CAD Algorithms and Performance of Malibu: An FPGA with Time-Multiplexed Coarse-Grained Elements

by

David Grant

B.A.Sc. Computer Engineering, University of Waterloo, 2002
M.A.Sc. Electrical and Computer Engineering, University of Waterloo, 2004

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE STUDIES
(Electrical and Computer Engineering)

The University Of British Columbia
(Vancouver)

August 2011

Abstract

Modern Field-Programmable Gate Arrays (FPGAs) are used to implement a wide range of ever-larger circuits, many of which have both coarse-grained and fine-grained components. Past research into coarse-grained FPGAs optimized for such circuits have only demonstrated a 10% density advantage. In contrast, time-multiplexing of fine-grained FPGAs has demonstrated a 14x density improvement. This leaves an open question whether a time-multiplexed, coarse-grained FPGA can provide a similar density advantage. Even more important is whether the coarse-grained circuit structure can be exploited by Computer-Aided Design (CAD) tools to significantly reduce compile times.

This thesis investigates a new type of FPGA in which coarse-grained, time-multiplexed resources are added to a traditional FPGA. Through time-multiplexing, density and compile time are improved. By retaining the fine-grained logic and routing resources, performance does not suffer as much as in past attempts.

This thesis also presents two CAD flows, M-CAD and M-HOT, to compile Verilog for this new FPGA. Both flows speed up compile times by more than 10x, which has not been demonstrated with any other flow (even flows that sacrifice quality). They can also achieve a circuit density greater than modern FPGAs, and can trade density for performance, something most FPGA CAD flows cannot do. At maximum density, the M-HOT flow achieves a 26.1x compile time speedup, 2.5x the density, and 0.5x the performance of a commercial FPGA and CAD tool. At maximum performance, M-HOT achieves 1.0x density, and 0.7x performance.

In contrast, M-CAD is a bit faster than M-HOT but achieves a lower quality result.

In M-CAD, there are situations where the placer needs temporal information from the scheduler to make good decisions. Instead, M-HOT divides the circuit into heights to keep the integrated placement, routing, and scheduling problem tractable, but compile time suffers if there are too few heights. Although we show there is at most a theoretical 1.6x or 2.0x clock frequency improvement still remaining in M-HOT or M-CAD, respectively, the amount achievable may be far less. Future work should focus on improving the front-end synthesis, the coarse/fine-grained interface, and the coarse/fine-grained partitioning to provide higher quality input to the back-end CAD flow.

Preface

[1] D. Grant and G. Lemieux. A spatial computing architecture for implementing computational circuits. In *Proc. Microsystems and Nanoelectronics Research Conference (MNRC)*, pages 41–44, Oct. 2008.

[2] D. Grant, G. Smecher, G. G. Lemieux, and R. Francis. Rapid synthesis and simulation of computational circuits in an MPPA. In *Proc. Field-Programmable Technology (FPT)*, pages 151–158, Dec. 2009.

[3] D. Grant, G. Smecher, G. G. Lemieux, and R. Francis. Rapid synthesis and simulation of computational circuits in an MPPA. In *Journal of Signal Process Systems*, pages 1–17, Dec. 2010.

[4] D. Grant, C. Wang, G. G. Lemieux. A CAD Framework for MALIBU: An FPGA with Time-multiplexed Coarse-Grained Elements. In *Proc. Field-Programmable Gate Arrays (FPGA)*, pages 123–132, Feb. 2011.

Parts of this thesis have been previously published as [1,2,3,4]. In all cases, except where mentioned below, I carried out the research and wrote manuscripts with input and editing from Dr. Lemieux.

Parts of Chapter 3 have been published in [1,3,4]. The initial instruction encoding in Section 3.5 was created by Dr. Lemieux. The minimum-width transistor Malibu block area estimates (Table 3.1), and the memory size estimates (Table 3.6) were compiled by Chris Wang.

The original version of the architecture instruction summary in Table 3.2 was compiled by Chris Wang, but has not been published. It has since been modified to present more information.

A brief summary of each of the sections in Chapter 4, excluding fine-grained synthesis, has been published in [2], and the summary of the fine-grained synthesis was published in [4].

An earlier version of Chapter 5 was published in [2] and [3] (the coarse-grained parts) and in [4] (the fine-grained parts). Rosemary Francis wrote the initial version of the coarse-grained scheduler in Section 5.5. Graeme Smecher wrote two benchmark circuits (*fft8* and *fft16*), and did the preliminary work on the upper bound by hand-coding these circuits for comparison. That comparison is not included in this thesis, but the continuation of this work is in Chapter 5.6.2 and Chapter 6.6.2.

Parts of Chapter 6 have been published in [4].

Table of Contents

List of Tables

List of Figures

List of Abbreviations

AES	Advanced Encryption Standard 73
ALAP	As-Late-As-Possible 132
ALM	Adaptive Logic Module 9
ASIC	Application-Specific Integrated Circuit 1
BLIF	Berkley Logic Interchange Format 75
CAD	Computer-Aided Design 1
CG	Coarse-Grained 41
CGI	Coarse-Grained Input 41
CGO	Coarse-Grained Output 41
CGRA	Coarse-Grained Reconfigurable Array 8
CLB	Configurable Logic Block 14
DFG	Data Flow Graph 8
FFT	Fast Fourier Transform 2
FG	Fine-Grained 41
FPGA	Field-Programmable Gate Array 1
HDL	Hardware Description Language 2
II	Initiation Interval 34
LUT	Look-Up Table 1
MPPA	Massively Parallel Processor Array 124
PE	Processing Element 22
SIMD	Single Instruction, Multiple Data 31
SL	Schedule Length 34
TSFPGA	Time-Switched Field-Programmable Gate Array 25
VLIW	Very Large Instruction Word 31
VPR	Versatile Place-and-Route 7

Acknowledgments

I would first like to thank my supervisor, Dr. Guy Lemieux. Without his help, encouragement, guidance, and insight, this work would not have been possible. Thanks also to Chris Wang, Rosemary Francis, Zhiduo Liu, and all the members of the SoC lab for help and input.

I am grateful for the support of my family, Norm, Sue, James, Steve, and Bryan. To my fiancé, Clara, I am grateful for your support and encouragement, I love you. And to all my friends, thank you for being there when I needed you.

I would also like to thank Deming Chen, Russell Tessier, and Graeme Smecher for providing several benchmark circuits, as well as the authors and the many additional contributors to the various open-source tools used in this research. Verilator: Wilson Snyder, Duane Galbi, and Paul Wasson. OdinII: Peter Jamieson, Kenneth B. Kent, Farnaz Gharibian, and Lesley Shannon. VPR: Vaughn Betz, Jonathan Rose, and Alexander Marquardt. ABC: Robert Brayton and Alan Mishchenko.

This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC). Equipment donations by CMC Microsystems are gratefully acknowledged.

Chapter 1

Introduction

1.1 Motivation

Modern Field-Programmable Gate Arrays (FPGAs) contain over 1 million Look-Up Tables (LUTs), over $1,000$ hard memory or multiplier blocks, and about 300 wires per row or column [7, 90]. In addition, they are continuing to grow according to Moore's law, roughly doubling in capacity every 18 to 24 months [80]. As a result, great demand is placed on FPGA Computer-Aided Design (CAD) tools to synthesize ever-larger circuits without degrading result quality or increasing run-time.

Commercial FPGAs play three important roles. First, although modern FPGAs are on average four times slower than Application-Specific Integrated Circuits (ASICs) [45], in many cases this is fast enough for the final circuit implementation, allowing FPGAs to be used in applications ranging from hand-held MP3 players [19] to the Large Hadron Collider (LHC) [16]. The ability to field-upgrade circuits to fix bugs or add new features is part of the attraction. Second, FPGAs are fast enough for emulation-based testing of a circuit intended for an ASIC. This is especially useful for testing systems-on-a-chip which implement a processor and run firmware. An estimated 90% of all ASIC designs are tested in FPGAs before fabrica-

tion [38]. Third, FPGAs are increasingly used to implement software applications which have been mapped to hardware. For example, an embedded processor (and the associated software) may be too slow for the intended application, and a high-performance processor may use too much power. FPGAs fill this gap. In all cases, a fast FPGA CAD flow is critical for a rapid product development cycle including design, simulation, and verification.

In these three roles, FPGAs are not just implementing fine-grained (bit-oriented) circuits. Modern FPGA usage has expanded to implementing a wide range of circuits. The research in this thesis is focused on the growing use of coarse-grained circuits. Although such circuits are dominated by word-wide signals, they may contain a small amount of fine-grained signals, primarily for control. These coarse-grained circuits are increasingly implemented on FPGAs for word-wide data processing and computation. For example, the aforementioned MP3 player [19] uses FPGAs as a bridge between the embedded processor and the memory and storage. In the LHC, FPGAs are used to compute real-time 32-bit Fast Fourier Transforms (FFTs). Additional examples include fluid dynamics [96], video processing [2], financial modeling [79], ray tracing [31], and nuclear simulation [28].

As with any circuit, a coarse-grained circuit can be specified at a behavioural level using a Hardware Description Language (HDL) like Verilog or VHDL. This makes it easy to build word-oriented circuits. Further, increasingly popular in industry are tools to automatically build such circuits. Full-system hardware generators like SOPCBuilder [6] and EDK [91], and automatic C-to-gates flows such as [3, 58, 75] generate large, word-oriented hardware circuits from software specifications.

This thesis investigates a way to implement these new, large, word-oriented circuits more efficiently than using a traditional FPGA. By "more efficiently", we mean improving the compile time of FPGA CAD tools, overcoming the strict capacity limit of FPGAs, and reducing the silicon area required by the circuit. Saving time and area means a faster time-to-market and lower costs for product development and maintenance.

Compile time is important for a rapid product development cycle, whether that is fixing bugs or adding new features. Faster compile times mean less waiting time, reducing the person-hours of a project and therefore reducing cost. It also reduces time-to-market, which can increase revenue particularly in the rapidly evolving technology industry.

One reason that FPGA CAD tools are slow is that FPGAs are still bit-oriented devices. FPGA synthesis involves decomposing a circuit into (potentially) millions of 2-input and 3-input logic gates, technology mapping the gates into LUTs, clustering the LUTs into logic blocks, and then placing and routing the clusters on the FPGA. This fine-grained approach can take several hours or days for a large circuit. Given these long runtimes, vendors are turning to other methods to increase CAD performance. Parallel compilation helps when powerful computer systems are available, but the achieved speedup in commercial tools is still limited. For example, QuartusII is up to 20% faster with four cores [5]. Incremental compilation [17] also helps during the design phase, but only when small successive changes are made to remove bugs and implement new features. A set of CAD tools that are always fast is more useful.

Circuit area is important because it translates directly into cost. Larger devices cost more, so being able to fit circuits onto smaller devices, whether for testing or product deployment, means cost savings. When mapping to an FPGA, every operation and communication in the circuit is assigned to dedicated resources and interconnect. While this gives a fast implementation, it is wasteful because the resources and interconnect are often not all used simultaneously.

Additionally, FPGAs have a strict capacity limit. FPGAs are fixed-density devices and therefore have an upper bound on the number of gates that can be implemented. If a circuit does not fit within this limit, the designer must buy a bigger device (if one exists and can be procured), or partition the circuit into two or more FPGAs. Both of these solutions can be expensive (new devices or engineering salaries), and will slow down product development. Running into this obstacle is a significant problem for using FPGAs to emulate large ASICs and for mapping large software applications into hardware.

Reducing compile time, reducing area, and overcoming the strict capacity limit are all possible by converting the circuit to firmware and running it on an embedded processor, similar to an embedded logic simulation. However, the resulting "circuit" would be very slow since every operation would be time-multiplexed on a single processor. The circuit performance (or speed, or user clock frequency) is important because it affects the usability of the synthesized result. If the circuit is too slow, it may not be usable in a commercial product, for example if it cannot meet real-time computational demands like decoding an MP3. If it is significantly slow, it may not be usable for prototyping ASICs either. When addressing the compile time, area, and capacity limit, the user clock frequency cannot be ignored.

To improve coarse-grained circuit mapping (addressing compile time, area, capacity limit, and user clock frequency), this thesis proposes a new, FPGA-like architecture with coarse-grained, time-multiplexed resources tightly coupled to traditional fine-grained FPGA resources. We have called this architecture "Malibu". The CAD tools in this thesis use the time-multiplexed, coarse-grained elements to fold computation in space and reuse resources, giving a performance/density tradeoff that depends on the amount of folding done. The only requirement for this is additional memory to store operations, but additional memory is more dense than having additional ALUs and dedicated interconnect for each operation.

The coarse-grained resources reduce the compile time by reducing the placement and routing problem size (e.g., no need to synthesize coarse-grained operations down to bits). By time-multiplexing these coarse-grained elements and the coarse-grained interconnect, the area cost of these large components can be amortized over many clock cycles (implementing a different part of the circuit each cycle). The improved density also allows larger circuits to be mapped into smaller devices by trading off the maximum clock frequency.

In this thesis we show the Malibu architecture and CAD tools (M-HOT) can achieve an average (geomean) 26.1x improvement in compile time compared to a commercial FPGA CAD tool (QuartusII v10.0). With this very fast compile time, we show a performance versus density

tradeoff on a fixed Malibu architecture ranging from the same density as a commercial FPGA (StratixIII) with 70% of the performance (26.1x compile speedup, 0.996x density, 0.707x performance) up to 2.5x the density with half the performance (26.1x compile speedup, 2.474x density, 0.513x performance).

The main focus of this work is on very fast compile times. Existing commercial and academic FPGA CAD tools have options to speed up the compile time, but the speedup is limited. In Appendix A we show that the compile time in QuartusII can be reduced by 15%, and in VPR by 7% by disabling optimizations, turning off timing-driven placement, and reducing the placement and routing effort levels. The Malibu approach is 26.1x faster than QuartusII and 8.4x faster than VPR; it is unlikely that compile times in existing tools can be further reduced by parallelism or algorithm tuning alone. Other approaches like Ultra-Fast placement [66] and various algorithms tested in [59] also reduce the compile times, but the Malibu approach is still several times faster as explained in Section 2.3.3.

The Malibu architecture can also achieve a higher density than traditional FPGAs by sacrificing performance (roughly 2.5x the density for half the performance). Similar density tradeoffs are possible with traditional FPGAs by varying the physical architecture and transistor sizes. Kuon et al. [46] report an area range of 3.6x (implementing the same circuit on FPGAs with different architecture parameters) and a circuit speed range of 2.6x. However, accessing these tradeoffs requires fabricating a new FPGA device (time consuming and expensive). In comparison, the advantage of the Malibu architecture is that the performance versus density tradeoff is possible without changing the underlying architecture or device deisgn. All it requires is a change to a compiler flag in the CAD tool. All of the results in this thesis were generated using the same Malibu device architecture.

1.2 Statement and Contributions of Thesis

The purpose of this thesis is to investigate mapping coarse-grained circuits onto an FPGA-like architecture comprised of fine-grained and time-multiplexed, coarse-grained resources. This thesis presents both a new architecture designed for implementing coarse-grained circuits, and the related CAD tools to synthesize circuits to the architecture.

There are three main contributions of this thesis: the Malibu architecture, the M-CAD flow, and the M-HOT flow. The first contribution is a new FPGA-like architecture for implementing coarse-grained circuits. This architecture is presented in Chapter 3. The Malibu architecture combines time-multiplexed, coarse-grained resources with the traditional fine-grained resources of an FPGA. The coarse-grained resources are ideal for performing word-oriented computation. The ALUs in Malibu are Verilog-specific and directly support common hardware operations not found in typical ALUs; for example, bit concatenation where two signals are joined together (`ab[7:0] = {a[3:0],b[3:0]}`) and unary logic reductions where the individual bits of a signal are ANDed, ORed, or XORed together (an XOR reduction is `parity[0] = ^a[7:0]`). In addition, the fine-grained resources can quickly compute and distribute the fine-grained control signals present in the circuit.

With the Malibu architecture, it is possible for CAD tools to fold a circuit in space and trade circuit performance for density—something that is only recently becoming possible with the latest academic and commercial FPGA CAD tools and devices.[1] This allows the same large circuit to run slowly on a small, inexpensive device, or run faster on a larger device. It also allows very large circuits to be developed and tested at a slower speed on current devices, with the intention of running such circuits at full-speed when larger devices eventually become available. The specific Malibu architecture configuration used in this thesis is focused on per-

[1] Tabula's synthesis flow can do this, but the Tabula architecture is only fine-grained, so everything is synthesized to fine-grained resources. The Tabula architecture is also limited to eight folds. SPR [32] can also do this, but it is solving a different problem than Malibu. SPR does not map circuits written in an HDL (it maps software kernels), and SPR focuses on high-quality results, so it is slow. More information is in Section 2.3.

formance, and yet it is 2.5x more dense than a traditional FPGA. However, the main focus of this work is on compile time, where the Malibu tools are 26.1x faster than FPGA CAD tools.

The second contribution is the M-CAD flow presented in Chapters 4 and 5. M-CAD can compile the full synthesizable subset of Verilog2005 into a configuration bitstream for Malibu. M-CAD builds on Verilator [71], as well as common FPGA CAD tools: Versatile Place-and-Route (VPR) [13], ABC [18], and OdinII [39]. These FPGA tools are designed for fine-grained logic, so they have been expanded, and new tools have been created, to add support for placing, routing, and scheduling the time-multiplexed coarse-grained logic. The significance of this is simultaneously handling coarse-grained and fine-grained components (and achieving a successful mapping solution). In addition, the tools presented in this thesis are a complete Verilog-to-bitstream flow for mapping circuits to Malibu. For FPGAs in academia, a complete flow is only recently becoming possible with the development of OdinII to provide front-end synthesis for ABC and VPR. However, OdinII only supports a limited subset of Verilog. For coarse-grained architectures in research, no such end-to-end tool flow exists in prior work.

Compared to QuartusII synthesis for a StratixIII FPGA, the M-CAD approach compile time is, on average, 38.7x faster. The M-CAD flow can also trade density for performance on a fixed architecture. At maximum performance it is roughly two-thirds the performance of a StratixIII and just over half the density (38.7x compile speedup, 0.652x performance, 0.582x density). At maximum density M-CAD is just under half the performance of a StratixIII and 2.5 times the density (38.7x compile speedup, 0.429x performance, 2.474x density). M-CAD achieves a very fast compile time and improved density (by sacrificing some performance) compared to a commercial FPGA.

The third contribution is the M-HOT flow presented in Chapter 6. An analysis of the M-CAD results showed the placement step accounted for 70% of the difference between the achieved F_{max} result and an upper bound on clock frequency. A common scenario is identified where placement could have been improved if the results of scheduling were known

beforehand, which is not possible in a segregated tool flow like M-CAD. To address this problem, M-HOT presents an integrated placement, routing, and scheduling approach. The algorithms in M-HOT are based on a modulo graph embedding Coarse-Grained Reconfigurable Array (CGRA) scheduler [62] which has been modified to add support for fine-grained resources. By using 50% more compile time than M-CAD, M-HOT achieves an average of 10% better frequency and 200% better density at maximum performance. Three benchmark circuits (*fft16*, *fft8*, and *me*) do not perform as well in M-HOT due to a small circuit depth. Future work could improve the compile time and quality of these benchmarks by detecting circuits with a small depth and changing some annealing parameters.

Compared to QuartusII/StratixIII, the M-HOT compile time is, on average, 26.1x faster. At maximum performance M-HOT achieves 70% of the performance of a commercial FPGA (StratixIII) with the same density (26.1x compile speedup, 0.996x density, 0.707x performance). At maximum density, M-HOT is about half the performance and 2.5x the density (26.1x compile speedup, 2.474x density, 0.513x performance). This is significant because it demonstrates an integrated placement, routing, and scheduling approach can be both fast and achieve a high quality result for coarse-grained circuits.

Just as FPGAs perform very well with fine-grained circuits, the architecture and tools in this thesis are focussed on performing well for coarse-grained circuits with some small amount of fine-grained control logic. Entirely fine-grained circuits will not perform well on Malibu.

The CAD tool investigation in this thesis is focussed on the steps after front-end logic synthesis, that is, the clustering, placement, routing, and scheduling. We use the Verilator open-source tool to perform front-end logic synthesis (generate a Data Flow Graph (DFG) from the source Verilog). Although Verilator produces an adequate result for our purposes, it is not ideal because it does not handle both coarse-grained and fine-grained parts of the circuit simultaneously. The results in this thesis suggest that future work should focus on a new front-end synthesis tool that can process and optimize both the coarse-grained and fine-grained parts

of the circuit, in addition to the interface between them.

1.3 Evaluation Metrics

In this thesis, three metrics are used to measure the quality of the synthesis result and the quality of the architecture. Recall from Section 1.1 that we are interested in reducing compile time, reducing area, and overcoming the strict capacity limit of FPGAs, all while maintaining good circuit performance. The three metrics we use to measure and evaluate these are:

- **Compile Time** – Also known as synthesis time, this is the elapsed time required to create a Malibu bitstream from the Verilog source. In this thesis, all results were generated on computers with identical specifications (dual-socket, quad-core 2.66 GHz Xeon X5355, 16 GB RAM, up to eight tasks per computer since the Malibu tools are single-threaded).

- **Density** – This is the number of logic operations per unit area. In Section 3.7.2 the number of StratixIII Adaptive Logic Modules (ALMs) per Malibu CLB is computed, and density comparisons are based on that ratio. A Malibu CLB contains both coarse-grained and fine-grained resources.

- **Circuit Performance** – Also known as circuit speed or user clock speed, this is the maximum user clock frequency of the synthesized circuit. In other words, it is the fastest clock which can be used with the circuit and still have it function correctly. This is measured in MHz.

The compile time is measured directly as the elapsed time from when the tools start to when they finish. A 10x improvement (one order of magnitude) in compile time would be a significant result for FPGA CAD, although given already long CAD runtimes, any improvement is desirable. Having an architecture that supports coarse-grained resources reduces the problem size for the CAD. For example, if every operation can be expressed in 32-bit words instead of

single bits, the problem size is reduced by 32x. Of course, that is just the general idea and the improvement, if any, varies by circuit. However, we will show this reduction is significant with up to a 269x compile time improvement over FPGA CAD flows.

The density is the number of operations in the circuit divided by the number of Malibu CLBs required to schedule the circuit. The silicon area of a Malibu CLB is used to normalize the density and compare it to the density of a circuit in a StratixIII and in a VPR/iFAR architecture. Any improvement in density is desirable because it means less silicon area, and thus, cost savings. By improving density, Malibu can also be used as a platform for anticipatory circuit development where large circuits can be developed and tested for FPGAs which do not yet exist. FPGA capacity has tracked, and continues to track, Moore's Law [29, 80], so capacities of devices in the near future can be predicted. Some circuit performance is lost through aggressive time-multiplexing to achieve a high density. However, Malibu can achieve 2.5x the density of modern FPGAs, with only a 50% loss in circuit performance (and a 26.1x faster compile time than traditional FPGA CAD). If the Malibu architecture is reconfigured with more memory to permit even more aggressive time-multiplexing, 10x the density of an FPGA is possible.

Circuit performance is calculated by dividing the system clock frequency (1 GHz) by the number of timeslots used to schedule the circuit (known as the Schedule Length, or SL). The overhead of time-multiplexing makes it difficult to match or exceed the performance of an FPGA. We originally aimed to keep the final result within $1/10^{th}$ that of a state-of-the-art FPGA (one order of magnitude), which we estimated would be acceptable for testing circuits. The results in this thesis show this is achieved. At maximum density for the Malibu architecture (2.5x density of an FPGA) we achieve 50% of the performance of an FPGA. At maximum performance, it is 70% of an FPGA.

In dealing with the above metrics, we have chosen to place power analysis outside the scope of this thesis. A significant amount of work would be required to create a detailed power model of the Malibu architecture and write the tools to perform power analysis. Instead, we

focus on obtaining the best possible density and performance results. Although we do not address power in this thesis, the Mosaic research project has shown that power reductions are possible in time-multiplexed, coarse-grained architectures [84].

1.4 Comparison to Related Work

This section presents a summary of related work to place the Malibu architecture and tools in context with previous research. A complete discussion of the differences with previous research is in Section 2.3.

The novelty of the Malibu architecture is in a unique combination of features to enable very fast compile times while simultaneously improving silicon density and maintaining circuit performance comparable to commercial FPGAs and CAD tools. Fast compile times are achieved by including coarse-grained resources in the underlying Malibu architecture, and making the Malibu CAD tools coarse-grained aware. This reduces the work the tools must do to map a circuit onto the architecture. The coarse-grained resources also include Verilog-specific operations not found in typical ALUs which further reduces compile time and helps improve circuit performance.

Previous work on fast FPGA CAD [59, 66] has looked at part of the synthesis flow (placement and/or routing), but not the complete flow including front-end logic synthesis. We show in Appendix A that front-end synthesis takes 3x longer, on average, than placement or routing, so it is important to the overall flow. In this thesis we show a significant compile time reduction for the entire flow, including front-end logic synthesis.

The Malibu architecture tightly couples time-multiplexed, coarse-grained resources with traditional fine-grained FPGA resources. Time-multiplexing improves density by amortizing the area cost of large coarse-grained ALUs over many clock cycles. The use of coarse-grained ALUs also improves density by configuration bit-sharing, which reduces instruction memory size by expressing word-wide operations with only a few bits. The ALUs also include

Verilog-specific operations to improve performance. Fine-grained resources in the architecture also improve performance by permitting quick computation and distribution of critical (usually control) signals.

Time-multiplexing has been applied to fine-grained FPGA logic in research [15, 27, 40, 52] and commercially by Tabula [37] to improve density. However, circuit performance tends to suffer compared to an FPGA, or the CAD tools are slow. In this thesis, Malibu uses coarse-grained resources to reduce the problem size and speed up the CAD. Malibu achieves very fast compile times with circuit performance within 50% of an FPGA, and density improvements similar to much of this past research.

Time-multiplexing has also been applied to CGRAs, both with [26, 83] and without [33, 57, 67, 69, 73] fine-grained resources, to accelerate software application kernels. The research presented in this thesis builds upon previous work and extends it to show compile time and density gains for coarse-grained circuits rather than software kernels.

Coarse-grained resources have been applied to FPGAs without time-multiplexing [21, 88, 94] to improve density, but with little (or no) improvement in compile times or performance compared to traditional CAD tools and FPGAs. The main difference in this thesis is that Malibu is primarily coarse-grained, and Malibu time-multiplexes the coarse-grained resources to improve density and utilization. Malibu also shows improvements in density and compile times, while maintaining circuit performance comparable to modern FPGAs.

1.5 Thesis Organization

This thesis is organized as follows. Chapter 2 presents an introduction to FPGA CAD and a background in related coarse-grained architectures, time-multiplexed architectures, and their associated CAD tools. Chapter 3 presents the Malibu architecture and experimentally determines typical architectural parameters. Chapter 4 presents the front-end synthesis which prepares the coarse-grained and fine-grained parts of the input circuit for further processing.

Chapter 5 details the M-CAD flow for mapping circuits to Malibu, and tests the approach using the evaluation metrics. Chapter 6 details the M-HOT flow, an alternative tool flow created to achieve better quality results over M-CAD, and also tests it with the same experiments as were done with M-CAD. Finally, Chapter 7 provides concluding remarks and directions for future research.

Chapter 2

Background

2.1 Overview

This chapter beings with a review of FPGA CAD in Section 2.2, and then presents related work in time-multiplexed architectures, coarse-grained architectures, and their associated CAD approaches in Section 2.3.

2.2 Introduction to FPGA CAD

A typical island-style FPGA is shown in Figure 2.1. An FPGA is a 2D grid of Configurable Logic Blocks (CLBs) connected by programmable interconnect. A CLB uses a number of LUTs to implement logic; each LUT can be programmed to implement any Boolean expression up to the number of inputs and outputs it has. For example, a 4x1 LUT has 4 inputs and 1 output. The interconnect uses programmable connection blocks and switch blocks to route or steer signals and connect the CLBs together and to the outside world through the I/O pads. The grey boxes in the figure are connection blocks; one horizontal wire can be programmed to connect to many vertical wires, or vice versa. There are many more wires in a real FPGA than shown in this figure. An FPGA with ten 4x1 LUTs in each CLB would typically have 22

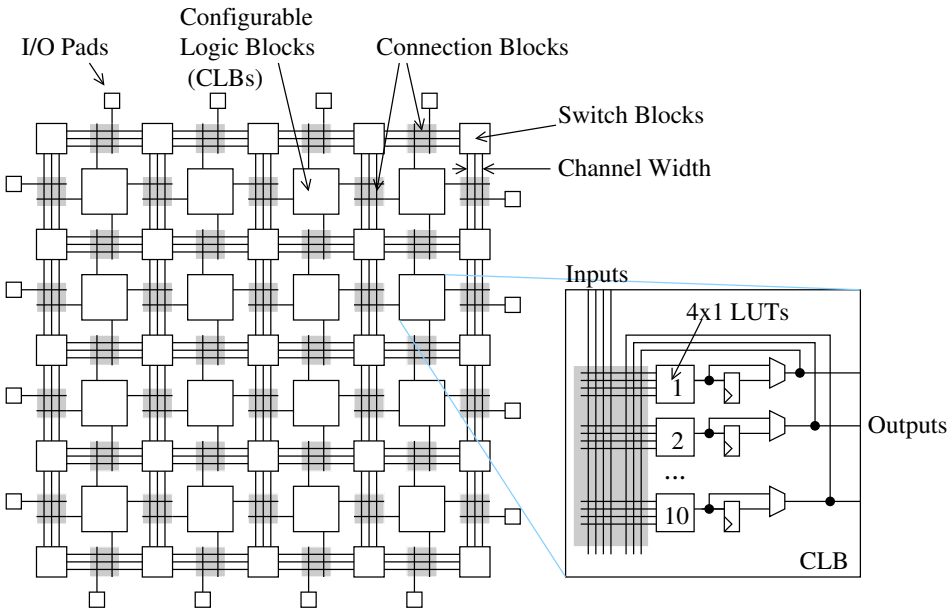


Figure 2.1: A typical island-style FPGA.

inputs and 10 outputs in each CLB [12]. Depending on the size of the supported circuits, the channel width could be anywhere from 10 to 300 wiring tracks.

To map logic circuits onto an FPGA, a CAD tool flow is required. The CAD tool flow transforms an input, usually described in an HDL like Verilog or VHDL, into a bitstream suitable for the FPGA. The tool flow can be generally split into four parts: front-end synthesis (including elaboration and technology mapping), clustering, placement, and routing. The entire flow is called synthesis. The front-end synthesis phase is also known as logic synthesis or technology-independent optimization. The back-end phase of clustering, placement, and routing is collectively known as physical synthesis. There is some ambiguity about whether technology mapping belongs in front-end or back-end synthesis since it performs technology-dependent logic optimizations. We have placed it in front-end synthesis so that back-end synthesis can assume the circuit only uses resources available on the targeted architecture (e.g., 4x1 LUTs, flip-flops, and I/Os in Figure 2.1). In this thesis, the term synthesis (alone with no modifiers) always refers to the complete flow including both front-end logic synthesis and

back-end physical synthesis.

Commercial tools like Altera's QuartusII [5] and Xilinx's ISE [92] implement a complete synthesis flow. Academic tools can also be used: for example OdinII (front-end synthesis), ABC (technology mapping), T-VPack (clustering), and VPR (placement and routing) collectively implement a complete flow. In Figure 2.1, the CAD tools would first do logic synthesis on the circuit, then technology-map the circuit into 4x1 LUTs, then cluster the LUTs into groups of ten per CLB, then place the CLBs on physical CLBs in the FPGA, and finally route the required connections. These steps are briefly explained next.

2.2.1 Front-End Logic Synthesis

Front-end logic synthesis begins by parsing the input (HDL, schematic, sequential code), and creating an unoptimized intermediate representation of the circuit. Part of this step is very similar to what a software compiler would do (lexical analysis, preprocessing, parsing, etc.).

The circuit description can be hierarchical. Unlike software, hardware cannot perform a function call, so all the instantiated modules must be elaborated (i.e., duplicated) to create a flat representation. Once flattened, technology-independent optimizations are applied. Many optimizations are the same as what a software compiler would perform (e.g., strength reduction, dead code elimination, constant folding, logic reduction), but for circuits additional objectives are required, like reducing the critical path (longest combinational path in the circuit).

The optimized code is then mapped to resources supported by the FPGA. This is where front-end synthesis really diverges from software compilers. In the example FPGA in Figure 2.1 there are only four resources: inputs, outputs, LUTs, and flip-flops. Commercial FPGAs include additional blocks like multipliers and memories, but the process of mapping is similar. A technology-mapping algorithm like BTWMap [25] or ABC's priority-based cut mapper [18] can be used. The circuit is reduced to 2-input or 3-input gates, then a ranking system adds gates to LUTs, limited by the 4-input, 1-output LUT constraint.

2.2.2 Clustering

Clustering accepts a technology-mapped netlist (a netlist is a graph representation of a circuit) and groups LUTs and flip-flops together into CLBs. The output is another netlist that contains only CLBs and macro blocks like multipliers, if present.

Clustering reduces the problem size for placement (fewer objects to place) and for routing (many nets become internal to the cluster). In Figure 2.1, each CLB can have up to ten LUTs and ten flip-flops clustered into it. In FPGA CAD, clustering is important to reduce the runtime of the placement step since VPR's simulated annealing placement requires $O(N^{1.33})$ time [68], where N is the number of logic elements in the circuit to be placed.

The academic clustering tool T-VPack uses a greedy algorithm to do this, aiming to simultaneously keep the number of inputs per CLB low, as well as fill each CLB [12]. Fewer inputs on a CLB means it is likely to route with fewer routing channels. The number of CLBs translates directly to area and a reduced problem size for placement.

2.2.3 Placement

Placement takes the clustered netlist and an architecture description, and assigns the clustered CLBs in the netlist to physical CLBs in the FPGA. Numerous placement approaches for FPGAs are summarized in [86]. In this thesis, simulated annealing is used for placement in Chapter 5, and for considering placement, routing, and scheduling together in Chapter 6. It is therefore worthwhile to explain how annealing works in relation to FPGA placement, and explain VPR's annealing schedule which is widely regarded as one of the best.

Simulated annealing [43] is a heuristic approach to global optimization. It performs well in large search spaces or when complex functions are required to describe the goodness (or badness, a.k.a., cost) of a particular configuration of the search space. It uses a "synthetic temperature" variable to control the acceptance rate of bad moves, and to determine when to exit. Simulated annealing often begins with a random configuration of the search space and

consists of two nested loops. The outer loop monitors the exit condition and decreases the temperature in each iteration. The inner loop performs random swaps in the search space, always accepting good moves, and accepting bad moves with a probability that decreases with the temperature. A "bad move" is one that increases the cost of the current solution. Some bad moves must be allowed to avoid getting stuck in a local minima. Near the end of annealing it is desirable to reject all bad moves, and a final greedy pass of the inner loop is used to do this.

The rate at which the temperature is decreased is called the annealing schedule. With a sufficiently large initial temperature, a sufficiently small cooldown rate, and with a sufficiently long run-time, it has been proven [36] that simulated annealing will converge to the global minimum (optimal) solution. For FPGA CAD, however, we only require a good solution to be found within a reasonable amount of time.

A simulated annealing algorithm for FPGA placement is shown in Figure 2.2. This algorithm uses VPR's annealing schedule. It is used in this thesis for annealing steps in both M-CAD and M-HOT. VPR computes the number of inner-loop iterations as $10 \times (N_{blocks})^{1.33}$, which is from [74]. N_{blocks} is the number of CLBs plus I/O pads. The inner loop randomly chooses two blocks to swap; the second block is also allowed to be an empty block location, and is chosen to be near the first block within a distance, D_{limit}. It then swaps them and computes the delta cost. If negative (a good move), then the swap is accepted. If not (a bad move), then the swap is only accepted if a function of the delta cost and temperature is less than a randomly generated number between 0 and 1:

$$e^{\frac{-delta\_cost}{temperature}} < rand\,[0,1) \tag{2.1}$$

The block selection is initially done over the entire chip, but as the temperature decreases, the area from which the second block can be chosen is restricted according to a distance

limit [13]:

$$D_{limit}^{new} = D_{limit}^{old} \cdot (1 - 0.44 + R_{accept}) \tag{2.2}$$

Where R_{accept} is the acceptance rate. VPR attempts to keep the acceptance rate as close to 0.44 as possible [13] by using D_{limit} to control the swap area and by changing the temperature reduction factor (between 0.5 and 0.96) in each outer-loop iteration based on the current acceptance rate. VPR exits when the temperature is below $0.005 \cdot \frac{cost}{N_{nets}}$.

The delta cost of the current solution is computed by VPR as a tradeoff between a wiring cost and a timing cost [53]:

$$delta\_cost = \lambda \frac{\Delta Timing\_Cost}{Prevous\_Timing\_Cost} + (1 - \lambda) \frac{\Delta Wiring\_Cost}{Previous\_Wiring\_Cost} \tag{2.3}$$

The wiring cost is based on a bounding box calculation for every net. The timing cost is based on a timing analysis that uses the slack on each path to calculate a criticality for the path, and then uses that criticality together with the delay information to compute the timing cost. The details of these computations are in [53]. The parameter λ is used to weigh one more than the other; by default, VPR uses $\lambda = 0.9$.

2.2.4 Routing

Once every CLB, block, and I/O pad has a location on the FPGA, the router determines how to configure the interconnect resources to connect all the elements together. In the circuit, the connections are represented as nets. Each net has exactly one source, and one or more sinks. The number of sinks on each net is the fanout. In Figure 2.1 these nets are routed using the wires, switch blocks, and connection blocks.

There are many routing options and approaches for FPGAs [47]. However, in this thesis, the performance of the router on fine-grained nets is not critical because the time-multiplexed pipelined routing network, which is used for the coarse-grained routing, is slow in comparison.

```
1  function simulated_annealing( initial_temperature )
2  {
3     temperature = initial_temperature
4     n_accept = 0
5     n_total = 0
6     distance_limit = size of the architecture
7
8     inner_iterations = 10 * exp( n_blocks, 1.33 )
9
10    cost = compute_total_cost()
11
12    while( 1 ) {
13
14       compute_timing()                              /* Compute the timing cost */
15
16       for(i=0; i<inner_iterations; i++) {
17          blk1 = choose_block()                      /* Choose an existing block (e.g., CLB) */
18          blk2 = choose_block(blk1, distance_limit) /* Choose a location to swap with, the
19                                                      * location returned here may be empty */
20          swap( blk1, blk2 )                         /* Swap them */
21
22          delta = compute_cost( blk1, blk2 )         /* Compute the cost difference for the swap */
23          if( delta <= 0 ) {
24             accept = true                           /* Always accept a good move */
25          } else {
26             e = exp( -delta / temperature )         /* Probalistically accept a bad move */
27             accept = (e < rand() ) ? true : false
28          }
29
30          n_total++;
31          if( accept ) {
32             cost += delta                           /* Commit the accepted move*/
33             commit_move( blk1, blk2 )
34             n_accept++;
35          } else {
36             swap( blk2, blk1 )                      /* Undo swap */
37          }
38       }
39
40       accept_rate = n_accept / n_total;             /* Compute accept rate */
41       distance_limit *= (1 - 0.44 + accept_rate);   /* Compute new distance limit */
42
43       if( accept_rate > 0.96 ) temperaure *= 0.5
44       else if(accept_rate > 0.8) temperature *= 0.9
45       else if(accept_rate > 0.15) temperature *= 0.95
46       else temperature *= 0.8
47
48       if( temperature < 0.005 * cost / n_nets ) {
49          break;                                     /* Exit condition */
50       }
51    }
52 }
```

Figure 2.2: Simulated annealing pseudocode. The annealing schedule and exit condition from timing-driven VPR [53] are used.

Still, we would like the fine-grained router to be fast and produce quality results, so we use the default VPR router.

The VPR router [11] is based on the PathFinder maze router [54]. It begins by routing each net, ignoring all congestion and resource conflicts, so that each net starts with the shortest possible route. It then iteratively rips-up every net and routes it again (rip-up and re-route), this time using a cost on each resource to discourage over-use. As the resource use is changed, the cost for the resource is updated using a combination of the current and past costs. Eventually, nets with alternative routing solutions are forced off high-demand resources, leaving only those nets which need them the most.

For a net with k sinks, the maze router is invoked k times to route the net. Routing is done using an A* search, expanding the *routing wavefront* along the lowest cost resources first until the first destination (one of the k sinks) is reached. The wavefront is generally then cleared, and the entire current route becomes the starting point for a new wavefront expansion to find the next sink. This is repeated until all k sinks are connected.

2.3 Related Work

The coarse-grained, time-multiplexed resources in Malibu are a departure from traditional FPGA architectures. This section presents related work in three categories. First, fine-grained, time-multiplexed architectures are presented. These architectures were among the first to demonstrate the benefits of time multiplexing to save silicon area, and in some cases to reduce compile times. Second, coarse-grained architectures are presented. These architectures leverage coarse-grained resources save silicon area and reduce compile times; however most are designed for software (not circuits). Some of these architectures also use time-multiplexing to maximize the use of large, coarse-grained ALUs. A few of these systems (e.g., RaPiD) also include fine-grained resources to implement control signals which allows the architecture, as a whole, to be decoupled from a controlling host processor and run independently. Third, re-

lated CAD tools for coarse-grained synthesis and fast fine-grained synthesis are presented. The CAD tools for Malibu are based partly on ideas from these tools.

Table 2.1 summarizes the features of the architectures presented in this section. It may be useful to refer to this table while reading the following sections. The columns for fine-grained resources and coarse-grained resources indicate whether such resources are available and what they are used for (data and control, control only, or data only). The number of contexts indicates how many operations may be time-multiplexed on the same resources (static means not time-multiplexed). The table then specifies the type host processor coupled to the architecture, whether global memory is used, and whether the architecture virtualizes resources. For architectures with virtual resources, the input typically only needs to be compiled once for any architecture size. Finally, the source language is given. Malibu is the only architecture to use both coarse-grained and fine-grained resources for implementing a circuit.

2.3.1 Fine-Grained, Time-Multiplexed Architectures

The benefits of time-multiplexing in FPGAs were first demonstrated with Dharma [15]. Dharma used an input-to-output levelization of the circuit to compute a level for each node, and required that all nodes with the same level be computed concurrently on the architecture. The area improvements in this work, if any, are not clear; however, there was a reported 18% improvement (reduction) in the critical path.

VEGA [40] extended this time-multiplexed work to allow the entire LUT-based FPGA to be multiplexed on a single CLB (if desired). VEGA demonstrated a 7-14x density improvement over commercial FPGAs of the same era (1995). Figure 2.3 shows the VEGA Processing Element (PE). The PE can be used alone, or can form a 2D mesh with other PEs. VEGA added a substantial amount of logic and memory around a 4x1 LUT and flip-flop for time-multiplexing. Every clock cycle, a single LUT is evaluated in each PE by reading inputs from the cache, evaluating those inputs with a LUT configuration from the logic instruction memory,

Table 2.1: Summary of related architectures. "AS" means application-specific, the value may change for a specific instance of the architecture for a specific application. "–" means that the feature is not used on the architecture.

	Fine-Grained Resources		Coarse-Grained Resources					
Architecture	**Used**	**Contexts**	**Used**	**Contexts**	**Host Processor**	**Global Memory**	**Virtual Resources**	**Source Language**
FPGA	yes	static	–	–	–	–	–	VHDL/Verilog
Dharma [15]	yes	∞[1]	–	–	–	–	–	Netlist
VEGA [40]	yes	∞[1]	–	–	–	–	–	Netlist
TSFPGA [27]	yes	∞[1]	–	–	–	–	–	Netlist
AFPGA [52]	yes	2	–	–	–	–	–	Netlist
Tabula [76]	yes	8	–	–	–	–	–	VHDL/Verilog
DP-FPGA [21]	yes	–	yes	static	–	–	–	Netlist
Ye and Rose [94]	yes	–	yes	static	–	–	–	Netlist
Wilton *et al.* [88]	yes	–	yes	static	–	–	–	Netlist
RaPiD [26]	control only	static	yes	mostly static[2]	risc	–	–	C-like (RaPiD-C)
ADRES [57]	–	–	yes	128	vliw	yes	–	C
MorphoSys [69]	–	–	yes	32	risc	yes	–	C-like
PipeRench [33]	–[4]	–	yes	256	sparc	yes	yes	C
Mosaic [83]	control only	AS[3]	yes	AS[3]	–	–	–	C-like (Macah)
WaveScalar [73]	–	–	yes	∞[1]	–	–	yes	C
TRIPS [67]	–	–	yes	∞[1]	–	–	yes	Assembly-like
SCORE [20]	Architecture Dependent						yes	C-like
Malibu	yes	static	yes	256	–	–	–	Verilog

[1]Up to the amount of context memory available.
[2]RaPiD's coarse-grained resources are mostly static but some are time-multiplexed. The number of contexts supported is unspecified.
[3]Mosaic is a family of architectures. These may be static or limited only by context memory.
[4]The PipeRench ALUs are built out of LUTs, but they are not user-programmable at the LUT level.

