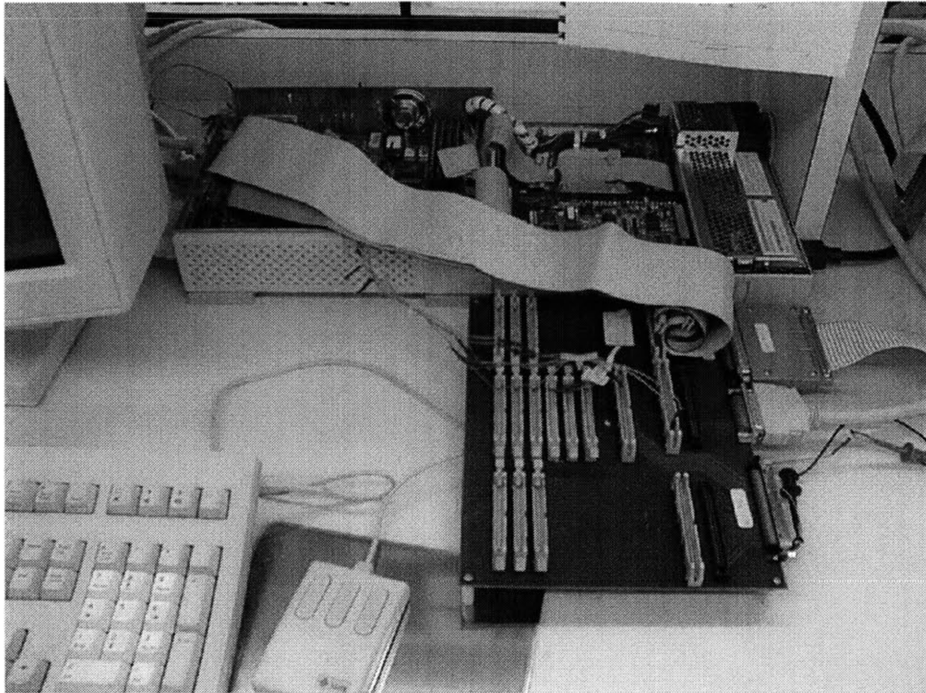


Figure B-7: SLIC interface

figuration at MIT. All of these files can be found in the include directory for the SLIC interface, both in the Raw Benchmark Suite release and in the DeepC release.

This interface operated successfully to verify the Raw Benchmark Suite results [10] as reviewed in Section 1.4.1.

B.2.3 Transit Host Interfaces

The oldest SBus interface to the emulator, described in this paragraph for archival purposes, was base on the MIT's Transit SBus Interface. Transit documentation describes the interface as follows:

> This note describes an SBus interface designed for providing interfacing from an SBus based machine to various custom hardware. This interface was designed primarily for communication with boundary-scan controllers and Transit nodes Nevertheless, it uses a minimal asynchronous interface making it suitable for providing interfacing to many devices. The interface is an SBus slave card that supports only single-word read and write operations. The interface card itself includes a boundary-scan controller that supports two independent boundary scan paths.

This interface connected an early *Argus* version of the IKOS VirtuaLogic Emulator to a host SPARC 10 Workstation. This interface had a 24 bit address, 32 bit data connection that was read and written with memory-mapped I/O. It operated in the 100KHz to 1MHz, making it usable for both simulation acceleration and reconfigurable computing applications.

Appendix C

Data For Basic Results

C.1 *Mode 0* Basic Results

Section 6.3.1 refers to the data in Tables C.1 and C.2.

Benchmark	DeepC Cycles	Rawsim, 1 tile Cycles (instr)	UltraSparc IIi gcc -O0/-O5	Speedup (Rawcc/DeepC)	Cross-technology (Sparc/DeepC)
adpcm	392K	3.07M(3.02M)	9.7M/1.5M	7.8X	3.8X
bubblesort	265K	3.16M(2.90M)	6.61M/1.6M	11.9X	6.0X
convolve	5.38K	72.4K(60.7K)	92.8K/61K	13.5X	11.3X
histogram	34.7K	247K(235K)	550K/81K	7.1X	2.3X
intfir	103K	714K(681K)	1.8M/540K	6.9X	5.2X
intmatmul-v1	111K	789K(788K)	2.5M/429K	7.1X	3.9X
intmatmul-v2	112K	819K(817K)	2.5M/429K	7.3X	3.9X
jacobi	94.4K	847K(842K)	1.5M/317K	9.0X	3.4X
life-v1	219K	1.90M(1.90M)	2.9M/567K	8.7X	2.6X
life-v2	115K	885K(885K)	1.3M/269K	7.7X	2.3X
median	203K	1.07M(1.02M)	4.2M/723K	5.3X	3.6X
mpegcorr	4.85K	35.7K(35.7K)	90.0K/14K	7.4X	2.9X
parity-v1	70.7K	516K(885K)	921K/112K	7.3X	1.6X
parity-v2	103K	686K(686K)	1.24M/240K	6.7X	2.3X
pmatch-v1	236K	2.57M(2.45M)	3.5M/1.0M	10.9X	4.2X
pmatch-v2	398K	2.93M(2.93M)	5.86M/807K	7.4X	2.0X
sor	35.6K	264K(2.63K)	482K/62K	7.4X	1.7X

Table C.1: Cycle counts

For each benchmark, I measured the number of clock cycles to execute with DeepC compiling to RTL Verilog, on the Raw Simulation (compiled by Rawcc), and on an UltraSparc IIi (compiled by gcc). The most relevant speedup is between DeepC and Rawsim. Comparing to UltraSparc is only used as a known reference point for a production system.

Benchmark	Non-memory Cell Area total (comb, seq)	Silicon area (mm^2, $lambda^2$)	Area Fraction of a Mips Core	latency	power @300MHZ total
adpcm	29421 (21024, 8397)	0.22 / 27M	5.8 %	3.26 ns	7.10 mW
bubblesort	10401(5968, 4433)	0.08 / 10M	2.0 %	3.02 ns	3.45 mW
histogram	15260 (8263, 6997)	0.11 / 14M	3.0 %	2.67 ns	3.59 mW
intfir	11885 (7388, 4497)	0.09 / 11M	2.3 %	2.77 ns	3.16 mW
intmatmul-v1	15083(9558, 5525)	0.11 / 14M	3.0 %	3.04 ns	4.36 mW
intmatmul-v2	13931(8774,5157)	0.10 / 13M	2.7 %	2.90 ns	3.97 mW
jacobi	12750(7437, 5313)	0.10 / 12M	2.5 %	2.93 ns	3.57 mW
life-v1	43728 (34699, 9029)	0.33 / 41M	8.6 %	3.04 ns	18.8 mW
life-v2	14797 (9456, 5341)	0.11 / 14M	2.9 %	2.89 ns	4.38 mW
median	29280 (14852, 14428)	0.22 / 27M	5.7 %	3.18 ns	8.57 mW
mpegcorr	14107 (8067, 6040)	0.11 / 13M	2.8 %	2.94 ns	5.26 mW
parity-v1	5266 (2312, 2954)	0.04 / 5M	1.0 %	2.63 ns	1.80 mW
parity-v2	6930 (3632,3298)	0.05 / 6M	1.4 %	2.60 ns	2.10 mW
pmatch-v1	11243 (5126, 6117)	0.08 / 10M	2.2 %	2.96 ns	3.44 mW
pmatch-v2	13896 (6139,7757)	0.10 / 13M	2.7 %	2.75 ns	4.77 mW
sor	13790 (8496, 5294)	0.10 / 13M	2.7 %	3.13 ns	6.12 mW

Table C.2: Power and area estimates

Data is reported after Synopsys synthesis to IBM SA27E ASIC technology. Area numbers shown assume a cell size of 3.762 um^2 with 50 percent routability. For this process, $lambda = 0.09\ u$. Area and Power estimates do not include RAM cost or overhead for pads and periphery circuitry. A clock rate of 300 MHz or more is achieved in each case. Power is estimated at a 300 MHz clock speed. The MIPS area is estimated as 25 percent of a RAW Tile and does not include RAM area.

C.2 *Mode I* Basic Results

Section 6.3.2 refers to the data in Tables C.3, C.4, C.5 and C.6.

Benchmark	Number of core cells (luts, flops)	latency
adpcm	2562 (414, 680)	20.47 ns
bubblesort	931 (201, 221)	24.25 ns
convolve	15864 (14041, 707)	37.59 ns
histogram	1223 (320, 349)	17.64 ns
intfir	1046 (521, 239)	17.64 ns
intmatmul-v1	1398 (776, 285)	32.93 ns
intmatmul-v2	1414 (733, 263)	32.63 ns
jacobi	1171 (458, 265)	17.91 ns
life-v1	3586 (1346, 440)	23.22 ns
life-v2	1829 (745, 366)	20.23 ns
median	2157 (827, 721)	20.38 ns
mpegcorr	1166 (278, 305)	17.21 ns
parity-v1	493 (118, 157)	13.28 ns
parity-v2	555 (174, 135)	14.35 ns
pmatch-v1	824 (191, 322)	15.05 ns
pmatch-v2	971 (249, 414)	14.55 ns
sor	1181 (276, 275)	18.91 ns

Table C.3: Synthesis to Xilinx Virtex-E-7 process

The timing file for Virtex-E-8 was not available at the time of this experiment, so delays are for Virtex-7. Virtex-E-8 data is used during place and route, so these numbers are pessimistic for timing.

C.3 Basic Sensitivity Results

Section 6.4 refers to the data in Tables C.7 and C.8.

Benchmark	Resources required Luts/Flops/Rams/States	Equiv. Gate Count gates / gates+ram	Clock Frequency (% routing)	Clock Cycles	Total Latency
adpcm	1544 / 669 / 27 / 23	12K / 310K	100 MHz (65%)	390K	3900 us
bubblesort	336 / 201 / 4 / 17	5K / 70K	80 MHz (60%)	260K	3300 us
convolve	1571 /426 / 2 / 30	44K / 50K	100 MHz (65%)	5400	54 us
histogram	464 / 294 / 14 / 29	7K / 240K	100 MHz (50%)	35K	350 us
intfir	466 / 202 / 10 / 16	5K / 170K	50 MHz (60%)	100K	2000 us
intmatmul-v1	723 / 250 / 14 / 25	7K / 240K	50 MHz (60%)	110K	2200 us
intmatmul-v2	725 / 230 / 14 / 34	7K / 240K	50 MHz (60%)	110K	2200 us
jacobi	573 / 248 / 28 / 35	7K / 470K	100 MHz (60%)	94K	940 us
life	1731 / 370 / 4 / 58	17K / 80K	80 MHz (60%)	220K	2750 us
life-v2	769 / 245 / 2 / 54	9K / 40K	100 MHz (50%)	120K	1200 us
median	1117 / 696 / 60 / 31	13K / 1M	100 MHz (50%)	200K	2000 us
mpegcorr	481 / 296 / 14 / 30	7K / 240K	100 MHz (60%)	4900	49 us
parity-v1	179 / 137 / 8 / 11	3K / 130K	100 MHz (60%)	70K	700 us
parity-v2	189 / 143 / 8 / 15	3K / 130K	100 MHz (60%)	100K	1000 us
pmatch-v1	293 / 276 / 15 / 17	5K / 80K	100 MHz (60%)	350K	3500 us
pmatch-v2	304 / 373 / 15 / 19	5K / 80K	100 MHz (60%)	400K	4000 us
sor	339 / 215 / 14 / 22	6K / 230K	100 MHz (50%)	36K	360 us

Table C.4: Synthesis results after place and route to Xilinx Virtex-E-8

Beside the clock frequency is the percent of the clock cycle devoted to routing. Apparently, the synthesizer is smart enough to add drivers for larger fanouts such that wire delay and logic delay is approximately balanced.

Benchmark	Switching Activity	Memory Activity	Quiescent Power	Luts Power	Regs Power	Rams Power	Total Power	Total Energy
adpcm	11.95 %	8 %	29	66	33	27	155 mW	607 uJ
bubblesort	13.07 %	96 %	9	23	9	84	125 mW	414 uJ
convolve	13.01 %	52 %	29	417	28	41	515 mW	28 uJ
histogram	8.86 %	27 %	58	29	24	99	210 mW	73 uJ
intfir	10.46%	21 %	14	18	4	32	68 mW	141 uJ
intmatmul-v1	9.68%	30 %	29	18	12	34	93 mW	206 uJ
intmatmul-v2	10.83%	30 %	9	24	8	62	103 mW	231 uJ
jacobi	10.2 %	46 %	14	37	20	151	222 mW	210 uJ
life-v1	12.74 %	43 %	14	86	38	23	161 mW	442 uJ
life-v2	9.94 %	41 %	9	48	23	15	95 mW	109 uJ
median	12.98 %	11 %	81	92	57	48	278 mW	564 uJ
mpegcorr	9.78 %	14 %	9	33	21	42	105 mW	5 uJ
parity-v1	16.7 %	47 %	9	22	8	72	111 mW	78 uJ
parity-v2	7.88 %	48 %	9	12	9	18	48 mW	49 uJ
pmatch-v1	9.3 %	32 %	14	25	14	69	122 mW	289 uJ
pmatch-v2	13.0 %	33 %	14	19	18	118	169 mW	674 uJ
sor	11.7 %	76 %	36	32	16	414	498 mW	177 uJ

Table C.5: More synthesis results after place and route to Xilinx Virtex-E-8

Synthesis is to the smallest Virtex part that fits each design. Memory activity is averaged across all RAMs.

Benchmark	MIPS core (1W@300MHZ)	VirtexE-8 (deepc) (from Table C.5)	Deep versus MIPS Energy Reduction	Deep versus MIPS Energy-Delay Reduction
adpcm	10 mJ	607 uJ	17X	44X
bubblesort	11 mJ	414 uJ	25X	81X
convolve	241 uJ	28 uJ	9X	39X
histogram	823 mJ	73 uJ	11X	27X
intfir	2.4 mJ	141 uJ	17X	20X
intmatmul-v1	2.6 mJ	206 uJ	13X	15X
intmatmul-v2	2.7 mJ	231 uJ	12X	14X
jacobi	2.8 mJ	210 uJ	13X	40X
life-v1	6.3 mJ	442 uJ	14X	33X
life-v2	3.0 mJ	109 uJ	27X	69X
median	3.6 mJ	564 uJ	6X	11X
mpegcorr	0.1 mJ	5 uJ	23X	57X
parity-v1	1.7 mJ	78 uJ	22X	53X
parity-v2	2.3 mJ	49 uJ	46X	103X
pmatch-v1	8.6 mJ	289 uJ	30X	108X
pmatch-v2	9.8 mJ	674 uJ	14X	36X
sor	0.9 mJ	177 uJ	5X	12X
sor-mem4	0.9 mJ	44 uJ	20X	25X

Table C.6: Energy and energy-delay comparison

Energy is computer from power and runtime. Power for the VirtexE-8 is reported from the Xilinx Power Estimator.

Benchmark	Base Speedup	Disambiguation Speedup	Predication Speedup	Unroll 4 Speedup
adpcm	3.7X	3.9M	7.8X	9.0X
bubblesort	3.1X	4.1X	11.9X	15.7X
histogram	7.1X	7.1X	7.1X	11.0X
intfir	6.9X	6.9X	6.9X	13.6X
intmatmul-v1	7.1X	7.1X	7.1X	9.7X
intmatmul-v2	7.3X	7.3X	7.3X	10.6X
jacobi	9.0X	9.0X	9.0X	11.3X
life-v1	6.3X	6.3X	8.7X	9.9X
life-v2	7.7X	7.7X	7.7X	10.1X
median	3.9X	4.0X	5.3X	9.2X
mpegcorr	7.1X	7.4X	7.4X	12.0X
parity-v1	7.3X	7.3X	7.3X	11.7X
parity-v2	6.7X	6.7X	6.7X	10.1X
pmatch-v1	3.7X	4.4X	10.9X	13.2X
pmatch-v2	3.7X	4.5X	7.4X	11.0X
sor	7.4X	7.4X	7.4X	9.3X

Table C.7: Cumulative contribution of specializations to cycle count reduction

The base speedup measures speedup versus a MIPS core. Disambiguation speedup is obtained when ECU is activated, while predication speedup is obtained with macro formation. Speedup from extra unrolling was not assumed in the previous basic results, although a significant additional gain is achievable in several instances.