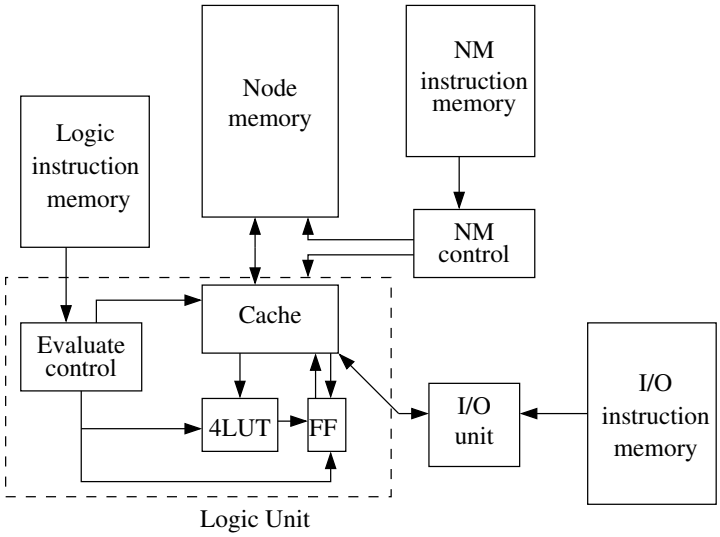


Figure 2.3: The VEGA PE [40]. The PE is replicated in a 2D mesh to form a complete system. The I/O unit includes a crossbar to route data between I/O units in adjacent PEs.

and writing the result back to the cache. The node memory (NM) instruction memory and the NM control execute pre-scheduled data transfers between the node memory and the cache. The I/O unit transfers data between the I/O units in adjacent PEs, the local cache, and the outside world. The number of configuration contexts supported by VEGA was only limited by the size of the memories used for logic instructions, nodes, and NM instructions.

The goal of VEGA was to maximize the logic density. VEGA also noted a speedup in compile time due to a reduction in routing complexity. However, the performance of mapped circuits peaked at 391 kHz, which was approximately 100x slower than state-of-the-art FPGAs in 1995. In this thesis, we time-multiplex the coarse-grained logic, not the fine-grained logic, and show similar density gains for coarse-grained circuits. However, our coarse-grained circuit speeds are only 2x slower on average compared to a state-of-the-art FPGA in 2010.

The Adaptive FPGA [52] (AFPGA) also used time-multiplexing to interleave two distinct circuits (or two distinct parts of the same circuit with no dependencies) on the same FPGA resources. AFPGA showed an 8-14% area improvement compared to FPGAs, but did not in-

vestigate circuit performance or compile time. Based on the AFPGA architecture description, the circuit performance should be relatively unchanged since it only added two-context memories to all the LUTs and routing resources. However, the AFPGA iterative mapping approach will likely increase compile time compared to traditional FPGA CAD tools.

Time-Switched Field-Programmable Gate Array (TSFPGA) [27] also demonstrated density improvements by time-multiplexing fine-grained elements, but took a radically different approach to the architecture building block. A TSFPGA subarray, shown in Figure 2.4a, integrates the interconnect with LUT memories. The subarrays are arranged into a 2D grid and connected using pipelined registers. The network inputs (xin0-xin3, yin0-yin3) come from other subarrays (not necessarily immediate neighbours) in the same row and column. These inputs can be directed into an Array Element (AE), shown in Figure 2.4b, or into the crossbar and routed to another subarray. In each timestep (clock cycle) the crossbar reads a new configuration from a context memory to route values. The results cited below use 64 routing contexts.

Also in each timestep, the output mux in the AE selects one input from the network inputs or the four AE LUT muxes, and directs it into the crossbar. Note that the four LUT muxes connect directly to the output mux (they do not form a chain as might be inferred from Figure 2.4b). Each of the four AE muxes has four configuration bits which are controlled by the subarray. It may take up to four timesteps to load all four configuration bits, so having four muxes ensures that one can always be ready with a value. The LUT memory connected to each of the four muxes contains data for many 4x1 LUTs.

Using this system, TSFPGA demonstrated a 2x improvement in density without any loss in circuit speed. The TSFPGA CAD tools also exhibited significant improvements in compile time due to reduced problem complexity. In this thesis, we take a similar approach to the coarse-grained element; inputs to the CLB may be directed to an ALU or may be routed. However, we limit the inputs to come only from the four immediate neighbours to avoid long

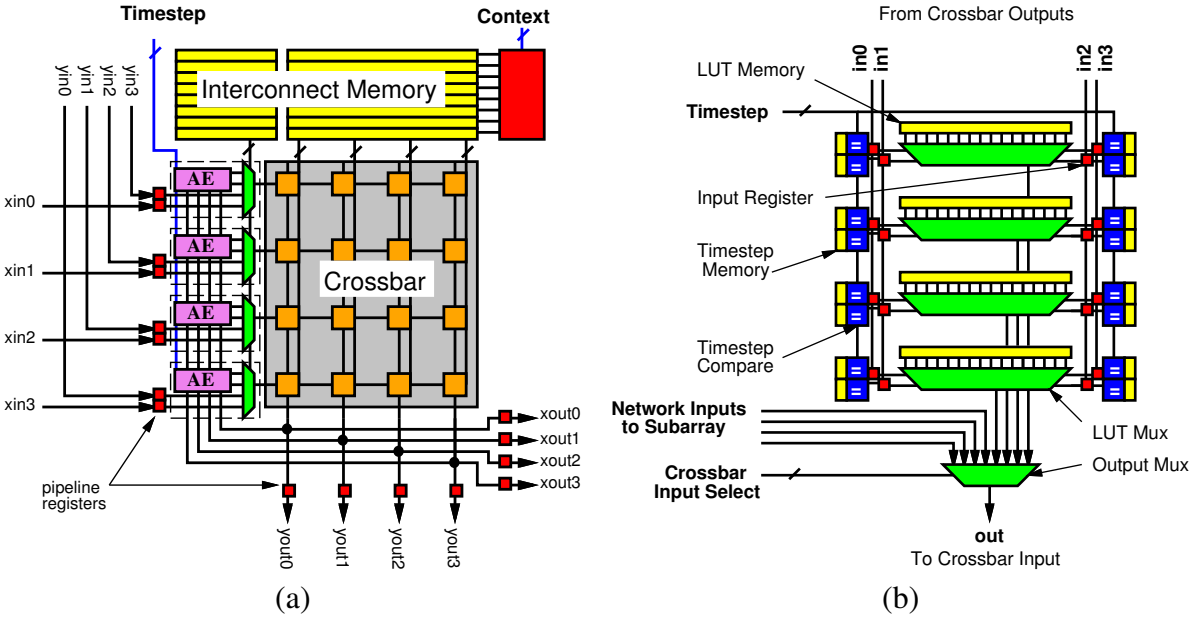


Figure 2.4: The TSFPGA subarray [27]. (a) The TSFPGA subarray, and (b) the subarray Array Element (AE). The subarray is replicated in a 2D grid to form a complete system. The xin0-xin3 inputs come from other subarrays in the same row. Similarly for yin0-yin3 in the same column.

wires.

Tabula is a commercial vendor with a time-multiplexed fine-grained device (ABAX [76]), and a set of CAD tools (Stylus [77]) to map circuits onto the device. Their tools synthesize HDL and can trade density for performance, offering up to a 2.5x density improvement over FPGAs, and comparable performance of 1.6 GHz divided by the number of configuration contexts, called "folds". The device and tools support up to eight folds.

The architectures presented in this subsection are all fine-grained. On such an architecture, just as on an FPGA, all coarse-grained operations are first decomposed into 2-input and 3-input operations and then implemented in LUTs. This increases compile time, and may not be the best use of configuration resources. This thesis investigates the use of coarse-grained resources to more directly implement coarse-grained operations. By not synthesizing down to fine-grained LUTs, compile time is significantly reduced (26.1x faster in this thesis compared

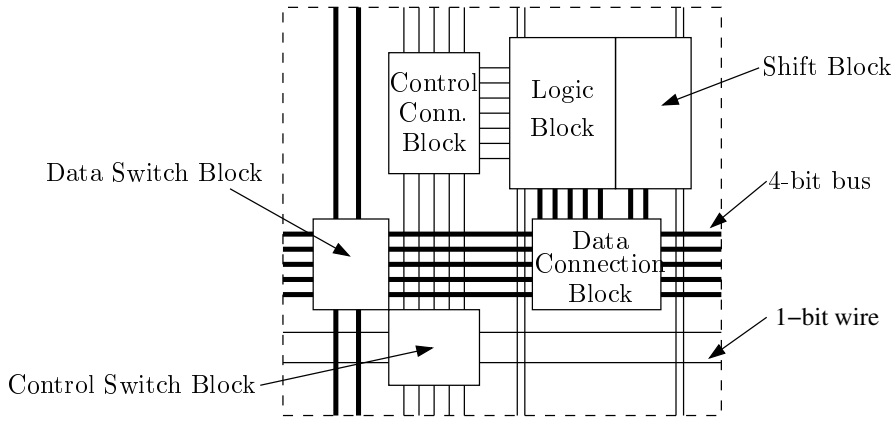


Figure 2.5: The DP-FPGA tile [21]. The tiles are replicated to form a 2D grid.

to QuartusII 10.0). Compared to modern FPGAs, this thesis also demonstrates a 2.5x density improvement (similar to Tabula and TSFPGA), although with half the clock frequency. The key advantage is the compile time reduction.

2.3.2 Coarse-Grained Architectures

DP-FPGA [21] (datapath-FPGA) has investigated using coarse-grained logic in an FPGA. By separating the routing networks for coarse-grained data and fine-grained control, the coarse-grained portions can save area by sharing configuration bits. The DP-FPGA tile is shown in Figure 2.5, where the 4-bit buses made use of configuration bit-sharing. Similarly, interconnect configuration-bit sharing and bus-based multiplexers were explored by Ye and Rose [94] using an architecture layout like the FPGA in Figure 2.1, but with some of the wires in each track converted to multi-bit buses. Their results show a 10% reduction in circuit area; however, the paper did not discuss time-multiplexing, which this thesis shows can also reduce area.

Wilton *et al.* [88] also investigated adding coarse-grained features to an FPGA by organizing the traditional FPGA LUTs into word-wide blocks, where each LUT computed a bit of the resulting word. These blocks were then connected using buses instead of individual wires. Area savings of 6x to 426x are reported compared to previous product-term-based pro-

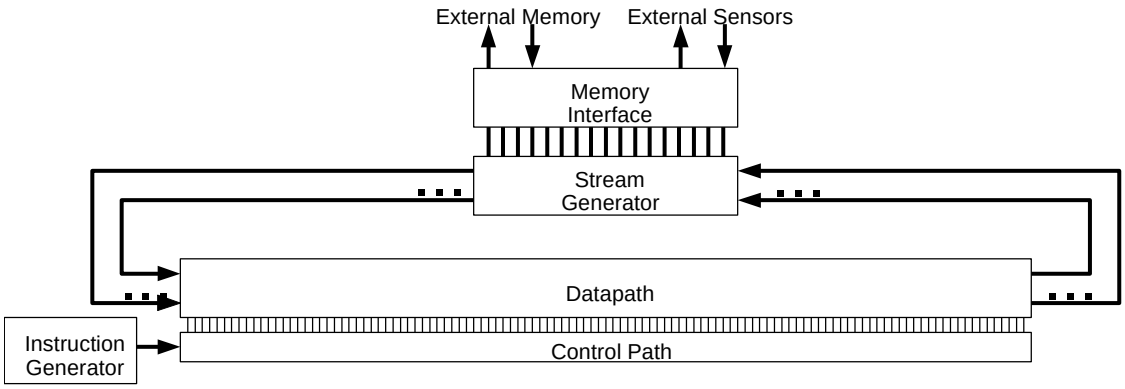
grammable cores (which are reported as 35% more dense than FPGAs [93]). However, this result is based on manually partitioning and mapping of each benchmark onto an architecture with different (optimized) parameter values for each benchmark. The final circuit speed is also not reported. As with DP-FPGA, the work by Wilton *et al.* did not explore time-multiplexing.

So far, the architectures presented in this section have added coarse-grained resources to an FPGA to improve density. However, without perfect placement these resources will not be 100% utilized, so further density improvements may be available through time-multiplexing.

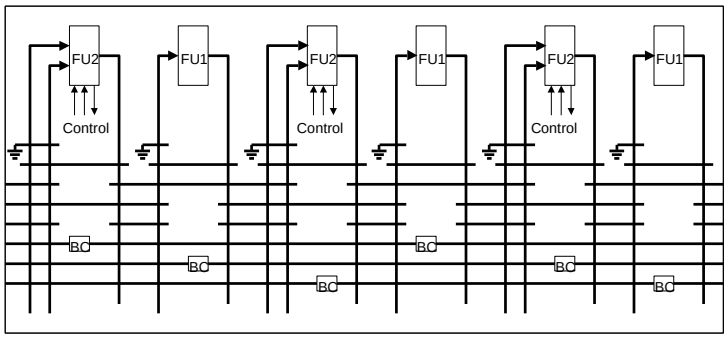
There has been a substantial amount of work done on coarse-grained architectures like Coarse-Grained Reconfigurable Arrays (CGRAs). These architectures are built from the ground up to be coarse-grained (they are not modifications of an FPGA). Also, they often use time-multiplexing more aggressively. However, these coarse-grained architectures almost exclusively deal with accelerating software "loop-kernels". The coarse-grained reconfigurable resources accelerate a portion of the software application, and the rest of the code is left for an attached host microprocessor. While this is a valuable approach for accelerating time-critical software, for our goal of mapping a complete circuit with similar density and performance as an FPGA, many of these previous architectures are unusable or require significant modifications.

RaPiD [26, 30] is perhaps the most well-known CGRA. It is a reconfigurable array with an attached RISC processor [26]. The high-level RaPiD architecture is shown in Figure 2.6a. The Datapath Cell, shown in Figure 2.6b, consists of functional units to perform word-oriented computation. Is replicated horizontally a number of times (depending on the target implementation) to form the complete Datapath block in Figure 2.6a. The Control Path Cell, shown in Figure 2.6c, consists of fine-grained resources with optional inverters to compute control signals for the datapath. It is also replicated horizontally to form the complete Control Path block.

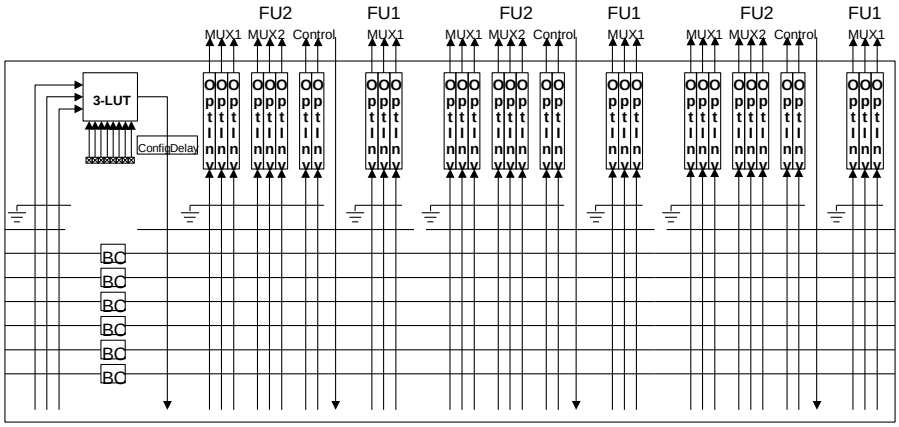
Most of the RaPiD resources are statically configured (configured once for each kernel), but some are dynamic and changed every clock cycle by a Control Path Cell. The data for a



(a)



(b)



(c)

Figure 2.6: The RaPiD architecture [26]. (a) The architecture block diagram, (b) the Datapath Cell, replicated horizontally to create the complete Datapath block, and (c) the Control Path Cell, also replicated horizontally to create the Control Path block. The "OptInv" block is an inverter with a bypass. The "BC" block is a a bus connector which may be connected or disconnected.

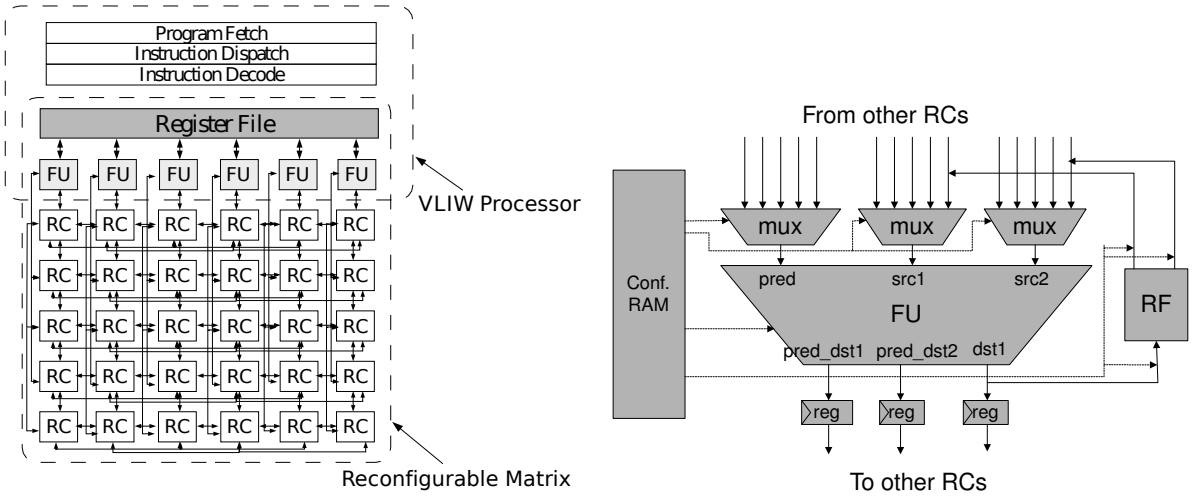


Figure 2.7: The ADRES core and reconfigurable cell [57]. (a) One possible configuration of the ADRES core, and (b) the ADRES Reconfigurable Cell (RC) consisting of a Functional Unit (FU) and Register File (RF).

RaPiD computation is streamed in and out of external memory using the Stream Generator, and the instructions for the computation are provided by the Instruction Generator. The fine-grained resources in RaPiD can be used to implement virtually any control structure, like state machines, so the architecture as a whole can function without external control from a host processor. However, RaPiD assumes that "the RaPiD datapaths will be integrated closely with a RISC engine on the same chip" [26]. The RISC processor is used to control the overall flow of the computation as well as perform unstructured computations (computation that involves branches or which does not have a regular or repeating pattern). However, the RaPiD authors note that most RaPiD applications do not execute any RISC instructions.

ADRES [57] is similar to RaPiD in that computation is directed through ALUs and registers, and a host processor (a VLIW processor in this case) oversees the entire operation. ADRES, however, organizes computation into a 2D array of Functional Units (FUs) and Reconfigurable Cells (RCs), where each RC consists of a FU and a Register File (RF), as shown in Figure 2.7. All communication with the host processor is done through the global register

file of the Very Large Instruction Word (VLIW) processor. In each Reconfigurable Cell (Figure 2.7b), a configuration RAM changes the mux, FU, and RF configuration inputs each clock cycle. The RC inputs can come from any other RC in the same row or column in a single clock cycle, which limits the size of the array and the maximum clock frequency. The coarse-grained resources in Malibu are similar to an ADRES RC, but they are allowed to communicate only with their immediate neighbours in one clock cycle (longer distance communication is done over multiple clock cycles); this ensures that Malibu can scale to a much larger architecture size.

MorphoSys [69] uses an architecture layout similar to ADRES, but with a Single Instruction, Multiple Data (SIMD) execution model. A context memory is used to reconfigure the fabric every clock cycle, supporting up to 32 different contexts. A RISC processor is used to oversee the reconfigurable fabric operation, execute code which is not part of a reconfigurable fabric kernel, and initiate data transfers to the fabric. Unlike ADRES, the processor does not share memory with the reconfigurable fabric. Instead, a DMA engine is used to stream data in and out of the fabric. The SIMD model used by MorphoSys means it cannot be used to implement a circuit.

Mosaic [83] is a family of CGRA architectures which includes both coarse-grained and fine-grained resources. Each Mosaic PE has four 32-bit functional units to implement datapath computation, and two 3x1 LUTs to implement control path logic. The PE also contains memories and register files to support time-multiplexing and data storage. As with most of the architectures in this section, Mosaic implements software loop kernels. In this thesis, we are interested in implementing coarse-grained circuits, and Malibu (this thesis) implements a complete Verilog CAD flow. Malibu also includes Verilog-specific operators in the ALU to make the circuit mapping more efficient.

PipeRench [33] is also a processor-controlled CGRA, but it is designed to be a coprocessor that can run independently from the main processor [34]. PipeRench supports multiple con-

current pipelined loop kernels, or more usually, multiple independent computations in a single "modulo" loop kernel where the tail wraps around to use the same physical resources as the head, all without the intervention of the main processor. The PipeRench architecture layout is similar to ADRES in Figure 2.7a, except that PipeRench calls each row of ALUs a stripe, and communication between ALUs is done through an interconnection network between each stripe.

PipeRench was significant because it was the first to support virtualized resources. The CAD tools compile the input using abstract "Pipe-Stage" resources, and only need to compile the input once. The Pipe-Stages are then dynamically scheduled on any and all available physical stages at runtime. This allows a kernel to run slowly on a small PipeRench architecture, or more quickly on a larger architecture. Similarly, WaveScalar [73] uses "waves" to virtualize and reuse existing hardware resources, TRIPS [67] uses "frames", and SCORE [20] uses "Compute Pages".

WaveScalar and TRIPS are designed to be standalone application-specific processors. Both implement one or more complete applications, usually computational or DSP in nature. WaveScalar is similar to PipeRench in that it uses interconnected ALUs to implement an application. TRIPS is more granular and uses four 16-core out-of-order processors which are partitioned at runtime to execute a stream of frames from a given application.

SCORE differs from these architectures with virtual resources (PipeRench, WaveScalar, and TRIPS) because it is not an architecture. SCORE is a model for implementing virtualized streaming computation on reconfigurable resources. It requires that the programmer manually partition the computation into streams, which are implemented on Compute Pages (reconfigurable resources) and Configurable Memory Blocks, or on a microprocessor. Most of the architectures in this section could implement a SCORE execution model; Malibu could as well.

Many of the architectures presented in this section use a host processor (ADRES, RaPiD,

PipeRench, MorphoSys). The processor is used to control computation and provide support for any unstructured computation which does not fit, or map nicely, onto the reconfigurable logic. Relying on the host processor to execute code is a sequential bottleneck that can slow down the implementation dramatically. Using the processor to control the configurable resources also means that it must regularly communicate with every reconfigurable resource in a reasonable amount of time, which can further limit the size of the CGRA and the performance. In this thesis we do not use a host processor, allowing Malibu to be a full replacement for an FPGA.

Even though the architectures presented in this section are not ideal for implementing coarse-grained circuits, they were valuable in both the creation of the CAD tools, and in providing insight into the design of the configurable resources for Malibu. Malibu distinguishes itself from previous approaches, such as those in this section, by: closely coupling the fine-grained and time-multiplexed, course-grained resources through an integrated interface in each CLB; using a time-multiplexed ALU which supports HDL operations like bit concatenation, unary logic reduction, and automatic truncation of results; implementing circuits (specified in Verilog), not software kernels; and providing a very fast compile time.

2.3.3 CGRA and Fast CAD

CGRA CAD approaches are well-documented in the literature (e.g., [32, 49, 50, 56, 95]), but ultimately these approaches have one of two drawbacks making them unusable for our purposes. First, they uses algorithms which assume the underlying architecture supports a global memory or has a host processor. Malibu has neither of these. Second, they depend on algorithms which do not scale as the problem size increases, which can result in long compile times as the CGRA architecture grows in size. Part of our motivation for bringing coarse-grained resources to an FPGA is to reduce the synthesis problem size and thus decrease the compile time for ever-larger FPGAs, so the algorithms used must scale.

Lee *et al.* [49] present a CGRA compilation approach that splits the mapping problem

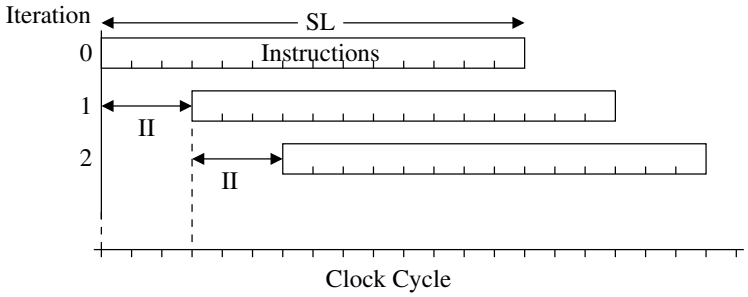


Figure 2.8: Initiation Interval (II) and Schedule Length (SL). In a modulo schedule, the same instruction sequence can be started in a new interval before previous intervals have finished. The waiting time before subsequent iterations can be started is the II. For the work in this thesis, SL equals the II, and the SL must be minimized.

into PEs, lines, and planes, and avoids 2D placement entirely. Reducing problem complexity like this is important for mapping to Malibu where the problem would be equivalent to 3D placement. However, these tools assume the existence of a global memory and a host processor.

Similarly, the graph-based CGRA mapping approach by Yoon *et al.* [95] will not scale up to large architecture sizes due to the NP-complete graph algorithms used. It is important to avoid such algorithms to achieve a fast compile time with large problem sizes.

A CGRA scheduler based on modulo graph embedding is presented by Park *et al.* [62, 63]. This scheduler breaks the scheduling problem into heights, scheduling the most critical (highest) heights first and working down a tree of ALAP arrival times. It repeatedly decreases the Initiation Interval (II) and re-runs the scheduler until the smallest II is found (the fastest recurrence), but places no restriction on the Schedule Length (SL). The difference between the II and SL is illustrated in Figure 2.8. The M-HOT scheduler is based on this approach, but we have made significant changes and additions. In particular, since II is the same as the SL in Malibu, we must minimize the SL (discussed next). M-HOT also runs scheduling only once, increasing the schedule length as necessary. This avoids multiple invocations of the algorithm to find a good quality solution. A description of all the differences is presented in Chapter 6.

Malibu attempts to minimize the SL for three reasons. First, a smaller SL means that each

CLB in the architecture can have less instruction memory. This reduces area and improves density. Second, reducing II to less than the SL does not improve the latency (input to output delay). Only minimizing the SL improves circuit latency. Third, compiling with an II<SL in Malibu is only valid on multi-cycle paths (combinational logic paths that are allowed to span more than one clock cycle), or on circuits which are strictly feed-forward (no cycles or feedback paths). Multi-cycle paths can be retimed into single-cycle circuits, and strictly feed-forward circuits are an interesting special case that we do not optimize for in this work. In this thesis, we assume that the II is equal to the SL.

Affinity [44, 62, 85] is a technique in CGRA scheduling used to place nodes with common descendants close together to minimize future routing costs. The work in this thesis uses the zero-based affinity definition from [62], which is described in Section 6.4.2.

The edge-centric modulo scheduler proposed by Park *et al.* [64] looks very promising for CGRAs, but will require some work before it can be used. It eliminates placement entirely by moving the placement decisions into the router. Hence, the computation on a path from a source to a sink will always be done in the processing elements visited by the route. Therefore, a router which can consider both congestion of routing and compute resources can perform the function of a placer. This approach depends on placing modulo paths first to reduce resource strain later. However, circuits have a large number of such paths (they are the output of every flip-flop, register, and memory), which may negate any gains of this approach.

Convergent scheduling [50] uses independent subroutines that each implement a heuristic to address a different mapping constraint, such as critical path reduction or load balancing. This approach is generic and applicable to any algorithm or architecture. However it is necessary to run many of the subroutines more than once to achieve a high-quality results. This may result in lengthy compile times.

The modulo scheduling mapping algorithm proposed in [56] is a modification of the DRESC tool [55]. The algorithm starts with a minimum II, and uses simulated annealing

to perform scheduling, increasing the II until a valid schedule is found. Like DRESC, the run-time of this approach is prohibitive. However, the dynamic schedule length implemented in this thesis in Chapter 6 was inspired by this work.

SPR [32] is an architecture-independent CGRA mapping tool for the Mosaic family of architectures [83]. Like Malibu, SPR can trade density for performance by leveraging time-multiplexing in coarse-grained resources. However, the focus of SPR is on achieving a high quality result with the smallest II for virtually any CGRA architecture specification. SPR often achieves an II that is less than ten. Our focus differs slightly in that we consider compile time to be of paramount importance, so the SPR iterative mapping approach is not usable for our work. We are also attempting to minimize the SL, not the II, for reasons discussed earlier in this section. However, the completeness and quality of solutions offered by SPR are worth investigating if the Malibu architecture could be expressed as an SPR architecture, and input circuits mapped to the SPR format.

Clustered VLIW architectures can also be viewed as spatial architectures, and many ideas from instruction scheduling tools [61, 65] can be adapted to CGRAs. In particular, list scheduling [24] is the most common instruction scheduling approach. It was used in this thesis for the same task (assigning instructions to coarse-grained timeslots in the Malibu architecture, see Section 5.5) after adding support for routing resources, communication delays, and fine-grained resources.

The Plasma/Teramac [8] system was a research project by HP Labs to build a circuit simulation platform with very fast synthesis speed. The platform used the Plasma FPGA [9], which was a custom-designed FPGA with far too many routing resources [51]. The idea was to over-design the hardware to achieve $O(n)$ placement and routing in the CAD tools. The whole Teramac system used 1,728 Plasma FPGAs to implement a 1 million gate circuit. It required 2 hours to place and route on computers of the day (1995). The work presented in this thesis can synthesize 1.6 million gates in about an hour on modern computer hardware, but it produces a

far more dense solution (requiring a single FPGA-sized device, not 1,728 of them).

The Ultra-Fast placement work by Sankar and Rose [66] demonstrates up to a 25x improvement in placement time compared to VPR with the "-fast" option (referred to as "VPR-fast") and achieves the same quality result as VPR-fast. The Ultra-Fast placer works by dividing the circuit into a number of levels and placing each level separately with a simulated annealer. This is similar to the approach described by Park in [63] for coarse-grained synthesis, but pre-dates it by seven years.

The Ultra-Fast placer is not timing-driven, nor is the version of VPR it was compared to. In this thesis, our algorithms, and modern VPR, are all timing-driven. Timing-driven placement requires 2.5x more time but improves the final circuit speed by 42% [53]. Our own testing in Appendix A shows that VPR routing takes approximately the same amount of time as timing-driven VPR-fast placement. However, front-end synthesis takes about 3x longer than routing. Even if Ultra-Fast placement could be reduced to take zero time, the overall CAD flow runtime would only be 20% faster. In this thesis, compared to the timing-driven VPR-fast flow and an iFAR architecture, the entire Malibu M-HOT flow shows an 8.4x compile time improvement with a 1.7x improvement in density. However, this also comes with a 2.8x performance (F_{max}) degradation.

Various FPGA placement and routing algorithms were tested together by Mulpuri and Hauck [59] to quantify the runtime and quality of results of the placement and routing steps in a CAD flow (excluding front-end synthesis). As with the Ultra-Fast placer, the version of VPR used by Mulpuri and Hauck was also not timing-driven. Comparing the original VPR (*inner_num*=10, PathFinder router) to a version of VPR with a *inner_num* down to 1 (same as VPR-fast) and a simple router, they report a 5x compile time speedup with a 2.5x quality reduction (2.5x increase in the critical path) using the MCNC benchmark circuits. Independent testing by Wrighton *et al.* [89] using the same MCNC benchmarks measured VPR-fast as 9x faster than the original VPR, with a 4% smaller critical path (larger F_{max}) and a 27% increase

in channel width (which means more area). Our own VPR testing in Appendix B with the benchmarks used in this thesis is closer to these latter results, finding a 3.6x faster compile time using VPR-fast (although this comparison uses timing-driven VPR and VPR-fast) with a 7% smaller critical path, but with a 5% increase in channel width. The Malibu results in this thesis are compared to a timing-driven VPR-fast, not the original VPR. The Malibu M-HOT tools are 8.4x faster than a complete VPR-based CAD flow (including front-end synthesis) that includes timing-driven VPR-fast, with a 1.7x density improvement and a 2.8x reduction in F_{max}. Comparing these results to the CAD tools in [59], the Malibu CAD tools are much faster and produce more dense results (less area), but have a similar quality degradation on the performance.

Malibu demonstrates improved compilation times and densities with a quality degradation comparable to previous fast FPGA CAD work. This previous work also excludes front-end synthesis and is not timing driven, which are both important features in modern CAD flows. For these reasons, we do not compare the results in this thesis directly to the Ultra-Fast placement work or other fast FPGA algorithms.

Fast CAD is not isolated to academic research. Modern commercial FPGA CAD tools also include options to reduce compilation time by sacrificing both area and performance. The commercial QuartusII tool used for comparison in this thesis includes such options. To investigate how fast QuartusII can go, we include a supplementary experiment in Appendix A where we have attempted to reduce compile time as much as possible by changing compilation configuration parameters to reduce compilation effort. Total compilation time is decreased by about 15% with a 13% reduction in F_{max}. In this thesis there is an average 26.1x decrease in compile time (M-HOT) with an average 50% reduction in F_{max} (M-HOT, maximum density). The Malibu approach offers significantly faster compile times.

Appendix A also investigates speeding up the academic VPR tool, which is also used for comparison in this thesis. By reducing placement and routing effort, we demonstrate a 7%

reduction in total compile time (including front-end synthesis) and a 25% reduction in F_{max}. Again, compared to the results in this thesis, the Malibu approach is significantly faster.

Chapter 3

Malibu Architecture

3.1 Overview

This chapter presents the Malibu architecture and describes how to implement a circuit using the coarse-grained and fine-grained resources available in the architecture. In Malibu, the word-wide operations from the source Verilog are mapped to the time-multiplexed ALUs, while the fine-grained logic, usually control logic, is extracted and mapped to the LUTs in the architecture. Before the details of the CAD flow can be presented in Chapters 4–6, the target architecture and the coarse-grained/fine-grained interface will be explained in this chapter.

The remainder of this chapter is organized as follows. In Section 3.2 the Malibu architecture is presented. Next, in Section 3.3, 21 benchmark circuits are introduced to evaluate the Malibu architecture and tools (for this and in later chapters). Then an experiment is done in Section 3.4 to determine architectural parameter settings for the Malibu architecture. Section 3.5 derives the instruction word encoding which is used in Section 3.7 to calculate the area of a Malibu CLB. Section 3.6 explains how a simulator was used to verify the benchmark results. Finally, concluding remarks about the architecture are in Section 3.8.

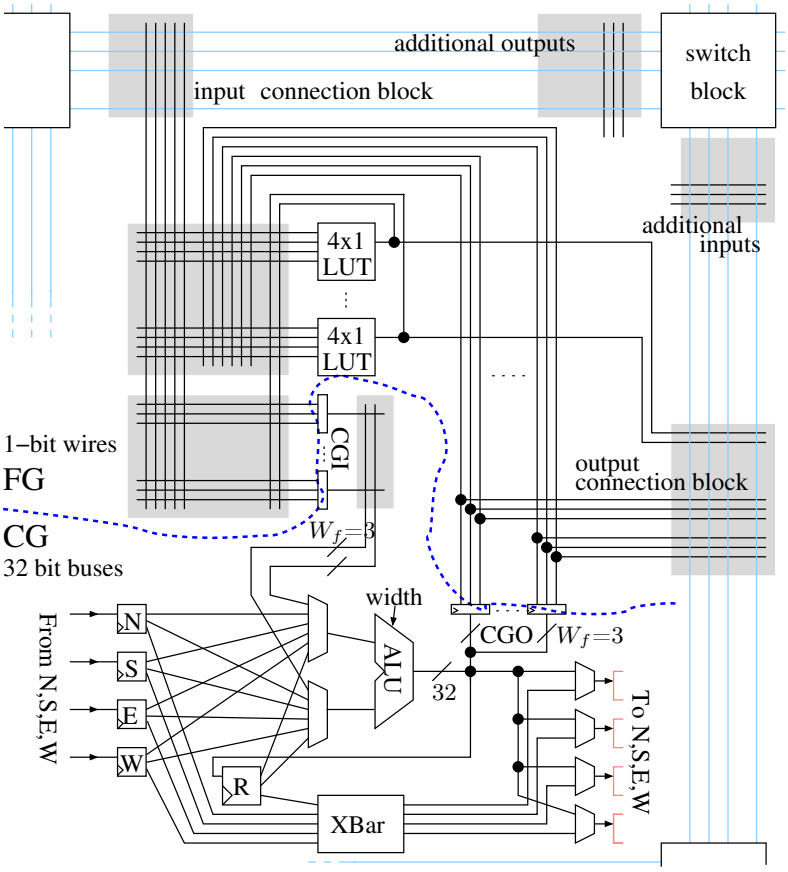


Figure 3.1: The Malibu architecture CLB. The coarse-grained (CG) part is added to a traditional FPGA CLB. Note the instruction memory and all control/address inputs have been omitted from this diagram.

3.2 The Malibu Architecture

The Malibu architecture CLB is shown in Figure 3.1. It starts with a standard Fine-Grained (FG) FPGA, as shown in Figure 2.1, and adds a time-multiplexed Coarse-Grained (CG) processing element to the CLB. The CG connects to the fine-grained components through Coarse-Grained Inputs (CGIs) and Coarse-Grained Outputs (CGOs). The Fine-Grained Width (W_f) parameter determines the width of the CG/FG interface. There are n_{CGI} CGIs, where each one aggregates a bundle of W_f signals from the fine-grained (FG) resources as an input to the ALU; the upper $32 - W_f$ bits of the CGIs are set to zero when used by the ALU. There are n_{CGO} CGOs, where each CGO latches the W_f least significant bits produced by a specific instruc-

tion, providing them to the LUTs or FPGA routing resources. When $W_f = 0$, the fine-grained resources (*all* traditional LUTs and interconnect) are excluded from the architecture, and the tools map the entire circuit to the CGs.

The CG contains an ALU for 32-bit computation. In addition to the two data inputs, the ALU uses a third input, *width*, to truncate (in other words, zero extend) the ALU results to a specified width. Many signals in a circuit are not exactly 32-bits wide and truncating these signals recreates the intended behaviour of the original Verilog. The exact use of the *width* field is discussed later in this section.

The CG also contains a local R memory which serves four purposes: i) storing intermediate results (simulating a wire), ii) preserving values between user clock cycles (simulating a register), iii) storing large 32-bit constants, and iv) storing data from user-instantiated memories. The CG also contains four neighbour memories, collectively referred to as NSEW, for storing results from the North, South, East, and West neighbouring CLBs. The crossbar (XBar) in the CG is discussed later in this section. Not shown in Figure 3.1 is an instruction memory and an instruction decoder which provides control signals to all the components.

Each CG is time-multiplexed; it always executes one instruction per system clock cycle from the instruction memory (not shown in Figure 3.1). All communication is explicitly pipelined and scheduled. As illustrated in Figure 3.2, the system clock differs from the user clock. On the active user clock edge, the instructions start executing, one per system clock cycle. Each CG contains a schedule with exactly *SL* instructions (the schedule length). At the end of the SL instructions, the CG pauses for the next user clock edge before starting over. One complete pass of the SL is required for each user clock cycle. We anticipate that a 1 GHz system clock can be achieved in 65nm technology using custom layout techniques. Therefore, the maximum achievable user clock frequency (the F_{max}) is $\frac{1}{SL} \cdot 1$ GHz.

The FG is a traditional FPGA CLB with the flip-flops removed, and with extra inputs and outputs to interface with the CG. The fine-grained resources are not time-multiplexed so that

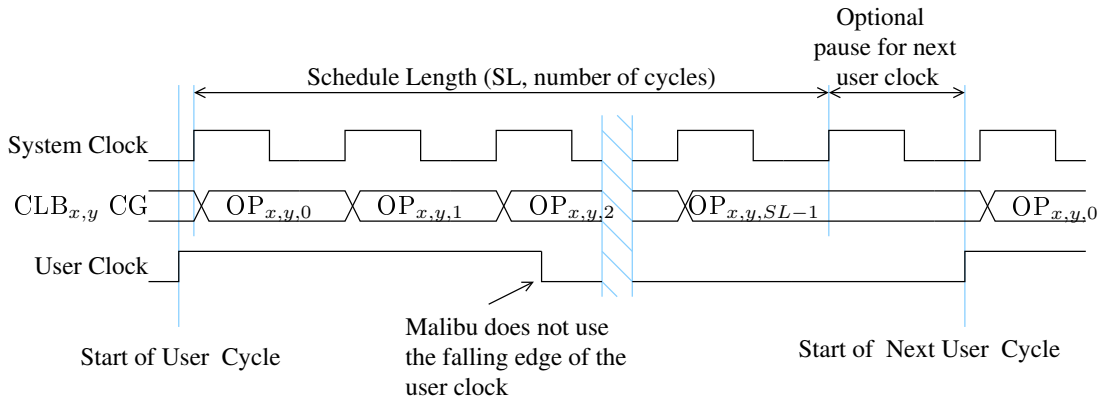


Figure 3.2: The user clock cycle.

they can be used to quickly distribute and compute combinational values. The state of any fine-grained flip-flop is stored in either the R or an NSEW memory and transfered to a CGO latch at the beginning of each user clock cycle so the value is stable for the duration of the user cycle. This was done to simplify the tools by having all signals/values which retain their value between user clock cycles stored in a coarse-grained memory.

The FPGA interconnect wires are directional and span only a single CLB in length. This is partly for simplicity, and partly due to the increased size of the CLB with the coarse-grained resources added. We also assume that the connection blocks and input/output blocks are fully-populated to simplify the mapping process. It is possible to change the architecture file passed to VPR for fine-grained routing (see Section 5.4) to accommodate more complex wire or connection block configurations. We save exploring the tradeoffs between such configurations as future work.

A complete list of the components in a Malibu CLB is shown in Table 3.1, including the estimated area using VPR's units of minimum-width transistor area (T). There is no area devoted to control-flow instructions in Malibu (like jump or branch) for three reasons:

1. There is no concept of "jumping over" part of a circuit in a real circuit. A synthesis tool will implement a conditional statement—whether for an FPGA, ASIC, or Malibu—by

Table 3.1: Malibu units and instructions.

Unit	Operators	ALU Instructions	Area (T)
Multiply	$\times$	MULS, MULU	35,000
Arithmetic	Arithmetic: +,–	ADD, SUB	1,995
	Comparison: $<, \leq, =, \neq$	LT, LTS, LEQ, LEQS, EQ, NEQ	
	Memory	LOAD, STORE	
Logic	Bitwise: $\&, \mid, \wedge, \sim$	AND, OR, XOR, NOT	3,208
	Ternary: ?:	MUX	
	Move	MOV	
	Sign extend	EXTS	
	Reductions: $\&, \mid, \wedge$	REDAND, REDOR, REDXOR	
Interconnect	4x4 Crossbar, CGIs, CGOs		5,482
	CG muxes		
BarrelShift	concat, $\ll$, $\gg$	CONCAT, BSL, BSR, LSL, LSR	1,791
FG	16 4-LUTs, LUT I/O		19,455
FG	Connection and Switch Blocks		17,519
		Total, Including Multiply	82,739[1]
		Total, No Multiply	47,739[1]

[1]This combined area is less than the sum of each block because of redundancy removal when combined.

creating the logic for all possible branches, and then creating a multiplexer to select the appropriate result. In a circuit all such paths are evaluated simultaneously because they are just logic components and wires. Since Malibu is designed for implementing circuits, it uses this same behaviour.

2. Malibu saves space in the instruction word by not storing write addresses or passing them to the register files (R,N,S,E,W). Instead, it generates write-addresses internally using a scheme that depends on perfect knowledge of which instructions are executed. This is discussed later in this section. Because of this, instructions cannot be skipped.

3. Malibu avoids synchronization between CGs by assuming the entire architecture can be statically and deterministically scheduled. If CGs were allowed to skip over code, addi-

tional hardware would be needed to synchronize the CGs, and additional effort would be required to create a valid schedule.

The area estimate in Table 3.1 for the Arithmetic, Logic, and BarrelShift blocks is from a manual gate-level design of each part, and the multiplier block is from [22]. For each block, the number of basic components (gates, muxes, etc.) were counted and converted into minimum-width transistors. For example, an OR gate requires 3 NMOS and 3 PMOS transistors, and a PMOS requires 1.5x the area of an NMOS, so the entire OR gate requires 7.5 T.

The area estimate for the interconnect and the FG is from a modified version of *transcount* [14], which reports the minimum-width transistors directly. The tool was modified several ways to compute the area of the Interconnect block:

- The crossbar and related CG muxes were modeled and added.
- The CG/FG interface logic and CGI/CGO registers were added using $n_{CGI} = 16$ and $n_{CGO} = 4$. These values are from the experiment in Section 3.4.
- The channel width area was added, based on a channel width of 120, computed by VPR. This value is also from Section 3.4.