Benchmark	Cell Area total (comb, seq)	latency	timing met?	power @300MHZ total
bubblesort	25381 (17891, 7490)	3.80 ns	no	6.89 mW
histogram	23764 (15144, 8194)	3.24 ns	yes	7.59 mW
intfir	18038 (11857, 6181)	3.09 ns	no	4.32 mW
intmatmul-v1	38471 (29397, 9074)	3.28 ns	yes	7.04 mW
intmatmul-v2	44483 (35125,9358)	3.28 ns	yes	6.73 mW
jacobi	29291 (19703, 9588)	3.07 ns	no	8.46 mW
life-v1	122795 (105739, 17056)	3.34 ns	no	39.4 mW
life-v2	48841 (14140, 34701)	3.29 ns	yes	10.3 mW
median	36329 (20515, 15814)	3.14 ns	yes	8.63 mW
mpegcorr	29090 (18716, 10374)	3.11 ns	yes	6.92 mW
parity-v1	11795 (6581, 5214)	3.10 ns	yes	2.73 mW
parity-v2	13947 (7642, 6305)	3.01 ns	yes	3.44 mW
pmatch-v1	15917 (8672, 7245)	3.14 ns	yes	3.70 mW
pmatch-v2	20790 (12603, 8187)	3.27 ns	yes	6.27 mW
sor	29520 (21233, 8287)	3.42 ns	no	14.5 mW

Table C.8: Synthesis to IBM SA27E process without bitwidth analysis

These results match the results in Table C.2, with the exception that bitwidth analysis is not used.

Appendix D

Data For Advanced Results

D.1 *Mode 0* Advanced Results

Section 6.5.1 refers to the data in Tables D.1, D.2 and D.3.

Benchmark	One Tile	Two Tiles router / crossbar	Four Tiles router / crossbar	Eight Tiles router / crossbar	Sixteen Tiles router / crossbar
intmatmul-v2	112K	276K / 113K	133K / 47K	70K / 23K	37K / 18K
jacobi	94K	126K / 82K	81K / 42K	83K / 26K	40K / 15K
mpegcorr	4.9K	12K / 5.8K	7.3K / 3.5K	5.9K / 2.7K	3.9K / 2.1K
life-v2	115K	152K / 76K	86K / 40K	51K / 22K	19K / 8.2K
parity-v2	103K	122K / 70K	71K / 35K	50K / 22K	28K / 10K
pmatch-v2	398K	829K / 333K	499K / 191K	597K / 356K	304K / 162K
sor	36K	66K / 32K	35K / 16K	27K / 9.0K	16K / 4.7K

Table D.1: Multi-tile cycle counts with and without router specialization

For the benchmarks compared, cycle counts are reported for one through sixteen tile systems. The router cases take more cycles because of intermediate hops through the routing network. The crossbar cases allow direct communication between tiles.

Benchmark	One Tile (cells)	Two Tiles router / crossbar	Four Tiles router / crossbar	Eight Tiles router / crossbar	Sixteen Tiles router / crossbar
intmatmul-v1	15K	74K / 56K	271K / 292K	1014K / 885K	3803K/ 2331K
intmatmul-v2	15K	116K / 126K	286K / 268K	584K / 843K	2172K/ 2754K
jacobi	13K	44K / 44K	97K / 91K	226K / 256K	537K/ 665K
mpegcorr	14K	44K / 41K	101K / 104K	227K / 265K	624K/ 790K
life-v2	15K	59K / 49K	127K / 113K	271K / 313K	512K/ 694K
parity-v2	7K	21K / 21K	40K / 43K	109K / 127K	287K/ 400K
pmatch-v1	11K	63K / 55K	143K / 143K	352K / 424K	1053K/ 1296K
pmatch-v2	14K	55K / 49K	151K / 161K	352K / 410K	1103K/ 1364K
sor	14K	56K / 49K	69K / 118K	292K / 302K	702K/ 775K

Table D.2: Estimated total cell area for IBM SA-27E

The reported IBM cell area for one through sixteen tiles, with and without router synthesis range from $7K$ to $3.8M$. For up to eight tiles the router cases consumed more area; at sixteen tiles the crossbars cases consumed more area. This cell area does not include the area of the router, only the area of the main tile.

Benchmark	One Tile	Two Tiles router / crossbar	Four Tiles router / crossbar	Eight Tiles router / crossbar	Sixteen Tiles router / crossbar
intmatmul-v1	4.4 mW	15.0 / 21.1 mW	51.2 / 78.4 mW	129.7 / 202.8 mW	351.3 / 634.3 mW
intmatmul-v2	4.0 mW	16.0 / 16.0 mW	47.1 / 66.4 mW	89.5 / 155.3 mW	247.6 / 470.4 mW
jacobi	3.6 mW	13.0 / 15.7 mW	28.7 / 34.1 mW	58.5 / 93.8 mW	106.8 / 223.2 mW
mpegcorr	5.3 mW	11.4 / 16.3 mW	24.9 / 37.5 mW	45.2 / 94.3 mW	116.2 / 255.5 mW
life-v2	4.4 mW	18.1 / 18.8 mW	35.6 / 43.8 mW	71.9 / 108.2 mW	132.0 / 241.8 mW
parity-v2	2.1 mW	6.7 / 7.5 mW	11.2 / 14.8 mW	30.8 / 48.1 mW	49.2 / 134.6 mW
pmatch-v1	3.4 mW	16.8 / 18.0 mW	38.1 / 55.9 mW	73.2 / 170.4 mW	247.7 / 472.2 mW
parity-v2	4.8 mW	15.8 / 18.8 mW	40.5 / 61.6 mW	87.6 / 158.7 mW	256.0 / 549.1 mW
sor	6.1 mW	18.3 / 16.3 mW	39.5 / 47.3 mW	63.7 / 134.5 mW	140.3 / 397.2 mW

Table D.3: Estimated total power for IBM SA-27E

Total power includes power for all tiles. Estimates do not include RAM power or power for routing logic.

D.2 *Mode I* Advanced Results

Section 6.5.2 refers to the data in Tables D.4, D.5, D.6, D.7 and D.8.

Benchmark	One Tile	Two Tiles router / crossbar	Four Tiles router / crossbar	Eight Tiles router / crossbar	Sixteen Tiles router / crossbar
intmatmul-v1	46MHZ	38 / 37 MHZ	34 / 31 MHZ	29 / 27 MHZ	23 / 24 MHZ
intmatmul-v2	47MHZ	34 / 34 MHZ	32 / 33 MHZ	26 / 29 MHZ	24 / 30 MHZ
jacobi	100MHZ	66 / 53 MHZ	63 / 58 MHZ	53 / 62 MHZ	51 / 67 MHZ
mpegcorr	95MHZ	77 / 89 MHZ	67 / 63 MHZ	53 / 55 MHZ	44 / 54 MHZ
life-v2	97MHZ	56 / 55 MHZ	52 / 48 MHZ	47 / 55 MHZ	43 / 53 MHZ
parity-v2	108MHZ	81 / 83 MHZ	93 / 96 MHZ	82 / 68 MHZ	65 / 63 MHZ
pmatch-v1	93MHZ	59 / 63 MHZ	59 / 56 MHZ	42 / 49 MHZ	38 / 68 MHZ
pmatch-v2	101MHZ	63 / 69 MHZ	52 / 47 MHZ	50 / 57 MHZ	44 / 51 MHZ
sor	100MHZ	79 / 60 MHZ	63 / 59 MHZ	43 / 58 MHZ	50 / 55 MHZ

Table D.4: Clock frequency after parallelization

Comparison of clock speed with (router) and without (crossbar) router specialization.

Benchmark	One Tile	Two Tiles router / crossbar	Four Tiles router / crossbar	Eight Tiles router / crossbar	Sixteen Tiles router / crossbar
intmatmul-v1	711	2302 / 2827	5071 / 4891	7908 / 6547	14677 / 15047
intmatmul-v2	725	4422 / 3993	4521 / 4181	5488 / 6462	15905 / 14220
jacobi	601	1205 / 1145	1326 / 1106	1345 / 1299	1374 / 1507
mpegcorr	416	878 / 854	1115 / 1038	1115 / 1189	1256 / 1237
life-v2	784	1695 / 1323	1903 / 1582	1977 / 2088	1808 / 1761
parity-v2	189	363 / 373	353 / 292	553 / 481	502 / 546
pmatch-v1	254	1447 / 1156	2036 / 3906	2517 / 4478	2896 / 2896
pmatch-v2	304	1180 / 922	1967 / 2235	2530 / 1929	3839 / 5117
sor	405	1141 / 760	1457 / 1345	1167 / 1891	1491 / 1316

Table D.5: Number of LUTs

The reported area results are for tile zero only. To approximate total area, multiply by the number of tiles in each case. Area does not include small switch area for router case.

D.3 Advanced Sensitivity Results

Section 6.6.1 refers to the data in Tables D.9.

Benchmark	One Tile	Two Tiles router / crossbar	Four Tiles router / crossbar	Eight Tiles router / crossbar	Sixteen Tiles router / crossbar
intmatmul-v1	252	415 / 470	627/ 581	627 / 617	546/ 809
intmatmul-v2	230	419 / 388	534/ 524	481 / 671	537/ 1061
jacobi	239	374 / 386	357 / 406	303 / 530	302 / 538
mpegcorr	281	384 / 380	417 / 461	384 / 552	440 / 837
life-v2	245	497 / 404	437 / 404	443 / 576	440 / 768
parity-v2	153	207 / 229	196 / 183	280 / 276	205 / 416
pmatch-v1	300	625 / 585	598 / 642	598 / 775	590 / 1918
pmatch-v2	373	589 / 486	649 / 649	598 / 1416	590 / 1118
sor	257	423 / 302	432 / 483	270 / 746	283 / 274

Table D.6: Number of registers

The reported register usage results are for tile zero only. To approximate total register usage, multiply by the number of tiles in each case. Register count does not include small switch area for router case.

Benchmark	One Tile	Two Tiles router / crossbar	Four Tiles router / crossbar	Eight Tiles router / crossbar	Sixteen Tiles router / crossbar
intmatmul-v2	10.8%	10.9 / 15.2%	12.5 / 12.7%	10.8 / 11.4%	13.2 / 14.0%
intmatmul-v2	9.7%	11.1 / 11.3%	12.5 / 12.8%	14.5 / 13.0%	12.0 / 15.0%
jacobi	10.2%	11.0 / 13.7%	11.6 / 12.3%	11.2 / 14.3%	10.9 / 14.2%
mpegcorr	9.8%	9.2 / 9.7%	10.0 / 11.4%	9.6 / 15.8%	11.2 / 10.9%
life-v2	9.9%	11.4 / 12.6%	11.7 / 15.0%	11.1 / 13.7%	10.68/ 12.5%
parity-v2	7.9%	9.6 / 10.5%	6.9 / 8.5%	8.7 / 12.6%	7.8 / 11.2%
pmatch-v1	9.3%	7.6 / 14.4%	11.3 / 18.1%	12.8 / 14.4%	12.8 / 12.7%
pmatch-v2	13.0%	10.0 / 14.3%	10.2 / 12.4%	11.6 / 12.0%	12.7 / 14.1%
sor	11.7%	12.8 / 11.7%	10.4 / 11.3%	11.9 / 13.1%	11.9 / 11.8%

Table D.7: Estimated switching activity

Switching activity is generated with the dynamic power estimation technique in Section 5.4.4.

Benchmark	One Tile	Two Tiles router / crossbar	Four Tiles router / crossbar	Eight Tiles router / crossbar	Sixteen Tiles router / crossbar
intmatmul-v2	59 mW	206 / 334 mW	881 / 791 mW	2007 / 1633 mW	7142 / 8155 mW
intmatmul-v2	41 mW	349 / 320 mW	741 / 722 mW	1668 / 1967 mW	7317 / 10270 mW
jacobi	71 mW	200 / 187 mW	410 / 339 mW	675 / 956 mW	1240 / 2332 mW
mpegcorr	63 mW	154 / 181 mW	326 / 329 mW	475 / 856 mW	1011 / 1213 mW
life-v2	80 mW	244 / 205 mW	484 / 476 mW	852 / 1288 mW	1347 / 1904 mW
parity-v2	30 mW	73 / 84 mW	109 / 113 mW	337 / 348 mW	421 / 641 mW
pmatch-v1	53 mW	166 / 246 mW	577 / 809 mW	1103 / 1950 mW	2268 / 4119 mW
pmatch-v2	51 mW	186 / 216 mW	453 / 552 mW	1209 / 1139 mW	3455 / 5953 mW
sor	84 mW	264 / 125 mW	407 / 387 mW	490 / 1188 mW	1435 / 1386 mW

Table D.8: Estimated power for Xilinx Virtex-8

For one tile, an accurate estimate of non-RAM power is determined with the Xilinx Virtex Power Estimator, Version 1.5, and the switching activity in Table D.7. For multiple tiles, a very rough estimate of the total power (for all tiles) is determined using the following formula: $Power = (1pJ * TotalRegisters + 10pJ * TotalLuts * SwitchingActivity) * ClockSpeed/1000$, where ClockSpeed is in MHz and Power is in mW. I obtained these conservative parameters experimentally by playing with the Virtex Power Estimator.

Benchmark	One Tile	Two Tiles router / crossbar	Four Tiles route / crossbar	Eight Tiles route / crossbar	Sixteen Tiles route / crossbar
intmatmul-v1	25	116 / 73	180 / 72	341 / 115	667 / 207
jacobi	35	115 / 54	160 / 71	238 / 80	254 / 87
mpegcorr	30	108 / 53	125 / 57	168 / 69	141 / 75
life-v2	54	169 / 84	256 / 106	346 / 124	254 / 106
pmatch-v1	18	115 / 56	204 / 81	455 / 171	1023 / 455
sor	22	87 / 41	113 / 48	158 / 55	179 / 63

Table D.9: Number of states used in scatter plot

This data is used in Figure 6-17 to show the correlation between the number of states and clock speed.

Appendix E

VPR Data and FPGA Layouts

E.1 Input Data and Results

This appendix contains data for the results reported in Section 6.3.2. To generate this data with VPR (the FPGA place and route tool introduced Section 5.3), I have used the architecture file and the commands in Figures E-1, E-2 and E-3.

Table E.1 contains three related tables. The first table includes the CLB count, minimum routing channel requirements, and the critical path timing. A CLB is a cluster of four 4-LUTs. The routing requirements, ranging from 25 to 66, were earlier graphed in Figure 6-8. The CLB count ranged from 75 for `parity-v1`, to 1147 for `convolve`. Note that `parity-v1` is small because of the simplicity of the function computed, while convolve is large because the source code is unrolled; the median for the other benchmarks is 261 CLBs. Even though the timing sensitive packer was used, the critical paths are much longer than when placing and routing with Xilinx's tool (see the clock speeds in Table C.4) because they do not use carry chains. The second table contains detailed data for blocks, CLBs, and total tracks, as well as the total nets, inputs and outputs. The routing area, measured in minimum-width transistors, is also in the second table. Low routing area assumes buffer sharing while high routing area does not. In the third table are even more statistics. These statistics include the inputs per CLB, the average and maximum bends in a wire, and the total, average, and maximum wire and segment lengths. Wire and segment length are in CLB units. VPR reports this data upon the completion of each place and route. See the VPR manual for a longer explanation of each item. This data is included here, without further discussion, for the benefit of FPGA architects and backend CAD tool developers.

E.2 Wire Length Distribution and Layouts

Figure E.2 lists the expected value of two-pin net lengths. The actual distribution for each benchmark is in Figure E-4 through E-16. These distributions provide information about the final layouts — longer nets correspond to more global routing. These final layouts are in Figures E-17 to E-29, culminating the evidence for my thesis. Real layouts are visual proof of my thesis — specialization of architectural mechanisms is effective in compiling high-level programs to gate-reconfigurable architectures.

Benchmark	Routing Channels	Total CLBs (4 4-Luts)	Critical Path (logic, net)
adpcm	43	632 (26x26)	276.5 (13.5, 263.0) ns
bubblesort	31	200 (15x15)	51.9 (8.5, 43.5) ns
convolve	66	1147 (34x34)	223.4 (7.3, 216.1) ns
histogram	28	286 (17x17)	66.2 (2.5, 63.8) ns
intfir	29	221 (15x15)	44.4 (13.2, 31.2) ns
intmatmul-v1	31	272 (17x17)	49.1 (6.5, 42.7) ns
intmatmul-v2	32	284 (17x17)	80.6 (6.5, 74.1) ns
jacobi	39	202 (15x15)	52.1 (8.5, 43.6) ns
life-v2	41	256 (16x16)	87.9 (5.4, 82.4) ns
median	43	399 (20x20)	49.7 (3.5, 46.2) ns
mpegcorr	34	303 (18x18)	53.7 (12.5, 41.3) ns
parity-v1	25	75 (9x9)	15.2 (6.5, 8.7) ns
parity-v2	25	82 (10x10)	36.1 (9.5, 2.7) ns
pmatch-v1	29	120 (11x11)	33.1 (5.5, 27.7) ns
pmatch-v2	29	165 (13x13)	22.6 (1.5, 21.1) ns
sor	33	267 (17x17)	211.8 (3.5, 208.3) ns

Benchmark	Blocks/clbs/tracks	nets/in/out	Routing area (min width transistor) total / perCLB
adpcm	839/632/1161	1840/45/162	2.61M-2.81M / 3865-4159
bubblesort	411/200/496	683/66/145	776K-843K / 3448-3748
convolve	1320/1147/2310	3196/38/135	6.25M-6.71M / 5408-5800
histogram	507/286/504	852/52/169	870K-941K / 3011/3255
intfir	422/221/464	727/51/150	748K-812K / 3224-3609
intmatmul-v1	485/272/558	875/53/160	952K-1.03M / 3296-3570
intmatmul-v2	497/284/576	887/53/160	990K-1.07M / 3424-3709
jacobi	341/203/624	602/22/116	958K-1.05M / 4258-4646
life-v2	358/256/697	733/8/94	1.09M-1.19M / 4276-4660
median	937/399/903	1284/194/344	1.67M-1.80M / 4164-4511
mpegcorr	559/303/646	938/70/106	1.12M-1.22M / 3461-3757
parity-v1	208/75/210	282/34/99	246K-270K / 3035-3328
parity-v2	184/82/275	274/8/94	348K-381K / 3482-3812
pmatch-v1	361/120/348	465/69/172	457K-501K / 3777-4134
pmatch-v2	464/165/406	588/98/201	594K-648K / 3517-3834
sor	393/267/594	763/19/107	1.01M-1.10M / 3499-3793

Benchmark	inputs per CLB	bends avg/max	Wire Length Tot/Avg/Max	Segments Tot/Avg/Max
adpcm	6.62	2.61/226	36446/19.8/607	12236/6.65/463
bubblesort	6.63	2.04/71	9102/13.3/265	3293/4.83/143
convolve	7.37	3.42/189	95671/29.9/979	29439/9.21/455
histogram	6.16	2.00/90	11298/13.3/455	4121/4.84/194
intfir	6.72	1.99/49	9077/12.1/145	3465/4.77/97
intmatmul-v1	6.77	2.11/37	12270/14.0/191	4379/5.0/77
intmatmul-v2	6.93	2.26/42	12643/14.3/198	4739/5.3/88
jacobi	8.27	3.01/75	11656/19.3/294	4211/7.1/148
life-v2	8.55	3.27/118	14781/20.1/304	5502/7.51/221
median	7.88	1.80/145	23515/18.3/588	8560/6.67/295
mpegcorr	6.42	2.25/63	14249/15.2/288	5002/5.33/124
parity-v1	6.37	1.47/29	2206/7.9/112	953/3.39/57
parity-v2	7.02	2.25/38	3145/11.5/131	1310/4.80/73
pmatch-v1	6.75	1.86/30	4837/10.4/145	1950/4.2/66
pmatch-v2	6.19	1.66/56	6025/10.3/196	2229/3.8/108
sor	7.06	2.56/64	12725/16.7/280	4603/6.0/129

Table E.1: VPR statistics

```
# NB:  The timing numbers in this architecture file have been modified
# to comply with our NDA with the foundry providing us with process
# information.  The critical path delay output by VPR WILL NOT be accurate,
# as we have intentionally altered the delays to introduce inaccuracy.
# The numbers are reasonable enough to allow CAD experimentation,
# though.  If you want real timing numbers, you'll have to insert your own
# process data for the various, R, C, and Tdel entries.

# Architecture with two types of routing segment.  The routing is
# fully-populated.  One length of segment is buffered, the other uses pass
# transistors.

# Uniform channels.  Each pin appears on only one side.
io_rat 20
chan_width_io 1
chan_width_x uniform 1
chan_width_y uniform 1

# Cluster of size 4, with 10 logic inputs.
inpin class: 0 bottom
inpin class: 0 left
inpin class: 0 top
inpin class: 0 right
inpin class: 0 bottom
inpin class: 0 left
inpin class: 0 top
inpin class: 0 right
inpin class: 0 bottom
inpin class: 0 left
outpin class: 1 top
outpin class: 1 right
outpin class: 1 bottom
outpin class: 1 left
inpin class: 2 global top   # Clock, global -> routed on a special resource.

# Class 0 -> logic cluster inputs, Class 1 -> Outputs, Class 2 -> clock.

subblocks_per_clb 4
subblock_lut_size 4

#parameters needed only for detailed routing.
switch_block_type subset
Fc_type fractional
Fc_output 0.5
Fc_input 0.5
Fc_pad 1
```

Figure E-1: VPR architecture file, virtex.arch

```
# All comments about metal spacing, etc. assumed have been deleted
# to protect our foundry.  Again, the R, C and Tdel values have been
# altered from their real values.

segment frequency: 0.3 length: 1 wire_switch: 0 opin_switch: 1 Frac_cb: 1. \
        Frac_sb: 1. Rmetal: 1.35 Cmetal: 0.4e-15
segment frequency: 0.6 length: 6 wire_switch: 0 opin_switch: 1 Frac_cb: 0.5 \
        Frac_sb: 0.5714 Rmetal: 1.35 Cmetal: 0.4e-15
segment frequency: 0.1 length: longline wire_switch: 0 opin_switch: 1 Frac_cb: 1. \
        Frac_sb: 1. Rmetal: 1.35 Cmetal: 0.4e-15

# Pass transistor switch.

switch 0  buffered: no  R: 1600 Cin: 9e-15  Cout: 9e-15 Tdel: 20e-12
switch 1  buffered: yes  R: 0 Cin: 4e-15  Cout: 9e-15  Tdel: 102e-12

# Logic block output buffer used to drive pass transistor switched wires.

# Used only by the area model.

R_minW_nmos 8229
R_minW_pmos 24800

# Timing info below.  See VPR manual for details.

C_ipin_cblock 4e-15
T_ipin_cblock 200e-12
T_ipad 0
T_opad 0
v_sblk_opin_to_sblk_ipin 0
T_clb_ipin_to_sblk_ipin 0
T_sblk_opin_to_clb_opin 0

# Delays for each of the four sequential and combinational elements
# in our logic block.  In this case they're all the same.

T_subblock T_comb: 1e-9 T_seq_in: .25e-9 T_seq_out: 1.2e-9
T_subblock T_comb: 1e-9 T_seq_in: .25e-9 T_seq_out: 1.2e-9
T_subblock T_comb: 1e-9 T_seq_in: .25e-9 T_seq_out: 1.2e-9
T_subblock T_comb: 1e-9 T_seq_in: .25e-9 T_seq_out: 1.2e-9
```

Figure E-2: Continuation of VPR architecture file

```
CLUSTER_SIZE       = 4
INPUTS_PER_CLUSTER = 10
LUT_SIZE           = 4              # 4 inputs per LUT.
WIDTH              = 46             # max width routing channel for device.
F_s                = 3              # Ordinary planar switchbox
INNER_NUM          = 1
NUM_ROWS           = 11
NUM_COLS           = 11
F_c                = 1              # F_c value for N=1
ARCH_FILE          = virtex.arch    # use this for N=1
ASTAR_FAC          = 1.2
INIT_T             = 10
```

Figure E-3: Settings for VPR

Benchmark	2-pin net length (avg)
adpcm	8.38
bubblesort	6.18
convolve	11.13
histogram	5.85
intfir	5.55
intmatmul-v1	6.13
jacobi	6.49
life-v1	out of memory
life-v2	6.47
median	6.73
mpegcorr	6.68
parity-v1	3.82
pmatch-v1	4.91
sor	6.37