The tool was also modified according to modifications made to the FG Block:

- The flip-flops in the FG were removed.
- Additional LUT inputs from the CGOs and LUT outputs to the CGIs were added.
- Additional CLB inputs and outputs directly to the CGOs and CGIs were added.

The combined area for the blocks is slightly less than the sum of individual unit areas due to redundancy removal when combined. The acronym MALIBU, an extension of ALU, originates from the name of these groups or units. Malibu comprises a total of 27 operations. Multiply

and comparison operations have both signed and unsigned variations. The signed version of an instruction ends with "S", and all other instructions are either unsigned or rely on two's complement for proper sign behaviour.

Table 3.2 lists each instruction supported by the ALU, and gives details of each operation. For the operands, *src*1 is always a memory location (R, N, S, E, or W), or a CGI, and *src*2 can be either a memory location, a CGI, or a signed 12-bit immediate value. Section 3.5 gives the precise instruction encoding.

Each instruction in the Malibu architecture executes in a single 1 GHz system clock cycle, except the LOAD and STORE instructions which require an extra cycle to index the R memory. Industry has demonstrated a 32x32 multiply in a single 1 GHz clock cycle in 65*nm* CMOS [78], so we believe that using a single-cycle ALU in Malibu at 1 GHz is reasonable.

Because a Verilog circuit often contains multi-bit signals which are less than 32-bits in width, the 32-bit ALU in each CG must generate results with the proper number of bits. Hence, the ALU must truncate the output result to the desired width, shown in the "output width" column in Table 3.2, by forcing the upper $32 - width$ bits to zero. The *width* is encoded in the instruction and is a separate input to the ALU as shown in Figure 3.1. When the ALU operation commences, it assumes the input operands are already of the correct width, and properly sign extended. This allows the input widths to be left unspecified, saving instruction bits. The CAD tools ensure the correct width of these inputs are provided by the upstream operations which appropriately choose their output width. If the upstream operation width does not match the desired input width, the tools will automatically insert zero-extend operations (EXT, which is mapped to a MOV instruction, see the bottom of Table 3.2) or sign-extend operations (EXTS) as necessary.

For some operations, the *width* field in the instruction encoding is used for a different purpose, and the output-width is implied:

Table 3.2: Malibu ALU operations.

Mnemonic	Operands	Operation Output	Output Width (bits)
ADD	*src*1,*src*2	*src*1 + *src*2	*width*
AND	*src*1,*src*2	*src*1 AND *src*2	*width*
BSL	*src*1,*src*2	{ 32-*width*{0}, *src*1[*width*-*src*2-1:0], *src*1[*width*-1:*width*-*src*2] }	32
BSR	*src*1,*src*2	{ 32-*width*{0}, *src*1[*src*2-1:0], *src*1[*width*-1:*src*2] }	32
CONCAT	*src*1,*src*2	{*src*1[32-*width*:0], *src*2[*width*-1:0]}	32
EQ	*src*1,*src*2	*src*1 == *src*2	1
EXTS	*src*2	{32-*width*{*src*2[*width*-1]}, *src*2[*width*-1:0]}	32
LEQ	*src*1,*src*2	*src*1 <= *src*2	1
LEQS	*src*1,*src*2	*src*1 <= *src*2	1
LOAD	*src*2	R[64 + *src*2]	–
LT	*src*1,*src*2	*src*1 < *src*2	1
LTS	*src*1,*src*2	*src*1 < *src*2	1
LSL	*src*1,*src*2	*src*1 << *src*2	*width*
LSR	*src*1,*src*2	*src*1 >> *src*2	*width*
MOV	*src*2	*src*2	*width*
MULU	*src*1,*src*2	*src*1 * *src*2	*width*
MULS	*src*1,*src*2	*src*1 * *src*2	*width*
MUX	*src*1,*src*2	CGI[*width*] ? *src*1 : *src*2	32
NEQ	*src*1,*src*2	*src*1 != *src*2	1
NOT	*src*2	~*src*2	*width*
OR	*src*1,*src*2	*src*1 OR *src*2	*width*
REDAND	*src*2	*src*2[*width*-1] AND *src*2[*width*-2] ... AND *src*2[0]	1
REDOR	*src*2	*src*2[*width*-1] OR *src*2[*width*-2] ... OR *src*2[0]	1
REDXOR	*src*2	*src*2[*width*-1] XOR *src*2[*width*-2] ... XOR *src*2[0]	1
STORE	*src*1,*src*2	R[64 + *src*2] ← *src*1	–
SUB	*src*1,*src*2	*src*1 - *src*2	*width*
XOR	*src*1,*src*2	*src*1 XOR *src*2	*width*
Mnemonics mapped to other Operations			**Conditions**
CONST	*src*2	MOV *src*2	*src*2 is immediate
EXT	*src*2	MOV *src*2	
GT	*src*1,*src*2	LEQ *src*2, *src*1	*src*2 is a register
		Rx ← LEQ *src*1, *src*2 ; NOT Rx	*src*2 is immediate
GTS	*src*1,*src*2	Same as GT but using LTES	
GTE	*src*1,*src*2	Same as GT but using LE	
GTES	*src*1,*src*2	Same as GT but using LTS	
REDNOT	*src*2	EQ *src*2, #0	

Note: *src1* is always a register. *src2* can be a register or a signed 12-bit immediate value.

- For `BSL` and `BSR`, *width* is used to specify the input width so the barrel shift can be done for the proper number of bits.
- For `CONCAT`, *width* is used to specify the number of bits to concatenate from the LSB input. The remaining $(32 - width)$ bits are taken from the lower bits of the MSB input.
- For `EXTS`, *width* is used to specify the number of bits in the input to sign extend.
- For `MUX`, *width* is used to specify which CGI to read the conditional input from. This multiplexer, which is not shown in Figure 3.1, is part of the ALU and has been included in all area calculations.
- For the reduction operations (`REDAND`, `REDOR`, `REDXOR`) *width* is used to specify the number of input bits to reduce.

The result of any instruction (except `LOAD` and `STORE`) can be written to one or more of the R, N, S, E, W memory, and a CGO register. Each of these memories operate synchronously using a single write port and up to three read ports. The instruction format (see Section 3.5) contains a *write enable* flag for each memory. To save space in the instruction word, the write address (write offset) for each destination memory is omitted and only the *write enable* is specified. The memory itself tracks which addresses are in use, and writes new data to the first unused address. The CAD tools use the same protocol to determine write-addresses and properly encode read-addresses in other instructions. All the operands and routing inputs in the instruction word include a *last read* flag to indicate to the hardware when a particular address can be marked as available.

All common Verilog operations can be easily mapped to the instructions in Table 3.2. For example, adding two 3-bit unsigned values into a 4-bit unsigned value in Verilog, written as `o = a + b`, can be expressed using 32-bit ALU operations in C language as:

```
o = ((a&0x7) + (b&0x7)) & 0xf
```

It could be implemented on Malibu using the following instructions, assuming `a` is in `R0` and `b` is in `R1`:

```
R2 ← AND R0, #0x7 (width=3)
R3 ← AND R1, #0x7 (width=3)
R4 ← ADD R2, R3 (width=4)
```

However, we apply optimizations during front-end synthesis to detect certain conditions. In this example, the first two operations are unnecessary except when the operation that generates `a` (or `b`) is wider than three bits. The width of that operation would not be 3-bits only if it also fans out to another operation which requires more than 3-bits, or if that instruction has a 32-bit implied output (`CONCAT`, `MUX`, `EXTS`, `BSL`, `BSR`). In these two cases, the fourth bit of `a` (or `b`, or both) must not be propagated into the `ADD` instruction; it must be truncated by an `AND` operation like above, or by an `EXT` operation which is what the synthesis tool would actually insert (or an `EXTS` operation if the numbers were signed). It does not matter if bits higher than the fourth propagate into the `ADD` instruction because the output is truncated to four bits, these higher bits would be immediately discarded. In all other cases, there is no truncation required and the front-end synthesizer would reduce the entire expression to a single operation, which also truncates the output to four bits:

```
R4 ← ADD R0, R1 (width=4)
```

To avoid introducing another memory block in the CG, all user-instantiated memory blocks are packed into offsets 64 to 127 in R. A user-instantiated memory is created by declaring an array in Verilog. Special `LOAD` and `STORE` instructions are used to access the memory data in R, and require one extra system cycle to perform the indexing. In this thesis, R has been sized to accommodate the user-memories in the benchmark circuits. However, if a user memory would exceed the space available in R, an error is reported. The problem of splitting a large user memory across multiple CG to avoid this error is left for future work.

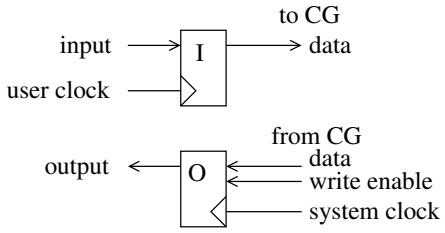


Figure 3.3: Malibu chip input/output logic. Input is captured once every user clock cycle and output is set immediately on write.

Figure 3.1 also shows a 4x4 routing crossbar (XBar). The crossbar is entirely combinational and writes values to the outgoing NSEW memories located in the four cardinal neighbours by taking values from the local incoming NSEW and R memories. Although there are 5 sources, the crossbar can be built with 4:1 muxes because each mux never accepts its own output direction as an input source. The crossbar routes coarse-grained signals concurrently with computation, and keeps CG communication off the FG routing resources. The ALU cannot write to the same NSEW memory as the crossbar in the same clock cycle. The CAD detects this condition and writes the ALU result to R instead, then schedules a transfer from R to the target NSEW in the next available cycle. However, the ALU-to-NSEW links are used often, so the ability of the ALU to write to NSEW directly is important for performance.

This coarse-grained routing architecture was chosen because it is simple and scalable to large array sizes. There are no long wires and there is no need to have a globally low-skew clock. Since each CG only needs to communicate with its four immediate neighbours in a single system clock cycle, a locally low-skew clock is sufficient. Alternate routing architectures are possible, for example, links which span multiple CLBs can be used to speed up data transfers, but this requires additional resources in each CLB to steer data into the right outgoing links and read data from the proper input links. Because of the complexity of such architecture exploration, it is left for future work and this thesis focuses on the CAD tools instead.

Chip input and output is accomplished using CLBs around the periphery of the architecture. Each CLB on the periphery has one CG neighbour channel that is not connected to another

CLB; for example CLBs along the top of the architecture have the N channel connected to I/O instead of a non-existent north neighbour. The CLBs in the corners have two unused channels, but only one is used for I/O to avoid creating a special case. The memory normally associated with the channel is replaced with an external I/O logic block shown in Figure 3.3 and the read/write address is not used (the I/O block contains only one value). The I/O provides an input value when the channel is read, and new inputs from off-chip are latched once per user clock cycle. This means the same input can be read as many times as necessary over the duration of the user cycle. When the channel is written, the output is latched immediately. The tools map circuit inputs and outputs to use these channels, and enforce the single input/output limitation in each CLB located on the periphery. For circuits that require more inputs we relax this constraint and report a warning, which would be an error in a commercial setting because a resource constraint is violated.

The schedule is pre-determined, making the entire architecture deterministic. It is the responsibility of the Malibu tools to schedule the code in each CG (controlling all of the memories, ALU, crossbar, and muxes), and to program each LUT, connection block, and switch block, so that data is always in the correct place at the correct time. Non-deterministic delays, such as waiting for input data from an external device, must be handled at the user-circuit level.

One limitation of this architecture is the assumption of a single user clock domain. For coarse-grained circuits created by a software-oriented programmer or generated by a C-to-gates flow, this is not necessarily a restriction; both will likely use a single clock and primarily use coarse-grained resources, making this type of architecture a natural fit. Nevertheless, it is important to address multiple clock domains in future work for coarse-grained circuits which require it.

Compared to previous architectures, Malibu is new in that it combines time-multiplexed, coarse-grained resources with traditional fine-grained FPGA resources. The coarse-grained resources reduce compile time and improve density, while the fine-grained resources provide a

fast mechanism to distribute control signals and improve the user-clock frequency of the final result. The Malibu architecture also uses an ALU that has specific support for HDL operations like bit concatenation and bit reduction. For the details presented in this chapter, please keep in mind this architecture is intended to be a starting point for many optimizations which have not yet been implemented, and to drive the creation of the CAD tools. The emphasis in this thesis is about understanding the CAD system for better synthesizing word-oriented circuits. A detailed architectural exploration is left for future work.

3.3 Benchmark Circuits

To evaluate the Malibu architecture and tools in this thesis, several Verilog benchmarks are used:

- The *chem*, *dir*, *honda*, *mcm*, *pr*, and *wang* benchmarks [72] are dataflow– and DSP-style combinational (non-pipelined) computational circuits described in behavioural Verilog.
- *me* is the motion estimation algorithm described in [35], written by the author. A block of 16x16 pixels is swept against a 32x32-pixel reference image, searching for the displacement which produces the lowest sum of absolute differences (SAD).
- *fft8* and *fft16* are 8- and 16-point complex FFTs respectively, implemented using a radix-2, decimation-in-time decomposition. These were written by Graeme Smecher.
- *jpeg_enc* is a JPEG encoder from [81].
- The other benchmarks are the 11 largest (in Quartus ALM count) from the IWLS 2005 benchmark set [23], excluding the circuits with a top-level entity name of the form "`sXXXXX`". These excluded circuits contain only 2-input logic gates and single-bit wires. They appear to be the gate-level output of another synthesis tool since all the internal

Table 3.3: Benchmark circuit list and StratixIII resource use. QuartusII synthesis used the default settings.

		QuartusII/StratixIII					
				Memory (bits)			
	Circuit	ALMs	18×18	Registers	M9K	M144K	MLAB
Coarse-Grained Only (CG-only)	fft16	6,412	84	10,232	0	0	0
	me	5,148	0	8,066	0	0	0
	chem	3,526	175	0	0	0	0
	fft8	2,075	28	3,842	0	0	0
	honda	1,216	52	0	0	0	0
	mcm	1,057	56	0	0	0	0
	wang	797	24	0	0	0	0
	pr	646	18	0	0	0	0
Good Circuits (Good)	ac97_ctrl	1,254	0	2,199	0	0	0
	aes_core	1,154	0	540	0	0	0
	dir	1,150	8	600	0	0	0
	spi	488	0	229	0	0	0
	pci_master	137	0	138	0	0	0
Impaired Circuits (Impaired)	ethernet	6,868	0	10,553	0	0	0
	wb_conmax	5,349	0	1,090	0	0	0
	dma	1,714	0	1,756	0	0	1,536
	tv80	850	0	347	0	0	0
	jpeg_enc	791	64	1,476	0	0	0
	systemcaes	716	0	675	0	0	0
	des	298	0	1,986	0	0	0
	systemcdes	237	0	190	0	0	0

nodes and nets are named with a sequential number. Such pre-synthesized single-bit circuits are not the type of circuit we expect to efficiently support; they require no coarse-grained resources and there is no opportunity to extract word-oriented operations from the Verilog source. It may be possible to recover the coarse-grained features of such circuits with an intelligent high-level analysis algorithm, but that is outside the scope of this thesis. However, we have ensured that our tools can successfully map such circuits.

To get an idea of the size of these benchmarks, Table 3.3 shows the ALM count generated using QuartusII 10.0 targeting a StratixIII (EP3SL340F1760C2) FPGA. These benchmarks are further discussed in Section 4.6 after the front-end synthesis has been presented in Chapter 4.

After the results in Chapters 5 and 6 were generated, an investigation of each benchmark was done to determine why it mapped well (or poorly) to the Malibu architecture. As a result, the benchmarks have been categorized into one of three groups:

- ***CG-only*** – Coarse-grained only circuits. These are circuits with no fine-grained signals at all, and thus will not make use of the fine-grained resources in Malibu. These circuits map well (good performance, good density, good compile time) to the Malibu architecture.

- ***Good*** – Coarse-grained circuits with a small amount of fine-grained control logic (maps well onto Malibu), or circuits that use programming constructs which map well onto Malibu.

- ***Impaired*** – These circuits use Verilog structures which do not map well to Malibu. These circuits are written in a style which is not efficiently supported by either our front-end synthesis, the back-end mapping flow (M-CAD and M-HOT), or the Malibu architecture, or some combination thereof. It may be possible to improve the mapping results of these circuits by rewriting the Verilog into another style, improving the CAD tools, improving the architecture, or some combination of these approaches.

These three groups, the benchmark circuits, and the Verilog structures which map poorly onto Malibu are discussed further in Section 4.6.

3.4 Architectural Parameter Values

The Malibu architecture was introduced in Section 3.2, but the architectural parameter values were not specified. These parameters must be completely specified to calculate the area and evaluate density. In this section we perform an experiment to choose a channel width, the number of LUTs per CLB, the size and number of the CGI and CGO structures, and the size

of the R, N, S, E, W, and instruction memories. The tools are first run in an exploratory mode, allowing the number of resources to float, so that the natural demand for each resource can be determined. Then, the tools are run in regular mode, where architecture constraints are enforced, to see the impact on the benchmarks.

Table 3.4 shows the resource usage results of the M-CAD flow run in exploratory mode for $W_f = 1$. For each benchmark, the architecture array size was kept square and swept from 3x3 to 48x48 CLBs. The result with the smallest schedule length (SL) is shown. The smallest schedule length represents the fastest user circuit $\left(F = \frac{1\text{ GHz}}{SL}\right)$. In the case of a tie, the smallest array size is reported. The architecture parameter values chosen from this data will allow the maximum speed of the circuits to be realized.

At the bottom of Table 3.4 are fixed values chosen for the architectural parameters. In addition, the largest user memory in the benchmarks is 2 kbit. Since R is also used to implement user memories, we add 64 more entries (x32 bits wide) to R for a total of 128 enties. The upper 64 entries are only accessible with the special `LOAD` and `STORE` instructions. As previously mentioned, splitting large user memories across multiple CLBs is left for future work.

The benchmarks constrained by these architectural restrictions are bolded along with the specific values that exceed the fixed parameter values. The results for the remaining (non-bolded) benchmarks will not change under the fixed parameters. To see the impact of these restrictions, Table 3.5 shows the results for the benchmarks after being synthesized with restricted fixed parameter values. The results that are unchanged are not shown. Note that in Table 3.5 the channel width is always reported as 120 because VPR was invoked with a fixed channel width and not allowed to search for the smallest channel width.

For the *ethernet*, *wb_conmax*, *dma*, and *tv80* benchmarks, the major constraint was the channel width. In these cases, to successfully build the benchmark with the fixed parameters the tools had to select a larger array. The *ethernet* benchmark went from 10x10 CLBs to 20x20, and the other three increased as well. In doing so, the maximum number of LUTs per CLB

Table 3.4: Resources required at the smallest schedule length for $W_f = 1$. Bolded results exceed the fixed architectural parameter value specified at the bottom of the table.

				FG Resources per CLB				CG Resources per CLB				
	Circuit	SL_{float}	**CLBs**	**CW**	**LUTs**	**CGI**	**CGO**	**R**	**N**	**S**	**E**	**W**
CG-only	fft16	22	16x16	0	0	0	0	10	3	3	2	3
	me	18	20x20	0	0	0	0	6	2	2	1	2
	chem	25	8x8	0	0	0	0	6	2	2	2	2
	fft8	14	16x16	0	0	0	0	5	2	2	2	2
	honda	22	6x6	0	0	0	0	8	1	2	1	1
	mcm	14	6x6	0	0	0	0	2	2	2	1	1
	wang	12	4x4	0	0	0	0	6	1	2	1	1
	pr	12	4x4	0	0	0	0	5	2	1	1	1
Good	ac97_ctrl	25	12x12	74	11	12	2	8	4	5	2	2
	aes_core	40	8x8	26	5	3	2	21	4	5	2	2
	dir	37	6x6	36	5	4	4	4	5	5	3	3
	spi	31	6x6	12	2	3	2	2	2	2	2	2
	pci_master	34	4x4	30	9	12	**5**	6	1	1	1	1
Impaired	**ethernet**	61	10x10	**200**	**28**	16	4	17	4	3	2	2
	wb_conmax	51	20x20	**208**	16	10	3	12	10	10	4	5
	dma	98	16x16	**164**	14	9	**5**	12	4	5	2	2
	tv80	109	20x20	**160**	13	11	**5**	28	3	4	2	2
	jpeg_enc	193	40x40	10	2	3	2	**257**	**23**	**23**	2	2
	systemcaes	65	48x48	14	1	3	1	**65**	5	3	2	2
	des	191	28x28	96	13	10	**5**	6	2	2	1	1
	systemcdes	46	20x20	26	5	5	**6**	4	3	2	2	1
	Arch Value:	**256**		**120**	**16**	**16**	**4**	**64 (+64)**[1]	**16**	**16**	**16**	**16**

CW: Fine-grained channel width.
[1]+64 Entries for user-instantiated memory mapped to R.

was reduced, as was the usage of the R, N, S, E, and W resources. This is expected given the increase in the number of resources available in the increased array size. For the *wb_conmax* and *dma* benchmarks there is a further interesting result; the schedule length of the solution is unchanged despite the circuit being forced to spread out on a larger array to stay within resource usage constraints. The tools use a criticality measurement to prioritize placement of the critical path. For these benchmarks, and all the circuits in Table 3.5 with $\frac{SL_{float}}{SL_{fixed}} \geq 1.00$, the post-scheduling critical path is not lengthened when architectural restrictions are enforced.

The *jpeg_enc* and *systemcaes* circuits show a similar critical-path result but without any

Table 3.5: Resource usage with fixed parameter values for $W_f = 1$. SL_{float} is from Table 3.4.

	Circuit	SL_{fixed}	$\frac{SL_{float}}{SL_{fixed}}$	CLBs	FG Resources per CLB				CG Resources per CLB				
					CW	LUTs	CGI	CGO	R	N	S	E	W
Good	pci_master	34	1.00	4x4	120	11	12	4	6	1	1	1	1
Impaired	ethernet	84	0.73	20x20	120	10	16	4	8	4	6	2	2
	wb_conmax	51	1.00	48x48	120	7	5	4	4	5	5	4	4
	dma	98	1.00	20x20	120	7	12	4	5	4	3	3	3
	tv80	204	0.53	40x40	120	6	8	4	6	3	3	2	2
	jpeg_enc	190	1.02	40x40	120	3	16	1	1	2	2	2	2
	systemcaes	65	1.00	48x48	120	1	3	2	14	5	2	2	2
	des	191	1.00	28x28	120	13	10	4	6	2	2	1	1
	systemcdes	46	1.00	20x20	120	5	5	4	3	3	2	2	1
	Arch Value:	**256**			**120**	**16**	**16**	**4**	**64 (+64)**	**16**	**16**	**16**	**16**

change in the array size. In exploratory mode there is no penalty for bunching up computation on a single CLB, because no constraints are violated, so *jpeg_enc* was free to use 257 entries in one of the R memories. Each instruction takes one cycle to compute and write a result back to R. However, it also takes one cycle to compute and write a result to a neighbour memory (N,S,E, or W). So, with restrictions on R, both *jpeg_enc* and *systemcaes* end up distributing the computation among several CLBs; each CLB does a portion of the computation (without exceeding the R memory limit) and passes the result to the next CLB. Doing this achieves the same critical-path length (or smaller for *jpeg_enc*), with the same array size, but with a more even distribution of resource use. The schedule length reduction in *jpeg_enc* is a little unexpected, but not impossible given that placement is done with a heuristic (simulated annealing). Given enough time, or enough repeated trials, the placer on the unconstrained architecture might also find this solution. Also, since the difference in this case is small (3 out of 190 timeslots), there is no concern that the heuristics are converging poorly (e.g., by getting easily stuck in a local minima).

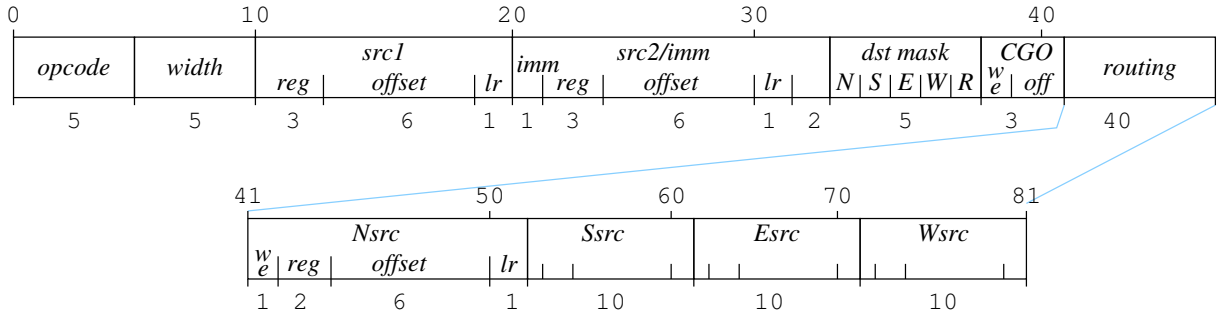


Figure 3.4: The 81-bit Malibu instruction word.

The remaining three benchmarks, *des*, *systemcdes*, and *pci_master* fit within the resource constraints by relocating a few coarse-grain to fine-grain interface registers (CGI/CGOs) to another CLB. Since the fine-grained routing network is fast, there is no penalty for this alternative solution, and the schedule-length results are unchanged.

3.5 Instruction Format

Now that all necessary parameters have a fixed value, it is possible to calculate an instruction encoding, shown in Figure 3.4. This encoding is required to estimate the silicon area of the instruction memory.

Each CG instruction has been encoded into 81 bits, including all of the source addresses, all required flags, and crossbar control. In contrast, there are well over 1,000 configuration bits in a traditional VPR-style CLB.[1] However, after time-multiplexing, the CG requires $SL \times 81$ bits. If the user requests a small array for a large circuit, upwards of 1024 instructions per CLB might be required. However, as demonstrated in Section 3.4, only 256 instructions per CLB are needed to achieve the maximum-frequency results across all of the benchmark circuits. A long-term goal is to significantly reduce this value through optimizations in the front-end synthesis, architecture, and CAD steps presented in this thesis.

The parts of the instruction word are as follows:

[1]Ten 6-LUTs require 640 bits, the 60 LUT inputs require at least 5 bits each, plus bits needed to configure flip-flops and all of the interconnect.

- *opcode* (5 bits) – Encodes the 27 different operations.
- *width* (5 bits) – Specifies the output width for automatic truncation.
- *src1* (10 bits) – The first source operand is always a memory/register file or a CGI. Three bits are required to select the input memory/register file or CGI, and six bits to specify the address (offset) within the input. The largest input is the R memory with 64 entries, so the offset must be large enough to encode any of the 64 addresses in R.[2] The final bit is the *last read* flag to indicate when an addresses is not needed in the memory anymore.
- *src2/imm* (13 bits) – The first bit indicates whether *src2* is a memory source or is immediate data. If it is memory, then it requires the same additional 10 bits as *src1*. If it is immediate, then all 12 remaining bits are sign extended to 32 bits and used as the second operand.[3]
- *dst mask* (5 bits) – The destination mask is a single bit for each output memory (R, N, S, E, W) to indicate whether the data produced by the ALU should be written to the next available memory offset.
- *CGO* (3 bits) – The first bit is write-enable, and the next two indicate to which CGO offset to write.
- *Nsrc*, *Ssrc*, *Esrc*, *Wsrc* (each 10 bits) – Similar to *src1*, but with a write-enable bit. If enabled, it specifies the source location of the value to write to the north (south, east, west) neighbour channel.
- For `LOAD` and `STORE` instructions, the base offset of the user-memory within the R memory is a 6 bit value which is encoded into the *dst mask* and *CGO* fields. The upper

[2]Note that R actually has 128 entries, but the upper 64 entries are reserved for user-defined memories and are only accessed by the `LOAD` and `STORE` instructions.

[3]An early version of the architecture used 13 bits for both *src1* and *src2/imm* fields. Although *src2/imm* can be made smaller, we chose to keep it at 13 bits to save re-running all of the experimental results.

four bits are stored in the N, S, E, and W bits in the *dst mask* field, and the lower two bits are stored in the offset of the *CGO* field. The *dst mask:R* bit is 1 for `LOAD` and 0 for `STORE`. The *CGO*:*we* bit is always set to 0 to disable writing to the CGOs.

3.6 Verification of Results

A cycle-accurate simulator for Malibu has been developed to verify the CAD flow synthesis results on the Malibu architecture. As input, the simulator uses the bitstream created by the Malibu CAD tools (M-CAD or M-HOT) and a stimulus waveform in the QuartusII 9.x Vector Waveform File (VWF) format. We have created a stimulus input for every benchmark circuit. As output, the simulator creates a waveform also in the VWF format.

QuartusII was also used to synthesize and simulate each circuit for the largest StratixIII FPGA. The same stimulus VWF inputs were used in the QuartusII 9.x simulator to create a repertoire of known-good output waveform results. All the results in this thesis for $W_f = 0$ have been checked against the known-good results and verified to be correct.

The simulator does not implement the fine-grained resources, so these results were not verified through simulation. Instead, several small benchmarks were created to exercise the capabilities of the fine-grained resources, and the results were verified manually. We are confident that the fine-grained results are also functionally correct.

3.7 CLB Area

The area of a Malibu architecture CLB is the sum of the hardware blocks and the memories:

$$area = \#CLBs \times (area\_blocks + area\_memories) \tag{3.1}$$

The area of *area_blocks* is the total area from Table 3.1. For all results in this thesis, one in

Table 3.6: Malibu memory area estimates.

		SRAM μm^2		eDRAM μm^2		Flash μm^2	
Memory	**Specification**	**Area**	**Per Bit**	**Area**	**Per Bit**	**Area**	**Per Bit**
NSEW	32x16, 3R1W	5,430	10.615	–	–	–	–
R	32x128, 3R1W	27,244	6.651	–	–	–	–
Instr.	81x256, 1RW	20,717	0.999	6,013	0.290	1,421	0.0686

five CLB columns contain a multiplier, so the average per CLB is:

$$area\_blocks = \frac{4}{5} \cdot 47,739 + \frac{1}{5} \cdot 82,739 = 54,739T \tag{3.2}$$

VPR/iFAR sets the area of one minimum-width transistor (1T) for logic at $\approx 0.5\mu m^2$ in 65nm:

$$area\_blocks = 54,739 \times 0.5 = 27,369.5\mu m^2 \tag{3.3}$$

The value of *area_memories* (the R, NSEW, and instruction memories) is found by using the CACTI memory modelling tool [60] as shown in the SRAM columns in Table 3.6. The NSEW and R memories need very fast read and write access, so they are implemented in SRAM. The instruction memory is primarily read-only and accessed sequentially, allowing it to be pipelined. It may be implementable in SRAM, eDRAM, or flash. To be consistent with the StratixIII FPGA, we also implement it in SRAM.

Using the CACTI SRAM results, the area required for the memory is:

$$area\_memories = 4 \times 5,430 + 27,244 + 20,717 = 69,681\mu m^2 \tag{3.4}$$

Therefore the total area for a Malibu CLB array is:

$$area = \#CLBs \times (27,369.5 + 69,681) = \#CLBs \times 97,050.5\mu m^2 \tag{3.5}$$

As a sanity-check, a 32-bit ARM core in 65nm with a 32-bit ALU which includes a 32-bit multiplier and no cache is approximately the same size, 0.1 mm^2 [10].

There are other memory technologies like eDRAM and flash which are smaller than SRAM. Although they are slower, they may be usable as the instruction memory since it can be accessed sequentially in a pipelined fashion. Table 3.6 also shows alternative memory configurations for the instruction memory which may further reduce the area of a Malibu CLB and improve density. These instruction memory areas were estimated using technology parameters for eDRAM [41] and flash [48]. However, additional work is required to verify these estimates and whether they can be used in Malibu.

3.7.1 Area for Comparison to VPR/iFAR

To convert back to VPR minimum-width transistor-area (T), the same conversion factor of 1T$\approx 0.5\mu m^2$ in 65nm is used:

$$area = \#CLBs \times 194,101T \tag{3.6}$$

This value is used in Chapters 5 and 6 to compute the area of various sized architectures after the CAD tools have been explained.

3.7.2 Area for Comparison to QuartusII/StratixIII

To compare the area of a Malibu array to a StratixIII FPGA, the area of the Malibu CLB is converted into equivalent ALMs. To do this, the area of a StratixIII ALM is needed. We estimate this ALM area using the following procedure:

1. The size (and area) of the StratixII components are measured on an enlarged die photo of an EP2S60 device shown in Figure 3.5.

2. These measured areas of the EP2S60 are then scaled up to the EP2S180 using the ratios

Table 3.7: StratixII resources.

		EP2S60	EP2S180	Ratio
Logic	ALM	24,176	71,760	2.97
	M512	329	930	2.83
	M4K	255	768	3.01
	18×18	144	384	2.67
	MRAM	2	9	4.5
	I/O	718	1,170	1.63
	die size	unknown	≈600 mm^2	

in Table 3.7. We convert to the EP2S180 because we have a good estimate of the physical size of that device, 600mm^2 (David Lewis, private communication).

3. The scaled areas for the EP2S180 are then scaled down to a physical area based on a 600mm^2 device size, keeping the same core aspect ratio.

4. The physical area for the logic (that is, the core area excluding the MRAMs and periphery I/O) is divided by the number of ALMs in the EP2S180.

5. Since the StratixII was fabricated in 90nm, and the StratixIII in 65nm, the result is divided by 2 to convert to 65nm.

6. Since the StratixII and StratixIII use the same ALM, this area-estimate result is similar for the StratixIII.

The details of these calculations are shown in Table 3.8. Overall, this gives us an approximate StratixIII ALM area of 2,674μm^2 (or 26,740μm^2 per LAB). Using the estimated CLB area computed previously in this section, the number of StratixIII ALMs per Malibu CLB is $\frac{97{,}050.5}{2{,}674} \approx 36.3$. This means that each Malibu CLB is the same silicon area as 36.3 StratixIII ALMs (or nearly 4 LABs). If one Malibu CLB implements the same user logic as 36.3 ALMs, the architecture density would be the same. The area/density comparisons in Chapters 5 and 6

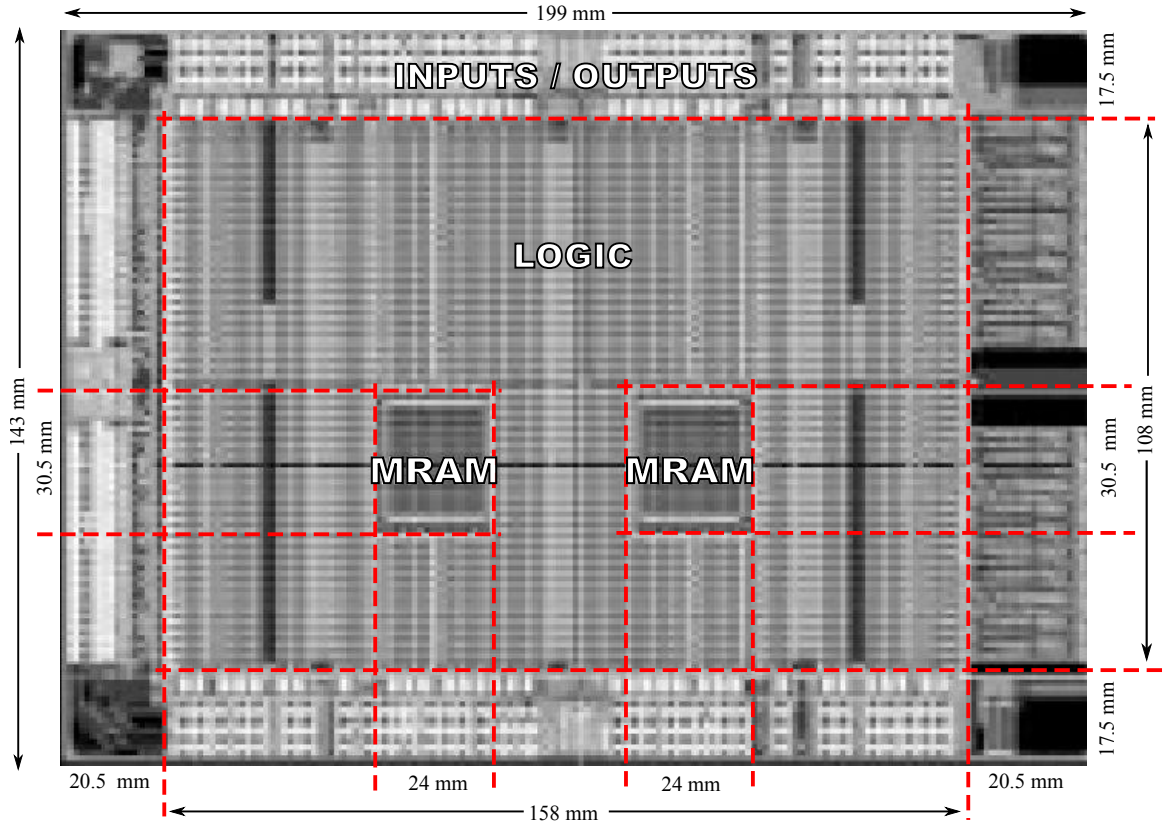


Figure 3.5: StratixII EP2S60 die photo. Dimensions are for a full-page blowup of the figure (not actual chip dimensions). Photo courtesy of Mike Hutton/Altera.

compute the number of Equivalent StratixIII ALMs (eALMs) by multiplying the number of Malibu CLBs by 36.3. Doing this allows the Malibu area and density to be compared directly to QuartusII/StratixIII.

3.8 Conclusions

This chapter has presented the Malibu architecture and computed the area of a Malibu CLB. The CAD tools (presented in Chapters 4–6) were used to determine architectural parameter values. This constrained some benchmark circuits, mostly the *Impaired* circuits, leading to larger Malibu array sizes for a successful synthesis. The tools were able to work around these fixed restrictions and find a successful synthesis solution for all the benchmarks.

Table 3.8: StratixIII ALM area calculation.

	Item	Width (mm)	Height (mm)	Area (mm^2)	Aspect Ratio
Enlarged EP2S60 (From Figure 3.5)	MRAM1	24.0	30.5	732.0	
	MRAM2	24.0	30.5	732.0	
	Core (excluding I/O)	158.0	107.5	16985.0	1.47
	Chip	199.0	143.0	28457.0	
	I/O (Chip - Core)	41.0	35.5		
	Logic Only (Core - MRAMs)			15521.0	
Enlarged Estimated EP2S180	Logic Only (3x EP2S60 Logic)			46563.0	
	MRAM (4.5x EP2S60 MRAMs)			6588.0	
	Core (Logic + MRAMs, same aspect ratio as EP2S60)	279.5	190.2	53151.0	1.47
	Chip (Core + I/O width and height)	320.5	225.7	72325.5	
Physical (Scaled to $600mm^2$) EP2S180	Chip (Scaled to $600mm^2$)	28.6	21.0	600.0	
	Core (Same aspect ratio as EP2S60)	25.4	17.3	438.0	1.47
	MRAMs			54.3	
	Logic (Core - MRAMs)			383.7	
	Per ALM (Logic / 71,760)			0.005348	
	Per ALM in 65nm (converted from 90nm)			0.002674	
				= 2,674μm^2	

Once the architecture parameter values were fixed, the area of the Malibu CLB was computed, and was determined to be 70% memory (by silicon area). Using alternate technologies like eDRAM or flash could reduce the memory area, leading to further density gains. However, that work is outside the scope of this thesis.

Finally, the Malibu CLB area was expressed in minimum-width transistors for comparisons to VPR/iFAR, and in equivalent ALMs for comparisons to QuartusII/StratixIII in Chapters 4–6.

Chapter 4

Front-End Synthesis

4.1 Overview

Mapping a circuit onto a hybrid time-multiplexed coarse-grained/fine-grained architecture is an unexplored problem, and there is little in the literature to directly guide the creation of such a CAD tool. Regardless of the approach taken, however, the first step must be the same: the Verilog source (or VHDL) must be parsed, elaborated, optimized, and expressed in some intermediate format for the tools to continue processing. This is called front-end synthesis.

Figure 4.1a shows an academic FPGA CAD tool flow. In this flow, the commercial tool QuartusII is used to perform front-end synthesis. The OdinII [39] academic tool is being developed for front-end synthesis, but it currently only implements a subset of Verilog, and thus cannot be used to synthesize the benchmarks presented in Section 3.3. The remainder of the academic flow uses T-VPack and VPR, which have become the *de facto* tools for academic FPGA clustering, placement, and routing.

To map circuits into a hybrid time-multiplexed coarse-grained/fine-grained architecture such as Malibu, a new tool flow is needed. This thesis presents two such flows: Malibu-CAD (M-CAD) and Malibu-HOT (M-HOT). These two flows are shown in Figures 4.1b and 4.1c,

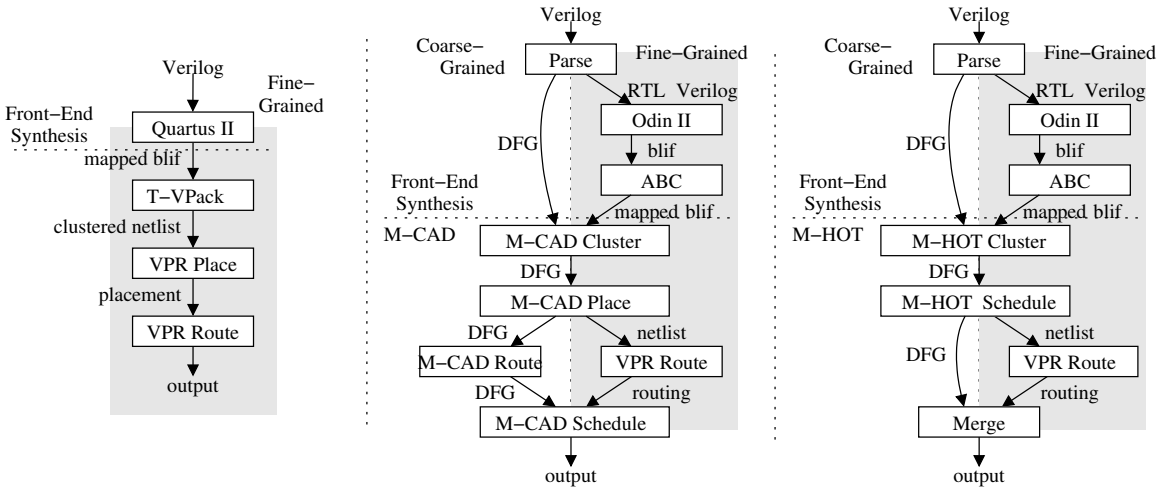


Figure 4.1: Three CAD flows: (a) Academic (traditional), (b) M-CAD, and (c) M-HOT. M-CAD follows the traditional place-then-route flow. M-HOT does placement, routing, and scheduling simultaneously in the same tool.

respectively. Both flows start with the same front-end synthesis step which is primarily based on the Verilator open source tool and is the subject of this chapter. Verilator is a Verilog to C++ synthesis tool that can parse, elaborate, and optimize the full synthesizable subset of Verilog 2005 at a coarse-grained level (that is, it preserves coarse-grained entities). This makes it a very capable front-end tool to generate input for the back-end synthesis flows, M-CAD and M-HOT.

However, since the Malibu architecture also has fine-grained resources, a second approach is needed to synthesize fine-grained logic and signals on to these resources. The fine-grained width threshold (W_f) is an architectural parameter used to separate the coarse-grained and fine-grained components. To simplify synthesis, all nodes and edges wider than W_f are implemented on coarse-grained resources, and nodes and edges of width W_f or smaller are flagged for fine-grained synthesis. The front-end synthesis flow generates an OdinII-compatible Verilog description of all the fine-grained components, and then OdinII and ABC are used for fine-grained synthesis. When both coarse-grained and fine-grained synthesis are done, either

the M-CAD or M-HOT flow is invoked. The M-CAD flow follows the same order of operations as a traditional FPGA CAD flow and is discussed Chapter 5. The M-HOT (Height-Oriented Tool) flow performs placement, routing, and scheduling simultaneously and is discussed in Chapter 6. Generally, the M-CAD flow runs a bit faster, while the M-HOT flow produces better quality results.

For a coarse-grained architecture, the front-end synthesis problem is somewhat the same as in fine-grained FPGA CAD, except that we do not wish to reduce the circuit to a gate level. Coarse-grained operations must remain intact to make use of the coarse-grained resources in Malibu. A secondary requirement is that, for good performance, the synthesis must extract and preserve as much parallelism as possible from the source circuit.

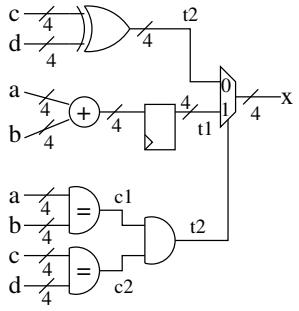
4.2 Circuit Representation

The Malibu CAD tools use a Data Flow Graph (DFG) representation of the circuit. Each graph node is a circuit operation and each graph edge is a communication event. Figure 4.2a gives an example Verilog specification of a circuit which operates as follows. On a rising clock edge, the circuit assigns a value to register `t1`. The circuit also determines if `a` equals `b` and `c` equals `d`. If they do, it assigns the next register value `t1` to the output `x`; otherwise it assigns `c XOR d`. The schematic of this circuit is shown in Figure 4.2b.