Table E.2: Expected value of 2-pin net lengths

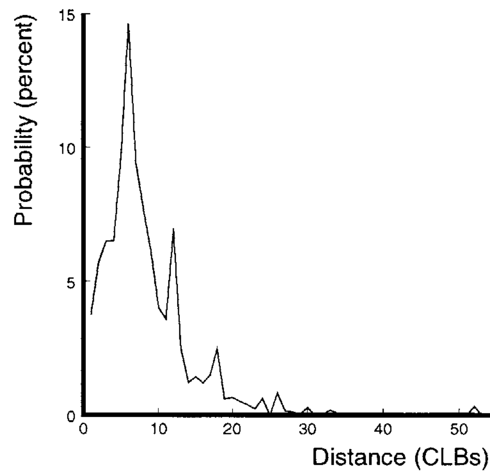


Figure E-4: adpcm wire length distribution

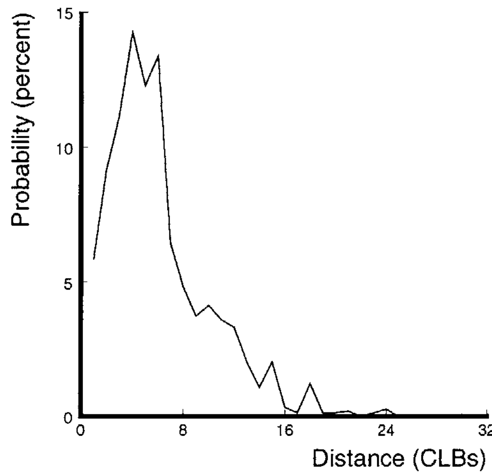


Figure E-5: bubblesort wire length

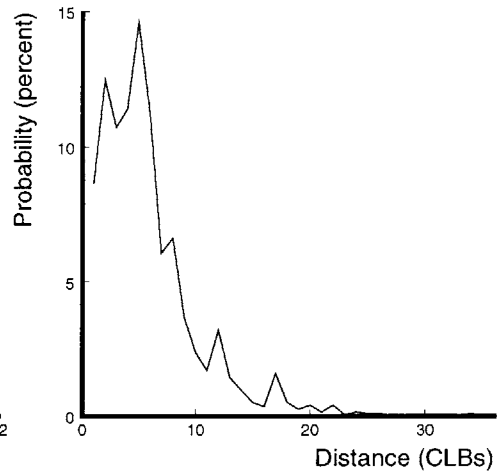


Figure E-7: histogram wire length

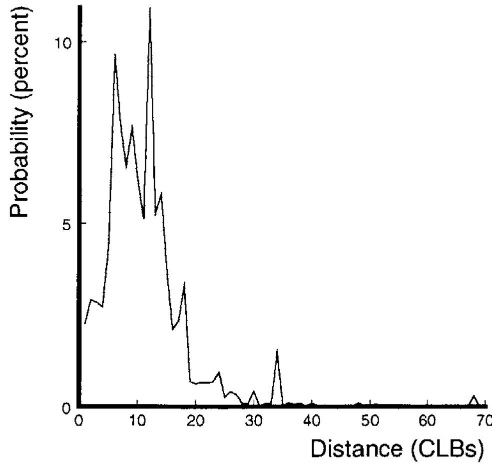


Figure E-6: convolve wire length

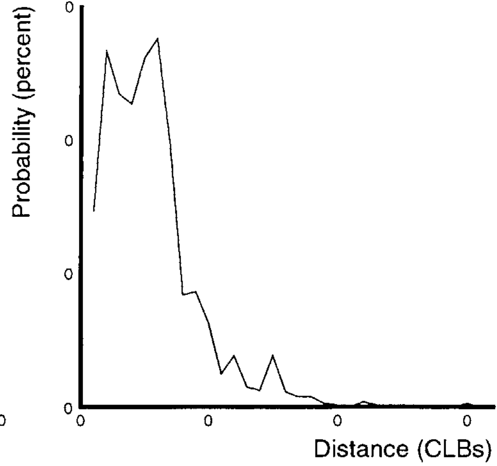


Figure E-8: intfir wire length

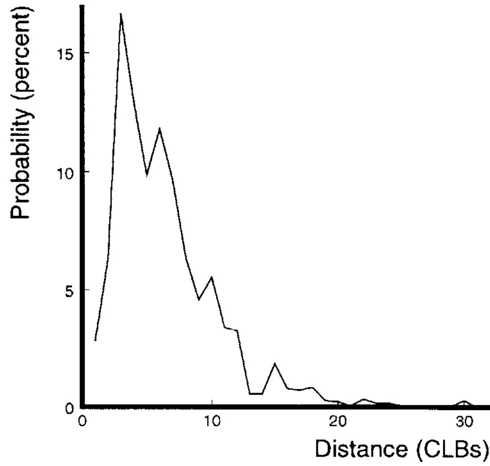


Figure E-9: intmatmul-v1 wire length

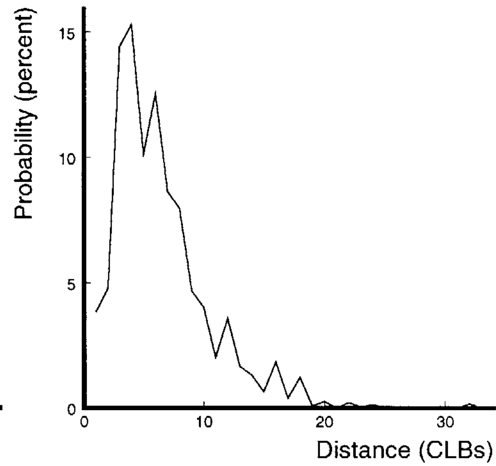


Figure E-11: life-v2 wire length

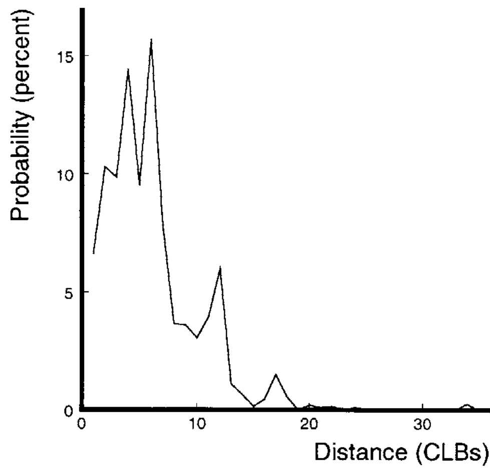


Figure E-10: jacobi wire length

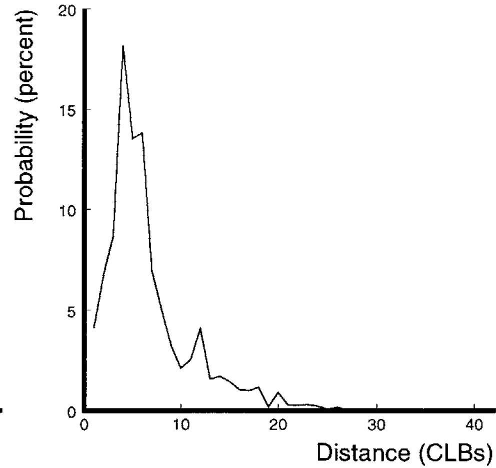


Figure E-12: median wire length

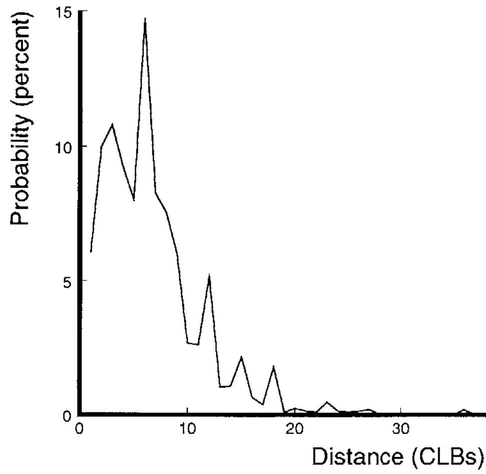


Figure E-13: mpegcorr wire length

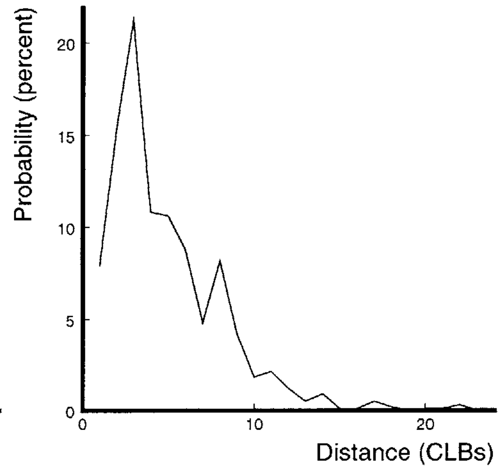


Figure E-15: pmatch-v1 wire length

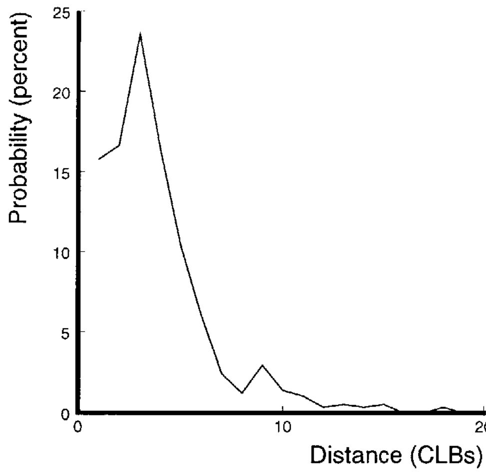


Figure E-14: parity-v1 wire length

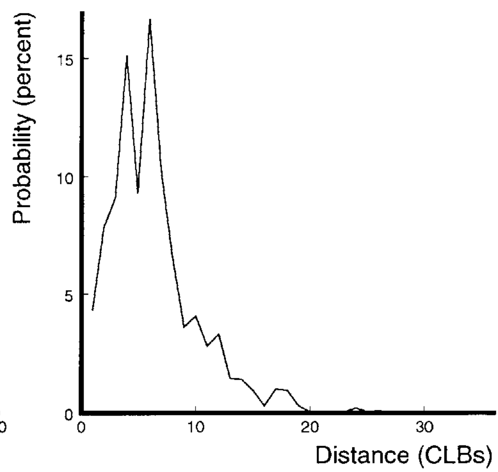
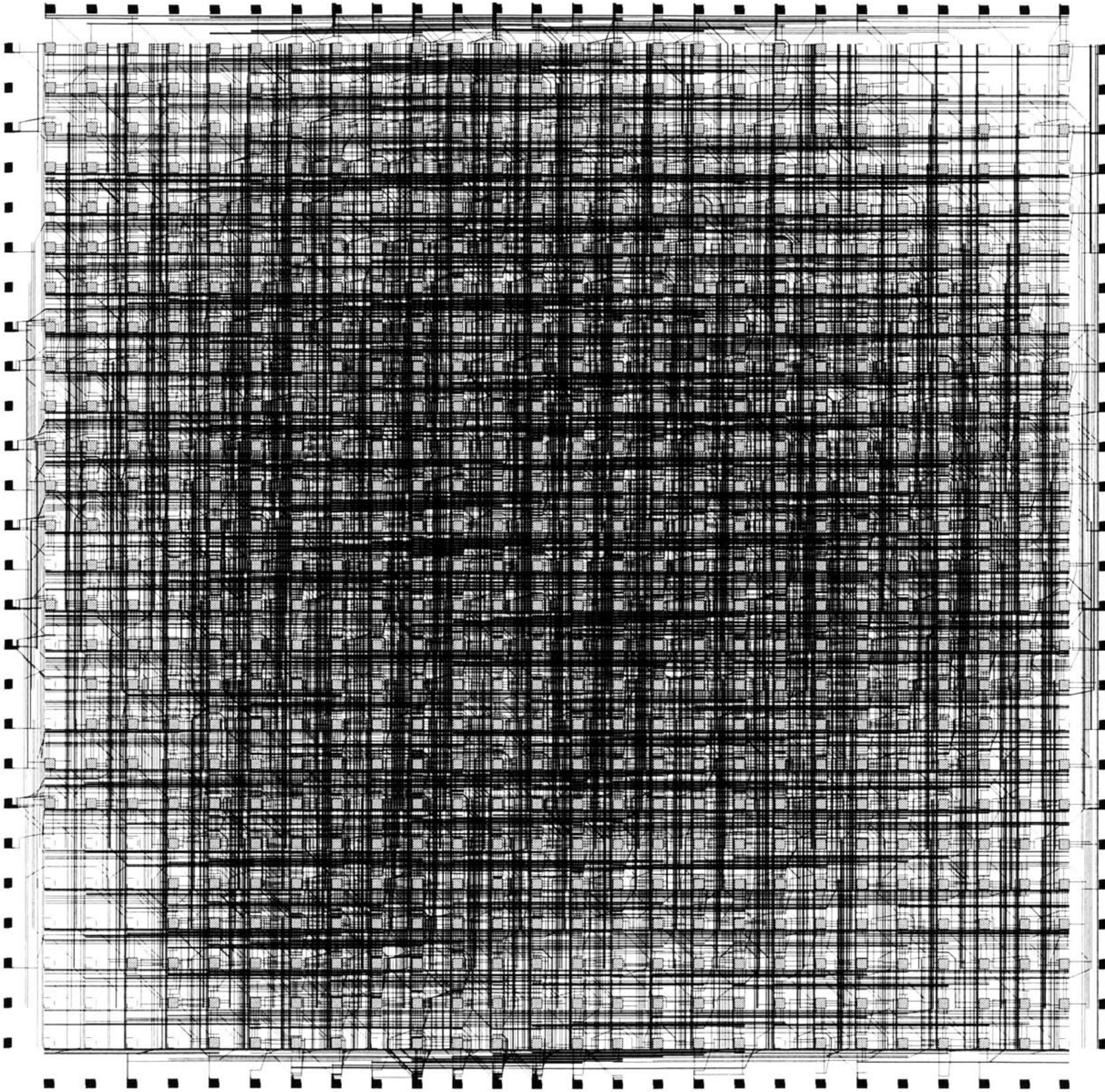
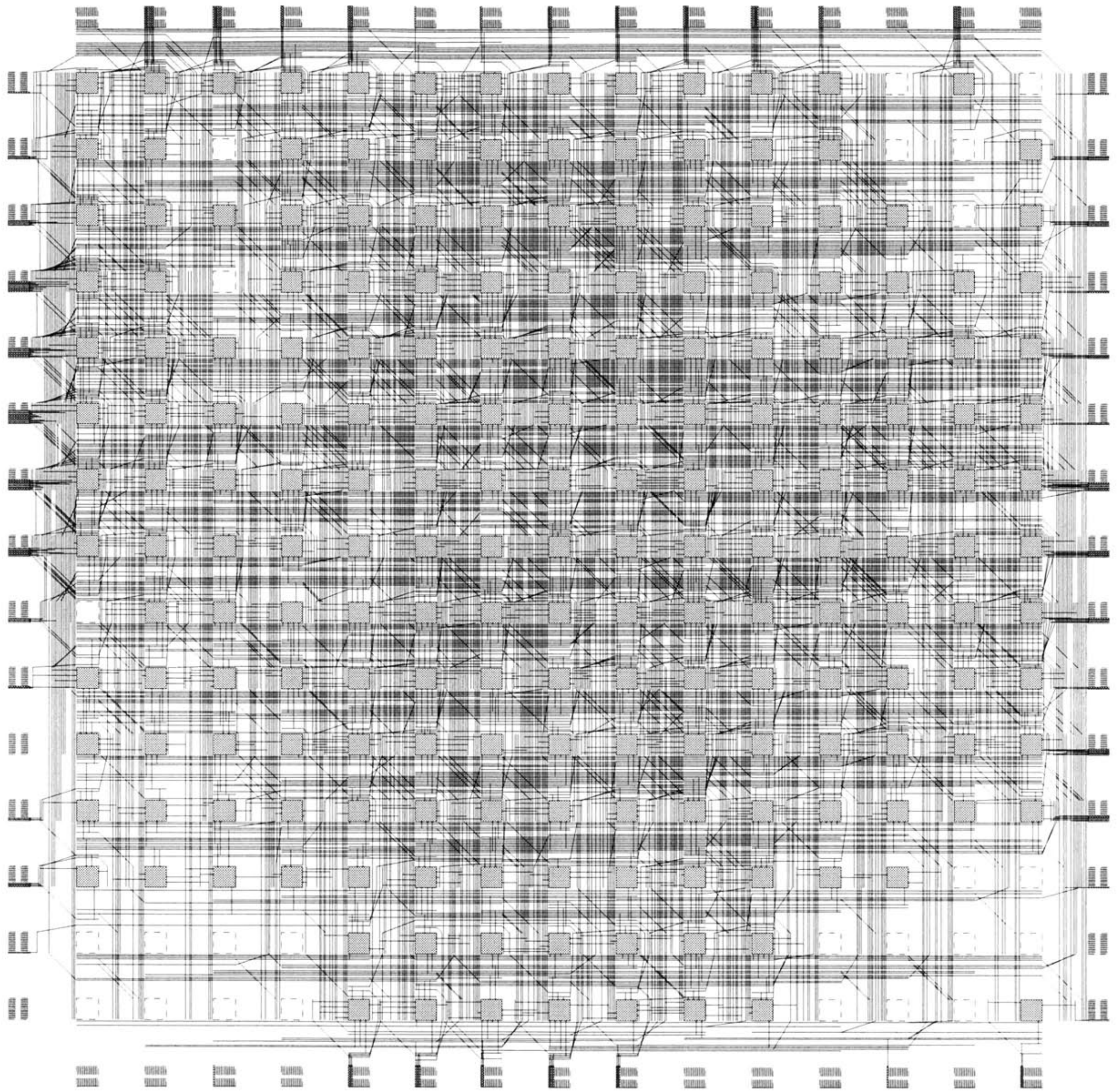


Figure E-16: sor wire length



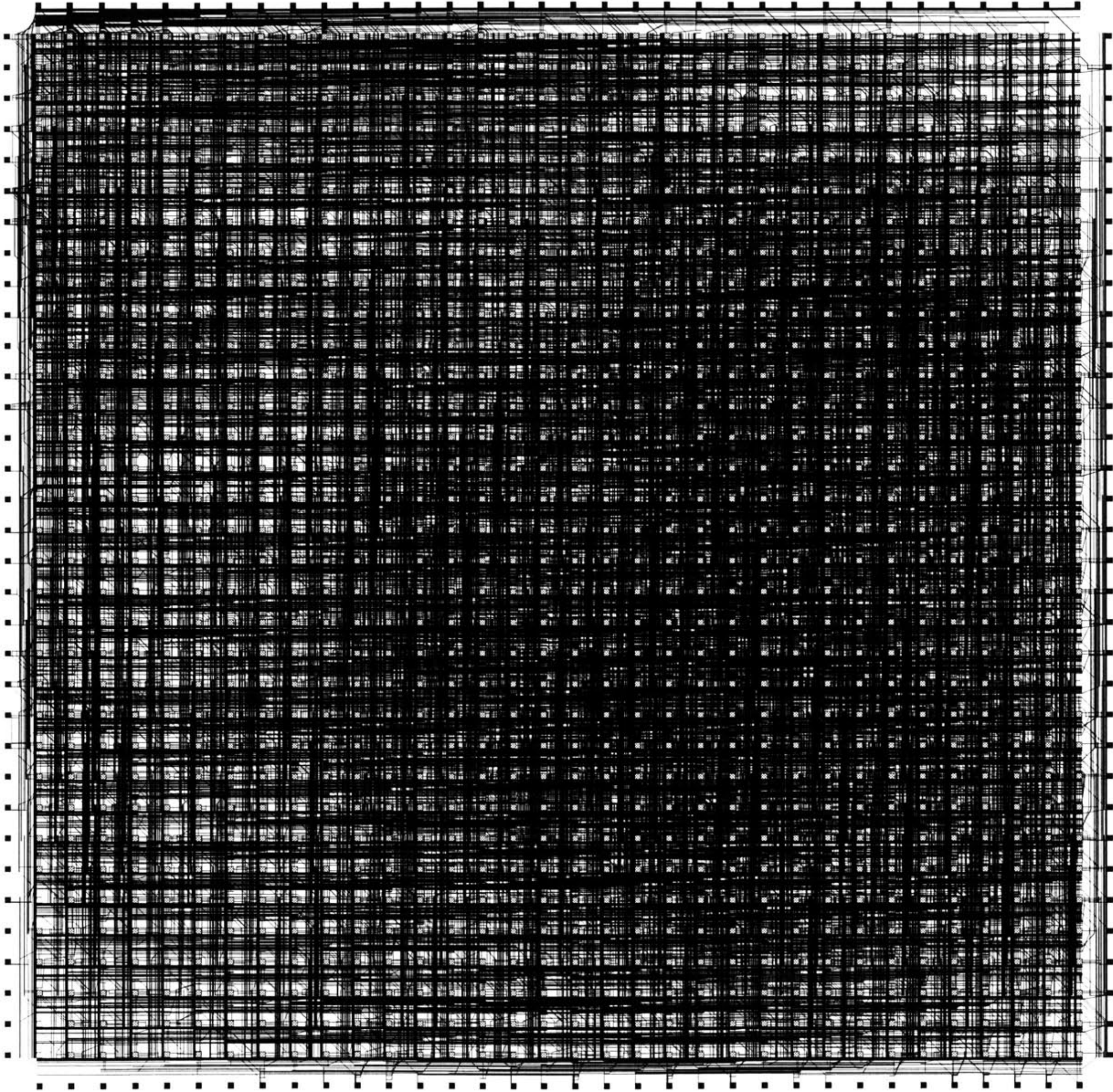
Routing succeeded with a channel width factor of 43.

Figure E-17: adpcm layout



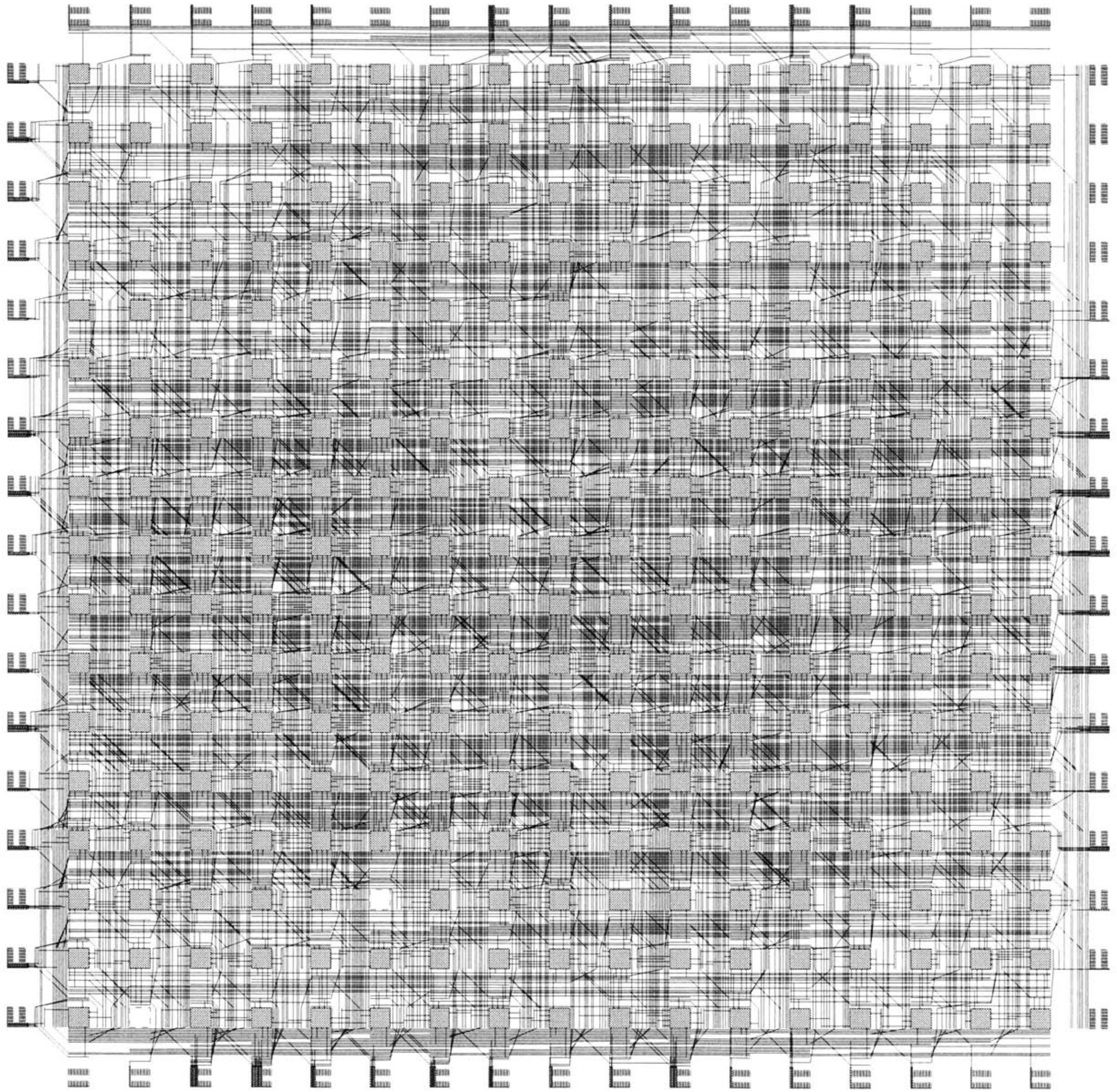
Routing succeeded with a channel width factor of 31.

Figure E-18: bubblesort layout



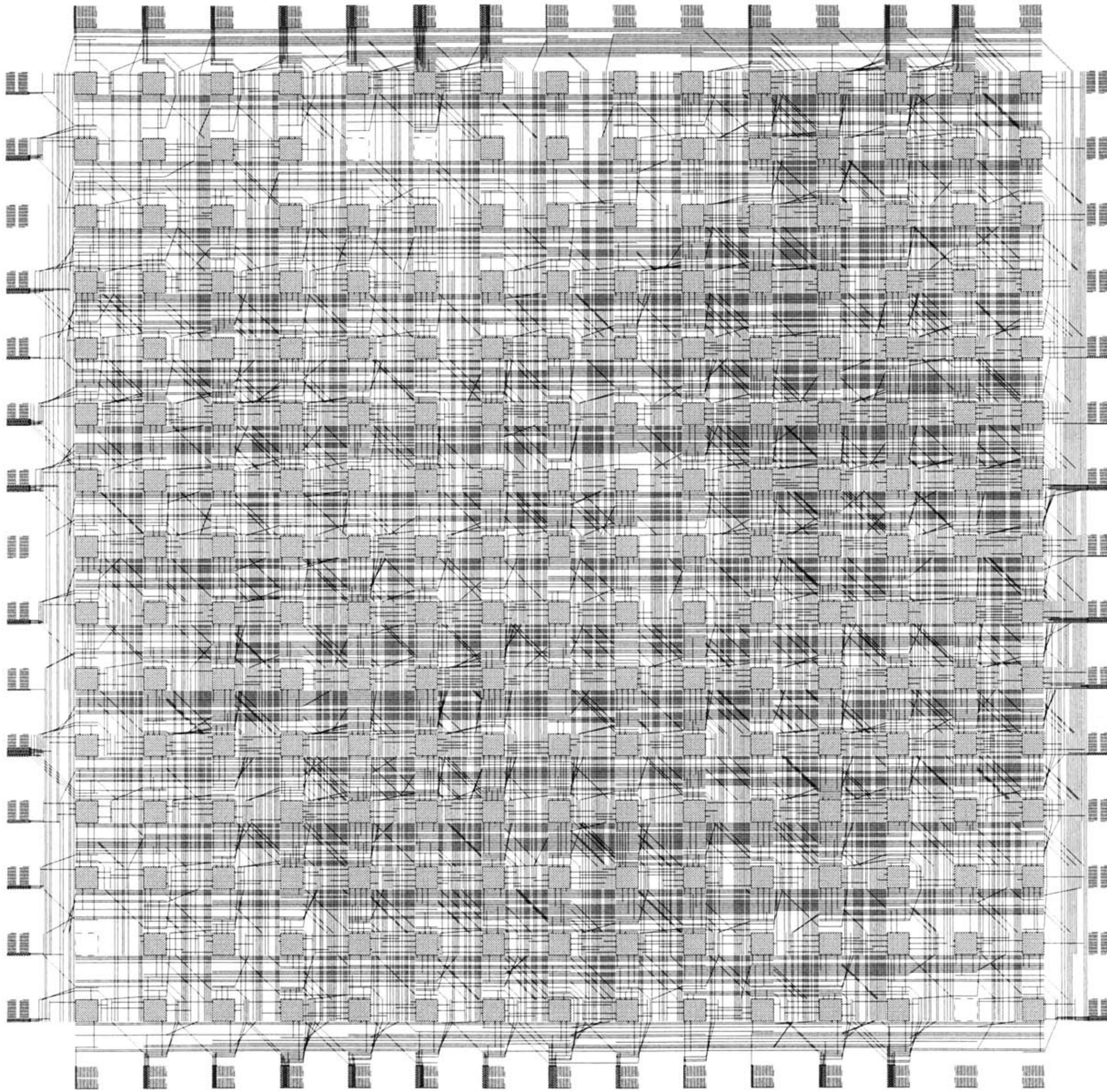
Routing succeeded with a channel width factor of 66.

Figure E-19: convolve layout



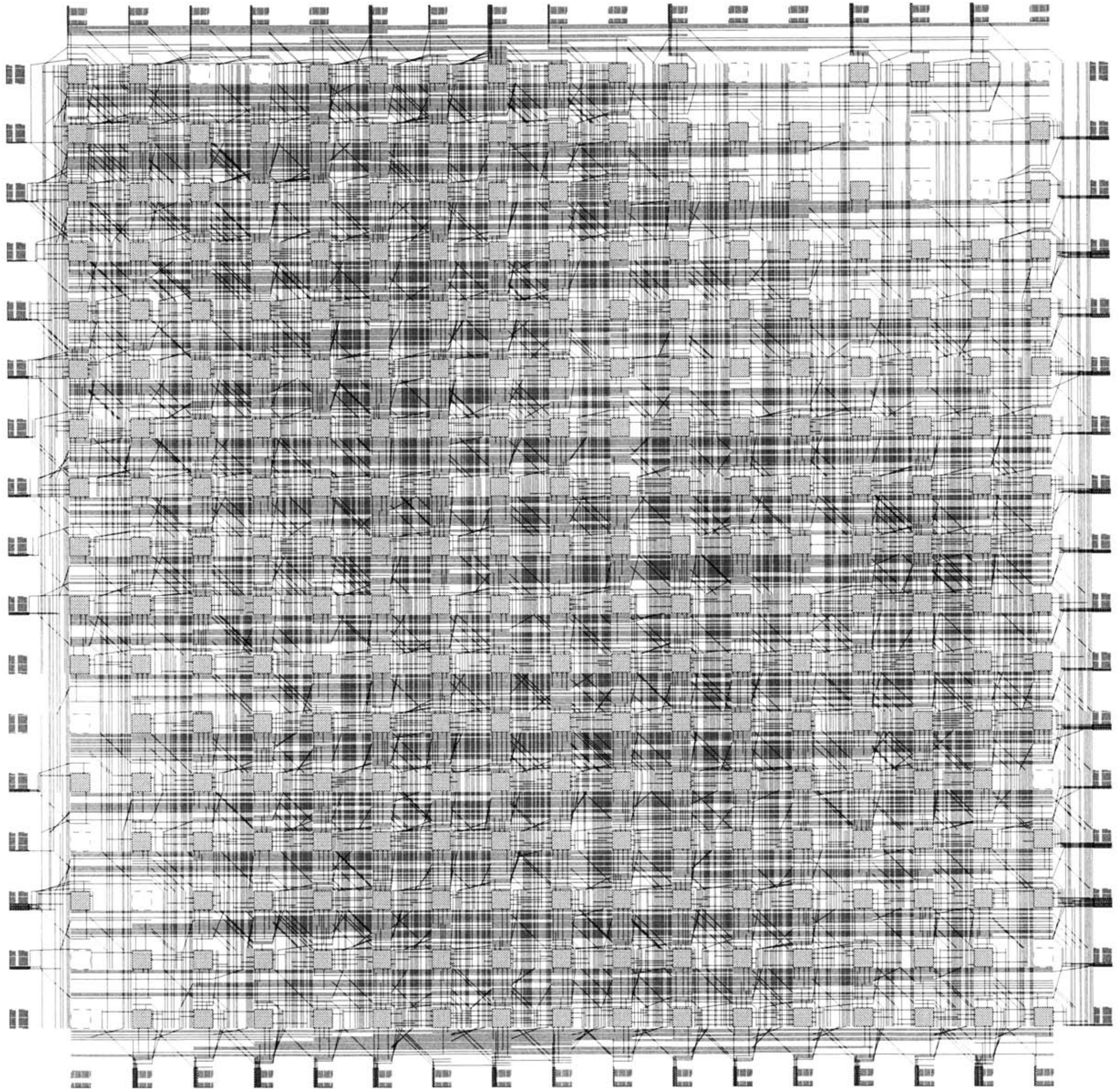
Routing succeeded with a channel width factor of 28.

Figure E-20: histogram layout



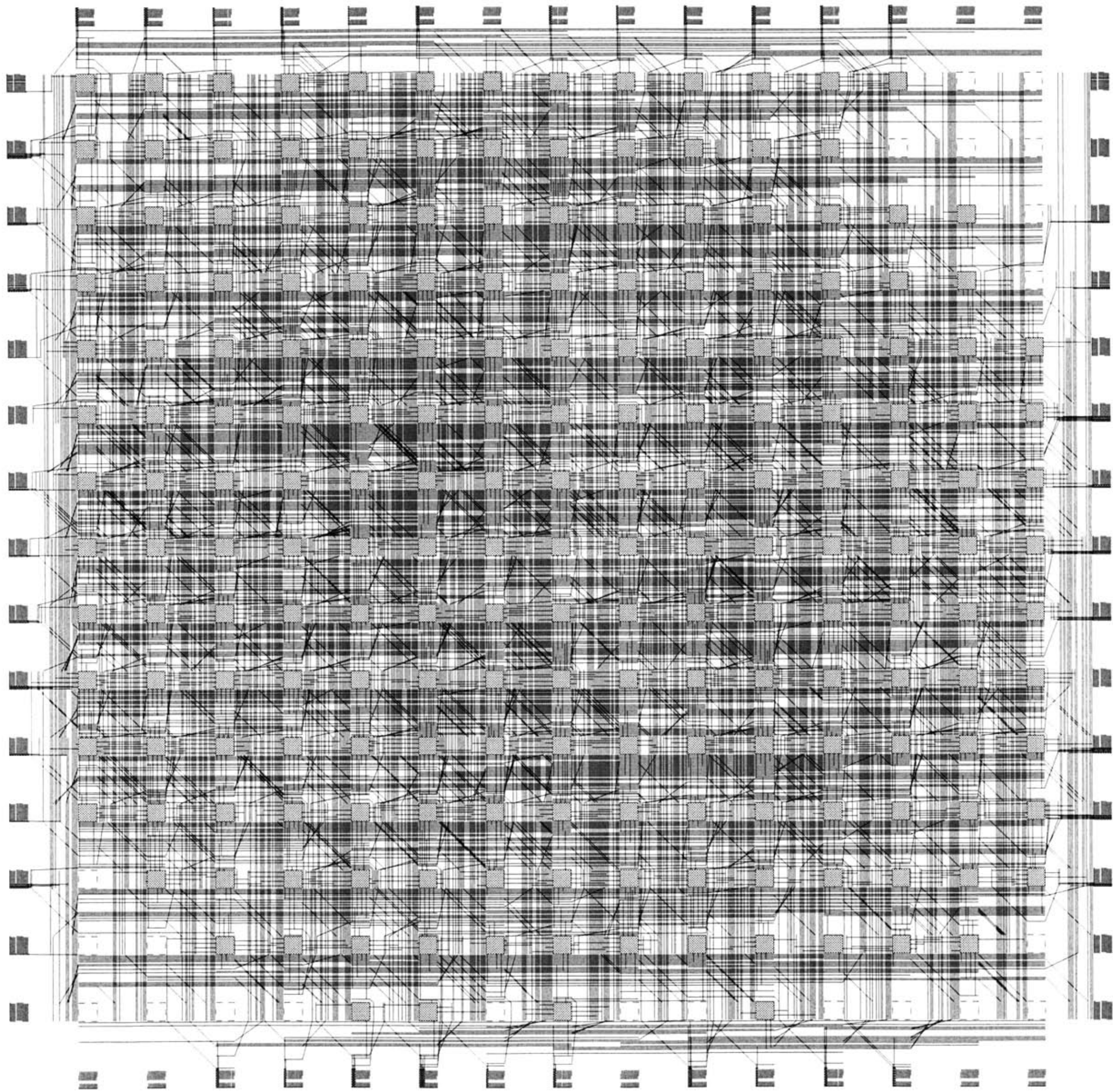
Routing succeeded with a channel width factor of 29.