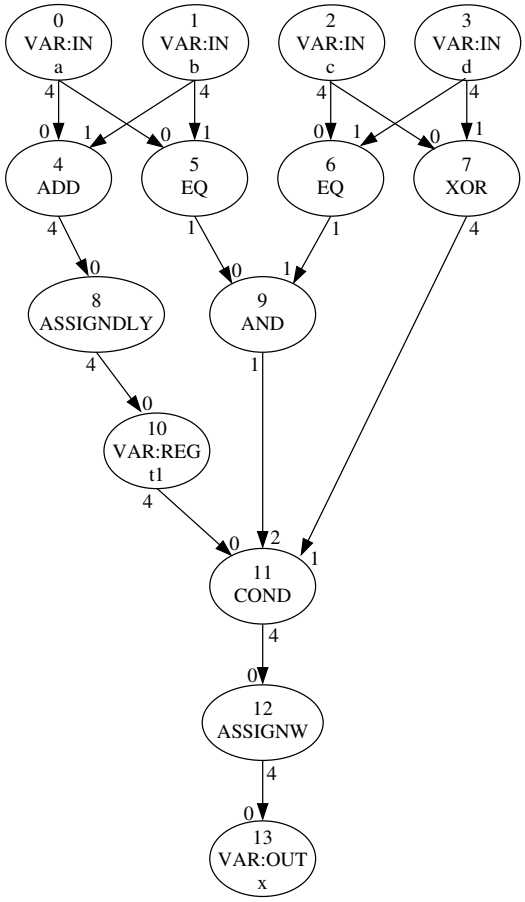
The DFG representation of this circuit is shown in Figure 4.2c. This was generated using a modified version of Verilator, which is discussed in the next section. Each node has a unique identifier, a type, an output width (the number outside the bottom of each node), a set of ordered sources starting at 0 (the numbers outside the top of each node), and a set of unordered sinks. For each sink, there is also a delay in system clock cycles which is not shown in the figure because it is calculated during placement.

```
1  module top(clk, a, b, c, d, x);
2    input clk;
3    input [3:0] a, b, c, d;
4    output [3:0] x;
5    reg [3:0] t1;
6    wire [3:0] t2;
7    wire c1, c2;
8
9    assign c1 = (a == b) ? 1'b1 : 1'b0;
10   assign c2 = (c == d) ? 1'b1 : 1'b0;
11   assign t2 = c ^ d;
12   assign x = (c1 & c2) ? t1 : t2;
13
14   always @(posedge clk) begin
15     t1 <= a + b;
16   end
17 endmodule
```

(a)



(b)



(c)

Figure 4.2: Example: Verilog source and Verilator output. (a) Verilog source for the circuit example used throughout this thesis, (b) a schematic representation of the circuit, and (c) the Verilator DFG output.

4.3 Parsing and Elaboration

The first step in front-end synthesis is to turn the source Verilog into a DFG representation of the circuit for further processing. Verilator [71] is used to parse the Verilog input and perform several optimizations. We do not use commercial tools like QuartusII for this because, even though the circuit is presumably coarse-grained at some point during the internal processing, there is no way to access these intermediate structures, and the output is a bit-oriented circuit.

Verilator is a compiled-code simulation tool. It normally synthesizes a circuit to a sequen-

tial language (C++), which in turn is compiled and executed on a desktop computer. Hence, if Verilator is allowed to run to completion, it will remove all the parallelism from the circuit. This is not desirable for Malibu, so Verilator was modified to stop and output a DFG after coarse-grained optimizations are applied but before it begins to serialize the graph. The optimizations that Verilator applies are:

- Module elaboration – Instantiated modules are included and duplicated as needed to create a single, flattened coarse-grained circuit with no hierarchy.
- Dead code elimination – Unused parts of the circuit are discarded.
- Constant folding – Operations with constant inputs are pre-calculated and the result is inserted as a constant input.
- "Free" hardware operation removal – Operations like bit selection (accessing a slice of a bus) are converted into shift and mask instructions. Unfortunately, Verilator performs this conversion for fine-grained logic too, so we rely on OdinII to re-discover the original operations when synthesizing the fine-grained parts of the circuit (which it does by detecting the shift and mask instructions). No parallelization or other information is lost by doing this, it only takes additional processing time.

These optimizations are done at the word-level, whereas FPGA CAD tools first convert the circuit to bit-level Boolean expressions, then apply logic optimizations. While the latter is effective for logic optimization purposes, it loses all the coarse-grained information.

Using Verilator in this manner works well, and the output is a high-quality DFG for many circuits. However, since Verilator is not a full-scale commercial-quality synthesis tool designed to perform full synthesis of parallel logic, it performs poorly in some cases (see Section 4.6). In one case (*wb_conmax*), the size of the fine-grained part of the circuit produced by Verilator is the same as the entire circuit when synthesized by QuartusII. This may be because some

optimizations are unavailable at the word-level, or because those optimizations have not been fully developed or implemented yet.

Figure 4.2c shows the Verilator DFG output. Verilator uses a verbose set of node types which must be mapped to the 27 instructions in the Malibu architecture. Notice that wires (`c1` and `c2`) have already been synthesized away, and the constants used by the ternary operators are optimized out. Also notice that the register (`t1`) remains because it is required to preserve the proper clock-edge behaviour. The entire DFG is coarse-grained except for node 9 which does a computation on fine-grained logic to control the conditional assignment in node 11.

In Figure 4.2c, there are some nodes which are not self-explanatory:

- `ASSIGNDLY` is a delayed assignment to a variable that only takes effect at the end of the current clock cycle.
- `ASSIGNW` is a non-delayed assignment to a variable that takes effect immediately.
- `COND` is a two-input multiplexer with a control signal. If the value on input 2 is zero, then input 0 is passed to the output, else input 1 is passed.
- `VAR:REG` is a register. It outputs a constant value for the duration of a user clock cycle. The value may change only at the end of a user clock cycle.

A complete list of the Verilator nodes, and their mappings to Malibu, can be found in Appendix C.

4.4 Coarse-Grained Synthesis

In addition to the 27 Malibu ALU operations in Table 3.1, our output circuit representation uses five other nodes to store information about the circuit:

- `INPUT` and `OUTPUT` – These nodes are for circuit I/O and are used by the placer to

allocate the appropriate input/output resources in CLBs around the periphery of the architecture. They are not executed by the ALU and do not appear in the final schedule.

- `CGO` and `CGI` – These nodes are the gateway between the coarse-grained and fine-grained operations. Nodes in the fanout cone of a `CGO` node are considered fine-grained down to the next `CGI` node. The placer uses these nodes to help track the CGI/CGO resource usage. They are not executed by the ALU and do not appear in the final schedule.

- `CONST` – This node provides a constant input to the circuit. Constants which are larger than 12-bits (too large to fold into the 12-bit immediate field of an instruction) are left in the circuit. The scheduler duplicates these nodes after placement (so that a `CONST` node never fans out to a node outside a CLB), and then implements the constant by allocating a dedicated offset in the relevant R memories for each constant value. We assume that when a bitstream is loaded into the Malibu architecture that these initial values in the R memory will be loaded as well.

Most of the nodes from Verilator are trivially converted to the 27 Malibu ALU operations and the above five placeholder nodes. For example, a `COND` node is replaced by a `MUX` operation. A complete listing of all such mappings is included in Appendix C. Even nodes like `CONCAT`, which would otherwise map into a series of masks and shifts, are directly supported. However, there are some non-trivial transformations required to legalize the DFG for the Malibu architecture:

- Verilator maps multiple writes to a single variable (a register, wire, or variable in the source Verilog) to a series of sequential IF statements which overwrite the same variable. While this is good for execution on a sequential processor (which is exactly why Verilator does it), this is not implementable in hardware. These assignments are detected and mapped to a binary decision tree that feeds a single write operation. This also allows

the computation of the written value to be (potentially) distributed among CLBs while having one instance of the final value. Since this conversion adds logic to the circuit, it could increase the critical path by the depth of the decision tree ($\lceil log_2 \rceil$ of the number of writes), or could cause some other path to become critical, reducing the maximum achievable use clock frequency.

- User-instantiated memories, represented as array operations in the DFG, are mapped to R memory locations. The clustering tool ensures all operations ot the same user memory reside in the same CLB.

- To keep memory usage low, multi-port ROMs in the user circuit are time-multiplexed rather than replicated. In most of the benchmarks a single ROM is independently accessed by only two or three reads. However, an extreme example is the Advanced Encryption Standard (AES) benchmark, *aes_core*, which uses a 256 byte substitution box (s-box) that is instantiated 20 times. It is good behaviour for Verilator to time-multiplex this memory because the C++ program it would normally create can read from the same memory as often as necessary. However, this is problematic for Malibu because the 20 separate LOAD operations are required to be in the same CLB to read from the same memory. It would be better to use a heuristic in Verilator to determine when to replicate and when to time-multiplex such memories, but we have not implemented that feature.

- Signals declared with the Verilog `reg` keyword will cause a register to be inserted in the DFG (if Verilator does not optimize it away). For these signals, the DFG node fanning out to a register (`VAR:REG`) is flagged as *registered*. If the node also fans out to a non-register it is duplicated first, and only one is marked as *registered*. All registers are then removed and replaced with wires leaving the node with the *registered* flag. This flag causes special treatment in the scheduler to recreate the expected clock-edge register behaviour.

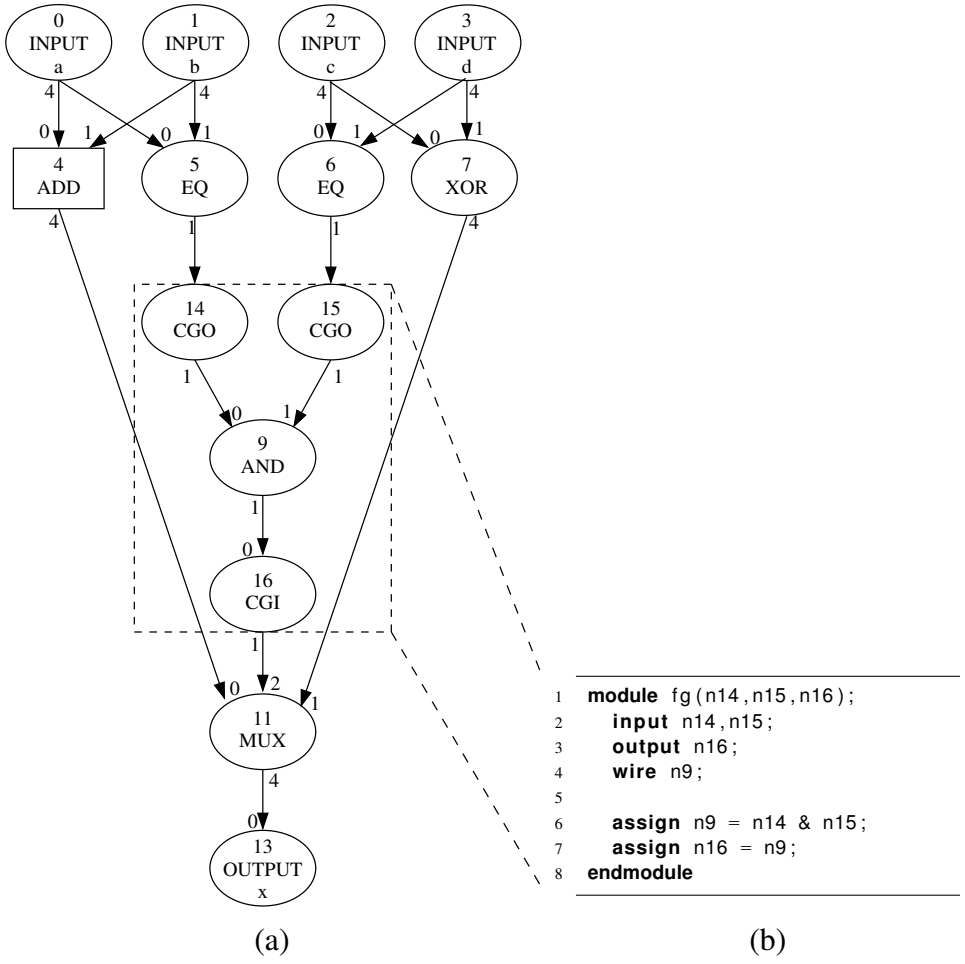


Figure 4.3: Example: DFG after coarse-grained synthesis. (a) The DFG, and (b) the generated RTL Verilog for fine-grained synthesis.

- Any constants less than or equal to 12 bits wide are folded directly into the instructions it feeds as an immediate operand.
- Nodes and edges of width $\leq W_f$ are considered fine-grained and marked to use the FG resources. `CGO` and `CGI` nodes are inserted around all such fine-grained logic.

Figure 4.3a shows the circuit after these transformations have been applied. The `ASSIGN` nodes have all been removed. Node 4 (ADD) is redrawn as a box to indicate it is *registered*; this means the output should only become available at the beginning of the next user cycle. The

VAR:REG node has been removed, and the VAR:IN and VAR:OUT nodes have been replaced. Finally, the fine-grained logic has been detected (using $W_f = 1$) and the appropriate CGO and CGI nodes have been inserted.

If there is no fine-grained logic in the circuit, or W_f has been set to 0, then fine-grained synthesis is skipped, and front-end synthesis is now complete. However, additional steps are required to process the fine-grained components, as described in the next section.

4.5 Fine-Grained Synthesis

The fine-grained parts of the circuit (DFG nodes and edges with width W_f or smaller) are written out from Verilator as OdinII-compatible RTL Verilog, synthesized to LUTs using OdinII and ABC, then merged back into the coarse-grained DFG for clustering. By using RTL Verilog, even multi-bit operations can be easily written (for cases when $W_f > 1$). OdinII elaborates the design to single-bit operations, and ABC cleans up any dangling logic (e.g., from an add operation where the carry bit is discarded) and technology-maps the logic to LUTs.

Figure 4.3b shows the RTL Verilog generated for the fine-grained part of the circuit in Figure 4.3a. The CGO and CGI nodes are used as inputs and outputs. In this example, the resulting fine-grained synthesis will return a single LUT.

After fine-grained synthesis, the LUTs are merged back with the coarse-grained operations to form a complete DFG, shown in Figure 4.4. To perform the merge, all nodes between the CGO and CGI nodes are deleted. Then, the Berkley Logic Interchange Format (BLIF) file output of ABC is used to create a new LUT node for each tech-mapped LUT, and the truth table data is attached to the node. The names of the CGO and CGI signals are not changed by OdinII or ABC, so input/output matching can be easily done. However, it is possible that some fine-grained logic will be optimized away, leaving some CGO nodes with no fanout (e.g., if a Boolean algebra reduction determines that a signal will have a constant value, all logic driving that signal will be removed). Thus, after merging, the dead-code elimination optimization is

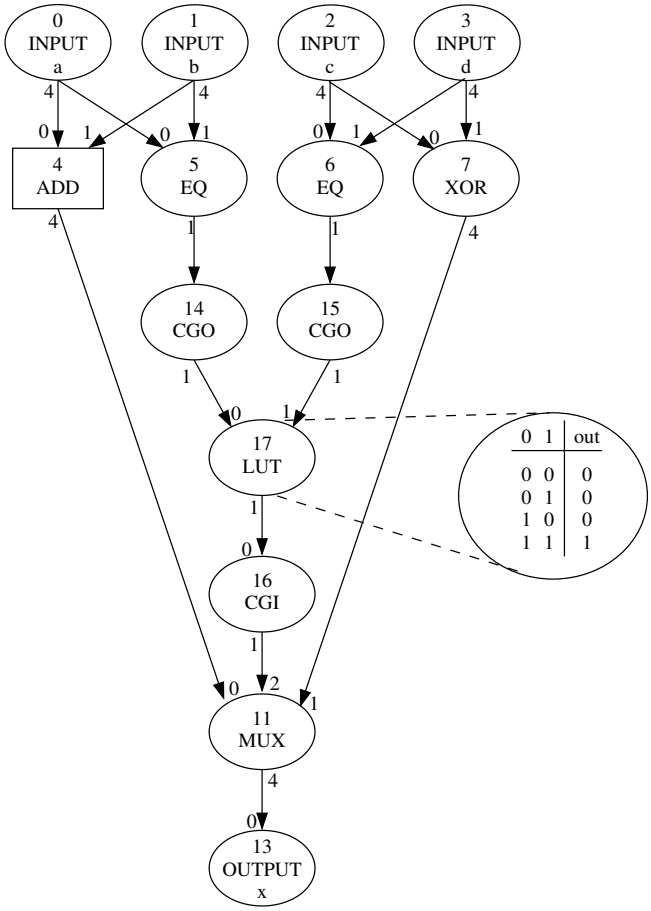


Figure 4.4: Example: DFG after fine-grained synthesis. Node 9 has been replaced by a LUT node which contains a truth table.

run again to delete any dangling logic.

Placing LUTs in CLBs is integrated with the placement tool and is done concurrently with placement of the CG nodes. The number of CGI and CGO interfaces needed in a CLB will change while performing CG placement, depending where nodes around the LUT are placed and the availability of CGI and CGO resources. For this reason, LUT placement must be considered concurrently with CG operation placement.

Table 4.1: Front-end synthesis results.

		QuartusII for StratixIII		QuartusII for VPR	Malibu Front-End Synthesis									
								% Nodes			Nodes in DFG			
	Circuit	ALMs	18×	4-LUTs	Nodes	Nets	%r	1b	2b	4b	$W_f = 0$	$W_f = 1$	$W_f = 2$	$W_f = 4$
CG-only	fft16	6,412	84	17,006	2,120	2,236	29	0	0	0	2,120	–	–	–
	me	5,148	0	14,388	5,954	7,020	14	0	0	0	5,954	–	–	–
	chem	3,526	175	36,143	568	714	0	0	0	0	568	–	–	–
	fft8	2,075	28	5,248	800	836	29	0	0	0	800	–	–	–
	honda	1,216	52	3,795	249	293	0	0	0	0	249	–	–	–
	mcm	1,057	56	3,067	232	288	0	0	0	0	232	–	–	–
	wang	797	24	2,275	134	152	0	0	0	0	134	–	–	–
	pr	646	18	1,893	176	194	0	0	0	0	176	–	–	–
Good	ac97_ctrl	1,254	0	3,538	4,911	6,097	8	47	10	6	4,911	3,187	3,033	2,936
	aes_core	1,154	0	5,021	3,380	3,970	1	8	1	8	3,380	2,191	2,229	2,209
	dir	1,150	8	6,620	884	1,190	6	22	3	16	884	884	846	616
	spi	488	0	987	664	856	6	37	4	9	664	505	504	469
	pci_master	137	0	325	957	1,342	8	71	3	16	957	728	716	563
Impaired	ethernet	6,868	0	19,626	9,693	13,686	15	61	4	6	9,693	7,528	7,219	6,574
	wb_conmax	5,349	0	16,098	17,917	23,558	2	40	8	22	17,917	13,566	13,782	8,017
	dma	1,714	0	5,071	18,514	23,650	6	41	3	9	18,514	14,018	14,241	8,284
	tv80	850	0	2,330	12,186	16,027	2	44	4	25	12,186	9,227	9,374	5,453
	jpeg_enc	791	64	2,836	4,486	5,882	11	13	0	11	4,486	4,197	4,197	4,580
	systemcaes	716	0	2,181	3,043	3,799	0	23	8	26	3,043	2,570	2,545	2,352
	des	298	0	865	4,114	5,497	0	34	0	2	4,114	3,856	3,856	3,829
	systemcdes	237	0	650	1,688	2,131	0	24	1	4	1,688	1,922	1,905	1,939

18×: Number of 18×18 multipliers required.

%r: Percent of nodes which are registered (output of a register).

%Nodes: Percent of nodes which output a one-bit (**1b**), two-bit (**2b**), and three- or four-bit (**4b**) value.

4.6 Benchmark Evaluation

Table 4.1 presents the results of front-end synthesis with QuartusII and Malibu. The first two QuartusII columns (ALM count and the number of 18×18 multipliers required) are for synthesis to a StratixIII FPGA. A complete list of the resources required for each benchmark on a StratixIII is in Table 3.3. The StratixIII ALM is most commonly thought of as a 6-input, 2-output lookup table (a 6x2 LUT), but is reconfigurable into different modes ranging from two 4-LUTs to some 8-input functions [4].

The third QuartusII column, the number of 4-LUTs, is taken from the QuartusII-generated BLIF for VPR. VPR only accepts a BLIF file as input. Unfortunately, we were unable to use QuartusII to produce a BLIF with hard multiplier blocks, so the 4-LUT BLIF for VPR uses LUTs exclusively.

Comparing the ALMs to the 4-LUTs, the results are mostly as expected: fewer ALMs than 4-LUTs are required, and benchmarks with multipliers use many more 4-LUTs.

The Malibu results in Table 4.1 are from the front-end logic synthesis described in this chapter (before clustering, placement, routing, or scheduling has been done). The "%r" column is the percent of nodes which are registered; this value is important in Chapter 6 for the M-HOT approach where all registered nodes are at $height = 0$. A large number of registered nodes means the M-HOT scheduler may do a disproportionate amount of work at $height = 0$, and thus may run slowly. The "%Nodes" columns are the percent of nodes which output a one-bit (1b), two-bit (2b), and three- or four-bit (4b) value. The fine-grained resources implement the 1b nodes when $W_f = 1$, the 1b and 2b nodes when $W_f = 2$, and the 1b, 2b, and 4b nodes when $W_f = 4$.

For most of the coarse-grained benchmarks (*CG-only*), the number of Malibu nodes is less than the QuartusII ALM count. This is expected because each node can have up to 64-bits of input and 32-bits of output, whereas an ALM is at most an 8-input, 2-output operation.

A node versus ALM comparison is not ideal, but it does give a sense for the relative size of each circuit, and for the number of entities the CAD tools must process. The Malibu front-end synthesis is operating as expected for these coarse-grained circuits. The exception is the *me* benchmark where QuartusII is able to better-optimize the result to give a circuit with fewer ALMs than nodes. This suggests that there are additional optimizations that are not being applied by Verilator during front-end synthesis.

For the *Good* circuits, the Malibu node count is 3-6x higher than the ALM count. This makes sense because of the one-bit node percentages (the 1b column in Table 4.1). In this table, all nodes are being mapped to CG operations, so circuits with one-bit operations are using unnecessary resources. It is possible to reduce the node count considerably after mapping to LUTs because a LUT can collapse many Boolean expressions into a single step (the LUT). The last four columns in Table 4.1 show a decreasing node count as more Malibu CG operations are migrated to the LUTs (as W_f is increased). For these results, fine-grained synthesis is performed. For the *Good* circuits, the Malibu node count is reduced to almost half. The $W_f = 0$ and $W_f = 1$ node counts for the *dir* benchmark are the same because there are a number of `COND` operations which have a single one-bit computation for the conditional input in the benchmark. When $W_f = 1$, that single node is implemented in a single LUT, giving the same node count.

The significant increase from ALMs to nodes for the *pci_master* benchmark is caused by a user-instantiated look-up table created with a case statement. Section 4.6.1 describes four Verilog structures, including this one, which map poorly to the Malibu architecture. For the *pci_master* benchmark, it uses about 200 more nodes than necessary. We are not modifying the benchmark circuits in this thesis, so we leave this result as-is. However, fixing this would also bring *pci_master* down near the same 2x range versus the ALM count.

For most of the *Impaired* circuits, QuartusII is able to generate a much smaller ALM count compared to the Verilator node count. This is true even when the different values for W_f are

considered in Table 4.1. The *des*, *dma*, and *tv80* benchmarks have a node count over 10x the ALM count. These same circuits also have the longest schedule lengths in Table 3.4, which means they have the lowest user clock speeds. The primary reason is that these benchmarks use Verilog structures which do not map well to coarse-grained architectures, as is explained in the next section. This further suggests the need for improvements to the front-end logic synthesis in future work.

4.6.1 Bad Verilog Structures

We have identified four bad Verilog structures that should be avoided when designing circuits for Malibu. Even though Verilog is a general language, skilled designers often write Verilog code to target a specific technology (e.g., a specific FPGA with specific resources). This approach would be needed for Malibu as well; the designer would need to learn to avoid the Verilog structures presented in this section.

To help alleviate the impact of these structures, we have implemented a technique to do parallel evaluation of case statements while preserving the priority final-assignment order as required by Verilog semantics. This helps for some situations in cases 3 and 4 below (case LUTs and serial cases), but it is not an ideal solution. The four bad structures are explained below and their usage by the benchmark circuits is summarized in Table 4.2 under the following headings:

1. **Bit-Level Coarse Ops** – Specifying coarse operations at the bit level. The type of operation is given (e.g., <<) followed by two values (`X`, `Y`), where `X` is the number of instances of that type of operation, and `Y` is the largest output-width of that operation.

2. **Bit Aggregating** – Using a bus to aggregate individual bits. Three numbers are specified: "`X`s, `Y`rd, `Z`wr". `X` is the total number buses used to aggregate individual bits, `Y` is the number of reads from such signals, and `Z` is the number of writes to such signals.

Table 4.2: Bad Verilog structures used in benchmark circuits.

		Bad Construct			
	Circuit	**Bit-Level Coarse Ops**	**Bit Aggregating**	**Case LUT**	**Serial Case**
CG-only	fft16	–	–	–	–
	me	–	–	–	–
	chem	–	–	–	–
	fft8	–	–	–	–
	honda	–	–	–	–
	mcm	–	–	–	–
	wang	–	–	–	–
	pr	–	–	–	–
Good	ac97_ctrl	–	3s, 12rd, 0wr	–	10 (10)
	aes_core	–	–	512 (256)	–
	dir	–	–	–	–
	spi	–	–	–	–
	pci_master	–	–	15 (15)	–
Impaired	ethernet	<< (2, 10)	15s, 55rd, 56wr	10 (10)	16 (8)
	wb_conmax	–	15s, 124rd, 48wr	8 (8)	120 (16)
	dma	–	54s, 707rd, 208wr	247 (16)	146 (9)
	tv80	see footnote[1]	1s, 15rd, 0wr	8 (8)	720 (256)
	jpeg_enc	see footnote[2]	1s, 64rd, 64wr	4200 (4096)	–
	systemcaes	–	13s, 204rd, 56wr	16 (12)	–
	des	–	38s, 1778rd, 1058wr	–	512 (64)
	systemcdes	–	8s, 896rd, 384wr	12 (5)	512 (64)

Note: The notation used in this table is explained in Section 4.6.1.
[1]An 8bit adder/subtracter is built from 1bit adders with a carry chain. An 8-bit shift/rotate left and right are individually specified at the bit level.
[2]One third of this circuit (by node count) is a 16-bit divider specified at the bit-level. However, since Malibu does not have a divide instruction, this operation cannot be simplified.

3. **Case LUT** – Creating a look-up table with a case statement or logic. Two numbers are specified: "`X (Y)`". `X` is the total number of items in all case statements used to declare memories, and `Y` is the size of the largest memory.

4. **Serial Case** – Sequential evaluation of parallel cases. Again, two numbers are specified: "`X (Y)`". `X` is the total number of items in all serial case statements, and `Y` is the size of the largest serial case chain.

Specifying Coarse Operations at the Bit Level

High-level operations like add (+), subtract (−), multiply ($\star$), and shift (<< and >>) should be used to create adders, subtracters, etc. When designing for performance or area on an FPGA or ASIC it is sometimes beneficial to construct a wide adder out of full adders and manually create carry-chains, for example. However, specifying an adder in this way is wasteful on a coarse architecture and leads to longer compile times and reduced performance. Automatically detecting such manually-specified coarse structures is a challenging problem because of the vast number of ways an adder, for example, could be built from fine-grained components. It is further complicated because the tools do not know what the original intention of the designer was; something that may look like an adder, except for a wire or two, may not actually be an adder. Therefore, properly specifying coarse constructs for coarse-grained architectures should be a circuit design-style change. In this thesis, we have not modified any of the benchmark circuits.

Using a Bus to Aggregate Individual Bits

Verilator generates shift-right (`LSRI`) and mask (`ANDI` with a constant) operations to read individual bits of a multi-bit signal, and generates a sequence of concatenation (`CONCAT`) operations to combine individual bit-writes into a group to form a multi-bit signal. Figure 4.5 illustrates these with an example from the *dma* benchmark. Neither of these structures are ideal for a coarse architecture, as will be discussed next.

Each read from the `dma_irq` signal in Figure 4.5 generates shift and mask instructions, even for reads from the same bit. In the *dma* benchmark, there are three reads for each bit of this signal, for a total of 24 shift and 24 mask instructions (only two are shown in Figure 4.5). The tools could perform common subexpression elimination to reduce the amount of logic by scanning the descendants of a signal for identical `LSRI` and `ANDI` instructions. Even better would be to modify Verilator to not generate such nodes in the first place. Neither of

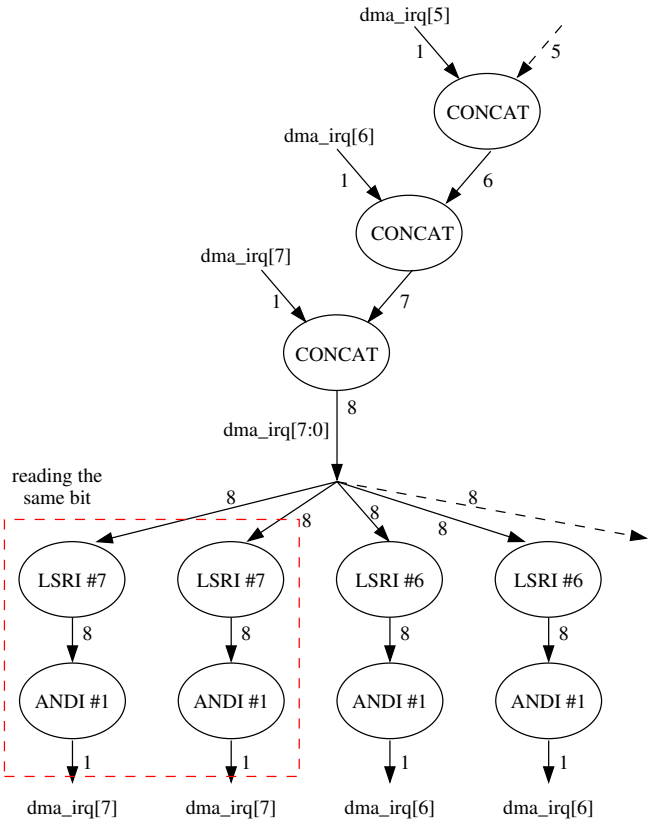


Figure 4.5: Using a bus to aggregate individual bits. Example of a bad structure from the *dma* benchmark. An 8-bit signal is used to store independent bits causing a concatenation sequence for writing, and individual `LSRI` and `ANDI` operations for reading.

these options have been explored because the focus of the thesis is on back-end synthesis. An alternate solution would be hardware support for a bit extraction operation in the CGO registers. This would eliminate the shift and mask operations completely.

To aggregate the multi-bit signal, a single concatenation chain is used, which can unnecessarily increase the critical path of the circuit by the number of bits in the signal (eight in the case for `dma_irq`). It is possible to improve this case in the tools by turning the chain into a tree using the same $n-1$ `CONCAT` operations, but with a depth of $\lceil log_2(n) \rceil$ instead of n. This optimization has been done because it is similar to the multiple-parallel-case logic presented later in this section.

An additional problem with this read and write behaviour that cannot easily be solved by the tools is the potential for a large fanout; `dma_irq` fans out to 24 operations in the `dma` benchmark. This causes delay and congestion when scheduling since not all 24 of these instructions can be scheduled in the timeslot immediately following the final `CONCAT`, so additional delay is inevitable. A smarter algorithm could, for example, realise that the lower bits of `dma_irq` are computed early, and thus do not need to go through the entire `CONCAT` before being separated again. However, the large fanout still creates congestion, putting pressure on resources that that could be used by other paths. A smart CAD algorithm or a circuit design-style change is what is needed to avoid this situation.

The ideal solution from a coarse-grained synthesis point of view for aggregating individual bits is to not declare individual bits as a multi-bit signal at all. Instead, they should be declared as individual signals (e.g., `wire dma_irq0, dma_irq1, ...`) with a final concatenation if a multi-bit aggregation is ever needed. Doing this automatically is challenging and would require careful graph analysis algorithms which are beyond the scope of this thesis. Doing this at design time places more work on the circuit designer but should produce a better result than relying on the tools to discover the parallelism. Either approach eliminates the concatenation chains and the shift-and-mask logic around the actual variable. Both also eliminate the fanout-congestion and allow the tools to potentially put each bit in a different CLB, increasing parallelism.

Creating a Look-Up Table with a Case Statement or Logic

For a coarse-grained architecture, a look-up table should be declared as an array (e.g., `wire [31:0] lut_data[0:63]` for a 64x32bit memory), and should be initialized with a Verilog `initial` block. Figure 4.6a shows a snippet of Verilog code from the *dma* benchmark which creates a look-up table using a case statement. Figure 4.6b shows code with the same functionality, but re-written to use a memory. Figures 4.6c and 4.6d are the DFGs generated

by Verilator from the two pieces of code. Using a case statement to specify a LUT creates a chain of conditional evaluations, one for each table entry, which must be evaluated in order according to Verilog case-statement semantics. This can increase the critical path (and thus the schedule length) by the number of entries in the lookup table.

A tool could automatically convert look-up tables specified in case statements to memories by scanning the DFG for a specific pattern (`CONST` nodes feeding chained `MUX` nodes in this case). The conditional input to to each `MUX` could also be checked, but it is complicated if the programmer has used don't-care conditions (e.g., `4'b0x1x`). A general approach for automatic conversion to cover all cases would be challenging to create, so such conversion is not explored in this thesis. However, because some benchmark circuits contain large look-up tables (e.g., the *aes_core* benchmark with a 256-entry look-up table, which adds 256 to the schedule length), we have added an optimization to automatically convert the sequential evaluation chain into a tree. This reduces the maximum depth from n to $\lceil log_2(n) \rceil$, where n is the number of case entries.

The ideal solution for a coarse-grained architecture is for the circuit designer to specify a look-up table as shown in Figure 4.6b. This removes any guesswork from the tools, removes the need for a decision-chain or a decision-tree, and allows for direct synthesis into a coarse-grained-compatible structure (a memory).

Sequential Evaluation of Parallel Cases

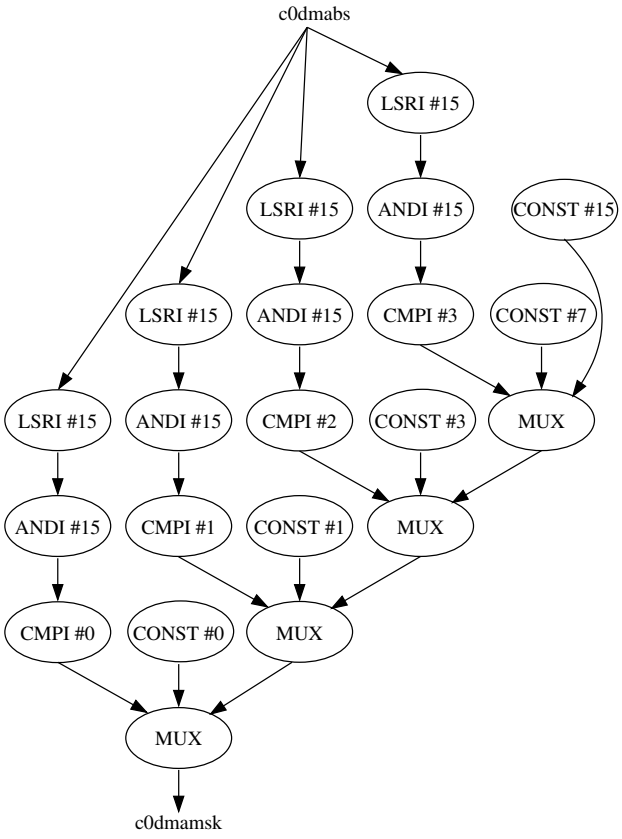
When a case statement is declared, Verilator synthesizes a chain of conditional evaluations which, according to Verilog semantics, must be evaluated in order. This sequential chain can lengthen the critical path. Unlike the look-up table case in the previous subsection, the entire case statement cannot be optimized into data storage. Some synthesis tools (e.g., Synopsis) support ways of annotating the source Verilog to aid the synthesis tools. Verilator does not support this feature and lacks support for a full-parallel-case, so it always evaluates case state-

```
1  always @(c0dmabs)
2  begin
3    case(c0dmabs[19:16])
4     'b0000: c0dmamsk = 'b0000;
5     'b0001: c0dmamsk = 'b0001;
6     'b0010: c0dmamsk = 'b0011;
7     'b0011: c0dmamsk = 'b0111;
8     default: c0dmamsk = 'b1111;
9    endcase
10 end
```

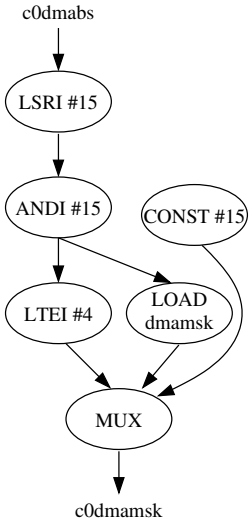
(a)

```
1  wire [3:0] dmamsk [3:0];
2  wire [3:0] c0dmabs19;
3  initial begin
4    dmamsk[0] <= 'b0000;
5    dmamsk[1] <= 'b0001;
6    dmamsk[2] <= 'b0011;
7    dmamsk[3] <= 'b0111;
8  end
9
10 assign c0dmabs19 = c0dmabs[19:16];
11 always @(c0dmabs)
12   c0dmamsk = (c0dmabs19 <= 'd4)
13       ? dmamsk[c0dmabs19] : 'b1111 ;
```

(b)



(c)



(d)

Figure 4.6: Example of a look-up table creation. (a) Verilog code for a lookup table from the *dma* benchmark (lookup table created with a case statement). (b) The same lookup table rewritten to use a memory. (c) Synthesized DFG for the code in (a). (d) Synthesized DFG for the code in (b). The memory-based lookup table synthesizes to fewer nodes, requires less instruction memory, and executes faster.

ments sequentially.

Instead of a sequential chain of comparisons, a binary decision tree can be used which only requires $\lceil log_2(n) \rceil$ depth. This decision tree is essentially an n-input multiplexer. The code to generate such a tree has been implemented and is used in all the results presented in this thesis. The conditional inputs to each `MUX` instruction are used to form the select inputs at each level of the decision tree, ensuring that the highest priority signals (the ones declared first) takes precedence over the others in the case of multiple case matches.

The *tv80* benchmark has a 256-case instruction decoder, which lengthens the critical path (and thus the schedule length) by 256, causing it to be over 400 timeslots. Using this decision tree transformation reduces the critical path contribution of the case statement to only $log_2(256) = 8$ timeslots. This shifts the critical path to a different part of the circuit, reducing the overall schedule length to 204.

4.7 Conclusions

This chapter has presented the common front-end synthesis approach for the M-CAD (Chapter 5) and M-HOT (Chapter 6) flows. While this is not a commercial-quality synthesis solution, it provides the features required for Malibu CAD, namely, it creates a coarse-grained DFG, it applies coarse-grained optimizations, and it allows the fine-grained parts of the circuit to be separated, synthesized, and merged.

An analysis of the benchmarks circuits used in this thesis lead to three classifications: *CG-only*, *Good*, and *Impaired*. In the *Impaired* circuits, four bad circuit design structures were identified which significantly contribute to the poor mapping onto Malibu (long schedule lengths, and larger area). The node-inflation shown in this chapter is a significant limitation in Malibu which contributes to the poor results. However, as demonstrated by the *CG-only* and *Good* benchmarks, it is possible to design circuits for the coarse-grained Malibu architecture without using these bad structures. The results of front-end synthesis for the *CG-only* and *Good*

benchmarks show the Malibu/Verilator solution is working well and gives node counts which are smaller than the QuartusII ALM count for the *CG-only* circuits.

There are many optimizations and improvements to coarse-grained front-end synthesis which can be investigated as future work. The automation of the conversion of the bad structures into Malibu-compatible ones will require careful and intelligent graph analysis techniques.

Next, in Chapters 5 and 6, we present back-end synthesis, and explore mapping the DFG output of front-end synthesis onto the Malibu architecture.

Chapter 5

M-CAD: An FPGA CAD Based Tool Flow

5.1 Overview

As a starting point for coarse-grained synthesis, the M-CAD flow is based on a well-studied fine-grained flow for mapping circuits to FPGAs—the VPR academic FPGA CAD tool flow—which is augmented to add support for word-oriented resources.

The tool flow begins with front-end synthesis as described in Chapter 4. Once the DFG has been constructed, the objective of M-CAD is to assign each coarse-grained node in the DFG to a timeslot in a CLB where it will be executed, assign each fine-grained node to a LUT within a CLB, and to route all edges (communication) over both the coarse-grained and fine-grained routing resources to connect the nodes. As with a traditional FPGA CAD flow, M-CAD is divided into several steps for clustering, placement, routing, and a step called scheduling to order the time-multiplexed operations over time. All steps are timing-driven, which means minimizing the schedule length for the time-multiplexed coarse-grained operations.

Compared with traditional FPGA synthesis, significant time can be saved by using and maintaining the coarse-grained elements in the circuit rather than decomposing all the operations down to 2-input gates. For example, a large circuit may have $\approx$ 1 million logic gates

which must be placed and routed. If these were all expressed as higher-level 32-bit operations (like add, subtract, etc.), a significant reduction in the number of operations can be achieved. For example, a single 32-bit add contains 32 full adders, where each full adder contains a 3-input XOR and three 2-input NAND gates. For a single 32-bit multiply, an even greater number of small gates are required. Not all operations in every circuit can be expressed as a high-level operation, but these examples show the potential. Avoiding synthesis from high-level operations down to gates saves time in two ways: i) reduced time to generate the DFG from the Verilog, and ii) reduced time to process (optimize, place, route, etc.) the circuit due to a reduced problem size.

The tool flow requires two inputs: the Verilog description of the circuit, and an architecture file. The parameter values from Table 3.5 are used to create the architecture file. These values act as constraints in the tool flow.

The remainder of this chapter is organized as follows. Sections 5.1–5.5 explain the M-CAD flow and show how it maps circuits onto the Malibu architecture. Section 5.6 presents experimental results and compares these to results from QuartusII and VPR with respect to density, compile time, and performance. Section 5.7 provides concluding remarks about M-CAD.

5.2 M-CAD Cluster

The first step after front-end synthesis is clustering. The clustering tool groups CG operations into clusters for each CLB to speed up placement. It also always groups some operations into the same cluster as follows:

- `CGO` nodes are placed in the same cluster as the source. This is because a fine-grained value should enter the fine-grained resources in the same CLB as the coarse-grained operation which produced the value. It would be inefficient to transfer this value over the

coarse-grained routing resources to another CLB, only to have it enter the fine-grained resources at that point.

- `CGI` nodes are replicated, with one copy placed in the same cluster as each sink. Similar to the reasoning above for `CGO` nodes, a coarse-grained instruction which uses a fine-grained value should have that fine-grained value routed directly to, and available in, the same CLB. The nodes are replicated because the signal could transfer back to the coarse-grained resources in several CLBs.

- All `LOAD` and `STORE` memory operations for the same user-memory must be in the same cluster.

The goal is to balance the number of operations in each cluster and to reduce the amount of communication by absorbing as many nets as possible into clusters. A partitioning algorithm can achieve exactly this, so M-CAD uses hMETIS [42], a well-known hypergraph (circuit) partitioning tool, to partition the graph using recursive bisection. To guide hMETIS in balancing the clusters, all nodes in the DFG are assigned a weight of 1, except the five placeholder nodes (`CONST`, `CGO`, `CGI`, `INPUT`, and `OUTPUT`) which are assigned a weight of 0. These latter nodes can be placed in any CLB for free, but they are subject to other constraints:

- `CONST` is replicated after placement for each CLB that needs it, so it does not matter where it is clustered.

- `CGO` and `CGI` are attached to their sources or sinks with high edge weights to ensure they are not separated. Similarly, related `LOAD` and `STORE` nodes are connected with high edge weights to ensure all operations involving for the same user memory reside in the same CLB. However, `LOAD` and `STORE` nodes (not edges) are assigned a (node) weight of 1 because they require an ALU cycle. The choice to use high edge weights to keep nodes together is an implementation decision; it makes it easier to export the graph,

run hMETIS, and then import the cluster information back into the DFG if the hMETIS input and output exactly match what is in the DFG.

- `INPUT` and `OUTPUT` are limited to one of each per cluster.