Figure E-21: intfir layout



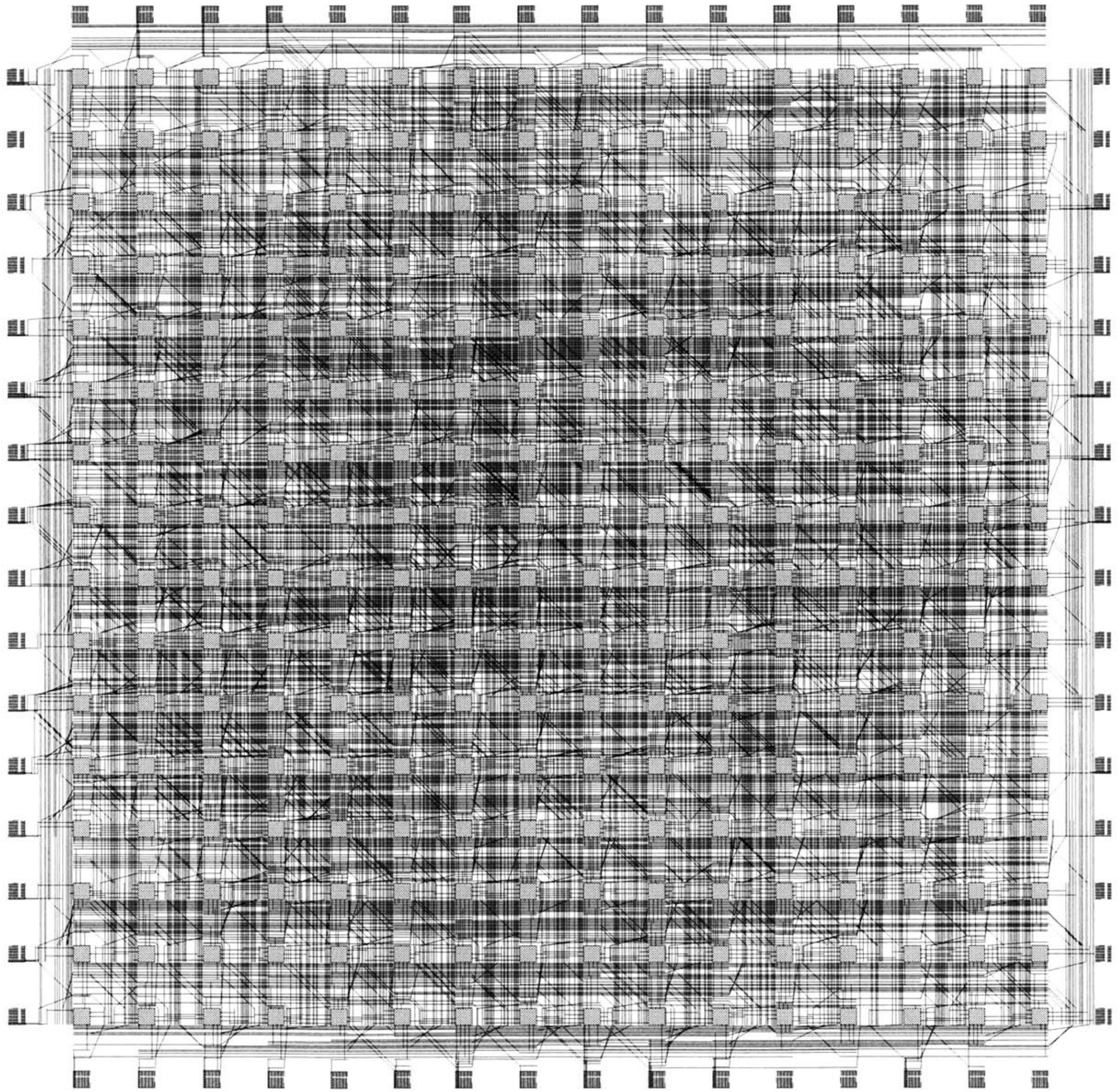
Routing succeeded with a channel width factor of 31.

Figure E-22: intmatmul-v1 layout



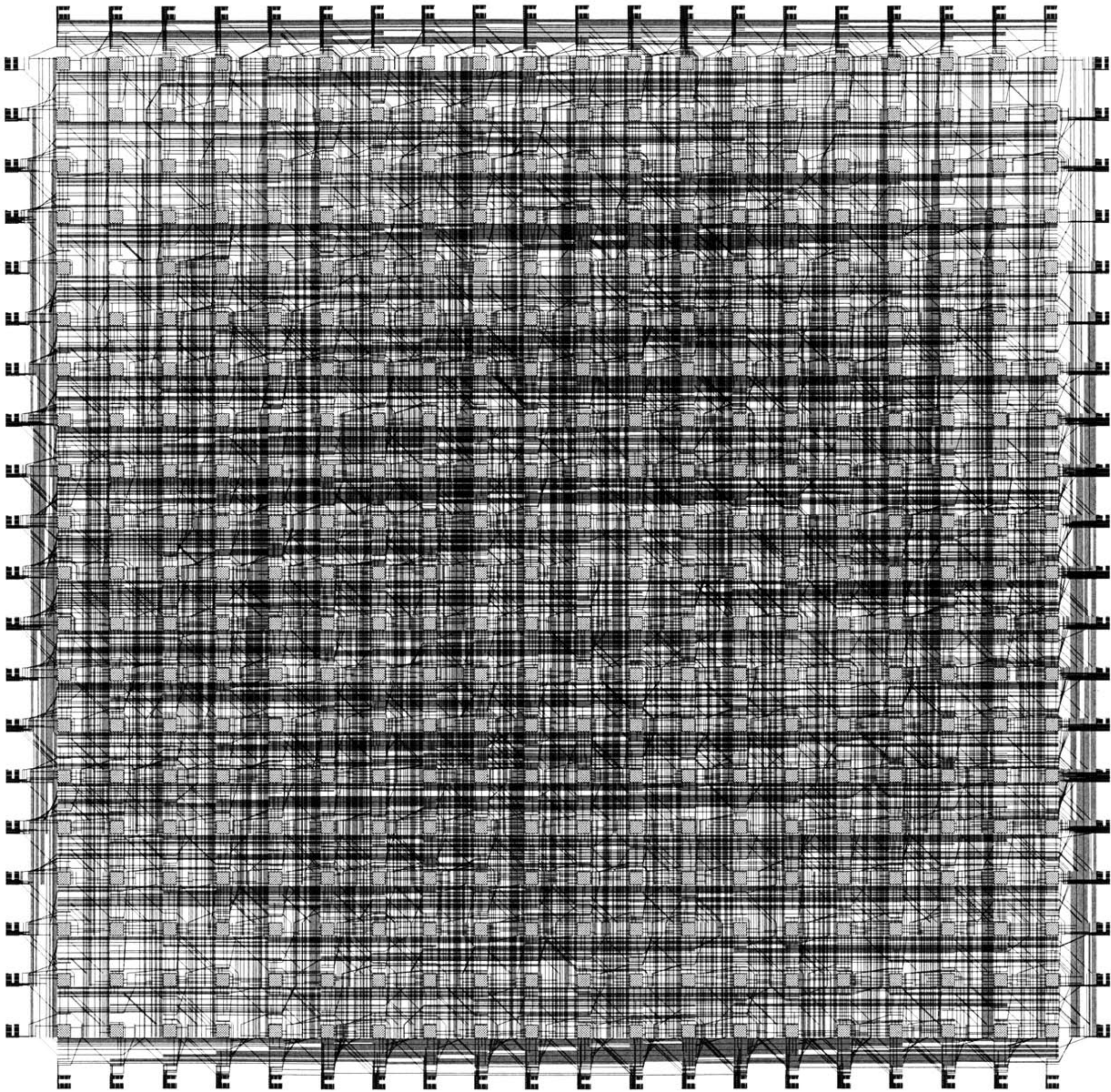
Routing succeeded with a channel width factor of 39.

Figure E-23: jacobi layout



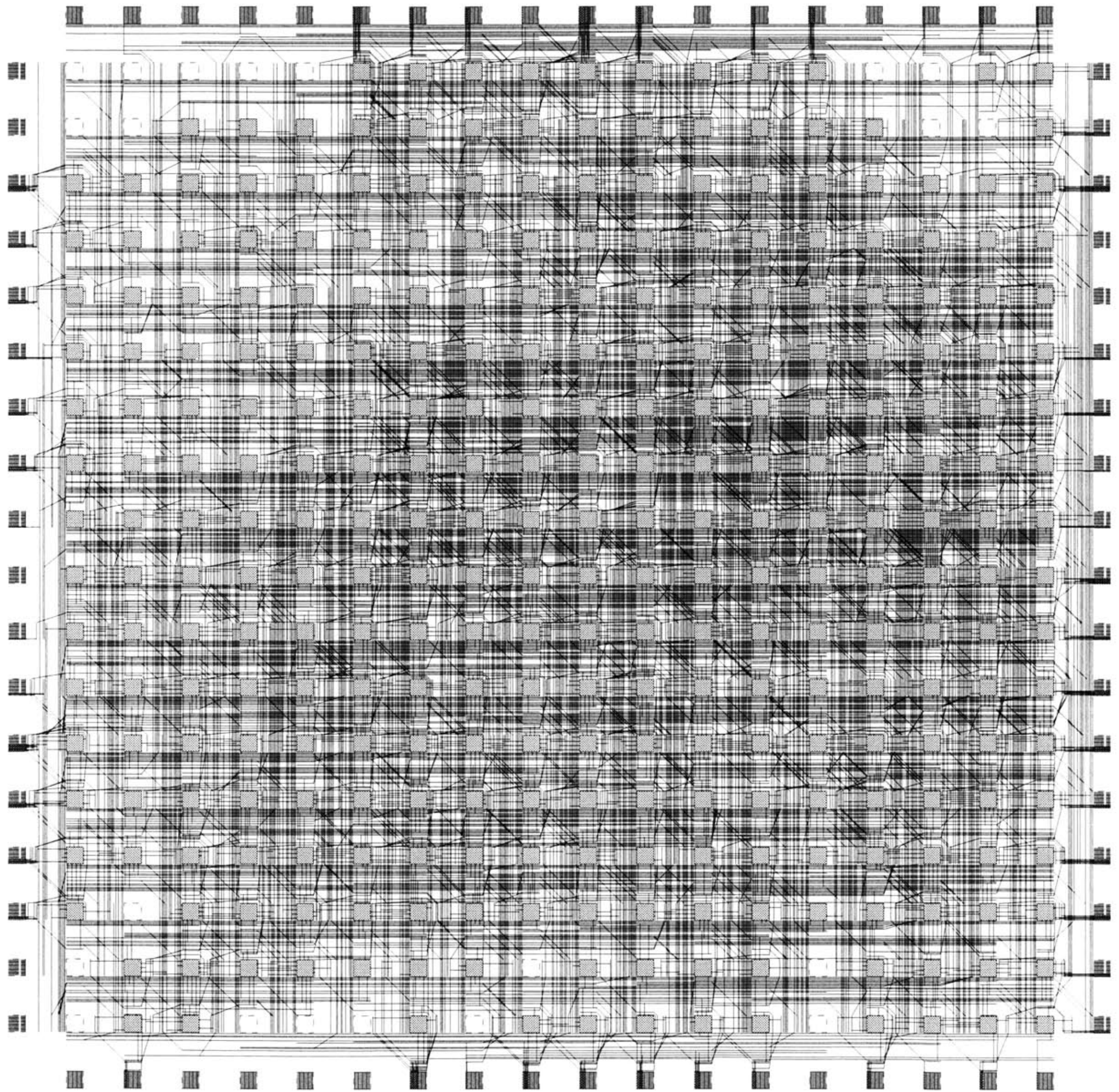
Routing succeeded with a channel width factor of 41.

Figure E-24: life-v2 layout



Routing succeeded with a channel width factor of 43.

Figure E-25: median layout



Routing succeeded with a channel width factor of 34.

Figure E-26: mpegcorr layout

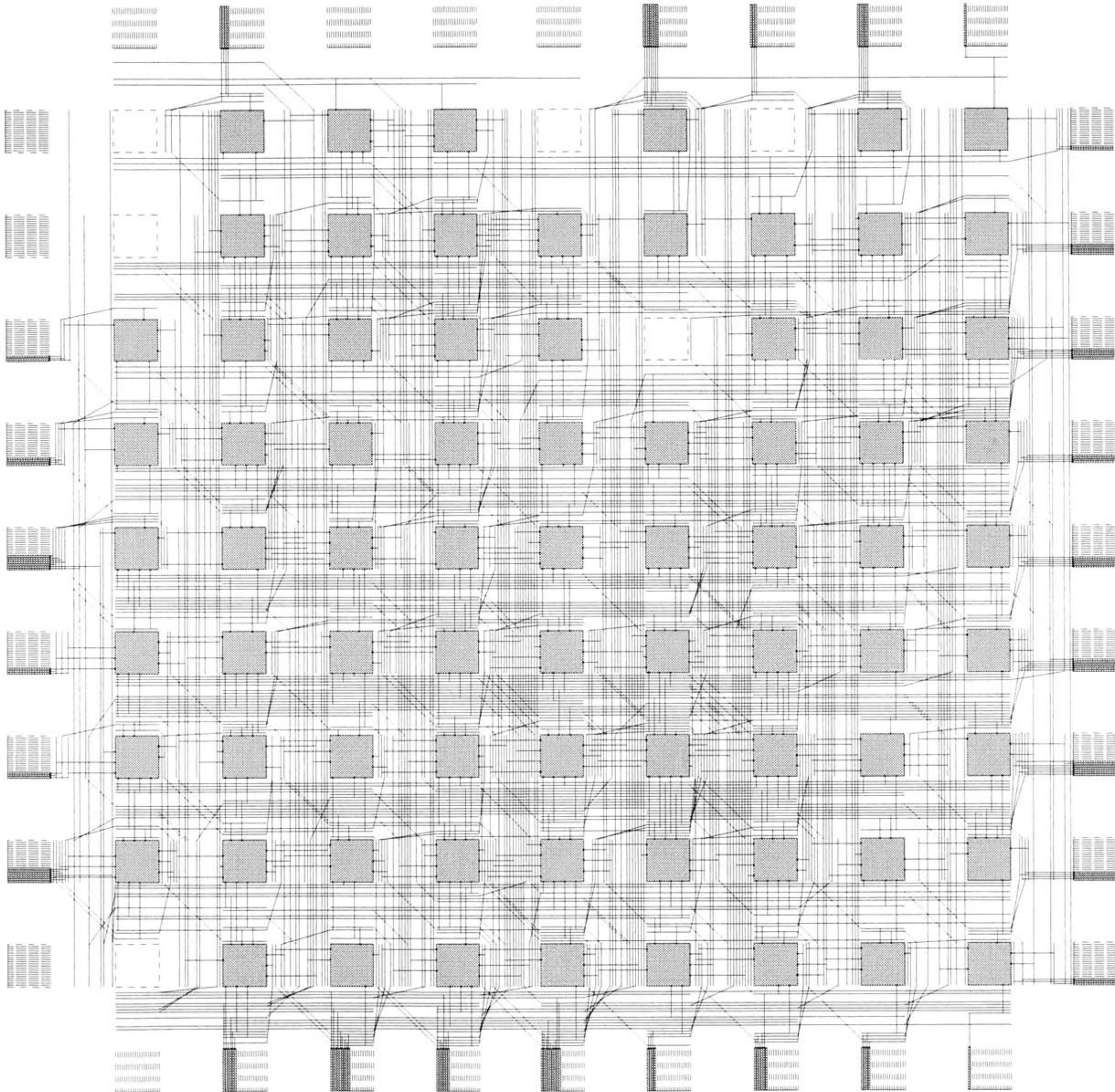


Figure E-27: parity-v1 layout

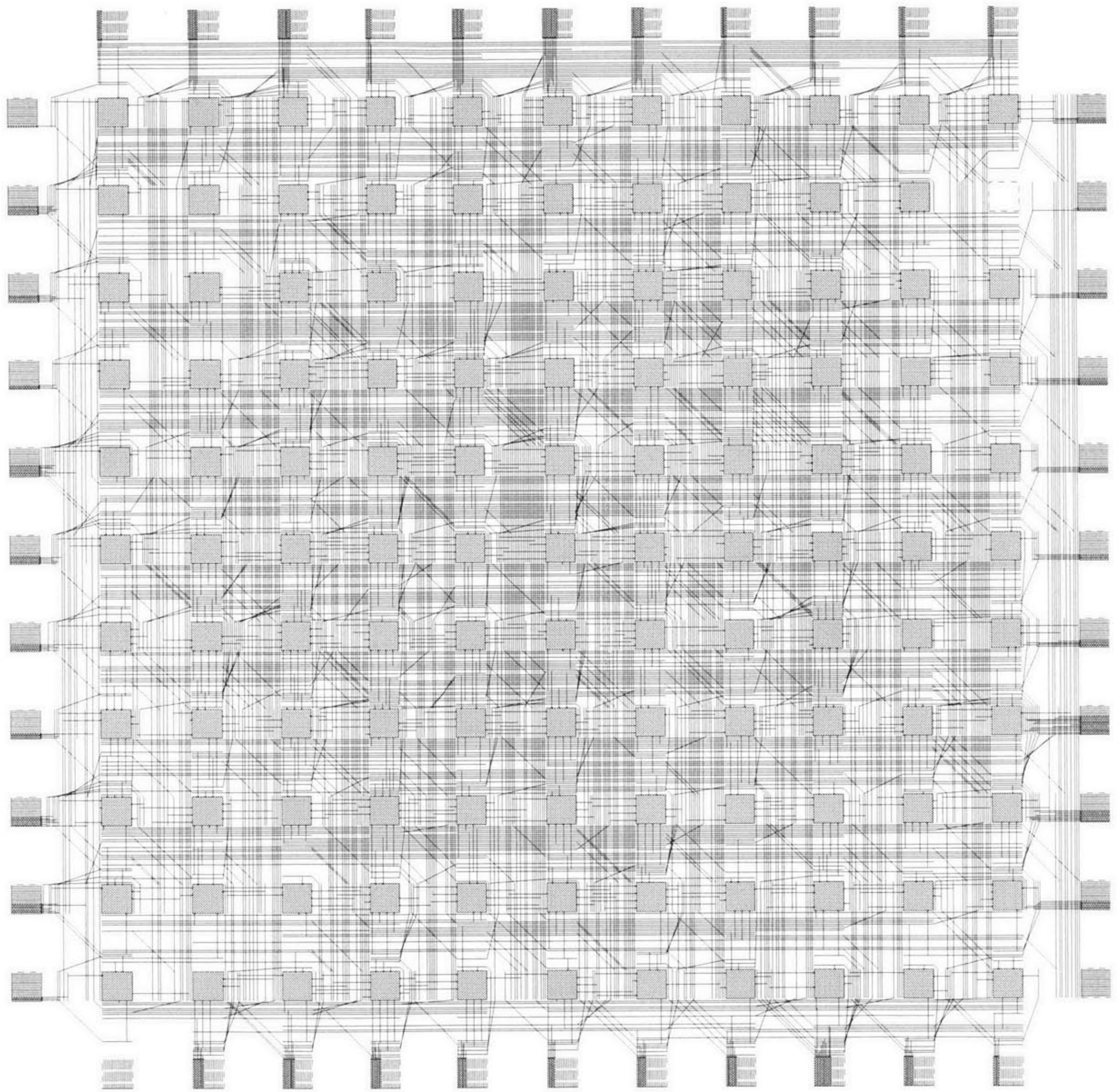
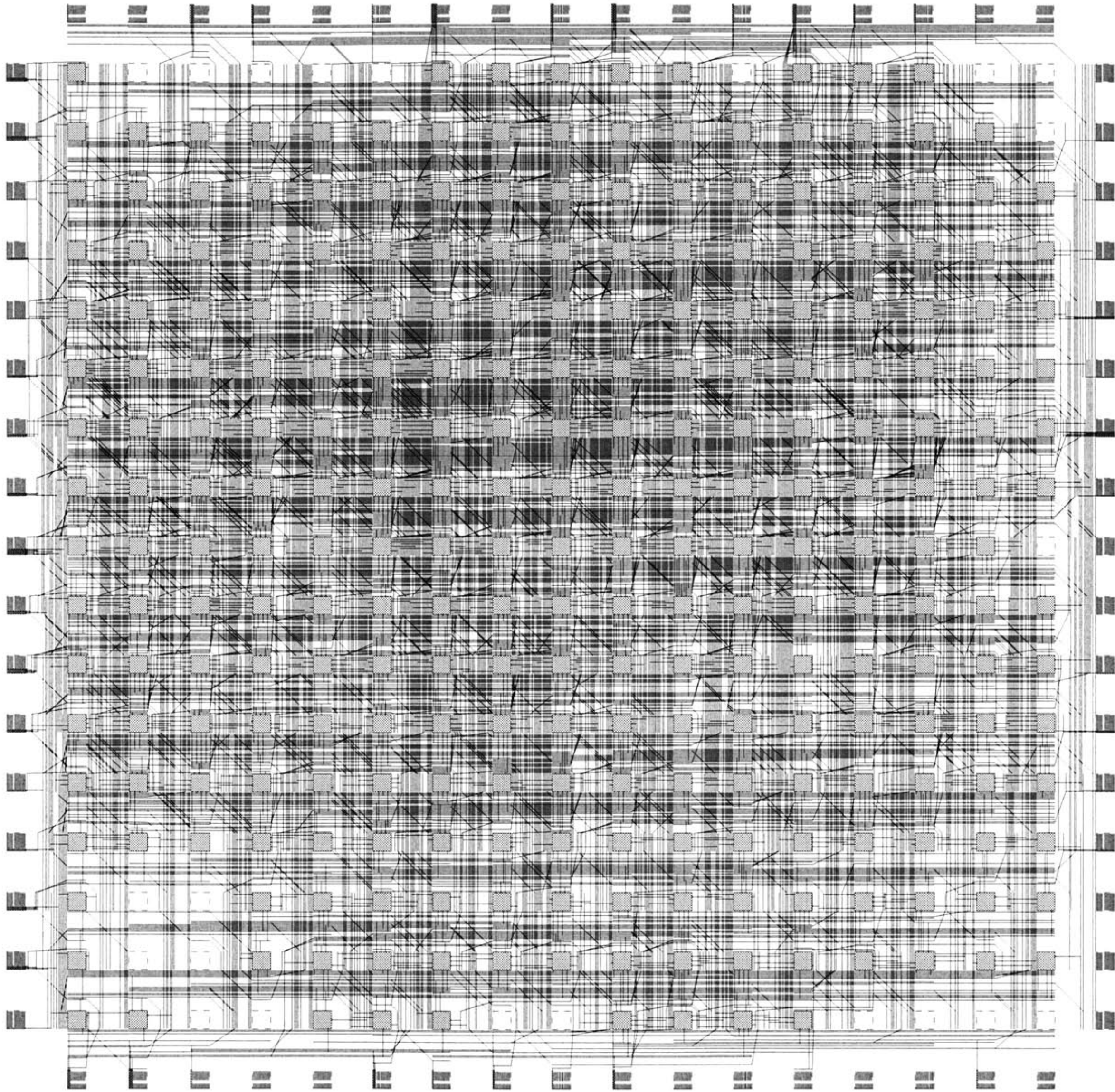


Figure E-28: pmatch-v1 layout



Routing succeeded with a channel width factor of 33.

Figure E-29: sor layout

Bibliography

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 248–259, 2000.

[2] Altera Corporation, 2610 Orchard Parkway, Jose, CA 95124. *Implementing Logic with the Embedded Array in FLEX 10K Devices*, 1996.

[3] S. P. Amarasinghe and M. S. Lam. Communication Optimization and Code Generation for Distributed Memory Machines. In *Proceedings of SIGPLAN '93, Conference on Programming Languages Design and Implementation*, June 1993.

[4] R. Amerson, R. Carter, B. Culbertson, P. Kuekes, and G. Snider. Teramac - Configurable Custom Computing. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 32–38, Los Alamitos, CA, 1995.

[5] C. S. Ananian. The Static Single Information Form. Technical Report MIT-LCS-TR-801, Massachusetts Institute of Technology, 1999.

[6] M. Annaratone, E. Arnould, T. Gross, H. T. Kung, M. Lam, O. Menzilcioglu, and J. A. Webb. The Warp Computer: Architecture, Implementation, and Performance. *IEEE Transactions on Computers*, C-36:1523–1538, 1987.

[7] K. Asanovic. *Vector Microprocessors.* PhD thesis, University of California, Berkeley, May 1998.

[8] K. Asanovic. Energy-Exposed Instruction Set Architectures. In *Work In Progress Session, Sixth International Symposium on High Performance Computer Architecture*, January 2000.

[9] J. Babb, M. Frank, and A. Agarwal. Solving Graph Problems with Dynamic Computation Structures. In *SPIE Photonics East: Reconfigurable Technology for Rapid Product Development & Computing*, Boston, MA, Nov. 1996.

[10] J. Babb, M. Frank, V. Lee, E. Waingold, R. Barua, M. Taylor, J. Kim, S. Devabhaktuni, and A. Agarwal. The RAW Benchmark Suite: Computation Structures for General Purpose Computing. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, April 1997.

[11] J. Babb, M. Rinard, A. Moritz, W. Lee, M. Frank, R. Barua, and S. Amarasinghe. Parallelizing Applications Into Silicon. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines (FCCM), Napa Valley, CA*, April 1999.

[12] J. Babb, R. Tessier, M. Dahl, S. Hanono, D. Hoki, and A. Agarwal. Logic Emulation with Virtual Wires. *IEEE Transactions on Computer Aided Design*, 16(6):609–626, June 1997.

[13] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, M. Walkden, and D. Zareksky. A MATLAB Compiler for Distributed, Reconfigurable, Heterogeneous Computing Systems. In *IEEE Symposium on FPGAs for Custom Computing Machines*, 2000.

[14] R. Barua. *Maps: A Compiler-Managed Memory System for Software-Exposed Architectures.* PhD thesis, M.I.T., Department of Electrical Engineering and Computer Science, January 2000.

[15] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal. Memory Bank Disambiguation using Modulo Unrolling for Raw Machines. In *Proceedings of the ACM/IEEE Fifth International Conference on High Performance Computing(HIPC)*, Dec 1998.

[16] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal. Maps: A Compiler-Managed Memory System for Raw Machines. In *Proceedings of the 26th International Symposium on Computer Architecture*, Atlanta, GA, May 1999.

[17] P. Bellows and B. Hutchings. JHDL - an HDL for Reconfigurable Systems. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 175–184, 1998.

[18] E. R. Berlekamp, J. H. Conway, and R. K. Guy. *Winning Ways.* Academic, London, 1982.

[19] A. A. Berlin and R. J. Surati. Partial Evaluation for Scientific Computing: The Supercomputer Toolkit Experience. In *Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida, June 1994 (Technical Report 94/9, Department of Computer Science, University of Melbourne)*, pages 133–141, 1994.

[20] V. Betz and J. Rose. VPR: A New Packing, Placement and Routing Tool for FPGA Research. In W. Luk, P. Y. Cheung, and M. Glesner, editors, *Field-Programmable Logic and Applications*, pages 213–222. Springer-Verlag, Berlin, 1997.

[21] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware Design in Haskell. In *International Conference on Functional Programming*, pages 174–184, 1998.

[22] K. Bondalapati and V. K. Prasanna. Dynamic Precision Management for Loop Computations on Reconfigurable Architectures. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 249–258, Los Alamitos, CA, 1999.

[23] S. Borkar, R. Cohn, G. Cox, T. Gross, H. T. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb. Supporting Systolic and Memory Communication in iWarp. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 70–81, June 1990.

[24] H. Bratman. An Alternate Form of the UNCOL Diagram. *Communication of the ACM*, 4(3), March 1961.

[25] R. Brayton, G. Hachtel, L. Hemachandra, A. Newton, and A. S-Vincentelli. A Comparison of Logic Minimization Strategies Using ESPRESSO - An APL Program Package for Partitioned Logic Minimization. In *Proc. Int. Symp. Circuits and Systems*, pages 43–49, Rome, May 1982.

[26] P. Briggs, November 2000. Private Communication.

[27] D. Brooks and M. Martonosi. Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance. In *5th International Symposium of High Performance Computer Architecture*, January 1999.

[28] M. Budiu and S. C. Goldstein. Fast Compilation for Pipelined Reconfigurable Fabrics. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 195–205, Monterey, CA, 1999.

[29] M. Budiu, M. Sakr, K. Walker, and S. C. Goldstein. BitValue Inference: Detecting and Exploiting Narrow Bitwidth Computations. In *European Conference on Parallel Processing*, pages 969–979, 2000.

[30] Burks, Goldstine, and von Neumann. Preliminary Discussion of the Logical Design of an Electronic Computing Instrument (1946). In *Perspectives on the Computer Revolution, Second Edition, Edited with commentaries by Zenon W. Pylyshyn and Liam J. Bannon.* Ablex Publishing Corporation, Norwood, New Jersey, 1989.

[31] W. Burleson, M. Ciesielski, F. Klass, and W. Lu. Wave-Pipelining: A Tutorial and Research Survey. *IEEE Transaction on VLSI Systems*, 6:464–473, 1998.

[32] T. Callahan, P. Chong, A. DeHon, and J. Wawrzynek. Fast Module Mapping and Placement for Datapaths in FPGAs. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 123–132, Monterey, CA, 1998.

[33] T. Callahan and J. Wawrzynek. Adapting Software Pipelining for Reconfigurable Computing. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2000.

[34] T. J. Callahan and J. Wawrzynek. Instruction-Level Parallelism for Reconfigurable Computing. In R. W. Hartenstein and A. Keevallik, editors, *Field-Programmable Logic: From FPGAs to Computing Paradigm*, pages 248–257. Springer-Verlag, Berlin, 1998.

[35] J. M. Cardoso and H. C. Neto. Fast Hardware Compilation of Behaviors into an FPGA-Based Dynamic Reconfigurable Computing System. In *Proc. of the XII Symposium on Integrated Circuits and Systems Design*, 1999.

[36] J. Carter, W. Hseih, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, L. Stoller, and T. Tateyama. Impulse: An Adaptable Memory System. In *Architectural Support for Programming Languages and Operating Systems*, October 1998.

[37] E. Caspi. Empirical Study of Opportunities for Bit-Level Specialization in Word-Based Programs. Master's thesis, University of California, Berkeley, Department of Electrical Engineering and Computer Science, Fall 2000. Also available as UCB//CSD-00-1126.

[38] E. Caspi, M. Chu, R. Huang, J. Yeh, Y. Markovskiy, A. Dehon, and J. Wawrzynek. Stream Computations Organized for Reconfigurable Execution (SCORE): Extended Abstract. In *10th Internation Conference on Field-Programmable Logic and Applications*, april 2000.

[39] Celoxica, Inc. *http://www.celoxixa.com.*

[40] J. Cong and Y. Ding. FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs. *IEEE Trans. Computer-aided Design*, 13(1):1–13, 1994.

[41] C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, and E.-N. Volanschi. Tempo: Specializing Systems Applications and Beyond. *ACM Computing Surveys*, 30(3es), 1998.

[42] H. Corporaal. *Transport Triggered Architectures; Design and Evaluation.* PhD thesis, Delft Univ. of Technology, September 1995.

[43] H. Corporaal and R. Lamberts. TTA Processor Synthesis. In *First Annual Conf. of ASCI*, May 1995.

[44] CynApps, Inc. Santa Clara, CA. *Cynlib: A C++ Library for Hardware Description Reference Manual*, 1999.

[45] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[46] W. J. Dally. Micro-Optimization of Floating-Point Operations. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 283–289, Boston, Massachusetts, April 1989.

[47] DAWN VME Products. *SLIC Evaluation Board User's Guide for DAWN VME PRODUCTS SLIC EB-1 Version 1.0*, June 1993.

[48] A. Dehon. *Reconfigurable Architectures for General-Purpose Computing.* PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, September 1996. Also available as MIT AI Lab Technical Report 1586.

[49] R. P. Dick and N. K. Jha. CORDS: Hardware-Software Co-Synthesis of Reconfigurable Real-Time Distributed Embedded Systems. In *Proceedings of ICCAD98*, pages 62–68, San Jose, CA, 1998.

[50] M. F. Dossis, J. M. Noras, and G. J. Porter. Custom Co-Processor Compilation. In W. Moore and W. Luk, editors, *More FPGAs*, pages 202–212. Abingdon EE&CS Books, Abingdon, England, 1993.

[51] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* PhD thesis, Yale University, 1985.

[52] P. Embree and B. Kimble. *C Language Algorithms for Digital Signal Processing.* Prentice Hall, Englewood Cliffs, 1991.