The tool can cluster code to varying degrees to target any number of CLBs, allowing a tradeoff between area (number of CLBs) and performance (frequency). By default, the clustering tool creates twice as many clusters as CLBs to give the placement tool some freedom. This method of clustering does not capture the time-multiplexed nature of the problem, and indeed, it would be challenging to do so without embedding a placement and scheduling tool inside the clustering algorithm (which we tried, and it was slow, as expected).

Our own testing has shown that, for Malibu, twice the number of clusters gives good quality results. Additional clusters increases the runtime significantly (the placer uses simulated annealing) and only decreases the overall schedule length by a cycle or two as the number of clusters approaches the situation where no clustering is being done.

Continuing the example from Figure 4.4, Figure 5.1 shows a possible clustering with only four clusters for two CLBs. This clustering violates the I/O restriction of one external input (the `INPUT` nodes) per CLB, so we shall assume that the architecture can support two inputs per CLB for corner CLBs. In this example there are only six operations which would be considered when balancing the number of operations per cluster (two `EQ`s, `ADD`, `XOR`, `LUT`, and `MUX`). Note that the `CGO` nodes remain in the same cluster as their sources, and the `CGI` node is in the same cluster as the sink.

5.3 M-CAD Place

The placement tool takes the clustering information in the DFG and assigns the clusters to CLBs. The goal is to keep the critical path small. The tool uses VPR's timing-driven simulated annealing placement algorithm [53] with two changes to the cost function. First, a different

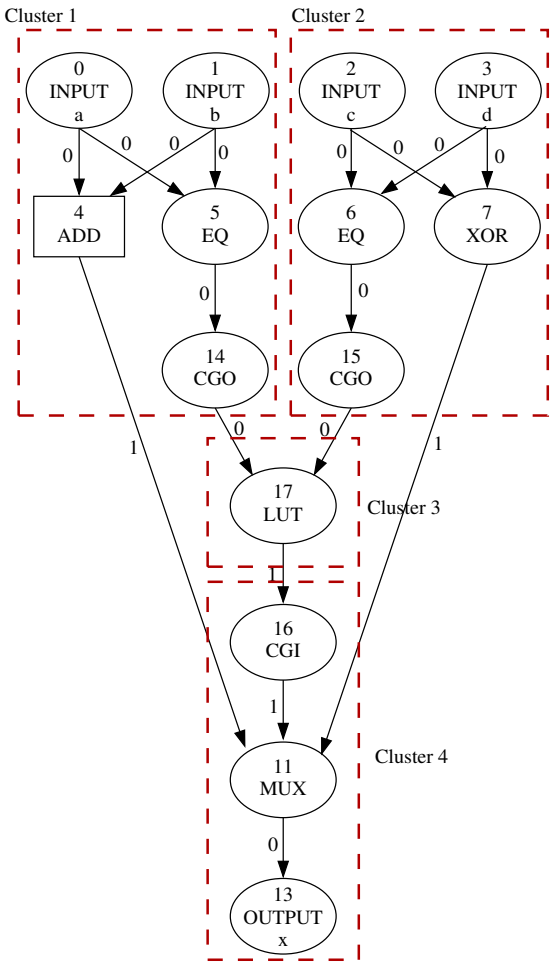


Figure 5.1: Example: M-CAD clustering for two CLBs. This is not an optimal clustering, and it is not even a valid clustering if the inputs per CLB are restricted to one. It was chosen to keep the example simple.

definition of "delay" is used in the cost function to handle both the fine-grained and coarse-grained operations being placed. Second, a parameter *penalty* discourages illegal placements.

To simplify placer delay estimates, all delays are expressed as integers. In the coarse-grained pipelined routing network in Malibu, the delay between two nodes is equal to the Manhattan distance (rectilinear distance), not a propagation delay along a wire as in conventional CAD. Time-multiplexing introduces an additional complication not found in regular FPGAs: two nodes within the same CG may be scheduled in timeslots far apart, causing additional de-

lay not modeled by the number of hops. Unfortunately this additional delay is not known until scheduling is complete, so at this stage we assume it is zero. Because of the feed-forward flow of information in the FPGA CAD approach, there is no opportunity to pass scheduling information back into an earlier stage of the tool flow. This effectively means we assume nodes are scheduled in consecutive timeslots in the same CLB. This is not a bad assumption because the scheduler tries to do exactly that. However, if there are two or more instructions that are ready to be scheduled at the same time, the ones which are not scheduled will incur this additional delay (again, only known after scheduling).

We used a simple one-net circuit in VPR with the iFAR architecture file[1] named n10k04l04.fc15.area1delay1.cmos65nmos to determine that a fine-grained signal travels just over 10 CLBs in one system clock cycle of 1.0 ns. Therefore, any FG signal traversing 10 CLBs or fewer has a delay of 1, 11-20 CLBs has a delay of 2, etc. The placer can thus estimate the fine-grained delays quite easily. The scheduling tool (Section 5.5) uses the actual delays from the VPR routing solution to make scheduling decisions (to ensure all fine-grained signals arrive before the consumer nodes are scheduled), so an estimate of the fine-grained delays during placement is sufficient.

The delay in clock cycles between two nodes i and j is:

$$delay(i,j) = \begin{cases} 1 & i,j \text{ are coarse-grained and placed in same CG} \\ mh(i,j) & i,j \text{ are coarse-grained and not in the same CG} \\ \left\lceil \frac{mh(i,j)}{10} \right\rceil & \text{otherwise (fine-grained)} \end{cases} \quad (5.1)$$

Where $mh(i,j)$ is the Manhattan distance (or rectilinear distance) between the CLBs for nodes i and j. For two nodes in the same CG, the ALU must execute an instruction in one timeslot to produce the value consumed by the next node in a subsequent timeslot, so the

[1] In this architecture, the length four wires are changed to length one to over-compensate for adding the coarse-grained resources to each CLB, this is explained in Section 5.4.

delay is at least one timeslot. As previously discussed, any additional delay due to nodes being assigned to timeslots far apart is unknown until scheduling is complete, so it is assumed to be zero. For adjacent CGs, this value is written to one of the NSEW memory locations so it is available in the adjacent CG in the next timeslot. For distant communication, the minimum delay (in cycles) is the Manhattan distance between the CLBs, but it may be longer depending on the availability of routing resources due to congestion. Again, this additional delay is unknown and cannot be known until after scheduling, so it is assumed to be zero.

The $delay(i, j)$ in Equation 5.1 is used with a slack and criticality computation to calculate the *timing_cost* of the circuit, which is part of the placement cost function. These are shown in Equations 5.2–5.5. The *slack*, *criticality*, and *timing_cost* computations are the same those in VPR's timing-driven placement algorithm [53]:

$$slack(i,j) = T_{required}(j) - T_{arrival}(i) - delay(i,j) \tag{5.2}$$

$$criticality(i,j) = 1 - \frac{slack(i,j)}{D_{max}} \tag{5.3}$$

$$timing\_cost(i,j) = delay(i,j) \cdot criticality(i,j)^{criticality\_exponent} \tag{5.4}$$

$$timing\_cost = \sum_{\forall i,j \in circuit} timing\_cost(i,j) \tag{5.5}$$

Where $T_{required}(j)$ is the latest possible arrival time for the signals at node j, $T_{arrival}(i)$ is the arrival time of the signals at node i, and D_{max} is the critical path delay. $T_{required}(j)$, $T_{arrival}(i)$, and D_{max} are all calculated using $delay(i, j)$ [53], so they are in the same integer delay units as previous described.

In addition to a timing cost, the VPR placement cost function uses a wiring cost. In Malibu, the wiring cost is the same as the delay because every length of wire means one additional cycle delay. The final placement cost function is similar to VPR's (Equation 2.3), but with the wiring

cost and λ removed:

$$\Delta C = \frac{\Delta timing\_cost}{previous\_timing\_cost} + penalty \tag{5.6}$$

The placer allows illegal placements to be considered. The *penalty* parameter adds a fixed cost of 1,000 each time one of the following violations occurs:

- the memory size is exceeded,
- unavailable CG or FG resources are used,
- too many CGI/CGO registers are used,
- too many CLBs are used, or
- too few CLBs are used (only if the tools are forcing a specific number of CLBs).

The fixed cost for too few CLBs is used to generate the performance versus area graphs as in Figure 5.9 and in Appendix D.

Returning to the clustering example in Figure 5.1, if the target architecture has two CLBs, then the placer may put clusters 1 and 3 in CLB0, and clusters 2 and 4 in CLB1 to balance the operations per CLB and also minimize the critical path. Other configurations are also possible. In Figure 5.1, the numbers on each edge are the delays as computed by the placer.

At the end of placement, if required, multiple small, user-instantiated memories are packed into the single R memory such that they do not overlap. The `LOAD` and `STORE` instructions are updated to reflect the required address offsets. The problem of splitting a large user memory across multiple CG is left for future work.

5.4 M-CAD Route

After placement, the coarse-grained and fine-grained signals are each routed using different methods. Fine-grained routing is done with VPR's PathFinder router as described below.

Coarse-grained routing is describe later in this section. M-CAD creates three files necessary to invoke VPR:

- netlist file – A netlist is created for the entire circuit. VPR requires a complete netlist with no dangling logic, so it is not possible to write only the fine-grained parts of the circuit. Each coarse-grained signal is also written to the netlist, but written as a single wire marked DNR (do-not-route). VPR was modified to ignore signals marked DNR. VPR uses these coarse-grained signals for timing and criticality information, but it otherwise ignores them.

- architecture file – An architecture file is created based on the 65nm iFAR [82] architecture (n10k04l04.fc15.area1delay1.cmos65nm). The length-four wires in the architecture were changed to length-one wires without changing the delay characteristics. This was done to over-compensate for the area of a CLB (with both the FG and CG components) being roughly four times larger (2x longer tile) than a Stratix III LAB. Also, the number of LUTs per CLB was changed from 10 to 16 to match the architecture parameter value from Table 3.4. The placement of the CLB pins was also changed to only the top or right of the CLB to mimic overhead routing (routing on a different metal layer in the physical device).

- placement file – A placement file is created with the information from the placer.

Figure 5.2 shows the routing output of VPR for the example circuit in Figure 5.1 with clusters 1 and 3 in CLB0 and clusters 2 and 4 in CLB1. The output shows the placement of the coarse-grained inputs and outputs, but they are not routed (they were marked DNR). This example is drawn with a channel-width of four (with only two wiring tracks used). Normally, there would be 120 wires between these blocks. The two routes shown represent node 15 in CLB1 from Figure 5.1 communicating with node 17 in CLB0, and node 17 in CLB0 communicating to node 16 back in CLB1.

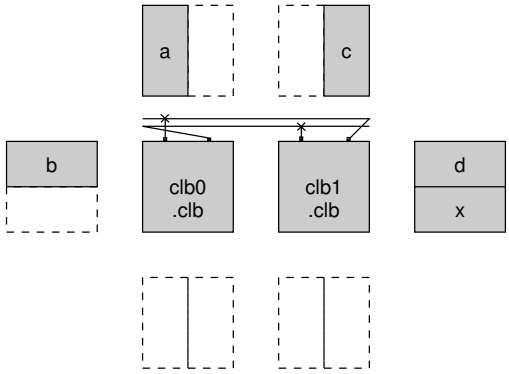


Figure 5.2: Example: fine-grained routing output from VPR.

The scheduler (in Section 5.5) reads the delay information from VPR and records the delay of each fine-grained link in the DFG. In this example, the delay of all the signals is 1.

The coarse-grained routing problem is different from traditional FPGA CAD because the Malibu coarse-grained routing network is time-multiplexed, so temporal as well as spatial decisions must be made. The spatial routing is done using a simple horizontal-then-vertical routing strategy where routes are created in the x-direction first (horizontal), and then in the y-direction (vertical). The router follows existing routes from the same source as far as possible before branching the route towards the new destination CG.

The temporal routing decisions are made during scheduling. When the endpoint of a route is to be scheduled, the scheduler follows each hop of the route, checking that the necessary CG resources are available. If a conflict arises, the route is held in place for as many timeslots as necessary until the resources are available at the next hop. Since the entire coarse-grained routing network is pipelined, holding a route in place just means marking the register where the routed value is currently stored as unavailable for additional clock cycles. For the F_{max} results in Section 5.6.1, it was never necessary to hold a value to avoid a routing conflict for any circuit. In practise, we have never seen a value held more than two cycles, except with targeted micro-benchmarks to exercise that specific condition. One such test is shown in Figure 5.3; the routes

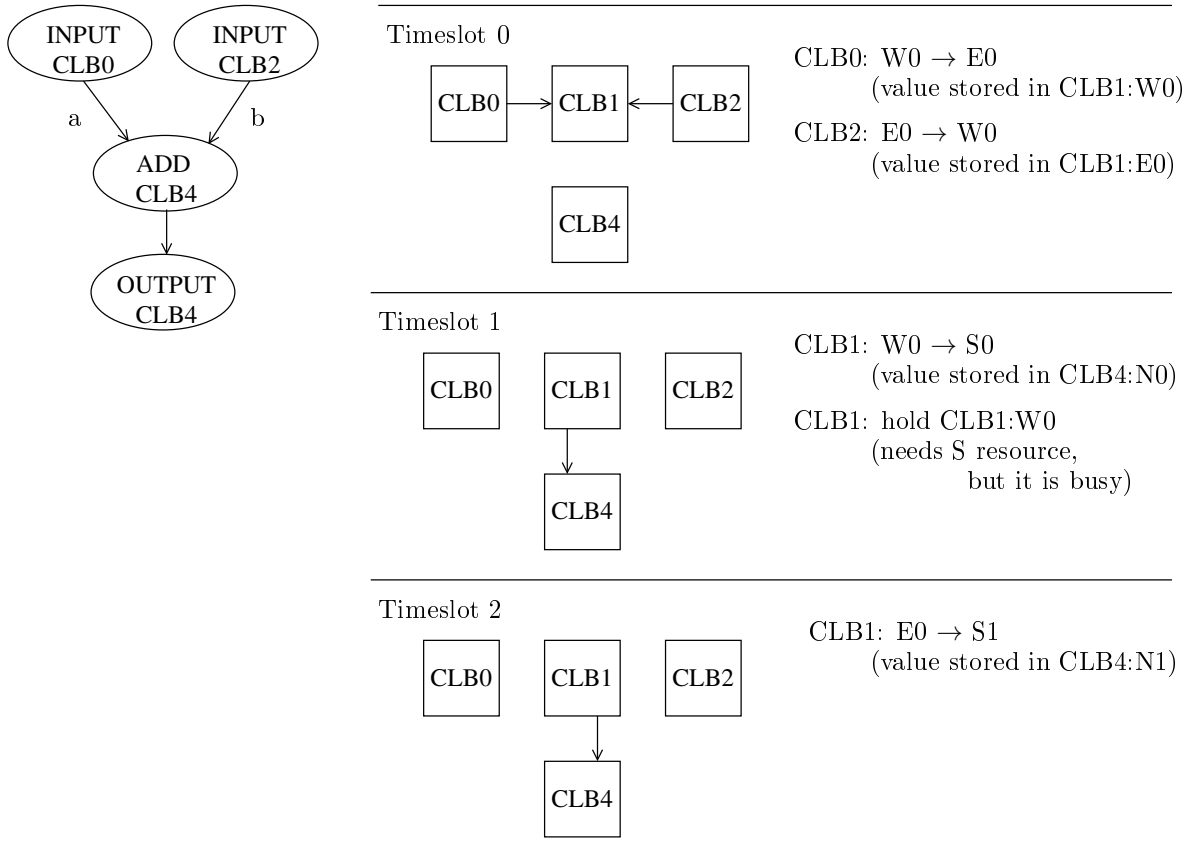


Figure 5.3: Microbenchmark for testing coarse-grained route collisions.

a and *b* from CLB0 and CLB2, respectively, collide in CLB1 and both need the S resource. In this case, route *a* proceeds first because of the order the scheduler visited the routes, and route *b* is held in CLB1:W at address 0 for an extra timeslot. Each node has a criticality so that the most critical nodes (and routes) are scheduled first. In the case of ties, like in this example, the outcome is deterministic and the smallest node IDs are scheduled first; the processing queue is populated in node-order in the scheduler. When the S resource becomes available in timeslot 2, route *b* can proceed.

The Malibu architecture was created with a single 32-bit link between neighbouring processors. The 32-bit links were selected to match the ALU width. The very rare need for hold slots suggest that these time-multiplexed links provide ample routing resources. Future architecture

work can look at reducing the area-overhead of these links, for example, by using 16-bit links and communicating 32-bit data in a pipelined fashion over two cycles.

5.5 M-CAD Schedule

The M-CAD schedule tool orchestrates the overall execution of code and movement of data to reproduce the behaviour of the original circuit. It assigns each instruction to a timeslot in a CG, it assigns each coarse-grained route-hop to a timeslot (resolving all routing collisions along the way), and it ensures all values are produced/consumed at the appropriate times on the (not-time-multiplexed) fine-grained resources. At this point all fine-grained nodes have been placed in a LUT and routed across the fine-grained resources, so perfect delay information is available.

The scheduling algorithm is a variation of list scheduling. The main loop of the tool is shown in Figure 5.4. It begins at $timeslot = 0$ and assigns as many operations as it can across all CGs in that timeslot. It then iterates over the sinks of the scheduled operations and uses the routing delay information to compute the minimum timeslot in which those sinks may be scheduled. It then moves on to the second timeslot, and so on. This timeslot-oriented approach ensures the scheduler is fast and is always making forward progress. NOP instructions are inserted in all timeslots that do not contain a circuit node after scheduling.

When fine-grained sinks are encountered, the scheduler propagates delay information down the fanout-cone of the fine-grained sink until it reaches coarse-grained nodes. This is essentially the same process as for coarse-grained sinks, except the scheduler does not need to find a timeslot for fine-grained nodes (the fine-grained resources are not time-multiplexed, and have already been assigned to a LUT in a CLB). The scheduler uses the fine-grained delay information (from the VPR routing solution) to compute the earliest timeslot in which these coarse-grained nodes (which depend on a fine-grained value) may be scheduled.

At each timeslot, nodes are considered in order of criticality as computed during placement.

```
ready_queue = all nodes flagged "end of cycle" or nodes with no parent
for( timeslot = 0 ; ; ) {
    if( is_empty( ready_queue ) ) {
        if( is_empty( next_queue ) ) {
            return;  /* Scheduling complete */
        } else {
            /* Swap queues, increase to next timeslot */
            ready_queue = next_queue
            timeslot++
            continue
        }
    }

    /* Find a schedulable node */
    node = dequeue( ready_queue )
    if( ! is_schedulable( node ) {
        enqueue( next_queue, node )
        continue /* Restart loop */
    }

    /* Create routes, record scheduled timeslot */
    schedule_routes( node )
    node.timeslot = timeslot

    /* Increment sched. count in all children, enqueue any that are now schedulable */
    foreach( node.children as child ) {
        if( child.width > Wf) {
            /* Coarse-grained child */
            child.parents_scheduled ++
            if( child.parents_scheduled == child.parents_len ) {
                enqueue( next_queue, child )
            }
        } else {
            /* Fine-grained child */
            process_fg_fanout_cone( child )
        }
    }
}
```

Figure 5.4: Main loop of the M-CAD scheduler.

This simple ordering reduces the final SL and thus increases the F_{max} by an average of 10% (recall $F_{max} = \frac{1\,\text{GHz}}{SL}$) across all the benchmarks used in this thesis.

The is_schedulable$(node)$ function checks whether $node$ is schedulable in the current $timeslot$. It may be scheduled in $timeslot$ if:

- The $timeslot$ is empty in the CG's ALU.
- All source signals have arrived in time.
- All internal CG resources required by the operation are available.

	Timeslot	CLB0	CLB1
One user cycle	0	`EQ N0,W0 → CGO0`	`EQ N0,E0 → CGO0`
	1	`NOP`	`XOR N0,E0 → R0`
	2	`ADD N0,W0 → E0`	`MUX W0,R0,CGI0 → E0`

Figure 5.5: Example: the final M-CAD code schedule. This is the output for the input given in Figure 4.2. Note that `E0` in CLB0 is the same as `W0` in CLB1.

- All routing resources required by the output of the operation (fine-grained and coarse-grained) are available for the first-hop of the route.

The schedule_routes(*node*) function assigns route-hops to timeslots. When a routing conflict arises, the value is held for as many timeslots as necessary for a free timeslot to be found in the next hop (as described in Section 5.4). When each route arrives at the destination CLB(s), the destination node(s) in the DFG are marked with the arrival timeslot(s); those nodes cannot be scheduled before their respective arrival timeslot.

The process_fg_fanout_cone(*node*) function increments the `parents_scheduled` attribute for all coarse-grained nodes on leaves of the fine-grained fanout cone rooted at node *node*. For those coarse-grained leaf nodes, it also enqueues the ones which have all of its parents scheduled into the next_queue. Finally, it updates the earliest timeslot that those nodes may be scheduled using the fine-grained delay information from routing.

At the end of scheduling, accesses to the NSEW and R memories are assigned specific offsets using a greedy approach. At this point, the CG operations and FG LUTs for each CLB are packed into a single output bitstream.

Figure 5.5 shows the final scheduling for the example started in Figure 4.2. Notice how the register behaviour of the `ADD` instruction is produced; the `MUX` instruction reads the `W0` input in the same timeslot as the `ADD` writes it, so it ends up reading the value written in the previous iteration of the schedule shown in Figure 5.5. The new value in `W0` will be used in the next user cycle.

During scheduling the ADD instruction would have been considered for $timeslot = 1$ in CLB0. However, since it is *registered*, the scheduler requires that all the child nodes (the MUX in this example) be scheduled so that a newly written value does not get used in the current cycle. This is how the proper register behaviour is produced, and this is what forces the ADD into timeslot 2. In CLB1, it is possible to swap the EQ and XOR, but the scheduler considers nodes in order of criticality, and the EQ instruction is part of a longer combinational chain (including a LUT), so it will always be considered first.

Not shown in Figure 5.5 is the fine-grained information that is part of the bitstream. One of the LUTs in CLB0 will have the truth-table from Figure 4.4, and the switch blocks and I/O blocks will be configured to connect CGO0 in CLB0 and CGO0 and CLB1 to the input of that LUT, and configured to connect the output of the LUT to CGI0 in CLB1 where it is used by the MUX instruction.

5.6 Experimental Results

In this section, the M-CAD flow is evaluated on the Malibu architecture. For baseline comparison, all the benchmarks were synthesized with QuartusII 10.0 for a StratixIII (EP3SL340F1760C2) FPGA as described in Section 3.3, and with VPR 5.0 using 65nm iFAR [82] architecture parameters (n10k04l04.fc15.area1delay1.cmos65nm—the same architecture M-CAD uses, but without the length-four wire change). The ten 4-LUT iFAR architecture was selected for area efficiency, and modified to place the CLB pins only on the top or right of the CLB to mimic overhead routing. A channel width of 100 was used, as recommended by the architecture file. The VPR "-fast" option was used to generate all the VPR results presented in this thesis.

All of the results in this section use the architecture configuration in Table 3.5. Since the tools can trade circuit performance for density, it is possible to use a single fixed architecture to target results for maximum performance, for maximum density, or anywhere in between.

We limit the discussion to these two extremes and present the maximum frequency results (and the density at maximum performance), and the maximum density results (and the frequency at maximum density).

For these two data points, maximum performance and maximum density, the fine-grained signal threshold (W_f) is tested at $W_f = 0, 1$, and 4. This is done to investigate the performance and density tradeoffs from adding the fine-grained resources. At $W_f = 0$, all of the fine-grained resources are excluded from the architecture (and the area of each CLB is adjusted appropriately). At $W_f = 1$, only one-bit signals are implemented on the fine-grained resources, and at $W_f = 4$ all four-bit signals (or smaller) are implemented on the fine-grained resources.

The results are separated into the *CG-only*, *Good*, and *Impaired* circuit categories as defined in Section 3.3. The analysis in this section focuses on the *CG-only* and *Good* circuits because they represent the types of circuits which would primarily be used on the coarse-grained Malibu architecture. For the *Impaired* benchmarks, it is only important for the tools to successfully synthesize the circuit; these benchmarks are included to demonstrate that this is being done.

We begin in Section 5.6.1 by examining the quality of synthesis measured by the maximum frequency (F_{max}), and compare that to traditional FPGA CAD tools. This is extended in Section 5.6.2 by investigating an upper bound on the frequency. Next, in Section 5.6.3, we evaluate the maximum density of each benchmark, and compare the frequency achieved at that density with the F_{max}. The M-CAD compile time is investigated in Section 5.6.4 and compared with other tools, along with tests using very large synthetic circuits up to ≈ 1.6 million gates in Section 5.6.5. Then, Section 5.6.6 looks at a breakdown of the longest paths in the circuits to get ideas for future research directions. Finally, in Section 5.6.7 we look at the M-CAD synthesis ability to trade area for performance.

5.6.1 Frequency (F_{max})

Table 5.1 shows the frequency (F_{max}) across all circuits for QuartusII, VPR, and M-CAD. For M-CAD there is data at both the maximum frequency and the maximum density. To reduce noise, the frequency is the fastest result of ten synthesis trials. For QuartusII, one synthesis trial is a complete compilation from Verilog to bitstream, for VPR it is a compilation from the QuartusII-generated BLIF to the routing output, and for M-CAD it is the back-end synthesis (clustering, placement, routing, and scheduling) since the front-end synthesis always gives the same result for the same circuit. Incidentally, the M-CAD data for the maximum performance at $W_f = 1$ is the same as the schedule length data from Tables 3.4 and 3.5 converted to frequency by assuming a 1 GHz system clock, i.e., $F_{max} = \frac{1000}{SL}$.

For the *CG-only* benchmarks, there are no signals small enough to use the fine-grained resources when $W_f \geq 1$, so the results for $W_f \geq 1$ results are the same as $W_f = 0$. At maximum performance, the geometric mean of the F_{max} results in Table 5.1 show that the *CG-only* circuits on Malibu are higher than the QuartusII/StratixIII implementation by almost 50%. This is an excellent result which shows performance gains are possible for coarse-grained circuits, which is what the Malibu architecture was designed for. At maximum density, the frequency is about 80% that of the FPGA. This is also an excellent result because, as will be shown later, the density (at this point) exceeds that of an FPGA by almost 5x.

For the *Good* benchmarks, the fine-grained resource requirements in the benchmark circuits cause the performance to drop compared to the *CG-only* benchmarks; these fine-grained signals are being computed on the coarse-grained resources and distributed using the coarse-grained pipelined routing network, which is slow. The addition of the fine-grained resources in the architecture at $W_f = 4$ almost doubles the F_{max} compared to $W_f = 0$ at maximum performance, and triples it at maximum density. Since many circuits do have some fine-grained control logic, this indicates that including fine-grained resources in the Malibu architecture is beneficial.

Table 5.1: M-CAD frequency results. Entries with a "–" are for coarse-grained benchmarks and the same as the $W_f = 0$ value. The "vs.QII" columns are the frequency speedup (M-CAD MHz / QuartusII MHz).

				M-CAD Maximum Performance						**M-CAD Maximum Density**					
		QuartusII	**VPR**	$W_f = 0$		$W_f = 1$		$W_f = 4$		$W_f = 0$		$W_f = 1$		$W_f = 4$	
	Circuit	**MHz**	**MHz**	**MHz**	**vs.QII**	**MHz**	**vs.QII**	**MHz**	**vs.QII**	**MHz**	**vs.QII**	**MHz**	**vs.QII**	**MHz**	**vs.QII**
CG-only	fft16	119.2	98.4	45.5	0.381	–	–	–	–	12.0	0.101	–	–	–	–
	me	201.7	66.0	55.6	0.275	–	–	–	–	27.0	0.134	–	–	–	–
	chem	11.3	27.1	40.0	3.554	–	–	–	–	21.7	1.932	–	–	–	–
	fft8	159.7	117.2	71.4	0.447	–	–	–	–	19.6	0.123	–	–	–	–
	honda	17.1	63.8	45.5	2.652	–	–	–	–	37.0	2.161	–	–	–	–
	mcm	24.9	93.7	71.4	2.864	–	–	–	–	62.5	2.506	–	–	–	–
	wang	19.3	79.7	83.3	4.320	–	–	–	–	76.9	3.988	–	–	–	–
	pr	24.0	87.6	83.3	3.469	–	–	–	–	66.7	2.776	–	–	–	–
	Geo. Mean (CG-only)				**1.445x**		**1.445x**		**1.445x**		**0.814x**		**0.814x**		**0.814x**
Good	ac97_ctrl	294.9	278.3	14.3	0.048	40.0	0.136	40.0	0.136	9.7	0.033	35.7	0.121	40.0	0.130
	aes_core	181.6	177.8	27.8	0.153	25.0	0.138	25.6	0.141	5.3	0.029	20.8	0.115	17.9	0.098
	dir	86.3	59.6	29.4	0.341	27.0	0.313	27.0	0.313	15.6	0.181	19.2	0.223	27.0	0.313
	spi	118.9	139.9	25.6	0.216	32.3	0.271	38.5	0.324	16.1	0.136	27.0	0.227	38.5	0.324
	pci_master	232.2	248.4	22.7	0.098	29.4	0.127	76.9	0.331	11.8	0.051	28.6	0.123	76.9	0.331
	Geo. Mean (Good)				**0.140x**		**0.182x**		**0.230x**		**0.065x**		**0.154x**		**0.214x**
Geo. Mean (CG-only and Good)					**0.588x**		**0.652x**		**0.713x**		**0.308x**		**0.429x**		**0.487x**
Impaired	ethernet	162.4	102.1	6.7	0.041	11.9	0.073	20.8	0.128	4.9	0.030	9.8	0.060	13.3	0.082
	wb_conmax	135.1	72.3	13.2	0.097	19.6	0.145	27.8	0.206	5.6	0.041	17.2	0.128	27.8	0.206
	dma	127.8	131.0	5.8	0.046	10.2	0.080	10.6	0.083	4.3	0.033	9.3	0.073	9.5	0.075
	tv80	96.8	105.3	7.0	0.072	4.9	0.051	64.0	0.661	2.2	0.023	3.1	0.032	12.8	0.132
	jpeg_enc	218.4	162.8	5.2	0.024	5.3	0.024	5.2	0.024	4.3	0.020	3.9	0.018	2.1	0.010
	systemcaes	120.7	154.5	15.2	0.126	15.4	0.127	14.7	0.122	4.6	0.038	4.5	0.038	5.6	0.046
	des	299.8	175.5	3.6	0.012	5.2	0.017	5.3	0.018	3.6	0.012	4.5	0.015	5.3	0.018
	systemcdes	169.6	219.7	21.3	0.125	21.7	0.128	18.9	0.111	10.5	0.062	16.4	0.097	17.9	0.105
	Geo. Mean (Impaired)				**0.053x**		**0.064x**		**0.098x**		**0.029x**		**0.045x**		**0.059x**
Geo. Mean (All)					**0.235x**		**0.270x**		**0.334x**		**0.126x**		**0.182x**		**0.217x**

Further, achieving $1/5^{th}$ the performance of a commercial FPGA is a good result for a time-multiplexed architecture.

For the *Impaired* benchmarks, M-CAD does successfully synthesize each benchmark, but the performance is poor, as expected. Interestingly for the *jpeg_enc*, *systemcaes*, and *systemcdes* benchmarks at maximum performance, the frequency decreases as W_f is increased from 1 to 4. This indicates that the fine-grained routing resources are being strained and, in these cases, it is faster to compute some of these fine-grained results using the coarse-grained resources. Further investigation into the coarse-grained/fine-grained partitioning in front-end synthesis is required.

Comparing the maximum performance and maximum density M-CAD results in Table 5.1, the performance at maximum density is 48%, 34%, and 32% lower than the performance at maximum speed for the *CG-only* and *Good* benchmarks for $W_f = 0$, 1, and 4 respectively. This is expected because at maximum density the tools are more aggressively time-multiplexing operations onto the coarse-grained ALUs to use less area. Further, the highest reduction in performance is at $W_f = 0$ because no fine-grained resources are being used; since there are no CGI, CGO, or LUT constraints, the tools are only constrained by the instruction memory (and the DFG itself) and can therefore time-multiplex even more aggressively. This performance reduction is matched by increases in density at $W_f = 0$, demonstrated in Section 5.6.3. Remember that the difference between the maximum density and maximum performance results is just a compiler flag. Both sets of results are achievable without changing the underlying architecture.

Table 5.2 summarizes the performance of M-CAD compared to the VPR data in Table 5.1. The results are comparable to the QuartusII results, although VPR/iFAR generally produced higher frequency results than QuartusII/StratixIII.

Table 5.2: M-CAD F_{max} speedup compared to VPR.

	M-CAD vs. VPR					
	Maximum Performance			**Maximum Density**		
	$W_f = 0$	$W_f = 1$	$W_f = 4$	$W_f = 0$	$W_f = 1$	$W_f = 4$
Geo. Mean (CG-only)	0.812x	0.812x	0.812x	0.457x	0.457x	0.457x
Geo. Mean (Good)	0.146x	0.190x	0.240x	0.068x	0.161x	0.222x
Geo. Mean (CG-only and Good)	0.389x	0.436x	0.482x	0.202x	0.292x	0.335x
Geo. Mean (Impaired)	0.062x	0.076x	0.115x	0.034x	0.053x	0.065x
Geo. Mean (All)	0.192x	0.221x	0.277x	0.103x	0.150x	0.175x

5.6.2 Frequency Upper Bound

To get a sense of how well M-CAD could perform, we compute an upper bound on the maximum achievable frequency by using the graph-depth of the DFG after front-end synthesis. The purpose of this investigation is to isolate the potential F_{max} improvements that might be obtainable with a perfect architecture and perfect placement, routing, and scheduling. To compute the upper bound we assume that:

- each coarse-grained node in the DFG which requires the ALU takes one system clock cycle,
- all sequences of fine-grained nodes collectively take a single clock cycle, regardless of the number of nodes (so the maximum frequency will increase as more nodes are considered fine-grained),
- all communication and all other nodes take zero time (zero-cost routing), and
- there are no resource constraints (perfect placement, no I/O limits, infinite memory resources).

In Figure 4.3a, the maximum graph depth is 3, making the frequency upper bound $\frac{1\text{ GHz}}{3} =$ 333 MHz. For $W_f = 1$ and 4, the depth is also 3. However, if the graph contained a second

fine-grained node immediately after node 9, then the depth for $W_f = 0$ would be 4, but the depth for all other values of W_f would still be 3 (all sequences of fine-grained nodes are assumed to execute in a single clock cycle).

Table 5.3 compares the frequency upper bound with the F_{max} from Table 5.1 for each benchmark. For this comparison we include the *Impaired* circuits because there is the most opportunity for performance gains with these circuits. The possible performance increase in F_{max} ranges from about 6x (*jpeg_enc* and *systemcaes*) to nothing (*me*). The geomean of the ratio across all circuits for $W_f = 0, 1$, or 4 is just over 0.5, meaning that there is, on average, at most a 2x performance improvement that might be found in M-CAD (back-end synthesis) and the Malibu architecture combined. In addition, there may be other opportunities to improve the results through changes to the front-end synthesis that are not captured in this data.

The same analysis was repeated to produce a post-placement lower bound on the schedule length. This captures how much performance is lost due to routing and scheduling. Table 5.4 shows the schedule length (SL) for each benchmark circuit after front-end synthesis, after placement, and after the final scheduling. The schedule length lower bound after placement is calculated assuming:

- each coarse-grained node in the DFG that requires the ALU takes one system clock cycle,
- all sequences of fine-grained nodes collectively take a single clock cycle, regardless of the number of nodes (so the lower bound could decrease as more nodes are considered fine-grained),
- all communication requires a number of cycles equal to the Manhattan distance, and
- there are infinite resources for routing.

The "Placement Ratio" column in Table 5.4 computes the fraction of the schedule length increase caused by placement (the rest is caused by routing and scheduling). The results are

Table 5.3: M-CAD frequency upper bound and actual F_{max}. Entries with a "–" are the same as the $W_f = 0$ value. The ratio is F_{max} / Bound; larger ratios are better.

		$W_f = 0$			$W_f = 1$			$W_f = 4$		
	Circuit	**Bound (MHz)**	**F_{max} (MHz)**	**Ratio**	**Bound (MHz)**	**F_{max} (MHz)**	**Ratio**	**Bound (MHz)**	**F_{max} (MHz)**	**Ratio**
CG-only	fft16	250.0	45.5	0.18	–	–	–	–	–	–
	me	55.6	55.6	1.00	–	–	–	–	–	–
	chem	58.8	40.0	0.68	–	–	–	–	–	–
	fft8	250.0	71.4	0.29	–	–	–	–	–	–
	honda	55.6	45.5	0.82	–	–	–	–	–	–
	mcm	100.0	71.4	0.71	–	–	–	–	–	–
	wang	111.1	83.3	0.75	–	–	–	–	–	–
	pr	100.0	83.3	0.83	–	–	–	–	–	–
	Geo. Mean (CG-only)			**0.58**			**0.58**			**0.58**
Good	ac97_ctrl	21.7	14.3	0.66	66.7	40.0	0.60	66.7	40.0	0.60
	aes_core	55.6	27.8	0.50	55.6	25.0	0.45	55.6	25.6	0.46
	dir	40.0	29.4	0.74	40.0	27.0	0.68	40.0	27.0	0.68
	spi	33.3	25.6	0.77	55.6	32.3	0.58	55.6	38.5	0.69
	pci_master	29.4	22.7	0.77	40.0	29.4	0.91	125.0	76.9	0.62
	Geo. Mean (Good)			**0.68**			**0.60**			**0.60**
Geo. Mean (CG-only and Good)				**0.62**			**0.59**			**0.59**
Impaired	ethernet	9.9	6.7	0.67	12.8	11.9	0.93	32.3	20.8	0.65
	wb_conmax	40.0	13.2	0.33	40.0	19.6	0.49	58.8	17.8	0.30
	dma	15.6	5.8	0.37	21.3	10.2	0.48	23.8	10.6	0.45
	tv80	18.9	7.0	0.37	18.9	4.9	0.26	66.0	64.0	0.97
	jpeg_enc	32.3	5.2	0.16	32.3	5.3	0.16	32.3	5.2	0.16
	systemcaes	71.4	15.2	0.21	76.9	15.4	0.20	90.9	14.7	0.16
	des	6.7	3.6	0.54	8.8	5.2	0.60	10.2	5.3	0.52
	systemcdes	28.6	21.3	0.74	28.6	21.7	0.76	30.3	18.9	0.62
	Geo. Mean (Impaired)			**0.38**			**0.41**			**0.40**
Geo. Mean (All)				**0.51**			**0.51**			**0.51**

fairly consistent for the *CG-only*, *Good*, and *Impaired* benchmarks. Overall, placement contributes 70.8% of the difference between the achieved F_{max} and the upper bound. This is not too surprising, because it is during placement where the constraints of the underlying architecture are imposed on the circuit (e.g., interconnect delays, I/O constraints, memory constraints). However, because of other factors on time-multiplexed architectures, discussed in the rest of this section, it is likely that the placement can be improved.

Table 5.4: M-CAD schedule length lower bound comparison for $W_f = 0$.

	Circuit	Schedule Length (SL) Lower Bound After: Front-End Synthesis	Place	Schedule	Total SL Increase (*Total*)	Placement SL Increase (*Place*)	Placement Ratio (*Place*/*Total*)
CG-only	fft16	4	17	22	18	13	0.722
	me	18	18	18	0	0	–[1]
	chem	17	23	25	8	6	0.750
	fft8	4	12	14	10	8	0.800
	honda	18	21	22	4	3	0.750
	mcm	10	12	14	4	2	0.500
	wang	9	11	12	3	2	0.667
	pr	10	12	12	2	2	1.000
	Geo. Mean (CG-only)						**0.728**
Good	ac97_ctrl	46	60	70	24	14	0.583
	aes_core	18	31	36	18	13	0.722
	dir	25	32	34	9	7	0.778
	spi	30	35	39	9	5	0.556
	pci_master	35	42	44	9	7	0.778
	Geo. Mean (Good)						**0.676**
Geo. Mean (CG-only and Good)							**0.706**
Impaired	ethernet	101	130	150	49	29	0.592
	wb_conmax	25	61	76	51	36	0.706
	dma	64	150	171	107	86	0.804
	tv80	53	124	143	90	71	0.789
	jpeg_enc	31	150	194	163	119	0.730
	systemcaes	14	51	66	52	37	0.712
	des	149	230	276	127	81	0.638
	systemcdes	35	44	47	12	9	0.750
	Geo. Mean (Impaired)						**0.712**
Geo. Mean (All)							**0.708**

[1]This value is excluded from the geomean since the ratio is undefined.

In traditional FPGA CAD, the placement and routing tasks are separate, and the tools can achieve a high-quality result even though placement operates with no routing information. A time-multiplexed architecture adds a temporal dimension to the problem that makes the lack of information more of a problem. This is demonstrated below.

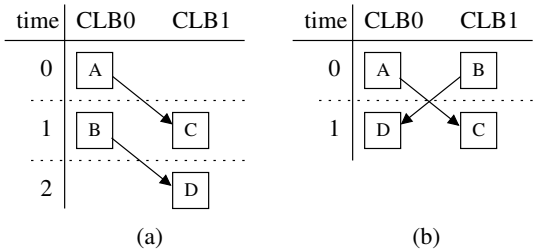


Figure 5.6: Example of bad placement. (a) The total wire length is 2, assume that A and C may not be placed on the same CLB, and (b) a better placement with the same wire length but requiring one less timeslot after scheduling.

Figure 5.6 illustrates one such problem where the placer needs post-scheduling information to do a better job. The best solution would be to place nodes A and C in CLB0 and nodes B and D in CLB1, but let us assume that the placement cost prefers to assign node A to CLB0, and node C to CLB1. The placer is aware of external communication, but it has no idea when (in which timeslot) such communication will take place. As a result, Figures 5.6a and 5.6b have the same cost of 2, even though Figures 5.6b achieves a better post-scheduling solution. While this specific problem is trivial to fix, finding a solution for the general case is more difficult. In an attempt to improve upon this, Chapter 6 presents a tool flow that performs simultaneous placement, routing, and scheduling.

Given that there are architectural constraints on the circuit which may necessitate an increase in the schedule length, the upper-bound frequency calculated in this section is likely an unachievable target. Since placement accounts for 70.8% of the overall schedule length increase after front-end synthesis, it is worth investigating whether enough effort has been expended by the placement algorithm itself.

Table 5.5 compares the previous upper-bound frequency with results generated using the placer with greater effort. To increase placement effort, an inner-loop multiplier of 10,000 was used instead of 10. The starting temperature was set to 100, which is very high, and the range limit which constraints the locality annealing swaps was disabled so any node in any CLB can be considered.

Comparing the first two "Ratio" columns in Table 5.5, the high-effort placer is able to achieve 57% of the upper bound across all circuits, whereas the M-CAD flow with the default options achieves 51%. However, if the improved F_{max} (high-effort result) is taken as a realistic and achievable upper bound and is compared to the F_{max} from Table 5.4, then the M-CAD flow achieves 91% of the improved F_{max} (the last "Ratio" column in Table 5.5). This leaves little room for circuit performance improvement from placement alone, and suggests future improvements should focus on front-end synthesis.

The front-end synthesis does perform well in many cases, but there is room for improvement. It is not able to produce the same quality solutions as commercial tools like QuartusII. This fact is reinforced by the frequency upper bound computed in this section; it is still slower than the QuartusII synthesis results in Table 5.1.

5.6.3 Area and Density

In this section, the area required to synthesize each benchmark is examined for the maximum performance and maximum density results on the architecture. The maximum density is the minimum area, which is the smallest number of CLBs required to synthesize the circuit without violating any architecture parameter values. These parameter values are the ones from Table 3.5.

Comparison to QuartusII/StratixIII

In Section 3.7 the area of a Malibu CLB was computed to be 97,050.5 μm^2 and the area of a StratixIII ALM was estimated to be 2,674μm^2. This means that a Malibu CLB is roughly equivalent in area to 36.3 ALMs. We compute the equivalent ALMs (eALMs) for Malibu by multiplying the number CLBs by 36.3 when $W_f \geq 1$. When $W_f = 0$, the area of a Malibu CLB, excluding all the fine-grained resources, is 97,050.5 - 18,487 = 78,563.5 μm^2. Therefore, a coarse-grained-only CLB is equivalent to 29.4 ALMs, and this value is used to compute the

Table 5.5: M-CAD high-effort schedule length comparison for $W_f = 0$.