[53] B. Fagin and C. Renard. Field Programmable Gate Arrays and Floating Point Arithmetic. *IEEE Transactions on VLSI Systems*, 2(3):365–367, September 1994.

[54] D. Filo, D. Ku, C. Coelho, and G. DeMicheli. Interface Optimization for Concurrent Systems Under Timing Constraints. *IEEE Transactions on Very Large Scale Integration Systems*, pages 268–281, September 1993.

[55] J. A. Fisher, P. Faraboschi, and G. Desoli. Custom-Fit Processors: Letting Applications Define Architectures. In *International Symposium on Microarchitecture*, pages 324–335, 1996.

[56] R. French, M. Lam, J. Levitt, and K. Olukotun. A General Method for Compiling Event-Driven Simulations. *32nd ACM/IEEE Design Automation Conference*, June 1995.

[57] D. Galloway. The Transmogrifier C Hardware Description Language and Compiler for FPGAs, 1995.

[58] M. P. Gerlek, E. Stoltz, and M. Wolfe. Beyond Induction Variables: Detecting and Classifying Sequences Using a Demand-Driven SSA Form. *ACM Transactions on Programming Languages and Systems*, 17(1):85–122, January 1995.

[59] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweeney, and D. Lopresti. Building and Using a Highly Parallel Programmable Logic Array. *Computer*, 24(1), Jan. 1991.

[60] M. Gokhale and J. Stone. Automatic Allocation of Arrays to Memories in FPGA Processors with Multiple Memory Banks. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 63–69, Los Alamitos, CA, 1999.

[61] M. Gokhale, J. Stone, and M. Frank. NAPA C: Compiling for a Hybrid RISC/FPGA Architecture. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines (FCCM), Napa Valley, CA*, Napa Valley, California, April 1998.

[62] S. C. Goldstein and M. Budiu. NanoFabrics: Spatial Computing Using Molecular Electronics. In *28th International Symposium of High Performance Computer Architecture*, 2001.

[63] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer. PipeRench: A Coprocessor for Streaming Multimedia Acceleration. In *26th International Symposium of High Performance Computer Architecture*, pages 28–39, 1999.

[64] C. E. D. C. Green and P. Franklin. *RaPiD – Reconfigurable Pipelined Datapath.* Springer-Verlag, Darmstadt, Germany, 1996.

[65] B. Greenwald. A Technique for Compilation to Exposed Memory Hierarchy. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, September 1999.

[66] S. Guo and W. Luk. Compiling Ruby into FPGAs. In *Field Programmable Logic and Applications*, Aug. 1995.

[67] R. K. Gupta and S. Y. Liao. Using a Programming Language for Digital System Design. *IEEE Design and Test of Computers*, 1997.

[68] R. K. Gupta and G. D. Michelli. Hardware-Software Cosynthesis for Digital Systems. *IEEE Design and Test of Computers*, 1993.

[69] Harel. On Folk Theorems. *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 12, 1980.

[70] J. R. Hauser and J. Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 12–21, Los Alamitos, CA, 1997.

[71] R. Helaihel and K. Olukotun. Java as a Specification Language for Hardware-Software Systems. In *Proceedings of the International Conference on Computer-Aided Design*, pages 690–699, 1997.

[72] J. Hennessy and D. Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann, Palo Alto, CA, second edition, 1996.

[73] J. Hoe. *Operation-Centric Hardware Description and Synthesis.* PhD thesis, Massachusetts Institute of Technology, June 2000.

[74] J. Holloway, G. L. Steele, Jr., G. J. Sussman, and A. Bell. SCHEME-79 - LISP on a Chip. *Computer*, 14(7):10–21, 1981.

[75] J. Howell and M. Montague. Hey, You Got Your Compiler in My Operating System! In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, March 1997.

[76] IBM Microelectronics. *ASIC SA-27E Data Book, Part I*, March 2001.

[77] A. IEEE. *IEEE Standard for Binary Floating-Point Arithmetic.* ANSI/IEEE Standard, New York, 1985.

[78] Intel, Santa Clara, California. *Intel Itanium Processor Specification Update*, June 2001.

[79] J. Janssen and H. Corporaal. Partitioned Register Files for TTAs. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 303–312, Ann Arbor, Michigan, Nov. 1995.

[80] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[81] Y. Kang, M. Huang, S. Yoo, Z. Ge, D. Keen, V. Lam, P. Pattnaik, and J. Torrellas. FlexRAM: Toward an Advanced Intelligent Memory System. In *International Conference on Computer Design (ICCD)*, October 1999.

[82] D. Kozen. Kleene Algebra with Tests. *ACM Transactions on Programming Languages and Systems*, 19(3):427–443, May 1997.

[83] A. Kuperman. An External I/O Interface for a Reconfigurable Computing System. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 1999.

[84] S. Larsen and S. P. Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 145–156, 2000.

[85] K. Leary and S. Level. DSP/C: A Standard High Level Language for DSP and Numeric Processing. In *Proceedings of the ICASSP*, pages 1065–1068, 1990.

[86] C. Lee. An Algorithm for Path Connections and its Applications. *IRE Transactions on Electronic Computers*, Sept. 1961.

[87] C. G. Lee. *Code Optimizers and Register Organizations for Vector Architectures.* PhD thesis, University of California, Berkeley, May 1992.

[88] R. Lee. Subword Parallelism with MAX. *IEEE Micro*, pages 51–59. 145, August 1996.

[89] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe. Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine. In *Architectural Support for Programming Languages and Operating Systems*, San Jose, California, October 1998.

[90] C. Lent and P. Tougaw. A Device Architecture for Computing with Quantum Dots. In *Proceedings of the IEEE*, volume 85(4), pages 541–557, April 1997.

[91] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood. Hardware-Software Co-Design of Embedded Reconfigurable Architectures. In *Proc. 37th ACM IEEE Design Automation Conference*, pages 507–512, Los Angeles, CA, 2000.

[92] Z. Li, K. Compton, and S. Hauck. Configuration Caching Management Techniques for Reconfigurable Computing. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines (FCCM), Napa Valley, CA*, April 2000.

[93] J. Liang, S. Swaminathan, and R. Tessier. aSOC: A Scalable, Single-Chip Communications Architecture. In *Proceedings of the IEEE International Conference on Parallel Architectures and Compilation Techniques*, Philadelphia, PA, 2000.

[94] W. B. Ligon III, S. McMillan, G. Monn, K. Schoonover, F. Stivers, and K. D. Underwood. A Re-Evaluation of the Practicality of Floating Point Operations on FPGAs. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 206–215, Los Alamitos, CA, 1998.

[95] P. Lowney, S. Freudenberger, T. Karzes, W. Lichtenstein, R. Nix, J. O'Donnell, and J. Ruttenberg. The Multiflow Trace Scheduling Compiler. In *Journal of Supercomputing*, pages 51–142, Jan. 1993.

[96] K. Mai, T. Paaske, N. Jayasena, R. Ho, and M. Horowitz. Smart Memories: A Modular Recongurable Architecture. In *International Symposium on Computer Architecture*, June 2000.

[97] M.-C. Marinescu and M. Rinard. High-Level Specification and Efficient Implementation of Pipelined Circuits. In *Proceedings of ASP-DAC Asia and South Pacific Design Automation Conference*, February 2001.

[98] A. Marshall, T. Stansfield, I. Kostarnov, J. Vuillemin, and B. Hutchings. A Reconfigurable Arithmetic Array for Multimedia Applications. In *International Symposium on Field Programmable Gate Arrays*, Monterey, Ca., Feb. 1999.

[99] H. Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Columbia University, 1992.

[100] Maya Gokhale and Brian Schott. Data-Parallel C on a Reconfigurable Logic Array. *Journal of Supercomputing*, September 1995.

[101] H. McGhan and M. O'Connor. PicoJava: A Direct Execution Engine For Java Bytecode. *IEEE Computer*, 31(10):22–30, 1998.

[102] G. D. Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[103] E. Mirsky and A. DeHon. MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 157–166, Los Alamitos, CA, 1996.

[104] C. A. Moritz, M. Frank, W. Lee, and S. Amarasinghe. Hot Pages: Software Caching for Raw Microprocessors. Technical Memo LCS-TM-599, Laboratory for Computer Science, Massachusetts Institute of Technology, Sept 1999.

[105] M. Oskin, F. Chong, and T. Sherwood. Active Pages: A Computation Model for Intelligent Memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 192–203, June 1998.

[106] Parallax Inc. *BASIC Stamp Programming Manual*, 2001.

[107] S. Park and K. Choi. Performance-Driven Scheduling with Bit-Level Chaining. In *Design Automation Conference*, pages 286–291, 1999.

[108] J. Patterson. Accurate Static Branch Prediction by Value Range Propagation. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, volume 37, pages 67–78, June 1995.

[109] S. Perissakis. *Balancing Computation and Memory in High Capacity Reconfigurable Arrays*. PhD thesis, University of California, Berkeley, 2000.

[110] J. Peterson, R. O'Connor, and P. Athanas. Scheduling and Partitioning ANSI-C Programs onto Multi-FPGA CCM Architectures. In *Proceedings, IEEE Workshop on FPGA-based Custom Computing Machines*, Napa, CA, Apr. 1996.

[111] D. Petkov. Efficient Pipelining of Nested Loops: Unroll-and-Squash. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2000.

[112] B. Pottier and J. L. Llopis. Revisiting Smalltalk-80 Blocks: A Logic Generator for FPGAs. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 48–57, Los Alamitos, CA, 1996.

[113] R. Razdan. *PRISC: Programmable Reduced Instruction Set Computers.* PhD thesis, Division of Applied Science, Harvard University, Harvard University Technical Report 14-94, Center for Research in Computing Technologies, May 1994.

[114] R. Rugina and M. Rinard. Pointer Analysis for Multithreaded Programs. In *Proceedings of the SIGPLAN Conference on Program Language Design and Implementation*, pages 77–90, Atlanta, GA, May 1999.

[115] R. Rugina and M. Rinard. Automatic Parallelization of Divide and Conquer Algorithms. In *Proceedings of the SIGPLAN Conference on Program Language Design and Implementation*, Vancouver, BC, June 2000.

[116] M. Saghir, P. Chow, and C. Lee. Exploiting Dual Data-Memory Banks in Digital Signal Processors. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 234–243, Cambridge, MA, October 1–5, 1996.

[117] H. Schmit and D. Thomas. Address Generation for Memories Containing Multiple Arrays. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17, 1998.

[118] R. Schreiber, S. Aditya, B. R. Rau, V. Kathail, S. Mahlke, S. Abraham, and G. Snider. High-Level Synthesis of Nonprogrammable Hardware Accelerators. Technical Report HPL-2000-31, Hewlett Packard, 2000.

[119] L. Semeria. Resolution of Dynamic Memory Allocation and Pointers for the Behavioral Synthesis from C. In *Design, Automation, and Test in Europe (DATE)*, pages 313–319, 2000.

[120] L. Semeria and G. D. Micheli. Encoding of Pointers for Hardware Synthesis. In *Proceedings of the International Workshop on IP-based Synthesis and System Design IWLAS'98*, pages 57–63, 1998.

[121] L. Semeria and G. D. Micheli. SpC: Synthesis of Pointers in C. Application of Pointer Analysis to the Behavioral Synthesis from C. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'98)*, pages 321–326, San Jose, November 1998.

[122] N. Shirazi, A. Walters, and P. Athanas. Quantitative Analysis of Floating Point Arithmetic on FPGA-based Custom Computing Machines. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 155–162, Los Alamitos, CA, 1995.

[123] A. Singh, L. Macchiarulo, A. Mukherjee, and M. Marek-Sadowska. A Novel High Throughput Reconfigurable FPGA Architecture. In *International Symposium on Field Programmable Gate Arrays*, pages 22–29, Monterey, CA, 2000.

[124] A. Smith, M. Wazlowski, L. Agarwal, T. Lee, E. Lam, P. Athans, H. Silverman, and S. Ghosh. PRISM II Compiler and Architecture. In *Proceedings IEEE Workshop on FPGA-based Custom Computing Machines*, pages 9–16, Napa, CA, April 1993.

[125] M. Smith. Extending SUIF for Machine-Dependent Optimizations. In *Proceedings of the First SUIF Compiler Workshop*, pages 14–25, Stanford, CA, Jan. 1996.

[126] D. Soderman and Y. Panchul. Implementing C Algorithms in Reconfigurable Hardware using C2Verilog. In *IEEE Symposium on FPGAs for Custom Computing Machines*, 1998.

[127] M. Stephenson, J. Babb, and S. Amarasinghe. Bitwidth Analysis with Application to Silicon Compilation. In *Proceedings of the SIGPLAN Conference on Program Language Design and Implementation*, pages 108–120, 2000.

[128] SUN Microsystems, Mountain View, California. *SPARC Architecture Manual*, 1988.

[129] Synopsys, Inc. *Behavioral Compiler User Guide, V 1997.08*, August 1997.

[130] systemc.org. *System C Version 1.1 User's Guide*, 2000.

[131] Tensila, Inc. *Application Specific Microprocessor Solutions - Overview Handbook*, 1998.

[132] R. G. Tessier. *Fast Place and Route Approaches for FPGAs*. PhD thesis, MIT, November 1998. Also available as MIT-LCS TR-768.

[133] W. Tsu, K. Macy, A. Joshi, R. Huang, N. Walker, T. Tung, O. Rowhani, V. George, J. Wawrzynek, and A. Dehon. HSRA: High-Speed, Hierarchical Synchronous Reconfigurable Array. In *ACM International Workshop on Field-Programmable Gate Arrays*, Monterey, CA, February 1999.

[134] A. V. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, and X. Ji. Adapting Cache Line Size to Application Behavior. In *International Conference on Supercomputing*, pages 145–154, 1999.

[135] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. H. Touati, and P. Boucard. Programmable Active Memories: Reconfigurable Systems Come of Age. *IEEE Transactions on VLSI Systems*, 4(1), March 1996.

[136] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring It All to Software: Raw Machines. *IEEE Computer*, 30(9):86–93, Sept. 1997. Also available as MIT-LCS-TR-709.

[137] M. Wan, H. Zhang, V. George, M. Benes, A. Abnous, V. Prabhu, and J. Rabaey. Design Methodology of a Low-Energy Reconfigurable Single-Chip DSP System. *IEEE Computer*, Feb. 1998.

[138] M. Weinhardt. Compilation and Pipeline Synthesis for Reconfigurable Architectures - High Performance by Configware. In *Reconfigurable Architecture Workshop*, April 1997.

[139] R. Weiss, G. Homsy, and T. Knight. Toward in Vivo Digital Circuits. In *Proceedings of DIMACS Workshop on Evolution as Computation*, 1999.

[140] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, C.-W. Tseng, M. Hall, M. Lam, and J. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *ACM SIGPLAN Notices*, 29(12), Dec. 1996.

[141] R. P. Wilson and M. S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. *ACM SIGPLAN Notices*, 30(6):1–12, 1995.

[142] S. Wilton, J. Rose, and Z. Vranesic. The Memory/Logic Interface in FPGAs with Large Embedded Memory Arrays. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(1), March 1999.

[143] XILINX, Inc., 2100 Logic Drive, San Jose, California, 95214. *The Programmable Gate Array Data Book and The XC4000 Data Book*, Aug. 1992.

[144] XILINX, Inc., 2100 Logic Drive, San Jose, California, 95214. *Virtex-E 1.8V Field Programmable Gate Arrays Datasheet, Version 1.7, September 2000*, 2000.

[145] T. Yang and A. Gerasoulis. List Scheduling with and without Communication. *Parallel Computing Journal*, 19:1321–1344, 1993.

[146] H. Zhang, M. Wan, V. George, and J. Rabaey. Interconnect Architecture Exploration for Low-Energy Reconfigurable Single-Chip DSP. In *Proceedings, Workshop on VLSI (WVLSI)*, Orlando, Florida, Apr. 1999.