		$W_f = 0$ from Table 5.3			High-Effort Placement		
	Circuit	Bound (MHz)	F_{max} (MHz)	Ratio	Improved F_{max} (MHz)	Improved Ratio	F_{max}/Improved F_{max} Ratio
CG-only	fft16	250.0	45.5	0.18	45.5	0.18	1.00
	me	55.6	55.6	1.00	55.6	1.00	1.00
	chem	58.8	40.0	0.68	43.5	0.74	0.92
	fft8	250.0	71.4	0.29	71.4	0.29	1.00
	honda	55.6	45.5	0.82	45.5	0.82	1.00
	mcm	100.0	71.4	0.71	83.3	0.83	0.86
	wang	111.1	83.3	0.75	83.3	0.75	1.00
	pr	100.0	83.3	0.83	83.3	0.83	1.00
	Geo. Mean (CG-only)			**0.58**		**0.60**	**0.97**
Good	ac97_ctrl	21.7	14.3	0.66	20.0	0.92	0.71
	aes_core	55.6	27.8	0.50	27.8	0.50	1.00
	dir	40.0	29.4	0.74	29.4	0.74	1.00
	spi	33.3	25.6	0.77	29.4	0.88	0.87
	pci_master	29.4	22.7	0.77	25.6	0.87	0.89
	Geo. Mean (Good)			**0.68**		**0.76**	**0.89**
Geo. Mean (CG-only and Good)				**0.62**		**0.66**	**0.94**
Impaired	ethernet	9.9	6.7	0.67	9.3	0.94	0.71
	wb_conmax	40.0	13.2	0.33	14.9	0.37	0.88
	dma	15.6	5.8	0.37	8.5	0.54	0.69
	tv80	18.9	7.0	0.37	7.2	0.38	0.97
	jpeg_enc	32.3	5.2	0.16	5.8	0.18	0.88
	systemcaes	71.4	15.2	0.21	15.2	0.21	1.00
	des	6.7	3.6	0.54	4.7	0.71	0.76
	systemcdes	28.6	21.3	0.74	21.3	0.74	1.00
	Geo. Mean (Impaired)			**0.38**		**0.44**	**0.85**
Geo. Mean (All)				**0.51**		**0.57**	**0.91**

equivalent ALMs for $W_f = 0$.

Table 5.6 compares the circuit area for the benchmarks implemented on Malibu with M-CAD and on a StratixIII FPGA using QuartusII. At maximum performance, Malibu requires more area than an FPGA, using on average (geomean) twice the area of an FPGA for the *CG-only* and *Good* benchmarks at $W_f = 0$. When the fine-grained resources are enabled ($W_f \geq 1$) the density becomes worse for the *CG-only* circuits because they cannot make use of such

Table 5.6: M-CAD area and density values compared to QuartusII/StratixIII. Entries with a "–" are same as the $W_f = 1$ value. The "Density" columns are the area reduction compared to QuartusII (QuartusII ALMs / Malibu equivalent ALMs (eALMs)).

			M-CAD Maximum Performance						M-CAD Maximum Density					
		QuartusII	$W_f = 0$		$W_f = 1$		$W_f = 4$		$W_f = 0$		$W_f = 1$		$W_f = 4$	
	Circuit	ALMs	eALMs	Dens.	eALMs	Dens.	eALMs	Dens.	eALMs	Dens.	eALMs	Dens.	eALMs	Dens.
CG-only	fft16	6,412	11,760	0.545	9,293	0.690	–	–	470	13.640	581	11.042	–	–
	me	5,148	11,760	0.438	14,520	0.355	–	–	1,880	2.738	1,307	3.940	–	–
	chem	3,526	1,058	3.331	2,323	1.518	–	–	264	13.335	327	10.795	–	–
	fft8	2,075	7,526	0.276	9,293	0.223	–	–	264	7.847	327	6.352	–	–
	honda	1,216	1,058	1.149	1,307	0.931	–	–	264	4.599	327	3.723	–	–
	mcm	1,057	1,058	0.999	1,307	0.809	–	–	470	2.249	327	3.236	–	–
	wang	797	470	1.694	581	1.372	–	–	264	3.014	327	2.440	–	–
	pr	646	1,882	0.343	581	1.112	–	–	264	2.443	327	1.978	–	–
	Geo. Mean (CG-only)			**0.786x**		**0.745x**		**0.745x**		**4.833x**		**4.517x**		**4.517x**
Good	ac97_ctrl	1,254	11,760	0.107	5,227	0.240	3,630	0.345	1,058	1.186	2,323	0.540	3,630	0.346
	aes_core	1,154	2,940	0.393	2,323	0.497	5,227	0.221	264	4.364	581	1.987	1,307	0.883
	dir	1,150	1,058	1.087	1,307	0.880	581	1.980	264	4.349	581	1.980	581	1.980
	spi	488	2,940	0.166	1,307	0.373	581	0.840	264	1.846	581	0.840	581	0.840
	pci_master	137	1,058	0.129	581	0.236	581	0.236	264	0.518	327	0.419	581	0.236
	Geo. Mean (Good)			**0.250x**		**0.392x**		**0.469x**		**1.847x**		**0.944x**		**0.654x**
Geo. Mean (CG-only and Good)				**0.506x**		**0.582x**		**0.637x**		**3.339x**		**2.474x**		**2.148x**
Impaired	ethernet	6,868	16,934	0.406	14,520	0.473	14,520	0.473	7,521	0.913	2,940	2.336	5,226	1.314
	wb_conmax	5,349	11,760	0.455	83,635	0.064	37,171	0.144	2,938	1.821	14,518	0.368	9,291	0.576
	dma	1,714	16,934	0.101	14,520	0.118	28,459	0.060	2,938	0.583	9,291	0.184	20,905	0.082
	tv80	850	7,526	0.113	58,080	0.015	58,080	0.015	5,759	0.148	2,940	0.289	5,226	0.163
	jpeg_enc	791	67,738	0.012	58,080	0.014	37,171	0.021	47,009	0.017	14,518	0.054	5,226	0.151
	systemcaes	716	67,738	0.011	16,934	0.042	11,760	0.061	264	2.708	1,307	0.548	2,323	0.308
	des	298	4,234	0.070	28,459	0.010	14,520	0.021	3,555	0.084	2,940	0.101	5,226	0.057
	systemcdes	237	4,234	0.056	14,520	0.016	11,760	0.020	470	0.504	2,940	0.081	2,940	0.081
	Geo. Mean (Impaired)			**0.075x**		**0.039x**		**0.049x**		**0.359x**		**0.241x**		**0.196x**
Geo. Mean (All)				**0.245x**		**0.207x**		**0.239x**		**1.428x**		**1.018x**		**0.863x**

resources, however the density improves for the *Good* circuits. Collectively, the benefits to the *Good* circuits outweigh the density drop for the *CG-only* circuits, improving density from 0.506x to 0.582x.

At maximum density, the M-CAD flow achieves almost 5x the density of an FPGA for the *CG-only* circuits due to aggressive time-multiplexing on the coarse-grained resources. For the *Good* circuits (which have fine-grained requirements), it is able to nearly match the density with $W_f = 1$. The limiting constraint for the maximum density results was often the number of LUTs per CLB, which was set to 16. More FG resources per CLB would help in this case, as would a heterogeneous architecture where some CLBs do not have the CG component. Also, a better CG/FG partitioning strategy and better front-end logic synthesis would help as well.

Comparison to VPR/iFAR

Table 5.7 shows the circuit area in units of millions of minimum-width transistor areas (T $\times 10^6$) for VPR and M-CAD at both the maximum performance and maximum density. The VPR area is taken directly from the output of VPR (the tile area multiplied by the number of CLBs used). The Malibu area is computed using the CLB area calculated in Section 3.7 and multiplied by the number of CLBs used.

The VPR area results are similar to the QuartusII/StratixIII results in Table 5.6. The VPR architecture is more dense than the StratixIII, so the Malibu results are correspondingly less dense relative to VPR than QuartusII.

Alternate Architectures

An alternate 10x density architecture configuration from Table 5.8 can also be used with Malibu. The 10x density architecture was designed with an appropriate amount of memory resources to achieve an average (geomean) 10x density compared to an FPGA across all the benchmarks. The use of this 10x density architecture is restricted to just the discussion in this

Table 5.7: M-CAD area and density values compared to VPR/iFAR. Entries with a "–" are same as the $W_f = 1$ value. The "Dens." columns are the area density compared to VPR (VPR T / M-CAD T).

			M-CAD Maximum Speed						M-CAD Maximum Density					
		VPR	$W_f = 0$		$W_f = 1$		$W_f = 4$		$W_f = 0$		$W_f = 1$		$W_f = 4$	
	Circuit	**T** $\times 10^6$	**T** $\times 10^6$	**Dens.**	**T** $\times 10^6$	**Dens.**	**T** $\times 10^6$	**Dens.**	**T** $\times 10^6$	**Dens.**	**T** $\times 10^6$	**Dens.**	**T** $\times 10^6$	**Dens.**
CG-only	fft16	48.12	62.85	0.766	49.69	0.968	–	–	2.51	19.155	3.11	15.498	–	–
	me	14.51	62.85	0.231	77.64	0.187	–	–	10.05	1.444	6.99	2.077	–	–
	chem	31.41	5.66	5.552	12.42	2.528	–	–	1.41	44.446	1.75	35.961	–	–
	fft8	11.72	40.22	0.291	49.69	0.236	–	–	1.41	8.291	1.75	6.708	–	–
	honda	3.22	5.66	0.569	6.99	0.461	–	–	1.41	2.277	1.75	1.843	–	–
	mcm	2.60	5.66	0.459	6.99	0.372	–	–	2.51	1.034	1.75	1.487	–	–
	wang	1.96	2.51	0.780	3.11	0.632	–	–	1.41	1.388	1.75	1.123	–	–
	pr	1.60	10.06	0.159	3.11	0.516	–	–	1.41	1.135	1.75	0.918	–	–
	Geo. Mean (CG-only)			**0.557x**		**0.528x**		**0.528x**		**3.734x**		**3.489x**		**3.489x**
Good	ac97_ctrl	2.84	62.85	0.045	27.95	0.102	19.41	0.146	5.65	0.503	12.42	0.229	19.41	0.146
	aes_core	3.65	15.71	0.232	12.42	0.294	27.95	0.131	1.41	2.584	3.11	1.176	6.99	0.523
	dir	11.06	5.66	1.956	6.99	1.583	3.11	3.562	1.41	7.828	3.11	3.563	3.11	3.563
	spi	0.79	15.71	0.050	6.99	0.113	3.11	0.254	1.41	0.557	3.11	0.254	3.11	0.254
	pci_master	0.46	5.66	0.081	3.11	0.148	3.11	0.148	1.41	0.326	1.75	0.264	3.11	0.148
	Geo. Mean (Good)			**0.153x**		**0.240x**		**0.303x**		**1.131x**		**0.577x**		**0.400x**
Geo. Mean (CG-only and Good)				**0.339x**		**0.390x**		**0.426x**		**2.358x**		**1.747x**		**1.517x**
Impaired	ethernet	21.70	90.51	0.240	77.64	0.279	77.64	0.279	40.20	0.540	15.72	1.380	27.95	0.776
	wb_conmax	73.06	62.85	1.162	447.21	0.163	198.76	0.368	15.70	4.653	77.63	0.941	49.68	1.471
	dma	7.76	90.51	0.086	77.64	0.100	152.18	0.051	15.70	0.494	49.68	0.156	111.78	0.069
	tv80	2.30	40.22	0.057	310.56	0.007	310.56	0.007	30.78	0.075	15.72	0.147	27.95	0.082
	jpeg_enc	1.64	362.02	0.005	310.56	0.005	198.76	0.008	251.24	0.007	77.63	0.021	27.95	0.059
	systemcaes	2.03	362.02	0.006	90.55	0.022	62.88	0.032	1.41	1.435	6.99	0.290	12.42	0.163
	des	0.60	22.63	0.027	152.18	0.004	77.64	0.008	19.00	0.032	15.72	0.038	27.95	0.022
	systemcdes	0.51	22.63	0.022	77.64	0.007	62.88	0.008	2.51	0.201	15.72	0.032	15.72	0.032
	Geo. Mean (Impaired)			**0.046x**		**0.024x**		**0.030x**		**0.220x**		**0.148x**		**0.120x**
Geo. Mean (All)				**0.159x**		**0.134x**		**0.155x**		**0.956x**		**0.681x**		**0.577x**

Table 5.8: Malibu memory area estimates for a 10x density architecture. The first three rows are the Malibu architecture from Table 3.6 used throughout this thesis. The 10x density architecture adds more memory to the R and instruction memories for more aggressive time-multiplexing. The size of the NSEW memories are unchanged.

			SRAM μm^2		eDRAM μm^2		Flash μm^2	
	Memory	**Specification**	**Area**	**Per Bit**	**Area**	**Per Bit**	**Area**	**Per Bit**
Table 3.6	NSEW	32x16, 3R1W	5,430	10.615	–	–	–	–
	R	32x128, 3R1W	27,244	6.651	–	–	–	–
	Instr.	81x256, 1RW	20,717	0.999	6,013	0.290	1,421	0.0686
10x Density	NSEW	32x16, 3R1W	5,430	10.651	–	–	–	–
	R	32x256, 3R1W	52,300	4.591	–	–	–	–
	Instr.	81x1024, 1RW	76,331	0.920	24,054	0.290	5,686	0.0686

subsection, all other results in this thesis use the architecture defined in Table 3.6.

Two coarse-grained benchmark circuits (*fft16* and *chem*) already exceed 10x density of an FPGA using the original architecture configuration. For the remainder of the benchmarks, to achieve an average 10x density improvement, M-CAD must time-multiplex the circuits more aggressively and be able to fit many operations into each CLB. Doing this requires increasing the size of the instruction memory and the R memory, which means further adjusting the ALM/CLB area estimate to include these larger memories. For most of the benchmark circuits, 10x density translates into a Malibu array size of 2x2 CLBs or fewer. Such small architecture sizes are not possible with the current Malibu input/output resource constraint that a CLB can only perform one input and output each user clock cycle. An alternative input/output mechanism would be required to map circuits with a large number of I/Os onto such a small Malibu array.

At 10x density, the average circuit speed is 3.30 MHz across all benchmarks for $W_f = 0$. This is about 0.01x ($1/100^{th}$) the speed of an FPGA. This is comparable to the VEGA density and performance [40], but accomplished with comparison to a modern FPGA using much larger benchmarks.

This result is only applicable for $W_f = 0$. Enabling the fine-grained resources means that all signals of width $\leq W_f$ will never be time-multiplexed. This means that additional LUTs and wires per CLB would be needed for the fine-grained signals, which makes the CLB tile larger and necessitates even more aggressive time-multiplexing to achieve 10x density. This was not explored.

Although we can reach 10x the density of an FPGA, this gain comes with a further loss in performance as the circuit must be heavily time-multiplexed. Additionally, for the current Malibu architecture, an alternative input/output mechanism would be required for circuits with a large number of I/Os. Again, to keep the analysis in this thesis focussed, the results of the 10x density architecture are not used outside this "Alternate Architectures" subsection.

5.6.4 Compile Time

Table 5.9 shows the complete Verilog-to-bitstream compile times for QuartusII, VPR (including the time for Quartus to build the input BLIF), and M-CAD. Also shown is the speedup compared to QuartusII, where speedup = $\frac{\text{QuartusII time}}{\text{M-CAD time}}$. The speedup compared to VPR is summarized in Table 5.10.

M-CAD shows a significant speedup compared to QuartusII ranging from 249x faster (*chem*) down to 1.3x faster (*dma*, *systemcaes*) for $W_f = 0$. The *CG-only* benchmarks show the highest speedups, as expected, because fine-grained synthesis is not invoked at all.

For $W_f = 1$, the speedup range changes from 249x faster (*chem*) to almost 72x slower (*wb_conmax*), with a geomean of 8.1x faster. We consider *wb_conmax* to be anomalous because a traditional VPR route takes 14 minutes, whereas VPR called by the Malibu to route the FG resources takes over 5 hours. We were unable to determine why VPR runs so slowly in this case.

In Appendix A we show that the compile time in QuartusII can be reduced by 15%, and in VPR by 7% by disabling optimizations, turning off timing-driven placement, and reducing

Table 5.9: M-CAD compile time and speedup versus QuartusII. Compile time is in seconds. Entries with a "–" are for coarse-grained benchmarks and the same as the $W_f = 0$ value.

				M-CAD					
		QuartusII	VPR	$W_f = 0$		$W_f = 1$		$W_f = 4$	
	Circuit	Time (s)	Time	Time	Speedup	Time	Speedup	Time	Speedup
CG-only	fft16	333.4	454.1	5.5	60.4	–	–	–	–
	me	220.5	111.9	13.5	16.3	–	–	–	–
	chem	311.9	3,663.0	1.2	249.8	–	–	–	–
	fft8	187.3	73.7	2.9	64.7	–	–	–	–
	honda	162.8	37.2	0.7	245.4	–	–	–	–
	mcm	153.4	25.9	0.8	200.6	–	–	–	–
	wang	148.2	22.2	10.6	13.9	–	–	–	–
	pr	145.6	18.7	1.3	112.9	–	–	–	–
	Geo. Mean (CG-only)				**77.0x**		**77.0x**		**77.0x**
Good	ac97_ctrl	156.8	20.2	14.0	11.2	48.2	3.3	64.0	2.5
	aes_core	169.6	155.0	5.9	28.6	12.2	13.9	9.9	17.2
	dir	186.8	93.8	2.4	77.2	8.4	22.2	5.7	32.6
	spi	138.1	9.5	3.9	35.3	3.2	42.6	3.4	41.1
	pci_master	140.1	10.0	2.0	70.5	6.7	20.8	5.7	24.6
	Geo. Mean (Good)				**36.1x**		**15.5x**		**17.0x**
Geo. Mean (CG-only and Good)					**57.5x**		**38.7x**		**16.9x**
Impaired	ethernet	306.6	207.2	69.0	4.4	544.0	0.6	543.1	0.6
	wb_conmax	319.8	804.7	96.6	3.3	22,997.1	0.01	9,787.1	0.03
	dma	206.4	64.3	157.2	1.3	2,033.5	0.1	7,871.3	0.03
	tv80	165.9	36.6	15.9	10.4	806.5	0.2	811.9	0.2
	jpeg_enc	149.6	31.3	91.7	1.6	141.5	1.1	125.8	1.2
	systemcaes	160.9	24.8	121.1	1.3	16.4	9.8	1,491.7	0.1
	des	137.3	19.8	7.7	17.8	216.6	0.6	232.3	0.6
	systemcdes	135.4	16.7	2.4	55.9	14.2	9.5	16.6	8.1
	Geo. Mean (Impaired)				**5.02x**		**0.56x**		**0.30x**
Geo. Mean (All)					**28.8x**		**8.08x**		**6.46x**

Table 5.10: M-CAD compile time speedup versus VPR.

	M-CAD vs. VPR		
	$W_f = 0$	$W_f = 1$	$W_f = 4$
Geo. Mean (CG-only)	36.14x	36.14x	36.14x
Geo. Mean (Good)	7.08x	3.03x	3.32x
Geo. Mean (CG-only and Good)	17.97x	12.50x	12.99x
Geo. Mean (Impaired)	1.52x	0.17x	0.09x
Geo. Mean (All)	6.83x	2.30x	1.80x

the placement and routing effort levels. Based on this minimal reduction, it is unlikely that algorithm tuning alone can reduce the compile time of QuartusII or VPR to be comparable to M-CAD.

These results also show that M-CAD can achieve fast compile times compared to FPGA CAD tools, particularly for coarse-grained circuits. The compile times were roughly the same for the maximum performance and for the maximum density results.

5.6.5 Synthesis Time for Very Large Circuits

It is difficult to acquire exceptionally large benchmark circuits that are real circuits. To test the compile time of the tools on very large circuits, we randomly generated *synthetic* benchmarks with a custom tool. The circuits are generated without any graph-depth or locality control, so they are not suitable for testing the quality of the tool output. For such testing, a more realistic random circuit generator would be required. However, we consider them adequate for testing runtime scaling.

The circuits have 32 inputs, 16 outputs, and contain $1,000$ to $50,000$ nodes. Each node is a 32-bit operation (add, subtract, invert, and, or, xor, multiply). To put this into perspective, if we assume the simplest case where each 32-bit operation requires 32 gates, the $50,000$ node benchmark contains 1.6 million gates.

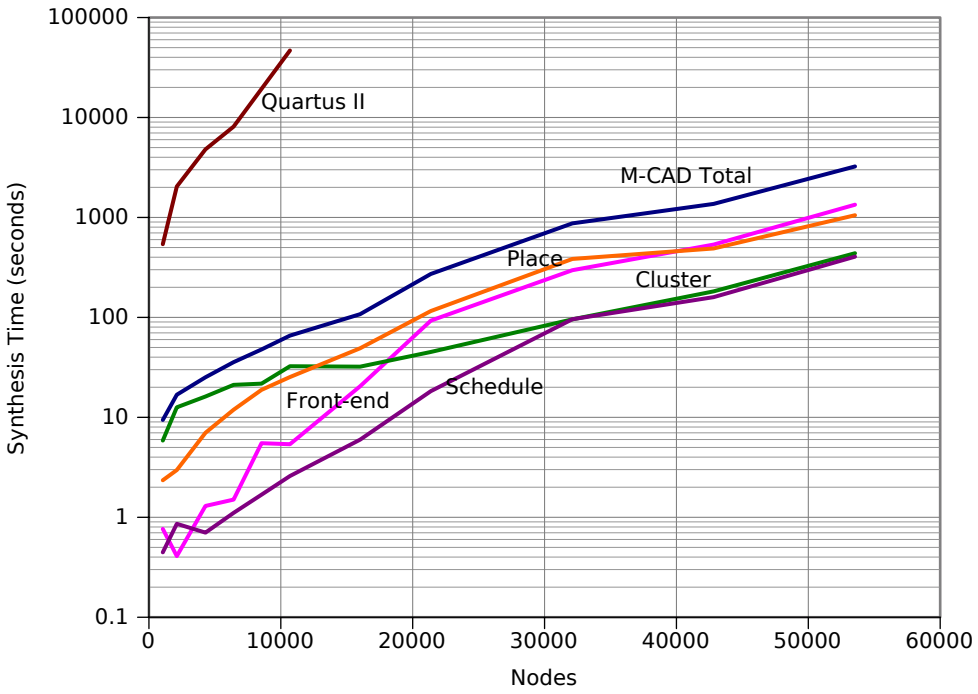


Figure 5.7: Synthesis time for very large circuits. A 32x32 CLB (1024 CLBs total) architecture was used.

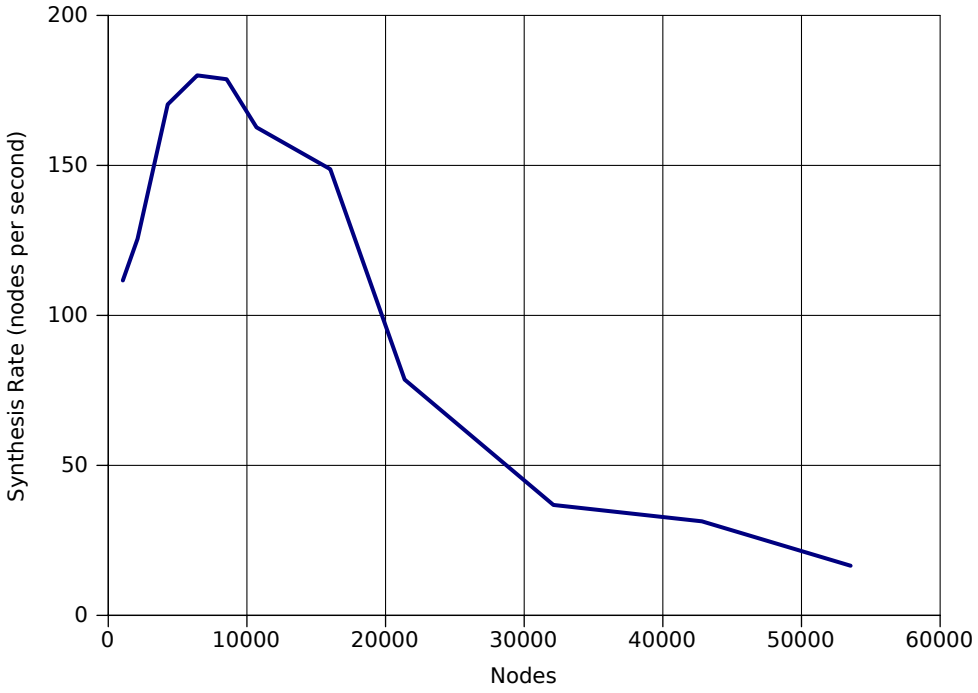


Figure 5.8: Synthesis rate (nodes per second) for very large circuits.

The circuits were also synthesized in QuartusII v10.0 with the largest StratixIII (EP3SL340-F1760C2) to produce a rough comparison with FPGA area usage and runtime. The 1,000 node circuit required 22,066 ALMs (16.2% of the device) and synthesized in 9 minutes. The 2,000 node circuit used 64,748 ALMs (47.6%) synthesized in 34 minutes. The 4,000 node circuit used 152,338 ALMs (112%) and stopped after 80 minutes as it exceeded the size of the device. The 6,000 node circuit exhausted the 32-bit 4GB memory limit and was forced to quit after 2 hours. The 10,000 node circuit exhausted the memory after 13 hours. The 50,000 node circuit ran for 24 hours without finishing the analysis and synthesis phase (before placement and routing). To be fair, QuartusII definitely attempts to do more optimizations than Verilator, and the benchmarks are not "FPGA-friendly" in that the circuits are entirely coarse-grained and very large. However, they are all valid circuits. The exact numeric comparisons are not important; what is of significance is that M-CAD can successfully synthesize these large circuits and QuartusII cannot.

Figure 5.7 shows the average compile time for M-CAD when run with 10 trials of the random benchmarks (10 different random benchmarks of the same size for each trial) compiled for an architecture of 1024 (32x32) CLBs. The total runtime curve is shown, plus a breakdown for each step in M-CAD. As the circuit size increases, front-end synthesis and placement dominate the runtime. For the 50,000 node circuit, the total compile time was 54 minutes.

To reduce the overall time, front-end synthesis (parsing, elaboration, optimization, and DFG generation) and placement are the two most likely targets. However, it is unlikely that the front-end synthesis can be made much faster, except by reducing the optimization done. The placement step can be improved by reducing the inner-loop iterations of the annealer, at the expense of quality. The number of inner-loop iterations is the same as in VPR, $10 \times n\_clusters^{1.3}$, which we have found to be a good balance between quality and speed. Increasing this causes longer run-times with little or no improvement in results, and decreasing it gives significantly poorer results. Beyond this, placement could be improved by changing to a fundamentally

different approach, e.g. analytical placement. Alternatively, it was recently shown by Smecher *et al.* [70] that a Massively Parallel Processor Array (MPPA) is capable of greatly accelerated placement by self-hosting a parallel simulated-annealing algorithm. Subsequent work by Wang *e*t al. [87] showed that a many-core system can speedup placement by 123x compared to VPR with only a small quality degradation.

Figure 5.8 shows the total compile time as a rate in nodes-per-second. The initial rise in rate, ending at $\approx 8,000$ nodes, is caused by amortization of the tool overhead. Beyond this, algorithmic complexity catches up. Scaling to even larger circuits than shown here may require heuristics with better algorithmic complexity or with reduced quality of results.

5.6.6 Longest-Path Analysis

The longest paths in each circuit can highlight where the M-CAD algorithms might be improved to reduce the overall schedule length. The length of a path in the scheduled circuit is the time required to traverse all compute and routing resources from inputs and registers, to outputs and registers. This includes the time spent waiting for resources.

Table 5.11 shows the longest-path analysis for the maximum performance results in Table 5.3. In the table, starting from the left, is the schedule length of the synthesized circuit (SL) which is reported as the F_{max} in Table 5.3. The next column is the longest path (Len), followed by the number of longest-paths (Count) because there is often more than one, each using a slightly different set of resources. The longest path will always be less than or equal to the schedule length; it can be lower in cases where an input does not immediately occur in the first timeslot due to scheduling decisions, or an output occurs before the last timeslot.

The next five columns are the longest-path breakdown, averaged over the number of longest-paths: the number of compute-only slots where an ALU is in use (ALU); the number of wait slots spent waiting for values to arrive or waiting for the ALU while it is busy with other computation (Wait); the number of compute-and-move slots where a value is computed

Table 5.11: M-CAD Longest-path breakdown for maximum performance results. All numbers are system clock cycles.

		M-CAD $W_f = 0$								**M-CAD $W_f = 1$**								
					Compute			**Route**					**Compute**			**Route**		
	Circuit	**SL**	**Len**	**Count**	**ALU**	**Wait**	**Move**	**Route**	**Hold**[1]	**SL**	**Len**	**Count**	**ALU**	**Wait**	**Move**	**Route**	**Hold**[1]	**FG**
CG-only	fft16	22	22	106	1.0	16.3	0.0	4.7	0.0	–	–	–	–	–	–	–	–	–
	me	18	18	20	1.6	11.0	0.1	5.2	0.0	–	–	–	–	–	–	–	–	–
	chem	25	25	1	5.0	13.0	0.0	7.0	0.0	–	–	–	–	–	–	–	–	–
	fft8	14	14	48	1.0	8.2	0.0	4.8	0.0	–	–	–	–	–	–	–	–	–
	honda	22	22	2	13.0	0.0	4.0	5.0	0.0	–	–	–	–	–	–	–	–	–
	mcm	14	14	1	7.0	0.0	0.0	7.0	0.0	–	–	–	–	–	–	–	–	–
	wang	12	10	10	6.2	0.0	1.6	2.2	0.0	–	–	–	–	–	–	–	–	–
	pr	12	12	2	8.0	0.0	1.0	3.0	0.0	–	–	–	–	–	–	–	–	–
Good	ac97_ctrl	70	70	8	1.8	64.9	0.0	3.4	0.0	25	25	2	1.0	21.0	0.0	3.0	0.0	0.0
	aes_core	36	36	1	13.0	0.0	0.0	23.0	0.0	40	39	1	13.0	5.0	0.0	21.0	0.0	0.0
	dir	34	34	1	5.0	25.0	1.0	3.0	0.0	37	37	6	17.0	11.0	1.0	8.0	0.0	0.0
	spi	39	36	2	16.0	0.0	2.5	17.5	0.0	31	16	1	10.0	0.0	0.0	3.0	0.0	3.0
	pci_master	44	43	2	10.5	28.0	0.0	4.5	0.0	34	15	1	1.0	13.0	0.0	1.0	0.0	0.0
Impaired	ethernet	150	116	10	3.5	104.4	0.0	8.1	0.0	84	61	1	4.0	54.0	0.0	3.0	0.0	0.0
	wb_conmax	76	76	1	1.0	71.0	0.0	4.0	0.0	51	36	1	13.0	0.0	3.0	17.0	0.0	3.0
	dma	171	171	1	16.0	123.0	0.0	32.0	0.0	98	98	7	2.0	89.0	2.0	4.0	0.0	1.0
	tv80	143	143	1	13.0	96.0	1.0	33.0	0.0	204	137	11	11.0	118.0	0.0	7.0	0.0	1.0
	jpeg_enc	194	194	23	3.0	176.7	0.0	14.3	0.0	190	190	22	2.4	180.7	0.0	6.7	0.0	0.2
	systemcaes	66	66	16	1.8	54.9	0.3	9.1	0.0	65	65	10	1.8	56.9	0.7	5.5	0.0	0.1
	des	256	186	1	104.0	0.0	12.0	70.0	0.0	191	158	1	120.0	0.0	3.0	31.0	0.0	4.0
	systemcdes	47	47	1	4.0	36.0	0.0	7.0	0.0	46	46	1	30.0	0.0	2.0	13.0	0.0	1.0

		M-CAD $W_f = 4$								
Good	ac97_ctrl	25	24	1	1.0	22.0	0.0	1.0	0.0	0.0
	aes_core	39	38	1	14.0	0.0	1.0	22.0	0.0	1.0
	dir	37	37	2	3.5	30.0	0.5	3.0	0.0	0.0
	spi	26	14	1	9.0	0.0	0.0	3.0	0.0	2.0
	pci_master	13	7	1	1.0	2.0	0.0	4.0	0.0	0.0
Impaired	ethernet	48	48	1	5.0	34.0	0.0	8.0	0.0	1.0
	wb_conmax	56	36	1	5.0	0.0	0.0	31.0	0.0	0.0
	dma	94	94	9	5.0	82.0	0.0	6.0	0.0	1.0
	tv80	16	16	1	14.0	0.0	0.0	1.0	0.0	1.0
	jpeg_enc	191	191	5	1.4	184.0	0.8	4.8	0.0	0.0
	systemcaes	68	68	5	1.6	61.2	0.2	4.8	0.0	0.2
	des	188	166	1	111.0	0.0	4.0	46.0	0.0	5.0
	systemcdes	53	39	2	25.0	0.0	2.0	11.0	0.0	1.0

[1]The number of hold slots are zero for all circuits (not just rounded to zero).

then transfered directly to a neighbouring CLB without using the crossbar (Move); the number of routing timeslots where progress is made (Route); and the number of routing hold timeslots where a value is held due to a route conflict (Hold).

For $W_f > 0$, the results for the *CG-only* circuits are equal to the $W_f = 0$ results because they use no fine-grained resources, so the results are not duplicated. On the far right of the table is an additional column (FG) for the number of timeslots that the scheduler allocated for fine-grained routing and computation. This value is only applicable for $W_f > 0$, because when $W_f = 0$ there are no fine-grained resources used.

Wait cycles are the extra cycles between the time an ALU operation is scheduled and the time of its most closely scheduled predecessor ALU operation. Some wait cycles are due to a ready operation waiting for the ALU, because it is busy servicing a large number of other operations that are also ready. Other wait cycles are due to an operation waiting for an operand to arrive over the routing network; in this case, the operand has already been computed but it must be transported. M-CAD tries to avoid wait cycles along the longest paths (most critical) by using slack and criticality and scheduling the most critical paths first. However, in doing this, other non-critical paths are forced to wait and incur wait cycles until they too become critical. To further reduce the number of wait cycles, it is possible to change the architecture by adding multiple ALUs per CLB, or adding longer interconnect wires that span multiple CLBs in a single clock cycle, or both. Future work will investigate these and other architectural options for reducing waiting time.

The criticality scheduling approach is also what causes several paths to have the same longest-length: a path is delayed up to the point where it become critical, and then it is treated with higher priority in the scheduler (causing other paths with lower criticality to be delayed, until they also become critical, etc.). There are a few singletons (circuits with a single longest path) with high ALU wait times, like *dma* and *wb_conmax* when $W_f = 0$. For these two circuits, an investigation revealed that the singleton is a modulo path that is lengthened near the end of

scheduling when the scheduler reconnects modulo routes. Up to that point, the singleton path has the same length as several other paths, but the scheduler is forced to increase the path length to complete a modulo route. The next chapter presents a CAD flow where this is less of a problem because placement, routing, and scheduling are done concurrently. The placer can get immediate feedback from the router and scheduler, and thus can avoid placements which unnecessarily increase the length of a critical path.

In Table 5.11, the number of compute timeslots and route timeslots are close, meaning that the longest path spends roughly the same amount of time actively computing and routing. This suggests that the tools may benefit from placing more emphasis on trying to use compute-and-move type operations. The zero hold slots (no values were rounded down to zero) shows that the horizontal-then-vertical routing strategy combined with the abundance of routing resources means there are indeed few routing conflicts.

5.6.7 Density Versus Performance Tradeoff

This section investigates the density versus performance tradeoff in M-CAD. We show that the M-CAD flow can trade density for performance, and that increasing W_f generally gives performance gains across all densities, not just at the F_{max} as was shown in Table 5.1.

Figure 5.9 shows the frequency of the largest benchmark, *ethernet*, for W_f =0, 1, and 4 over a range of architecture sizes from 3x3 CLBs to 48x48 CLBs.

For each architecture array size, the tools are forced to use all available CLBs so that the performance on the various sized architecture arrays can be seen. When the array becomes too large, there is a decrease in performance as communication delay dominates the schedule. If the array is too small, the synthesis result may not be viable if the fixed resource constraints (from Table 3.5) are violated. In these cases, the M-CAD flow still finishes the synthesis by overusing the necessary resources, but it reports an error. For completeness, these results are still included and marked with a dotted line on the left of each curve.

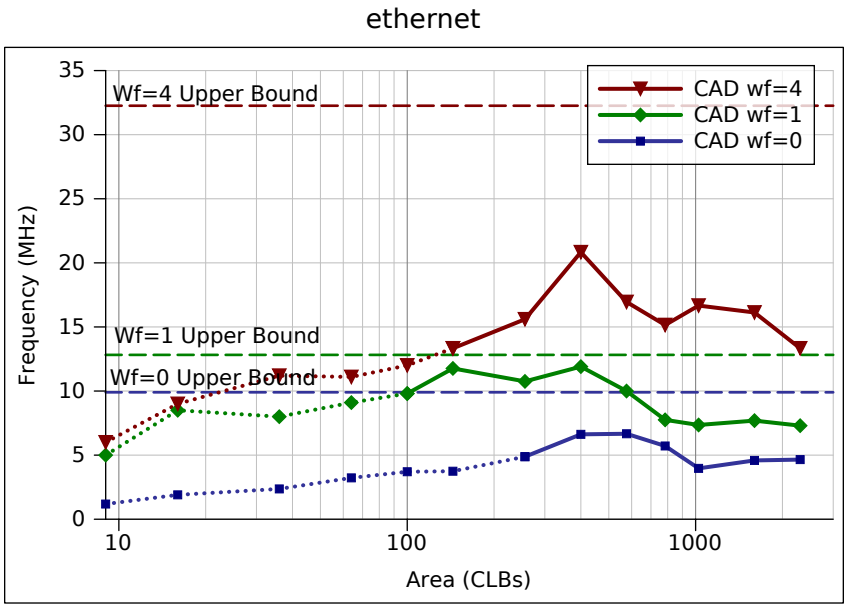


Figure 5.9: Frequency versus area (number of CLBs) for the *ethernet* benchmark. The dotted line represents synthesis results where architectural resource constraints were violated.

This graph demonstrates two things. First, it shows the Malibu tools can trade area (number of CLBs) for performance (MHz) by targeting any architecture array size and time-multiplexing more nodes (or fewer) on the CGs. This is useful for fitting a large design in a small architecture, for example. Second, it shows the tradeoff involving W_f. Increasing W_f from 0 to 1 causes the frequency to increase. This is expected since the fine-grained control logic is moved to the FG resources where it can be computed and distributed more quickly.

This graph is duplicated in Appendix D, which also contains individual area versus performance graphs for the other benchmark circuits. These two trends can be seen in all these graphs.

5.7 Conclusions

This chapter presented the M-CAD flow for mapping circuits to the Malibu architecture. It is a segregated flow based on the traditional place-then-route CAD model with an additional final

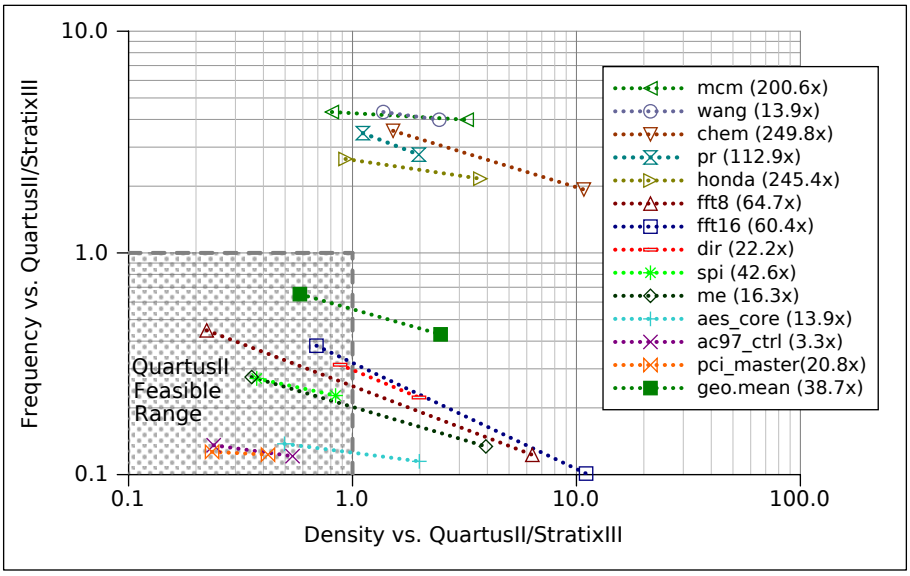


Figure 5.10: M-CAD $W_f = 1$ results summary. The *CG-only* and *Good* benchmark frequency versus density is shown relative to QuartusII/StratixII. The maximum density and maximum performance data points are shown for each benchmark. The compile time speedup compared to QuartusII is shown in the figure legend after the benchmark name.

step, scheduling, to temporally order the coarse-grained operations in each CLB.

The density, performance, and compile time results for each benchmark are summarized in Figure 5.10. The figure shows the density versus performance for each of the *CG-only* and *Good* benchmarks for M-CAD $W_f = 1$. The shaded area of the figure is the performance and density achievable with QuartusII on a StratixIII FPGA. Each curve has two data points, corresponding to the maximum performance and maximum density. The compile time speedup relative to QuartusII is included in the figure legend after each benchmark name. For all the circuits except *spi*, *ac97_ctrl*, and *pci_master*, M-CAD is able to achieve a density result outside the range reachable by QuartusII/StratixIII. For all the benchmarks, the compile time compared to QuartusII is significantly faster.

The results presented in this section show that the M-CAD flow performs well. A 38.7x improvement in compile time is achieved for $W_f = 1$ over FPGA CAD tools (excluding the

Impaired benchmark circuits). M-CAD can trade density for circuit performance, with results for $W_f = 1$ ranging from 2.474x density at 0.429x performance to 0.582x density at 0.652x performance compared to QuartusII/StratixIII. Additionally, when only the coarse-grained benchmarks are considered, the compile time is 77x, the density is nearly 5x, and the performance is 80% that of a commercial FPGA.

The problems caused by lack of information sharing between the tools—usually only a minor annoyance in FPGA CAD—are more of a problem when a temporal dimension is added to the architecture and the CAD. This was demonstrated with an example where the placer would benefit from the post-scheduling results, and reinforced with an upper bound frequency investigation. M-CAD was only able to achieve 50% of the upper bound frequency for the benchmark circuits, with 70% of the loss contributed by the placer (although this is partly due to the placer enforcing layout restrictions on the circuit). Compiling with additional placement effort did not appreciably improve the results. Even though an optimal placement, routing, and scheduling solution may not be achievable, a 2x performance gap does leave room for improvement. We believe that a better result is possible if scheduling decisions can be made during placement. This motivated the M-HOT flow described in the next chapter. M-HOT achieves better performance and density than M-CAD by integrating placement, routing, and scheduling into a single algorithm.

Chapter 6

M-HOT: A Height-Oriented Tool Flow

6.1 Overview

The M-CAD tool in Chapter 5 was able to quickly synthesize a circuit, but only came within 50% of the upper bound on frequency. This chapter describes an alternative approach to the CAD tools for Malibu. It integrates the placement, routing, and scheduling into a single algorithm to achieve a higher-quality solution. Although not as fast as M-CAD, M-HOT can still achieve fast compile times and improve both density and performance results. Just as with M-CAD, M-HOT can also trade density for performance.

The results in this chapter have led us to recommend that the Malibu architecture be used with M-HOT, and with a fine-grained width of 1 ($W_f = 1$). M-HOT provides a very fast compilation, 26.1x faster than QuartusII, and generates high-quality results. Using $W_f = 1$ balances the performance gains from increasing W_f with the density loss by doing the same, as will be shown in this chapter. At this setting, at maximum performance, M-HOT/Malibu achieves 70% the performance of an StratixIII FPGA and about the same density (26.1x compile speedup, 0.707x performance, 0.996x density). At maximum density, Malibu is about half the performance of the FPGA and almost 2.5x the density (26.1x compile speedup, 0.513x performance,

2.474x density). Both of these are reached without changing the underlying architecture, only an M-HOT compiler setting. We consider this an excellent result for a time-multiplexed coarse-grained architecture: the tools can compile circuits quickly, the architecture is close to a commercial FPGA in performance, and exceeds it in density.

This chapter is organized as follows. Sections 6.2–6.5 describe the M-HOT flow and how it maps circuits onto the Malibu architecture. Section 6.6 presents results of circuit synthesis and compares them to results from the M-CAD flow in the previous chapter. Section 6.7 details the technical differences between M-HOT and previous work. Section 6.8 summarizes the M-HOT flow and gives concluding remarks.

6.2 M-HOT Introduction

The Malibu height-oriented tool flow (M-HOT) is shown in Figure 4.1c. Compared to segregated flows like M-CAD, M-HOT is able to make better decisions because it has more accurate information about resource usage and the current schedule as it is created. M-HOT uses the same front-end synthesis as presented in Chapter 4. The remainder of the M-HOT flow is explained in the following sections. The overall flow of the M-HOT tool is as follows:

- Perform a minimal clustering on the input DFG.
- Levelize the DFG into an As-Late-As-Possible (ALAP) tree, resulting in the most time-critical operations at the highest levels of the tree.
- Starting from the top of the tree (the greatest height) and working down to height 0. Simulated annealing is done at each height to assign the nodes at that height to physical resources. Routing is also done at this time to all previously placed nodes.
- When each height is complete, locked down the nodes and anneal the next height until height 0 is done.

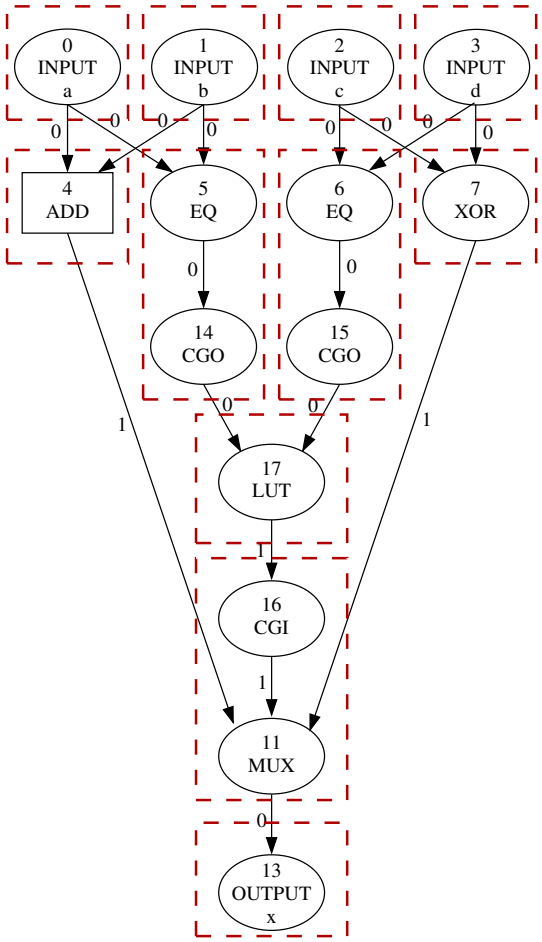


Figure 6.1: Example: M-HOT clustering for two CLBs. Each node is put in an individual cluster except the `CGI` and `CGO` nodes which must be kept with their respective sources and sinks.

- Invoke VPR to route the fine-grained signals and verify the timing estimates.

This overall flow is roughly based on a CGRA mapping tool that uses modulo graph embedding [62]. M-HOT adds a clustering step, support for fine-grained resources, scalability to larger architecture array sizes, and a dynamic schedule length to reduce compile time. The differences are detailed in Section 6.7.

6.3 M-HOT Cluster

The goal of M-HOT is to take advantage of simultaneous placement, routing, and scheduling. The benefits of clustering first in traditional FPGA CAD tools, and in M-CAD, are problem size reduction for placement, and reducing the interconnect demands for routing. However, M-CAD is very fast and there is no interconnect congestion on the Malibu architecture. Since clustering restricts the freedom of the placer, M-HOT only does a minimal clustering: `LOAD` and `STORE` operations for the same memory are clustered together, and `CGO` and `CGI` nodes are clustered with their respective sources or sinks as required by the CG/FG interface (see Section 5.2). By keeping each instruction unclustered, M-HOT has a greater ability to place instructions optimally.

This placement freedom also causes M-HOT to favour a "compute-and-move" model where computation is done while values are being routed through the current CLB *en route* to the next CLB location. The values are routed though the ALU instead of through the crossbar, so the required computation can be done while being routed.

Returning to the example in Figure 4.4, the M-HOT clustering is shown in Figure 6.1.

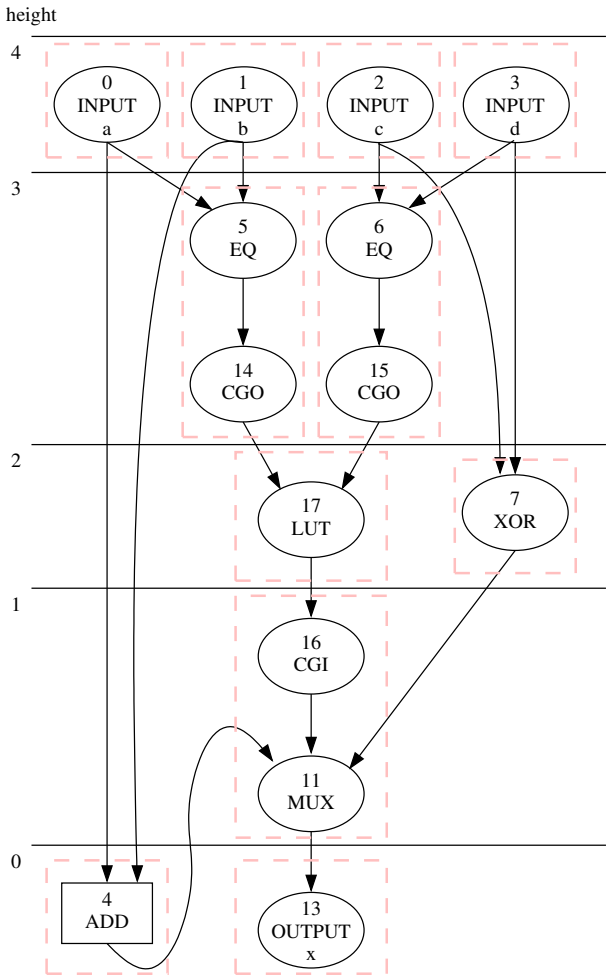
6.4 M-HOT Schedule

The top-level code of the M-HOT scheduler is shown in Figure 6.2a. The algorithm accepts a DFG and computes an ALAP height for each node. Figure 6.2b shows the ALAP result computed from Figure 6.1. At the bottom of the tree (height 0) are the outputs and all the *registered* nodes. The scheduler processes each height, starting at the largest (which are always inputs), because those nodes have the longest path to the outputs at the bottom, so they are the most critical.

The ALAP tree calculation ignores `CGI` and `CGO` nodes which will only appear in clusters with some other node by virtue of the clustering approach. These nodes are not real operations,

```
1  function M_HOT_main()
2  {
3    max_h = compute_ALAP_heights()
4    for( h = max_h; h >= 0; h-- ) {
5       nodes = all nodes at height h
6       affinity = compute_affinity(nodes)
7       anneal( affinity, nodes )
8    }
9    finalize_modulo_routes()
10 }
```

(a)



(b)

Figure 6.2: Example: M-HOT top-level code and ALAP tree. (a) M-HOT top-level code, and (b) the ALAP tree. All *registered* nodes are at height 0.

merely placeholders for resource usage. Another way to view it is the ALAP tree is constructed using clusters, rather than the nodes themselves.

At each height, a low-temperature anneal is done to assign coarse-grained operations to CGs and fine-grained operations to LUTs in the FGs. The coarse-grained and fine-grained nodes are annealed together, so either can be a move candidate. After choosing a move, the annealer invokes the router to determine the earliest timeslot for the current operation in the chosen destination CLB. To do this, it computes a route from every source to the current node

in the destination CLB and timeslot. Routing delays are computed the same as the M-CAD approach. When annealing at a height is complete, all nodes at that height are locked so they cannot be moved, and the next height is annealed.

If a node is *registered*, it requires special handling. A *registered* node is always the terminus of a path, so it is always at height 0 in the ALAP tree. The sinks of a *registered* node are not at height 0 unless the register is also an output, or the sink is another register. Therefore, when a *registered* node is encountered, not only must the routes from all sources to the node be computed, but the routes from the node to all sinks must be also computed. The latter are called "modulo routes", as they wrap around the schedule back to some earlier height.

Since the M-HOT approach lengthens the *SL* as required, instead of targeting a fixed *SL*, these modulo routes may become broken as additional timeslots are added. The last step of M-HOT in Figure 6.2a completes and reconnects any dangling routes.

At the heart of the annealer is a cost function. The annealing schedule is from VPR but with a lower fixed initial temperature (hard-coded to 0.01). The lower temperature causes the annealer to finish faster. Also, since nodes from previous heights are placed and locked in good positions, they serve as anchors to help the placement converge faster.

The cost function for M-HOT is the sum of five costs. Each cost is calculated for all nodes at the current height. A cost may look at nodes in other heights for placement information or to compute the affinity, but the actual calculation is only done for the current height. The cost function is:

$$cost = producer\_cost + affinity\_cost + parallel\_cost + register\_cost + penalty \tag{6.1}$$

The components of this cost function are described in the subsections below. The *producer_cost* keeps nodes at the current height near nodes of previous heights. The *affinity_cost* keeps nodes at the current height close if they share descendants which have not yet

been placed. These two costs have been adapted from [62]. The *parallel_cost* encourages nodes at each height to spread out for improved parallelism. The *register_cost* keeps nodes at the current height close if they share registers; it is similar to the *affinity_cost* but can use actual placement information. Finally, the *penalty* cost discourages illegal placements.

The *producer_cost*, *affinity_cost*, and *register_cost* are sums of weighted Manhattan distances, so they are just added together in the final cost. This is similar to how the cost function in [62] was constructed. The *parallel_cost* is small compared to the others, so it is also just added to the total cost; it only adds 1 to the total cost for each pair of nodes in the same height in the same CLB. It forces nodes to spread out, but not at the expense of additional scheduling delays. The *penalty* cost is large, adding $1,000$ for every resource violation, overpowering all the other costs. If scheduling finishes with a non-zero *penalty* cost, the solution is invalid.

6.4.1 Producer Cost

The *producer_cost* is the delay cost of all producer nodes at previous heights to all the consumer nodes in the current height being placed. Minimizing this cost means that the consumer nodes are placed near their respective producers, reducing the routing resource usage and reducing the delay for communicating values.

The calculation of *producer_cost* is shown in Figure 6.3. The function is passed a list of all the nodes at the current height, and it returns a single value, the *producer_cost*. For the CG operations, it uses actual routing information to compute the real cost (this includes any hold slots, although usually the Manhattan distance is the actual cost, but that cannot be guaranteed). For the FG operations it uses Equation 5.1 to estimate the delay through the fine-grained resources.[1] The total cost for either a CG or FG node is the sum of the timeslot differences from each source to the current node.

[1] At the end of scheduling, the VPR router is invoked to route all the fine-grained signals and to verify that Equation 5.1 has not under-estimated the delay on any signal. See Section 6.5.

```
function producer_cost( nodes_in_current_height )
{
   cost = 0
   foreach( nodes_in_current_height as node ) {
      foreach( node.sources as src ) {
         if( src is a register ) {
            foreach( src.sinks as sink ) {
               cost += mh( node.clb, sink.clb )
            }
         } else {
            /* The routing cost is the timeslot difference for both CG and FG resources.
             * Recall the definition of "delay" in Malibu is the number of timeslots
             * required to route a signal. Routing (and timeslots) for each node are
             * calculated before the cost. */
            cost += src.timeslot - node.timeslot
         }
      }
   }
   return cost
}
```

Figure 6.3: Calculation of *producer_cost*.

The idea of the *producer_cost* is the same as in [62], except there are two important modifications in this work. First, the FG computation has been added to support Malibu's FG resources, as described in the previous paragraph. Second, when the CG operation has a register as a source, the source register will not be placed until the last height (height 0) is processed. Because of this, the source register will not have a timeslot, so the *producer_cost* cannot be determined as described. Instead, the producer cost is set to the total Manhattan distance from the candidate CLB location of the current node to each of the already-placed children of the register. This helps keep the current node near the children of that source register thereby reducing lengthy fanout delays when the register is finally placed (it can be easily placed near all sinks of the register). In retrospect, since this is summed across all sinks of the source register, this may inflate the *producer_cost* when high-fanout registers are encountered; using the average sink distance or bounding box perimeter of the register net may be better in that case.

6.4.2 Affinity Cost

The *affinity_cost* keeps nodes with common descendants close together to reduce future routing costs. It is computed among all nodes at each height ($nodes(height)$), and is the product of the Manhattan distance (mh) between each pair of nodes at the height and the affinity between them:

$$affinity\_cost = \sum_{i,j \in nodes(height)} affinity(i,j) \times mh(i.clb, j.clb) \tag{6.2}$$

The $affinity(i,j)$ counts the common sinks between the nodes i and j in future graph levels (lower heights) which are not yet placed. It looks up to 3 levels deep, with a common sink being counted four times at level 1, twice at level 2, and once at level 3. This exponential weighting will heavily penalize two nodes with many common sinks in the very next level of the graph if they are placed too far apart. It is calculated as follows:

$$affinity(i,j) = \sum_{d=1..max\_d} 2^{max\_d-d} \times sinks(i,j,d) \tag{6.3}$$

Where, max_d is the maximum depth to search for common sinks ($max\_d = 3$ in this thesis) and $sinks(i,j,d)$ is number of common sinks of i and j at distance d. For M-HOT, there are three slight algorithmic exceptions. First, the M-HOT affinity calculation ignores `CGI` and `CGO` nodes because they are not real operations. Second, for registered nodes, the affinity calculation only includes nodes in the fanout that have not yet been placed (it is more likely that they have already been placed, and the *register_cost* handles that case). And third, nodes are only counted once in the event of a feedback path.

For example, at height 4 in Figure 6.2b, nodes 0 and 1 share node 5 one depth away, node 14 is ignored, node 17 two depths away, node 16 is ignored, and node 11 three depths

away. Hence, the affinity between nodes 0 and 1 is:

$$\mathit{affinity}(0,1) = 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 7 \tag{6.4}$$

Notice that node 4 is not included even though it is a direct descendent of nodes 0 and 1. It is four depths away in the ALAP tree, so there is extra time to route values to node 4, whereas for node 5 the values are needed immediately or the critical path is lengthened.

Similarly, the affinities for the other pairs of nodes at height 4 can be computed as:

$$\mathit{affinity}(2,3) = 1 \times 2^2 + 2 \times 2^1 + 1 \times 2^0 = 9 \tag{6.5}$$

$$\mathit{affinity}(0,2) = 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 3 \tag{6.6}$$

$$\mathit{affinity}(0,3) = \mathit{affinity}(1,2) = \mathit{affinity}(1,3) = \mathit{affinity}(0,2) \tag{6.7}$$

Finally, the affinity for a height is stored in a symmetric matrix:

$$\mathit{affinity} = \begin{bmatrix} 0 & 7 & 3 & 3 \\ 7 & 0 & 3 & 3 \\ 3 & 3 & 0 & 9 \\ 3 & 3 & 9 & 0 \end{bmatrix} \tag{6.8}$$

This matrix is computed before annealing each height, and it remains constant for the duration of the processing on that height.

6.4.3 Parallel Cost

This cost spreads out nodes at the same height so they are placed in *different* CLBs. It is a small cost relative to the Manhattan sums used in the other cost terms, but is especially useful

for breaking ties when a node could be placed in several locations. Without this cost, nodes tend to bunch up and require slightly more timeslots to schedule. The implementation in [62] uses skewed scheduling and prefers to place nodes "on the left" of the array, since this is where I/O is located. For Malibu, it is better to force the nodes to spread out because I/O is around the periphery.

The cost is computed only for nodes at the current height ($nodes(height)$). It is the number of node-pairs placed in the same CLB at the current height. This addition to the cost function reduces the average output schedule length by 3.3% (compared to not using this cost):

$$parallel\_cost = \sum_{i,j \in nodes(height)} \begin{cases} 1 & nodes[i].clb = nodes[j].clb, i \neq j \\ 0 & \text{otherwise} \end{cases} \tag{6.9}$$

6.4.4 Register Cost

Like the *affinity_cost*, the register cost is a forward-looking cost that keeps a node close to its ultimate destination registers. Although those registers have not yet been placed, the sinks of those registers have likely already been placed. While the backward-looking *producer_cost* keeps these sinks close together, the *register_cost* ensures the nodes in the fanin cone of each register will be placed on a straight line path to the sinks of the register, and thus close to the register too when it is finally placed. Park avoids problem in [62] by pre-placing all inputs, outputs, registers, and memories. If the circuit has a specific pin mapping this would be a reasonable approach for M-HOT too, however it is not always desirable to impose this restriction.

The *register_cost* is computed as shown in Figure 6.4. It is the sum of the Manhattan distances between each node in the current height and the sinks of any registers that have been placed in the node's respective fanout cone. While the registers themselves have not yet been placed, the sinks of those registers have been, so the Manhattan distance can be computed.

```
1  function register_cost_node( root_node, node, depth, maxdepth)
2  {
3     if( depth == maxdepth )
4        return 0
5
6     cost = 0
7     foreach( node.sinks as sink ) {
8        if( sink is a register ) {
9           foreach( sink.sinks as regsink ) {
10             cost += mh( root_node.clb, regsink.clb )
11          }
12       } else {
13          cost += register_cost_node( root_node, sink, depth + 1, max_depth )
14       }
15    }
16    return cost
17 }
18
19 function register_cost( nodes_in_current_height )
20 {
21    cost = 0
22    foreach( nodes_in_current_height as node ) {
23       cost += register_cost_node( node, node, 0, 3 );
24    }
25    return cost
26 }
```

Figure 6.4: Calculation of *register_cost*.

Using this cost reduces the average schedule length for circuits with registers by 4%. The value of *max_depth* used in this thesis is 3.

6.4.5 Penalty Cost

The *penalty* cost discourages invalid/illegal placements. It is computed the same way as the M-CAD approach in Section 5.3. The initial placement at any height may be invalid, resulting in many penalty costs. At the end of each annealing phase, however, the penalty costs are zero (or if they are not, the tools report an error that the solution violates resource constraints).

6.5 M-HOT Route

Placement, routing, and scheduling are done together with the M-HOT tool, so there is no opportunity to incorporate the actual FG routing delays into the flow without invoking VPR in the inner loop of the annealer, which would take far too long to run.

Instead, M-HOT initially estimates the FG routing delay using Equation 5.1 which is based on Manhattan distances and pre-characterized Elmore delays. At the end of scheduling, M-HOT calls VPR to compute the actual delay for each FG route. In all of our benchmark trials, we have found that even when the routing delay is underestimated (e.g., a route has to go around some CLBs to avoid congestion), it is close enough to the actual delay that the value will still arrive before it is needed. M-HOT flags any routes with timing violations and reports an error.

For coarse-grained routing, as in the M-CAD approach, a horizontal-then-vertical routing strategy is used. By making use of both time and space, it is impossible for a route to not (eventually) arrive at the destination, except in one case. It is possible that a producer does not have an available write port or write resource to start a route after it has been placed. This can happen when a producer node at one height is placed (and locked down), and then a route from a subsequent height goes through the CLB at the same timeslot as a previously placed producer and claims the routing resources. When a sink of the producer is finally placed, the router will be unable to route the producer to the sink, and will fail due to unavailable resources. When this happens, the producer cannot be moved (it is at a previous height and already locked down), nor can routing resources be re-claimed because they are used to route values between two other nodes which have also been locked down. The tendency of M-HOT to closely pack nodes causes this situation to occur frequently. This situation is the reason the link from R to the crossbar was added in Figure 3.1. When this situation occurs, the value is written to R instead, guaranteeing that the route can always be started. From R, the route proceeds normally when resources become available.

This problem never occurs in the M-CAD flow because of the timeslot-oriented scheduling approach.

6.6 Experimental Results

To evaluate M-HOT, it is compared to M-CAD, and to the two baseline synthesis flows, VPR and QuartusII. As was done for the M-CAD experimentation in Section 5.6, the maximum frequency, maximum density, and compile time are used to evaluate the tool. Results are presented for the maximum performance and maximum density on the architecture in Table 3.5, with the benchmark circuits divided into the same three categories (*CG-only*, *Good*, and *Impaired* as defined in Section 3.3). As before, the *CG-only* and *Good* represent the types of circuits which would primarily be used on the coarse-grained Malibu architecture, so we are most interested in those results. For the *Impaired* benchmarks, it is only important that the tools successfully synthesize the circuit; these benchmarks are included to demonstrate that this is being done.

All the results in this section are generated the same way as in Section 5.6, except that the M-HOT tool was used as a drop-in replacement for M-CAD. Therefore, the explanation and setup for each test is not repeated in this section. For those explanations, refer to the appropriate subsection in Section 5.6.

The subsections in this section mirror those in Section 5.6. Section 6.6.1 examines the quality of synthesis measured by the maximum frequency (F_{max}). Section 6.6.2 extends this beyond the benchmark set by investigating an upper bound on the frequency. Section 6.6.3 looks at the area/density required to synthesize each benchmark. The compile time for the benchmarks is in Section 6.6.4, and the compile time for very large circuits is in Section 6.6.5. Section 6.6.6 looks at a breakdown of the longest path for the maximum performance results. And finally, Section 6.6.7 looks at the area versus performance tradeoff.

6.6.1 Frequency (F_{max})

Table 6.1 shows the maximum frequency results for M-HOT. The VPR comparison is summarized in Table 6.2. For the *CG-only* benchmarks, M-HOT is able to achieve a 40% improve-

Table 6.1: M-HOT maximum frequency and comparison to QuartusII. Entries with a "–" are for coarse-grained benchmarks and the same as the $W_f = 0$ value. The "vs.QII" columns are the frequency speedup (M-HOT MHz / QuartusII MHz).

				M-HOT Maximum Performance						**M-HOT Maximum Density**					
		QuartusII	**VPR**	$W_f = 0$		$W_f = 1$		$W_f = 4$		$W_f = 0$		$W_f = 1$		$W_f = 4$	
	Circuit	**MHz**	**MHz**	**MHz**	**vs.QII**	**MHz**	**vs.QII**	**MHz**	**vs.QII**	**MHz**	**vs.QII**	**MHz**	**vs.QII**	**MHz**	**vs.QII**
CG-only	fft16	119.2	98.4	34.5	0.289	–	–	–	–	15.2	0.127	–	–	–	–
	me	201.7	66.0	25.6	0.127	–	–	–	–	19.2	0.095	–	–	–	–
	chem	11.3	27.1	50.0	4.443	–	–	–	–	24.4	2.167	–	–	–	–
	fft8	159.7	117.2	55.6	0.348	–	–	–	–	22.7	0.142	–	–	–	–
	honda	17.1	63.8	55.6	3.242	–	–	–	–	52.6	3.071	–	–	–	–
	mcm	24.9	93.7	83.3	3.342	–	–	–	–	83.3	3.342	–	–	–	–
	wang	19.3	79.7	111.1	5.760	–	–	–	–	111.1	5.760	–	–	–	–
	pr	24.0	87.6	100.0	4.163	–	–	–	–	100.0	4.163	–	–	–	–
	Geo. Mean (CG-only)				**1.400x**		**1.400x**		**1.400x**		**0.990x**		**0.990x**		**0.990x**
Good	ac97_ctrl	294.90	278.35	17.5	0.059	43.5	0.147	52.6	0.178	11.2	0.038	40.0	0.136	52.6	0.178
	aes_core	181.55	177.80	29.4	0.162	30.3	0.167	29.4	0.162	5.7	0.031	11.0	0.061	23.8	0.131
	dir	86.32	59.61	33.3	0.386	34.5	0.399	37.0	0.429	16.4	0.190	27.8	0.322	37.0	0.429
	spi	118.86	139.89	27.0	0.227	52.6	0.443	52.6	0.443	20.8	0.175	52.6	0.443	52.6	0.443
	pci_master	232.18	248.36	29.4	0.127	40.0	0.172	125.0	0.538	14.9	0.064	37.0	0.160	125.0	0.538
	Geo. Mean (Good)				**0.161x**		**0.237x**		**0.312x**		**0.076x**		**0.180x**		**0.299x**
Geo. Mean (CG-only and Good)					**0.609x**		**0.707x**		**0.786x**		**0.369x**		**0.513x**		**0.625x**
Impaired	ethernet	162.42	102.06	9.5	0.059	12.7	0.079	32.3	0.199	8.9	0.055	12.3	0.076	27.0	0.166
	wb_conmax	135.10	72.35	13.2	0.097	22.7	0.168	20.0	0.148	7.8	0.057	22.7	0.168	12.2	0.090
	dma	127.75	130.97	12.0	0.094	15.4	0.120	17.9	0.140	8.3	0.065	14.1	0.110	12.7	0.099
	tv80	96.80	105.26	4.1	0.043	5.7	0.059	33.3	0.344	3.7	0.038	5.0	0.052	31.2	0.323
	jpeg_enc	218.39	162.81	23.8	0.109	28.6	0.131	27.8	0.127	11.2	0.051	20.4	0.093	27.8	0.127
	systemcaes	120.71	154.50	31.2	0.259	40.0	0.331	50.0	0.414	5.3	0.044	23.8	0.197	50.0	0.414
	des	299.76	175.49	6.5	0.022	8.6	0.029	10.1	0.034	6.3	0.021	8.6	0.029	10.1	0.034
	systemcdes	169.58	219.71	25.6	0.151	27.8	0.164	30.3	0.179	13.7	0.081	25.6	0.151	26.3	0.155
	Geo. Mean (Impaired)				**0.082x**		**0.108x**		**0.161x**		**0.048x**		**0.094x**		**0.138x**
Geo. Mean (All)					**0.284x**		**0.346x**		**0.430x**		**0.170x**		**0.268x**		**0.352x**

Table 6.2: M-HOT F_{max} speedup compared to VPR.

	M-HOT vs. VPR					
	Maximum Performance			Maximum Density		
	$W_f = 0$	$W_f = 1$	$W_f = 4$	$W_f = 0$	$W_f = 1$	$W_f = 4$
Geo. Mean (CG-only)	0.787x	0.787x	0.787x	0.556x	0.556x	0.556x
Geo. Mean (Good)	0.168x	0.248x	0.326x	0.079x	0.188x	0.312x
Geo. Mean (CG-only and Good)	0.406x	0.479x	0.539x	0.242x	0.349x	0.434x
Geo. Mean (Impaired)	0.097x	0.127x	0.189x	0.057x	0.110x	0.163x
Geo. Mean (All)	0.235x	0.287x	0.360x	0.141x	0.224x	0.295x

ment in clock frequency over the QuartusII/StratixIII result at maximum performance. This is slightly less than the M-CAD value in Table 5.1, but comparable. At maximum density, M-HOT almost matches the frequency of QuartusII/StratixIII at 0.99x, which is a definite improvement over the 0.814x result from M-CAD. As will be shown later in Section 6.6.3, the density at this maximum density point is almost 5x that of a StratixIII.

For the *Good* benchmarks, QuartusII achieves a higher frequency than M-HOT. As with M-CAD, the addition of fine-grained resources to the architecture ($W_f > 0$) doubles the performance of these circuits for the maximum performance results (from 0.161x to 0.312x), and triples it for the maximum density results (from 0.076x to 0.299x). As will be shown later in Section 6.6.3, the density is also improved for these benchmarks compared to the M-CAD flow by 2x at $W_f = 0$ and by 1.15x for other values of W_f. Overall, M-HOT is producing better quality results.

For the *Impaired* benchmarks, M-HOT does successfully synthesize each benchmark, but the performance is poor, as expected. On average M-HOT achieves $1/10^{th}$ of the StratixIII performance at maximum performance, and down to $1/20^{th}$ at maximum density. With these circuits it is only important that M-HOT achieves a successful synthesis, which it is doing.

The *CG-only* and *Good* benchmarks represent the types of circuits that would primarily be

used on the coarse-grained Malibu architecture, so we are most interested in those results. The geometric mean of the combined *CG-only* and *Good* speedups in Table 6.1 shows that increasing the amount of fine-grained resources in Malibu ($W_f = 0$ to $W_f = 4$) improves the frequency speedup for both maximum performance (0.607x to 0.786x) and maximum density (0.369x to 0.625x). However, as will be shown later in Section 6.6.3, the density at maximum density worsens as W_f increases from 1 to 4 due to the circuit needing more fine-grained resources. Therefore, we have chosen $W_f = 1$ as a balance point between improving the frequency and losing density. At this point (*CG-only* and *Good* benchmarks, $W_f = 1$), the M-HOT frequency at maximum performance is 0.707x that of a StratixIII, and at maximum density it is 0.513x (about half). This is an excellent result, particularly at maximum density, because M-HOT achieves half the speed of a StratixIII with 2.5x the density (see Table 6.6), and can compile these benchmarks 26.1x faster than QuartusII on average.

Table 6.3 summarizes the clock frequency advantage of M-HOT over M-CAD. The results are computed by dividing the geometric mean of M-HOT speedup compared to QuartusII by the geometric mean of the M-CAD speedup.[2] The QuartusII result is cancelled in the division, leaving just the M-HOT speedup over M-CAD:

$$speedup = \frac{\text{M-HOT speedup vs. Quartus}}{\text{M-CAD speedup vs. Quartus}} = \frac{\frac{\text{M-HOT MHz}}{\text{Quartus MHz}}}{\frac{\text{M-CAD MHz}}{\text{Quartus MHz}}} = \frac{\text{M-HOT MHz}}{\text{M-CAD MHz}} \tag{6.10}$$

On average, M-HOT outperforms M-CAD for all the frequency results, except for the *CG-only* circuits at maximum performance. For these results, the M-CAD and M-HOT results are about the same, but M-HOT achieves over twice the density compared to M-CAD for the same performance (see Table 6.7). For the other results, the M-HOT frequency is on average up to 29% better than M-CAD (for just the *CG-only* and *Good* benchmarks, excluding the *Impaired*

[2]Since the geometric mean is distributive over division (when there are the same number of items in the set), the result is the same as if the geometric mean was computed after the division was done individually for each benchmark.

Table 6.3: M-HOT F_{max} speedup compared to M-CAD.

	M-HOT (Table 6.1) / **M-CAD** (Table 5.1)					
	Maximum Performance			**Maximum Density**		
	$W_f = 0$	$W_f = 1$	$W_f = 4$	$W_f = 0$	$W_f = 1$	$W_f = 4$
Geo. Mean (CG-only)	0.969	0.969	0.969	1.216	1.216	1.216
Geo. Mean (Good)	1.150	1.301	1.357	1.165	1.166	1.409
Geo. Mean (CG-only and Good)	1.035	1.085	1.103	1.196	1.196	1.287
Geo. Mean (Impaired)	1.562	1.682	1.645	1.658	2.067	2.511
Geo. Mean (All)	1.210	1.282	1.284	1.355	1.473	1.650

benchmarks). Interestingly, the *Impaired* benchmarks show the most gains over M-CAD, but these are not the types of circuits the Malibu architecture is designed to handle.

Although the M-HOT F_{max} is usually higher than the M-CAD F_{max}, there are three circuits where it gives lower speeds (*fft16*, *fft8*, and *me*). The reason for this is that the maximum depth of these benchmarks is small (4, 4, and 10 respectively) and they have a large number of nodes, thereby having a large number of nodes per height in the ALAP tree. Since M-HOT does no clustering, it is essentially annealing one quarter of the circuit at each height for *fft16* and *fft8*. Furthermore, M-HOT only does a low-temperature anneal to be fast, but it finishes before a good solution is found.

It is possible to recover the lost performance in *fft16*, *fft8*, and *me* by changing the annealer to use the M-CAD annealing schedule at each height. Doing this improves the results to be 1% better than the M-CAD result, but it lengthens the compile time for each benchmark by almost 10x. For example, the *fft16* benchmark increases from 76.5 seconds to 690 seconds (from Table 5.9 the runtime for *fft16* with M-CAD is 5.5 seconds). To avoid adding a special case to the code, and because we consider compile time a high-priority, this was not done for the results reported in this thesis. All the results use the same M-HOT annealing schedule. Future work should look for ways to speed this up, perhaps by reducing the amount of effort

required by the annealer through intelligent clustering.

6.6.2 Frequency Upper Bound

To evaluate how well M-HOT might perform, the upper bound analysis that was done with M-CAD in Section 5.6.2 is repeated for M-HOT using the same four assumptions. Since the upper bound is computed using the output of front-end synthesis, it is the same for M-HOT or M-CAD. The difference is that M-HOT achieves a higher quality result, so is closer to this bound than M-CAD.

Table 6.4 compares the frequency upper bound with the F_{max} from Table 6.1 for each benchmark. The possible performance increase in F_{max} ranges from about 7x (*fft16*) to nothing (*honda*, *wang*, *pr*, *pci_master*, *ethernet*, *des*, and *systemcdes*). The geometric mean of the ratio across the *CG-only* and *Good* circuits for $W_f = 0, 1,$ or 4 is at least 0.64, meaning that there is, on average (geometric mean), a potential 1.6x performance improvement to be found in M-HOT by improving the back-end synthesis algorithms. In addition, there may be other opportunities to improve the results through changes to the front-end synthesis that are not captured in this data.

Since M-HOT does placement, routing, and scheduling at the same time, it is impossible to decompose the difference from the actual result to the upper bound into the individual contributions. However, since there are unavoidable architectural constraints on the schedule, it is quite possible that this upper bound is unachievable. To achieve a more realistic evaluation, we next compare to M-HOT using a high-effort anneal.

The high-effort anneal uses M-HOT with an inner-loop multiplier of 10,000, instead of 10 at each height. The starting temperature was set to 25, which is high for a low-temperature anneal. The annealing range limit which constraints the locality annealing swaps was disabled so any node in any CLB can be considered.

Table 6.5 compares this high-effort anneal to the regular M-HOT results and the upper

Table 6.4: M-HOT frequency upper bound and actual F_{max}. Entries with a "–" are the same as the $W_f = 0$ value. The ratio is F_{max} / Bound; larger ratios are better.

		$W_f = 0$			$W_f = 1$			$W_f = 4$		
	Circuit	**Bound (MHz)**	**F_{max} (MHz)**	**Ratio**	**Bound (MHz)**	**F_{max} (MHz)**	**Ratio**	**Bound (MHz)**	**F_{max} (MHz)**	**Ratio**
CG-only	fft16	250.0	34.5	0.14	–	–	–	–	–	–
	me	55.6	25.6	0.46	–	–	–	–	–	–
	chem	58.8	50.0	0.85	–	–	–	–	–	–
	fft8	250.0	55.6	0.22	–	–	–	–	–	–
	honda	55.6	55.6	1.00	–	–	–	–	–	–
	mcm	100.0	83.3	0.83	–	–	–	–	–	–
	wang	111.1	111.1	1.00	–	–	–	–	–	–
	pr	100.0	100.0	1.00	–	–	–	–	–	–
	Geo. Mean (CG-only)			**0.56**			**0.56**			**0.56**
Good	ac97_ctrl	21.7	17.5	0.81	66.7	43.5	0.65	66.7	52.6	0.79
	aes_core	55.6	29.4	0.53	55.6	30.3	0.55	55.6	29.4	0.53
	dir	40.0	33.3	0.83	40.0	34.5	0.86	40.0	37.0	0.93
	spi	33.3	27.0	0.81	55.6	52.6	0.95	55.6	52.6	0.95
	pci_master	29.4	29.4	1.00	40.0	40.0	1.00	125.0	125.0	1.00
	Geo. Mean (Good)			**0.78**			**0.78**			**0.82**
Geo. Mean (CG-only and Good)				**0.65**			**0.64**			**0.65**
Impaired	ethernet	9.9	9.5	0.96	12.8	12.7	0.99	32.3	32.3	1.00
	wb_conmax	40.0	13.2	0.33	40.0	22.7	0.57	58.8	20.0	0.34
	dma	15.6	12.0	0.77	21.3	15.4	0.72	23.8	17.9	0.75
	tv80	18.9	4.1	0.22	18.9	5.7	0.30	66.0	33.3	0.51
	jpeg_enc	32.3	23.8	0.74	32.3	28.6	0.89	32.3	27.8	0.86
	systemcaes	71.4	31.2	0.44	76.9	40.0	0.52	90.9	50.0	0.55
	des	6.7	6.5	0.96	8.8	8.6	0.98	10.2	10.1	0.99
	systemcdes	28.6	25.6	0.90	28.6	27.8	0.97	30.3	30.3	1.00
	Geo. Mean (Impaired)			**0.59**			**0.69**			**0.70**
Geo. Mean (All)				**0.62**			**0.66**			**0.67**

bound analysis. Comparing the "Ratio" column to the "Improved Ratio" column, the high-effort placer is able to achieve 70% of the upper bound across all circuits, whereas the M-HOT tool with the default options achieves 62%. If the improved F_{max} (high-effort result) is taken as a realistic and achievable upper bound and is compared to the actual F_{max}, then the M-HOT tool result is 88% of the realistic F_{max} bound. Compared to the previous chapter where M-CAD achieved 91% of a realistic upper bound, the same observation can be made: there is

Table 6.5: M-HOT high-effort schedule length comparison for $W_f = 0$.

		$W_f = 0$ from Table 6.4			High-Effort Placement		
	Circuit	**Bound (MHz)**	F_{max} **(MHz)**	**Ratio**	**Improved** F_{max} **(MHz)**	**Improved Ratio**	F_{max}**/Improved** F_{max} **Ratio**
CG-only	fft16	250.0	34.5	0.14	34.5	0.14	1.00
	me	55.6	25.6	0.46	27.0	0.49	0.95
	chem	58.8	50.0	0.85	50.0	0.85	1.00
	fft8	250.0	55.6	0.22	55.6	0.22	1.00
	honda	55.6	55.6	1.00	55.6	1.00	1.00
	mcm	100.0	83.3	0.83	83.3	0.83	1.00
	wang	111.1	111.1	1.00	111.1	1.00	1.00
	pr	100.0	100.0	1.00	100.0	1.00	1.00
	Geo. Mean (CG-only)			**0.56**		**0.57**	**0.99**
Good	ac97_ctrl	21.7	17.5	0.81	25.6	1.18	0.68
	aes_core	55.6	29.4	0.53	30.3	0.55	0.97
	dir	40.0	33.3	0.83	34.5	0.86	0.97
	spi	33.3	27.0	0.81	38.5	1.15	0.70
	pci_master	29.4	29.4	1.00	34.5	1.17	0.85
	Geo. Mean (Good)			**0.78**		**0.94**	**0.85**
Geo. Mean (CG-only and Good)				**0.64**		**0.69**	**0.93**
Impaired	ethernet	9.9	9.5	0.96	14.7	1.49	0.65
	wb_conmax	40.0	13.2	0.33	14.1	0.35	0.94
	dma	15.6	12.0	0.77	12.0	0.77	1.00
	tv80	18.9	4.1	0.22	10.8	0.57	0.38
	jpeg_enc	32.3	23.8	0.74	26.3	0.82	0.90
	systemcaes	71.4	31.2	0.44	31.2	0.44	1.00
	des	6.7	6.5	0.96	6.5	0.96	1.00
	systemcdes	28.6	25.6	0.90	25.6	0.90	1.00
	Geo. Mean (Impaired)			**0.59**		**0.72**	**0.82**
Geo. Mean (All)				**0.62**		**0.70**	**0.88**

little room for circuit performance improvement from simple changes to back-end synthesis. Instead, future work should focus on front-end synthesis.

Comparing the M-HOT high-effort results with the M-CAD high-effort results in Section 5.6.2, M-HOT is able get closer to the computed upper bound than M-CAD (70% vs. 57%). This is expected and is due, at least in part, to the M-CAD information sharing problem. Since M-CAD placement lacks scheduling information, the M-CAD heuristics sometimes

cannot distinguish between two placements where one has a better (smaller) post-scheduling schedule length. M-HOT does not have this problem, so it is expected to perform better.

6.6.3 Area and Density

In this section, the area and density of each benchmark is examined at maximum performance and maximum density. The maximum density occurs at the minimum area which is the smallest number of CLBs required to synthesize the circuit without violating any architecture parameter values. These parameter values are the ones from Table 3.5.

Comparison to QuartusII/StratixIII

In Section 3.7 the area of a Malibu CLB was computed to be 97,050.5 μm^2 and the area of a StratixIII ALM was estimated to be 2,674μm^2. This means that a Malibu CLB is roughly equivalent in area to 36.3 ALMs. Without any FG components, the Malibu CLB is slightly smaller at 29.4 ALMs. The equivalent ALMs (eALMs) for Malibu is computed by multiplying the number CLBs by 36.3 when $W_f \geq 1$, and by 29.4 when $W_f = 0$.

Table 6.6 compares the circuit area for the benchmarks implemented on Malibu with M-HOT and on a StratixIII FPGA using QuartusII. At maximum performance with $W_f = 0$, Malibu achieves twice the density of an FPGA for the *CG-only* circuits, and about half the density for the *Good* circuits. When the fine-grained resources are enabled ($W_f \geq 1$) the density becomes worse for the *CG-only* circuits because they cannot make use of such resources, however the density is still better than an FPGA. For the *Good* benchmarks, the density initially gets worse for $W_f = 1$, but then improves when $W_f = 4$.

At maximum density, the M-HOT results are the same as the M-CAD results, because the minimum architecture array size is a function of the resource requirements of each benchmark circuits. The difference is that M-HOT can schedule the instructions on these resources more efficiently and achieve a higher F_{max} as shown in Table 6.3. For the *Good* circuits, although

Table 6.6: M-HOT area and density values compared to QuartusII/StratixIII. Entries with a "–" are same as the $W_f = 1$ value. The "Dens." columns are the area reduction compared to QuartusII (QuartusII ALMs / Malibu equivalent ALMs (eALMs)).

			M-HOT Maximum Performance						M-HOT Maximum Density					
		QuartusII	$W_f = 0$		$W_f = 1$		$W_f = 4$		$W_f = 0$		$W_f = 1$		$W_f = 4$	
	Circuit	ALMs	eALMs	Dens.	eALMs	Dens.	eALMs	Dens.	eALMs	Dens.	eALMs	Dens.	eALMs	Dens.
CG-only	fft16	6,412	4,234	1.515	3,630	1.766	–	–	470	13.640	581	11.042	–	–
	me	5,148	4,234	1.216	9,293	0.554	–	–	1,880	2.738	1,307	3.940	–	–
	chem	3,526	1,058	3.331	1,307	2.698	–	–	264	13.335	327	10.795	–	–
	fft8	2,075	1,058	1.961	3,630	0.572	–	–	264	7.847	327	6.352	–	–
	honda	1,216	470	2.585	327	3.722	–	–	264	4.599	327	3.723	–	–
	mcm	1,057	470	2.247	581	1.820	–	–	470	2.249	327	3.236	–	–
	wang	797	265	3.012	327	2.440	–	–	264	3.014	327	2.440	–	–
	pr	646	265	2.441	327	1.977	–	–	264	2.443	327	1.978	–	–
	Geo. Mean (CG-only)			**2.182x**		**1.628x**		**1.628x**		**4.833x**		**4.517x**		**4.517x**
Good	ac97_ctrl	1,254	1,882	0.666	3,630	0.345	3,629	0.346	1,058	1.186	2,323	0.540	3,629	0.346
	aes_core	1,154	2,940	0.393	3,630	0.318	2,323	0.497	264	4.364	581	1.987	1,307	0.883
	dir	1,150	1,058	1.087	1,307	0.880	581	1.980	264	4.349	581	1.980	581	1.980
	spi	488	470	1.037	581	0.840	581	0.840	264	1.846	581	0.840	581	0.840
	pci_master	137	1,058	0.129	581	0.236	581	0.236	264	0.518	327	0.419	581	0.236
	Geo. Mean (Good)			**0.520x**		**0.453x**		**0.583x**		**1.847x**		**0.944x**		**0.654x**
Geo. Mean (CG-only and Good)				**1.257x**		**0.996x**		**1.097x**		**3.339x**		**2.474x**		**2.148x**
Impaired	ethernet	6,868	11,752	0.584	3,629	1.892	7,114	0.965	7,521	0.913	2,940	2.336	5,226	1.314
	wb_conmax	5,349	7,526	0.711	14,518	0.368	11,759	0.455	2,938	1.821	14,518	0.368	9,291	0.576
	dma	1,714	7,526	0.228	14,518	0.118	28,455	0.060	2,938	0.583	9,291	0.184	20,905	0.082
	tv80	850	7,521	0.113	3,630	0.234	7,114	0.119	5,759	0.148	2,940	0.289	5,226	0.163
	jpeg_enc	791	67,693	0.012	20,905	0.038	5,226	0.151	47,009	0.017	14,518	0.054	5,226	0.151
	systemcaes	716	2,940	0.244	3,630	0.197	2,323	0.308	264	2.708	1,307	0.548	2,323	0.308
	des	298	4,231	0.070	2,940	0.101	5,226	0.057	3,555	0.084	2,940	0.101	5,226	0.057
	systemcdes	237	1,058	0.224	3,629	0.065	3,629	0.065	470	0.504	2,940	0.081	2,940	0.081
	Geo. Mean (Impaired)			**0.162x**		**0.177x**		**0.165x**		**0.359x**		**0.241x**		**0.196x**
Geo. Mean (All)				**0.576x**		**0.515x**		**0.533x**		**1.428x**		**1.018x**		**0.863x**

Table 6.7: M-HOT density improvement factor versus M-CAD.

	M-HOT (Table 6.6) / **M-CAD** (Table 5.6)					
	Maximum Performance			**Maximum Density**		
	$W_f = 0$	$W_f = 1$	$W_f = 4$	$W_f = 0$	$W_f = 1$	$W_f = 4$
Geo. Mean (CG-only)	2.775	2.184	2.184	1.000	1.000	1.000
Geo. Mean (Good)	2.081	1.157	1.176	1.000	1.000	1.000
Geo. Mean (CG-only and Good)	2.484	1.711	1.721	1.000	1.000	1.000
Geo. Mean (Impaired)	2.157	4.554	3.381	1.000	1.000	1.000
Geo. Mean (All)	2.354	2.484	2.226	1.000	1.000	1.000

F_{max} improves from $W_f = 1$ to $W_f = 4$, the density drops considerably. This is because too many LUTs are required, and this limits the achieved density.

Including fine-grained resources in the architecture, which increases the CLB size, is not helpful for maximum density. However, including fine-grained resources is helpful for performance, so again we recommend $W_f = 1$ as a mid-point between density and performance advantages.

Table 6.7 compares the M-HOT density to the M-CAD density, computed the same way as the frequency comparison shown in Table 6.3. Compared to M-CAD, the M-HOT approach is doing an excellent job scheduling the coarse-grained instructions giving nearly 2.5x better density than the M-CAD approach for $W_f = 0$, and is 1.7x better for other values of W_f with the *CG-only* and *Good* results. Just as with the frequency results, the *Impaired* benchmarks show the most gains over M-CAD. However, in this case, the gains come with longer compile times, as will be demonstrated in Section 6.6.4.

Comparison to VPR/iFAR

Table 6.8 shows the transistor area in units of millions of minimum-width transistor areas (T $\times 10^6$) for VPR, and for M-HOT using both the maximum performance and maximum density results. The VPR area is taken directly from the output of VPR (the tile area multiplied by

the number of CLBs used). The Malibu area is computed using the CLB area calculated in Section 3.7 and multiplied by the number of CLBs used.

The VPR area results are similar to the QuartusII/StratixIII results in Table 6.6. For the *CG-only* and *Good* circuits, at maximum density the density gets worse as W_f increases from 0 to 4, and at maximum performance it initially gets worse $W_f = 0$ to $W_f = 1$ but then improves to $W_f = 4$. This leads to the same recommendation as with the QuartusII density comparison. $W_f = 1$ is a balance point between performance advantages and density improvement.

6.6.4 Compile Time

Table 6.9 shows the compile time for M-HOT and the speedup relative to QuartusII. The speedup relative to VPR is summarized in Table 6.10. Comparing M-HOT to QuartusII, the *CG-only* and *Good* benchmarks combined show an 18x to 26.7x (geomean) improvement in compile time.

There are a few benchmarks that take a very long time to synthesize with M-HOT. Most of the *wb_conmax* time is spent in fine-grained routing with VPR, as was the case with M-CAD. The *ethernet* and *tv80* benchmarks are very slow at $W_f = 0$, but then improve as the fine-grained resources are used. These benchmarks have a number of fine-grained nodes at a low height, so the ALAP tree is bottom heavy and the annealer spends a lot of time on the last few heights. Annealing becomes faster by moving these nodes into the fine-grained network. The *dma* benchmark is similarly bottom-heavy, but it requires more time to anneal the last few heights at $W_f = 4$ compared to $W_f = 1$.

Table 6.11 compares the M-HOT compile time to the M-CAD compile time from Table 5.9. M-HOT is slower than M-CAD for word-oriented circuits, but faster for circuits with fine-grained components. This is because M-HOT does routing during placement (instead of after it), so it always actively routing signals to evaluate swaps in the simulated annealer. For fine-grained nets it estimates the route delay using Equation 5.1 and invokes VPR at the end of

Table 6.8: M-HOT area and density values compared to VPR/iFAR. Entries with a "–" are same as the $W_f = 1$ value. The "Dens." columns are the density compared to VPR (VPR T / M-HOT T).

			M-HOT Maximum Performance						M-HOT Maximum Density					
		VPR	$W_f = 0$		$W_f = 1$		$W_f = 4$		$W_f = 0$		$W_f = 1$		$W_f = 4$	
	Circuit	**T** $\times 10^6$	**T** $\times 10^6$	**Dens.**	**T** $\times 10^6$	**Dens.**	**T** $\times 10^6$	**Dens.**	**T** $\times 10^6$	**Dens.**	**T** $\times 10^6$	**Dens.**	**T** $\times 10^6$	**Dens.**
CG-only	fft16	48.12	22.63	2.127	19.41	2.479	–	–	2.51	19.155	3.11	15.498	–	–
	me	14.51	22.63	0.641	49.69	0.292	–	–	10.05	1.444	6.99	2.077	–	–
	chem	62.81	5.66	11.104	6.99	8.989	–	–	1.41	44.446	1.75	35.961	–	–
	fft8	11.72	5.66	2.071	19.41	0.604	–	–	1.41	8.291	1.75	6.708	–	–
	honda	3.22	2.51	1.280	1.75	1.842	–	–	1.41	2.277	1.75	1.843	–	–
	mcm	2.60	2.51	1.033	3.11	0.836	–	–	2.51	1.034	1.75	1.487	–	–
	wang	1.96	1.41	1.387	1.75	1.123	–	–	1.41	1.388	1.75	1.123	–	–
	pr	1.60	1.41	1.134	1.75	0.918	–	–	1.41	1.135	1.75	0.918	–	–
	Geo. Mean (CG-only)			**1.686x**		**1.257x**		**1.257x**		**3.734x**		**3.489x**		**3.489x**
Good	ac97_ctrl	2.84	10.06	0.283	19.41	0.146	19.41	0.146	5.65	0.503	12.42	0.229	19.41	0.146
	aes_core	3.65	15.71	0.232	19.41	0.188	12.42	0.294	1.41	2.584	3.11	1.176	6.99	0.523
	dir	11.06	5.66	1.956	6.99	1.583	3.11	3.562	1.41	7.828	3.11	3.563	3.11	3.563
	spi	0.79	2.51	0.313	3.11	0.254	3.11	0.254	1.41	0.557	3.11	0.254	3.11	0.254
	pci_master	0.46	5.66	0.081	3.11	0.148	3.11	0.148	1.41	0.326	1.75	0.264	3.11	0.148
	Geo. Mean (Good)			**0.318x**		**0.277x**		**0.357x**		**1.131x**		**0.577x**		**0.400x**
Geo. Mean (CG-only and Good)				**0.888x**		**0.703x**		**0.774x**		**2.358x**		**1.747x**		**1.517x**
Impaired	ethernet	21.70	62.81	0.345	19.41	1.118	38.04	0.570	40.20	0.540	15.72	1.380	27.95	0.776
	wb_conmax	73.06	40.22	1.816	77.63	0.941	62.88	1.162	15.70	4.653	77.63	0.941	49.68	1.471
	dma	7.76	40.22	0.193	77.63	0.100	152.15	0.051	15.70	0.494	49.68	0.156	111.78	0.069
	tv80	2.30	40.20	0.057	19.41	0.119	38.04	0.061	30.78	0.075	15.72	0.147	27.95	0.082
	jpeg_enc	1.64	361.78	0.005	111.78	0.015	27.95	0.059	251.24	0.007	77.63	0.021	27.95	0.059
	systemcaes	2.03	15.71	0.129	19.41	0.104	12.42	0.163	1.41	1.435	6.99	0.290	12.42	0.163
	des	0.60	22.61	0.027	15.72	0.038	27.95	0.022	19.00	0.032	15.72	0.038	27.95	0.022
	systemcdes	0.51	5.66	0.089	19.41	0.026	19.41	0.026	2.51	0.201	15.72	0.032	15.72	0.032
	Geo. Mean (Impaired)			**0.100x**		**0.108x**		**0.101x**		**0.220x**		**0.148x**		**0.120x**
Geo. Mean (All)				**0.386x**		**0.345x**		**0.357x**		**0.956x**		**0.681x**		**0.577x**

Table 6.9: M-HOT compile time and speedup versus QuartusII. Compile time is in seconds. Entries with a "–" are for coarse-grained benchmarks and the same as the $W_f = 0$ value.

				M-HOT					
		QuartusII	VPR	$W_f = 0$		$W_f = 1$		$W_f = 4$	
	Circuit	**Time (s)**	**Time**	**Time**	**Speedup**	**Time**	**Speedup**	**Time**	**Speedup**
CG-only	fft16	333.4	454.1	76.5	4.4	–	–	–	–
	me	220.5	111.9	390.9	0.6	–	–	–	–
	chem	311.9	3,663.9	2.6	119.7	–	–	–	–
	fft8	187.3	73.7	8.4	22.4	–	–	–	–
	honda	162.8	37.2	0.6	268.9	–	–	–	–
	mcm	153.4	25.9	0.8	196.6	–	–	–	–
	wang	148.2	22.2	10.6	14.0	–	–	–	–
	pr	145.6	18.7	0.9	169.0	–	–	–	–
	Geo. Mean (CG-only)				**30.9x**		**30.9x**		**30.9x**
Good	ac97_ctrl	156.8	20.2	241.2	0.7	37.9	4.1	34.1	4.6
	aes_core	169.6	155.0	52.9	3.2	18.1	9.4	18.9	9.0
	dir	186.8	93.8	6.7	27.8	3.3	56.0	3.0	61.6
	spi	138.1	9.5	5.4	25.4	4.7	29.5	4.8	28.7
	pci_master	140.1	10.0	8.0	17.5	2.2	62.3	2.4	58.9
	Geo. Mean (Good)				**7.6x**		**20.9x**		**21.2x**
Geo. Mean (CG-only and Good)					**18.0x**		**26.1x**		**26.7x**
Impaired	ethernet	306.6	207.2	3,574.2	0.1	421.3	0.7	377.3	0.8
	wb_conmax	319.8	804.7	10,857.1	0.03	27,838.2	0.01	1,950.1	0.2
	dma	206.4	64.3	11,165.3	0.02	3,083.6	0.1	8,417.0	0.02
	tv80	165.9	36.6	1,811.0	0.1	138.8	1.2	78.2	2.1
	jpeg_enc	149.6	31.3	267.9	0.6	62.4	2.4	98.3	1.5
	systemcaes	160.9	24.8	183.8	0.9	24.6	6.6	1492	0.1
	des	137.3	19.8	92.5	1.5	19.9	6.9	19.6	7.0
	systemcdes	135.4	16.7	14.8	9.1	6.2	21.8	6.8	20.0
	Geo. Mean (Impaired)				**0.27x**		**1.06x**		**0.79x**
Geo. Mean (All)					**3.64x**		**7.79x**		**7.01x**

Table 6.10: M-HOT compile time speedup versus VPR.

	M-HOT vs. VPR		
	$W_f = 0$	$W_f = 1$	$W_f = 4$
Geo. Mean (CG-only)	14.50x	14.50x	14.50x
Geo. Mean (Good)	1.49x	4.10x	4.16x
Geo. Mean (CG-only and Good)	5.47x	8.44x	8.49x
Geo. Mean (Impaired)	0.08x	0.32x	0.24x
Geo. Mean (All)	1.04x	2.28x	2.04x

Table 6.11: M-HOT compile time speedup versus M-CAD. An entry of 0.4 means that M-CAD is 1/0.4=2.5x faster than M-HOT.

	M-CAD (Table 5.9) / **M-HOT** (Table 6.9)		
	$W_f = 0$	$W_f = 1$	$W_f = 4$
Geo. Mean (CG-only)	0.40	0.40	0.40
Geo. Mean (Good)	0.21	1.35	1.25
Geo. Mean (CG-only and Good)	0.31	0.64	0.62
Geo. Mean (Impaired)	0.05	1.88	2.67
Geo. Mean (All)	0.16	0.96	1.08

scheduling to verify the route timings. Reducing this routing burden, combined with the separate low-temperature annealing at each height, results in a faster overall flow.

Despite a few slow-compiling circuits (*wb_conmax* and *dma*) M-HOT is a reasonably fast approach. There is much work that can be done to improve the synthesis speed of M-HOT, but as is, 17 of the 21 benchmarks show a speedup compared to QuartusII ($W_f = 1$). Of those that compile slower, 3 of the 4 are *Impaired* benchmarks. Overall, the M-HOT approach is faster than QuartusII, but not as fast as M-CAD. However, given the circuit density and performance advantages with M-HOT, it should be used instead of M-CAD for coarse-grained circuit compilation. Using $W_f = 1$ is again preferred because of the increase in compilation speed compared to $W_f = 0$ with no significant further advantage by going to $W_f = 4$. For

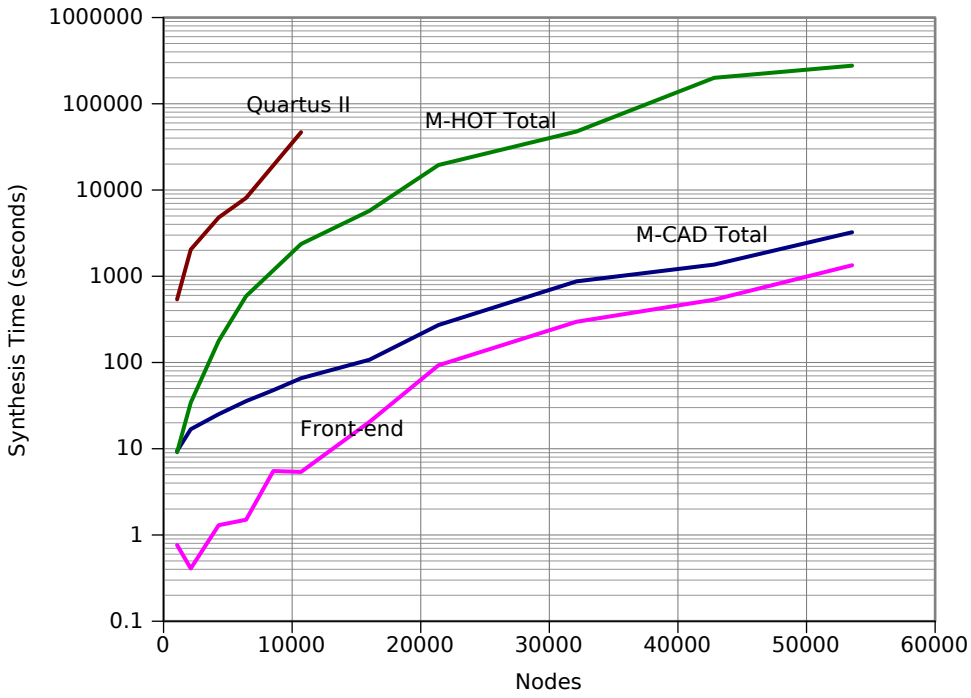


Figure 6.5: M-HOT compile time for very large circuits. A 32x32 CLB (1024 CLBs total) architecture was used.

circuit performance and density, $W_f = 1$ is a midpoint between improved performance and loss in density.

6.6.5 Compile Time for Very Large Circuits

This section tests the compile time performance of M-HOT using the same large randomly generated circuits that were used in Section 5.6.5. Figure 6.5 shows the average compile time for M-HOT when run with 10 trials of the random benchmarks (a different random benchmark of the same size is used for each trial) compiled for an architecture of 1024 (32x32) CLBs. The M-CAD and QuartusII data from Figure 5.7 are also shown for comparison.

The M-HOT runtime is approximately 42x higher (slower) than M-CAD on average (excluding the data below 10,000 nodes), and about an order-of-magnitude below QuartusII. For the $50,000$ node circuit, the total compile time was 4,609 minutes (3 days, 4 hours, 49 min-

utes). Further, these very large circuits synthesize successfully on Malibu, whereas they exceed the capacity of the largest modern FPGAs.

6.6.6 Longest-Path Analysis

This section presents a breakdown of the longest path for the maximum frequency results in Table 6.1, and compares this breakdown to the longest path results from M-CAD given in Table 5.11. The length of a path in the scheduled circuit is the time required to traverse all compute and routing resources from inputs and registers, to outputs and registers. The longest path can be less than or equal to the schedule length, but not greater than it. An example of when it will be less than the schedule length is given in Section 5.6.6.

Table 6.12 shows a breakdown of the longest path for the M-HOT results. Starting from the left is the schedule length (SL) of the synthesized circuit which is reported as F_{max} in Table 6.4. The next column is the longest path (Len), followed by the number of longest-paths (Count). The next five columns are the longest-path breakdown, averaged over the number of longest-paths: the number of compute-only slots where an ALU is in use (ALU); the number of slots spent waiting for values to arrive or waiting for the ALU while it is busy with other computation (Wait); the number of compute-and-move slots where a value is computed then transfered directly to a neighbouring CLB without using the crossbar (Move); the number of routing timeslots where progress is made (Route); and the number of routing hold timeslots where a value is held due to a route conflict (Hold).

The M-HOT longest path results in Table 6.12 are similar to the M-CAD results in Table 5.11, but there are three interesting observations. First, M-HOT has fewer singletons (circuits with only one longest path). Fewer singletons means the tool is doing a better job of keeping a single path from lengthening too much; instead multiple paths are critical. By integrating the placer, router, and scheduler into a single flow, M-HOT can avoid some scenarios where, previously, the M-CAD flow would have had no choice but to lengthen the schedule to

Table 6.12: M-HOT Longest-path breakdown for maximum performance results. All numbers are system clock cycles.

		M-HOT $W_f = 0$								**M-HOT $W_f = 1$**								
					Compute			**Route**					**Compute**			**Route**		
	Circuit	**SL**	**Len**	**Count**	**ALU**	**Wait**	**Move**	**Route**	**Hold**[1]	**SL**	**Len**	**Count**	**ALU**	**Wait**	**Move**	**Route**	**Hold**[1]	**FG**
CG-only	fft16	29	29	43	1.2	23.7	0.1	4.0	0.0	–	–	–	–	–	–	–	–	–
	me	39	39	16	1.8	31.1	0.1	5.9	0.0	–	–	–	–	–	–	–	–	–
	chem	20	19	10	5.5	0.0	9.9	3.6	0.0	–	–	–	–	–	–	–	–	–
	fft8	18	18	25	1.6	12.9	0.3	3.2	0.0	–	–	–	–	–	–	–	–	–
	honda	18	18	4	7.2	0.0	7.5	3.2	0.0	–	–	–	–	–	–	–	–	–
	mcm	12	11	5	4.6	0.0	4.2	2.2	0.0	–	–	–	–	–	–	–	–	–
	wang	9	9	19	4.2	0.0	3.3	1.5	0.0	–	–	–	–	–	–	–	–	–
	pr	10	10	6	3.5	0.0	5.5	1.0	0.0	–	–	–	–	–	–	–	–	–
Good	ac97_ctrl	57	53	8	1.8	49.4	0.5	1.4	0.0	23	22	24	3.5	11.5	1.1	5.3	0.0	0.6
	aes_core	34	33	4	7.2	0.8	5.2	19.8	0.0	33	32	8	7.6	6.2	5.1	12.2	0.0	0.8
	dir	30	28	18	4.2	10.7	7.2	5.9	0.0	29	29	16	4.4	17.1	3.6	3.6	0.0	0.3
	spi	37	27	2	4.0	17.5	1.5	4.0	0.0	19	18	1	5.0	9.0	1.0	2.0	0.0	1.0
	pci_master	34	29	5	1.4	24.8	1.0	1.8	0.0	25	24	4	4.8	4.5	13.5	1.0	0.0	0.2
Impaired	ethernet	105	82	13	2.5	73.8	0.6	5.1	0.0	79	72	1	3.0	64.0	0.0	4.0	0.0	1.0
	wb_conmax	76	76	10	5.0	45.6	2.0	23.4	0.0	44	44	21	5.2	24.7	3.1	10.5	0.0	0.5
	dma	83	83	42	8.0	41.6	8.1	25.3	0.0	65	65	20	7.1	48.2	2.8	5.9	0.0	1.0
	tv80	243	242	14	13.9	196.0	8.9	23.3	0.0	176	175	15	37.9	20.7	73.7	37.5	0.0	5.3
	jpeg_enc	42	42	9	3.2	29.1	1.2	8.4	0.0	35	35	12	3.9	19.3	1.2	9.8	0.0	0.8
	systemcaes	32	30	19	3.4	18.4	1.1	7.1	0.0	25	25	37	3.2	14.2	0.7	6.5	0.0	0.4
	des	155	154	1	2.0	147.0	1.0	4.0	0.0	116	116	1	3.0	105.0	0.0	8.0	0.0	0.0
	systemcdes	39	39	2	1.5	34.5	1.0	2.0	0.0	36	36	2	1.0	30.0	1.0	4.0	0.0	0.0

		M-HOT $W_f = 4$								
Good	ac97_ctrl	19	18	11	4.2	7.1	1.3	4.5	0.0	1.0
	aes_core	34	34	5	8.8	7.0	6.0	11.4	0.0	0.8
	dir	27	26	52	4.4	10.2	9.4	1.7	0.0	0.3
	spi	19	18	1	1.0	14.0	0.0	3.0	0.0	0.0
	pci_master	8	8	1	3.0	0.0	4.0	0.0	0.0	1.0
Impaired	ethernet	31	31	31	4.9	15.4	1.7	8.1	0.0	0.9
	wb_conmax	50	50	6	5.3	15.5	1.8	26.2	0.0	1.2
	dma	56	56	20	9.0	37.6	1.4	6.2	0.0	1.8
	tv80	30	30	11	4.4	10.6	1.0	13.1	0.0	0.9
	jpeg_enc	36	35	40	5.6	20.0	1.4	6.4	0.0	1.7
	systemcaes	20	20	15	3.3	9.3	0.4	6.7	0.0	0.3
	des	99	59	1	15.0	0.0	20.0	23.0	0.0	1.0
	systemcdes	33	33	1	10.0	0.0	19.0	3.0	0.0	1.0

[1]: The number of hold slots are zero for all circuits (not just rounded to zero).

connect one last path. M-HOT, instead, would change the placement (if a lower-cost alternative could be found by the annealer).

Second, there are no routing hold slots on the longest path in M-HOT. In fact, there are no hold slots on any path in any benchmark. This means, as before with M-CAD, that the horizontal-then-vertical routing strategy is not causing any routing congestion.

Third, a large portion of the longest path, for most benchmarks, is still wait slots. The M-HOT approach favours a "compute-and-move" strategy where a value is computed while it is routed. In these results, there is no apparent increase in the number of such operations on the longest path compared to M-CAD. However, the M-HOT longest paths are different than the M-CAD longest paths (verified by checking that the sources and sinks of the paths are different). To see that M-HOT actually does favour a "compute-and-move" strategy, Table 6.13 shows the total number of "compute-and-move" operations for each benchmark for M-CAD and M-HOT. On the far right of this table is the ratio of the number of these operations in M-HOT / M-CAD. M-HOT contains 18% to 30% more of these operations (*CG-only* and *Good* benchmarks), on average. So, while the number of compute-and-move operations along the longest path is the same between M-HOT and M-CAD, the total number is 18% to 30% greater in M-HOT.

6.6.7 Density Versus Performance Tradeoff

This section investigates the density versus performance tradeoff in M-HOT. We show that, like the M-CAD tool, the M-HOT tool can trade density for performance, and that increasing W_f generally gives performance gains across all densities, not just at the F_{max} as was shown in Table 6.1.

Figure 6.6 shows the frequency of the largest benchmark, *ethernet*, for W_f =0,1,4 over a range of architecture sizes from 3x3 CLBs to 48x48 CLBs. This figure contains the M-CAD data from Figure 5.9, as well as the M-HOT data. Appendix D contains the individual area

Table 6.13: Total number of compute-and-move operations.

		M-CAD			M-HOT			M-HOT / M-CAD		
	Circuit	$W_f = 0$	$W_f = 1$	$W_f = 4$	$W_f = 0$	$W_f = 1$	$W_f = 4$	$W_f = 0$	$W_f = 1$	$W_f = 4$
CG-only	fft16	2,520	–	–	4,671	–	–	1.85	–	–
	me	3,095	–	–	9,154	–	–	2.96	–	–
	chem	822	–	–	1,156	–	–	1.41	–	–
	fft8	986	–	–	1,131	–	–	1.15	–	–
	honda	285	–	–	303	–	–	1.06	–	–
	mcm	329	–	–	295	–	–	0.90	–	–
	wang	121	–	–	120	–	–	0.99	–	–
	pr	153	–	–	134	–	–	0.88	–	–
	Geo. Mean (CG-only)							**1.28**	**1.28**	**1.28**
Good	ac97_ctrl	7,410	6,835	6,765	8,911	6,538	6,345	1.20	0.96	0.94
	aes_core	2,626	3,286	3,919	4,305	5,028	4,877	1.64	1.53	1.24
	dir	980	2,179	2,239	1,472	2,173	2,327	1.50	1.00	1.04
	spi	1,068	1,044	892	995	959	874	0.93	0.92	0.98
	pci_master	1,492	1,505	1,357	2,192	1,466	1,338	1.47	0.97	0.99
	Geo. Mean (Good)							**1.32**	**1.05**	**1.03**
Geo. Mean (CG-only and Good)								**1.30**	**1.19**	**1.18**
Impaired	ethernet	18,710	16,096	17,292	26,653	17,352	16,134	1.42	1.08	0.93
	wb_conmax	27,594	34,511	36,205	53,026	38,360	39,779	1.92	1.11	1.10
	dma	25,249	28,016	35,218	46,477	24,503	26,286	1.84	0.87	0.75
	tv80	20,647	23,025	17,381	23,934	21,106	16,628	1.16	0.92	0.96
	jpeg_enc	13,772	11,257	9,881	7,517	8,259	9,197	0.55	0.73	0.93
	systemcaes	10,475	7,195	6,114	5,136	5,433	5,599	0.49	0.76	0.92
	des	3,632	7,226	7,519	6,855	7,833	8,056	1.89	1.08	1.07
	systemcdes	1,704	3,185	3,367	2,431	3,138	3,276	1.43	0.99	0.97
	Geo. Mean (Impaired)							**1.20**	**0.93**	**0.95**
Geo. Mean (All)								**1.26**	**1.08**	**1.09**

versus performance graphs for each benchmark circuit.

For each architecture array size, the tools are forced to use all available CLBs so that the performance on the various sized architecture arrays can be seen. When the array becomes too large there is a decrease in performance as communication delay dominates the schedule. If the array is too small, the synthesis result may not be viable if the fixed resource constraints (from Table 3.5) are violated. In these cases, the M-HOT tool still finishes the synthesis (overusing the necessary resources) but reports an error. For completeness, these results are still included

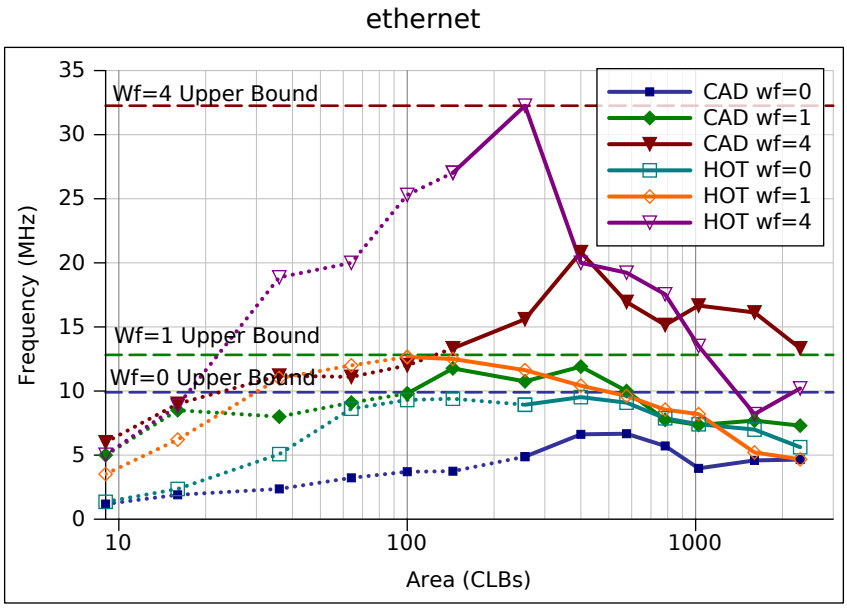


Figure 6.6: Frequency versus area (number of CLBs) for the *ethernet* benchmark. The dotted line represents synthesis results where architectural resource constraints were violated.

and marked with a dotted line on the left of each curve.

This graph demonstrates three things about the M-HOT tool. First, the M-HOT frequency results are mostly higher for all architecture sizes than M-CAD, demonstrating the superiority in the quality of results in the M-HOT approach. Second, this figure (and the other graphs in Appendix A) show that the M-HOT tool can trade area (number of CLBs) for performance (MHz) by targeting any architecture array size and time-multiplexing more code (or less) on the CGs. Third, it shows the tradeoff involving W_f. Increasing W_f from 0 to 1 causes the frequency to also increase as expected since the fine-grained control logic is moved to the FG resources where it can be computed and distributed more quickly.

6.7 Comparison to Previous Work

The M-HOT flow was developed after an analysis of the M-CAD results in the previous chapter. It was likely that the performance and density of circuits mapped to Malibu could be improved

with an algorithm that could use information from routing and scheduling during placement. The challenge was to create such an algorithm and make it fast.

M-HOT does simultaneous placement, routing, and scheduling of both coarse-grained and fine-grained resources. It supports a heterogeneous architecture (recall only one in five CLBs have multipliers, and only CLBs around the periphery support I/O).

The overall flow of M-HOT is similar to the modulo graph embedding (MGE) scheduler by Park [62]; the DFG is divided into a number of heights, and each height placed, routed, and scheduled separately. This flow allows the algorithm to be fast while retaining some benefits of doing placement, routing, and scheduling at the same time. The MGE scheduler, however, targets a CGRA architecture similar to ADRES [57]. It only supports coarse-grained elements, and maps software loop kernels, not circuits. The specific differences between M-HOT scheduler step and the MGE scheduler scheduler are as follows:

- The MGE scheduler was tested on CGRAs with up to 16 (4x4 array) processing elements. Park noted that the scheduler should scale up to larger architectures. In this thesis, we use M-HOT on architectures up to 48x48 CLBs, demonstrating that the general approach does indeed scale.

- M-HOT places, routes, and schedules the fine-grained and coarse-grained operations together. To do this, the delay model used throughout M-HOT is the same unified delay model that was used in Chapter 5. M-HOT also interfaces with the VPR router (as was done in Chapter 5) to verify the fine-grained routing delays at the end of scheduling. The MGE schedule has no support for fine-grained resources.

- The MGE scheduler pre-places inputs, outputs, registers, and memories. In M-HOT, inputs for the most critical paths in the DFG are at the highest level of the ALAP tree (see Section 6.4), so these inputs are placed first anyway. Other inputs are placed as needed,

and thus have the opportunity to use the affinity cost from the currently unfolding schedule to be placed in locations near where they are needed. The same is true for outputs, except that the affinity cost is used to encourage paths toward an output-capable CLB when required. Registers in M-HOT are simply nodes in the DFG with the *registered* flag enabled, they do not need any extra resources so there is no reason to pre-place them. Finally, there is memory in each CLB, and not pre-placing the memories means the scheduler can choose the best location for the memory based on the current schedule and minimize the schedule length.

- The MGE scheduler used a fixed II for scheduling. It uses multiple invocations of the tool to search for the lowest II, but is not guaranteed to find the true minimum II. M-HOT minimizes the SL, not the II, for reasons discussed in Section 2.3.3. M-HOT also uses a variable-length schedule that is increased as needed during scheduling. This is sub-optimal, but it means the tool is only invoked once.

- M-HOT uses a *parallel_cost* in the cost function to encourage nodes at each height to spread out to many CLBs, encouraging parallelism.

- M-HOT uses a *register_cost* in the cost function to cause multiple DFG paths that end at the same user register (logic reconvergence) to tend toward the CLB which holds the register state. We believe that the MGE scheduler used the affinity to do this, but that is not explicitly stated in [62]. Registers are always at the last height (height=0) of the ALAP tree. Because of this, the sinks of the registers will already be placed and have position information available. The *register_cost* uses the actual placement information, just as the *producer_cost* uses the actual placement information, rather than using the affinity heuristic.

- M-HOT uses a *penalty* cost in the cost function to discourage illegal placements and

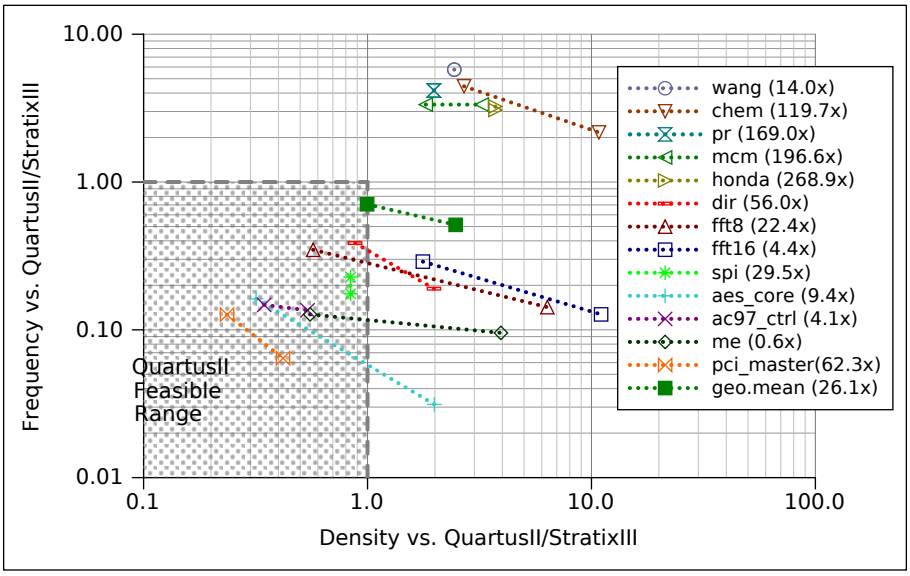


Figure 6.7: M-HOT $W_f = 1$ results summary. The *CG-only* and *Good* benchmark frequency versus density is shown relative to QuartusII/StratixII. The maximum density and maximum performance data points are shown for each benchmark. The compile time speedup compared to QuartusII is shown in the figure legend after the benchmark name.

resource overuse.

6.8 Conclusions

This chapter presents the M-HOT flow which breaks from the traditional FPGA-CAD flow of placement-then-routing, and performs integrated placement, routing, and scheduling to achieve a higher-quality solution. Although the compile time in M-HOT is 1.5x longer than M-CAD ($W_f = 1$), it is still 26.1x faster than both QuartusII and VPR. M-HOT also improves the circuit performance by an average of 10% over M-CAD and simultaneously improves the density by an average of nearly 2x.

The results in this chapter have led us to recommend that the Malibu architecture be used with M-HOT with a fine-grained width of 1 ($W_f = 1$). M-HOT provides a very fast compilation and generates high-quality results. Using $W_f = 1$ balances the performance gains from

increasing W_f with the density loss. At maximum performance, M-HOT/Malibu achieves 70% the performance of an StratixIII FPGA and about the same density (26.1x compile speedup, 0.707x performance, 0.996x density). At maximum density, Malibu is about half the performance of the FPGA and almost 2.5x the density (26.1x compile speedup, 0.513x performance, 2.474x density).

These results are summarized in Figure 6.7. The shaded area of the figure is the performance and density achievable with QuartusII/StratixIII. The figure shows the density, performance, and compile time results achieved for the *CG-only* and *Good* benchmarks for M-HOT $W_f = 1$. The maximum performance and maximum density is plotted for each benchmark. The compile time speedup relative to QuartusII is shown in brackets after the benchmark name in the figure legend. For all the circuits except *spi*, *ac97_ctrl*, and *pci_master*, M-HOT is able to achieve a density result outside the range reachable by QuartusII/StratixIII. For all the benchmarks except *me*, the compile time compared to QuartusII is much faster. Five circuits achieve both a higher frequency and a more dense result than QuartusII.

Among these results, M-HOT sometimes generates solutions with poor performance. In particular, larger circuits with a small height (e.g., 4) have long schedule lengths because the low-temperature annealer gives up before a good solution is reached. This situation can be detected and worked-around by dynamically adjusting the annealing schedule, but that may cause even longer compilation times. Since these circuits already have long compile times, this may not be a good idea. Instead, future work should investigate other approaches to placement that are faster than annealing, or at least scale better to larger numbers of nodes.

The frequency upper bound which was computed for M-CAD is also compared to M-HOT. M-HOT achieves 64% of this upper bound, whereas M-CAD only achieves 50%. Due to architectural constraints, the entire 1.6x improvement may not be achievable, however an M-HOT experiment using a high-effort annealer showed that some improvement is possible. To realize additional performance and density gains, future work should focus on improving the

front-end synthesis, the coarse/fine-grained interface, and the coarse/fine-grained partitioning strategy to provide a higher quality input to M-HOT.

Chapter 7

Conclusions

7.1 Thesis Conclusions

Modern FPGAs implement a wide range of circuits which have both coarse-grained and fine-grained components. The capacity of these FPGAs continues to double every 18 to 24 months [29, 80] as predicted by Moore's Law. As such, great demand is placed on FPGA CAD tools to synthesize these ever-larger circuits faster, and without loss in quality.

In particular, there is a trend toward using circuits with coarse-grained components. This is partly because of full-system generators and C–to-gates hardware flows that automatically generate large, word-oriented datapaths from software specifications. Unfortunately, coarse-grained circuits do not map well to traditional fine-grained FPGAs; the compilation is slow and there is a fixed device capacity limit.

We look at addressing these two issues in this thesis, as well as improving density, and maintaining good circuit performance relative to an FPGA. Improved density, a soft capacity limit, and fast compile times are all important for reducing costs (either silicon area or engineering salaries). Additionally, compile time, a soft device capacity limit, and circuit performance are important for a rapid product development cycle and reducing the time-to-market for new

technology.

To improve coarse-grained circuit mapping (addressing compile time, area/density, capacity limit, and performance), time-multiplexed coarse-grained resources are added to the traditional fine-grained FPGA CLB to create Malibu, a new type of FPGA architecture. Two mapping approaches are demonstrated for this new architecture: M-CAD, which is based on traditional FPGA CAD tools with a scheduling step added at the end, and M-HOT, which is based on an integrated placement, routing, and scheduling approach.

To evaluate the Malibu architecture and tools, three metrics are used: compile time, density, and performance. The compile time metric is important for a rapid product development cycle, whether that is fixing bugs or adding new features to circuits. Current compile times for FPGA CAD tools range from several minutes for small circuits to many hours, or even days, for very large circuits. This wastes time and hurts designer productivity.

Density is an important metric because density translates directly into cost. Larger devices cost more. Being able to fit circuits onto smaller devices, whether for testing or product deployment, results in a cost savings. Improving density over FPGAs has an additional benefit: the Malibu architecture can be used for anticipatory large-circuit development and testing for large FPGA devices which do not exist yet, but will soon, as predicted by Moore's Law.

Performance is important for data throughput, whether emulating an ASIC for testing, or using the FPGA in a final product. FPGAs already come within $1/4^{th}$ the performance of an ASIC [45], which is fast enough for both simulation and direct use in commercial products. When considering a new FPGA-like architecture for coarse-grained circuits, performance must not be degraded too much.

These metrics are measured using the synthesis results from a number of benchmark benchmark circuits. The circuits are divided into three categories: *CG-only*, *Good*, and *Impaired*. The first two categories are coarse-grained circuits with small amounts of fine-grained logic. These are the types of circuits that we expect would be implemented on Malibu, so all com-

parisons in this thesis are done with these circuits. The *Impaired* circuits use Verilog structures which do not map well to Malibu. These circuits are written in a style which is not efficiently supported by either our front-end synthesis, the back-end mapping flow (M-CAD and M-HOT), or the Malibu architecture, or some combination thereof. An experienced Malibu developer would avoid using these particular structures and design a circuit in a way that works well with Malibu. However, in this thesis, we have not modified any of the benchmark circuits. The data for the *Impaired* benchmarks is shown throughout the thesis, but we only require that M-CAD and M-HOT successfully synthesize these circuits.

The Malibu architecture, the M-CAD flow, and the M-HOT flow form the three main thesis contributions. We use the three aforementioned metrics with the benchmark circuits to assess the architecture and tool flows, and to compare them with a commercial FPGA (StratixIII) and FPGA CAD tool (QuartusII). This comparison and evaluation of Malibu has led us to recommend using the M-HOT flow with a fine-grained width (W_f) of 1 for a balance between the performance gains by pushing more resources onto the fine-grained resources and the density loss of doing so.

The first thesis contribution is the Malibu architecture presented in Chapter 3. The Malibu CLB is a unique combination of time-multiplexed, coarse-grained resources and traditional fine-grained FPGA CLBs. We explain the architecture in detail and show how it implements a circuit. Then, an experiment is done to determine suitable values for all the architecture parameters. With the architecture fully specified, an 81-bit instruction word for use by the coarse-grained resources is derived, and area of the Malibu CLB is computed to be 97,050.5μm^2. The area is used to compute the density of M-CAD and M-HOT synthesis results.

Compared to FPGAs, the coarse-grained resources allow a much faster compile times (up to 269x faster in this thesis), and the time-multiplexing feature can improve density over a StratixIII FPGA up to 2.5x on average. The architecture also allows the Malibu CAD flows (M-CAD or M-HOT) to trade density for performance. An average (geomean) density range

of 0.996x up to 2.474x is demonstrated.

The second thesis contribution is the M-CAD flow presented in Chapters 4 and 5. M-CAD is based on algorithms used by FPGA CAD tools [13, 18, 39, 71], but augments these with support for coarse-grained, time-multiplexed resources. New placement and routing tools are created based on these algorithms to handle both types of resources simultaneously. Finally, a new scheduling step is added at the end of the flow to temporally order the coarse-grained operations in each CLB. To our knowledge this is the first time a complete Verilog-to-bitstream flow has been created for an FPGA-like architecture with both coarse-grained and fine-grained resources.

We show that M-CAD improves compile times over FPGA CAD tools by an average factor of 38.7x (geometric mean of speedups for M-CAD, $W_f = 1$, *CG-only* and *Good* benchmarks). Having coarse-grained resources in the architecture means that coarse-grained synthesis can be used (as opposed to synthesizing down to 2–input logic gates), which significantly reduces the problem size and thus improves the compile time. We also avoid the use of long-running iterative algorithms to keep runtime low. Very large synthetic circuits, with up to 50,000 32-bit operations (approx 1.6 million gates), are used to extrapolate performance results. M-CAD synthesizes the 50,000 node circuit in 54 minutes, whereas QuartusII quits after 24 hours without finishing.

To test the performance limit, a performance upper bound is calculated using the post-front-end synthesis graph-depth of the circuit. We find that the M-CAD synthesis results reach just over half of this value, meaning there is less than a 2x improvement possible without changing the front-end synthesis.

Finally, M-CAD can trade density for circuit performance without changing the underlying architecture. Compared to QuartusII/StratixIII, the performance/density range is from 0.582x density and 0.642x performance at maximum performance to 2.474x density and 0.429x performance at maximum density. These results are summarized in Table 7.1 and Figure 7.1. If

the architecture parameter values are changed to include more memory for additional time-multiplexing, a 10x density is possible with 0.01x performance.

The major drawback of the M-CAD approach is the separation of information in the algorithms used. The separation is not a problem in FPGA CAD, but adding another dimension to the mapping problem, time, exemplifies the inefficiencies. Specifically, there are cases where placement needs the order of the coarse-grained operations (the output of scheduling) to avoid bad decisions. Since we avoid iterative approaches this is not possible, and the placement step ends up accounting for 70% of the loss from the upper bound to the actual synthesis result.

The third thesis contribution is the M-HOT flow presented in Chapter 6. M-HOT does integrated placement, routing, and scheduling to avoid the aforementioned problem with the separation of information. It is based on a modulo graph embedding CGRA scheduler [62]. It constructs an ALAP tree representation of the circuit and places (in space and time) and routes each height separately, starting from the maximum height where the most critical nodes are.

We also use the same performance upper bound that was calculated in the M-CAD analysis to show that M-HOT achieves 64% of this maximum. Hence, there is at best a 1.6x performance improvement possible in M-HOT. The large circuit synthesis results confirm that M-HOT can also successfully synthesize large circuits on the Malibu architecture, but it takes longer, just over 3 days for the 50,000 node circuit.

The M-HOT approach is not without drawbacks. It generates slightly inferior solutions to M-CAD on large circuits with a small height (e.g., less than 10) because the low-temperature annealer gives up before a good solution is found. This situation can be detected and worked-around by dynamically adjusting the annealing schedule, but that increases the compile time. With the already long compile times for circuits with imbalanced ALAP trees, this may not be a good idea. A better way of recovering placement quality is needed.

The M-CAD and M-HOT results are summarized next. Table 7.1 summarizes the numerical results, and Figure 7.1 shows the performance versus density tradeoff graphically for $W_f = 1$.

Table 7.1: Summary of important results. Geometric mean of the *CG-only* and *Good* benchmarks. All results are compared to a QuartusII synthesis for the largest StratixIII FPGA (EP3SL340F1760C2).

		Density vs. Performance Range			
		At Max Performance		At Max Density	
Result	Compile Time	Perf.	Density	Perf.	Density
M-CAD $W_f = 0$	57.5x	0.588x	0.506x	0.308x	3.339x
M-CAD $W_f = 1$	38.7x	0.652x	0.582x	0.429x	2.474x
M-CAD $W_f = 4$	16.9x	0.713x	0.637x	0.487x	2.148x
M-HOT $W_f = 0$	18.0x	0.609x	1.297x	0.369x	3.339x
M-HOT $W_f = 1$	**26.1x**	**0.707x**	**0.996x**	**0.513x**	**2.474x**
M-HOT $W_f = 4$	26.7x	0.786x	1.097x	0.625x	2.148x

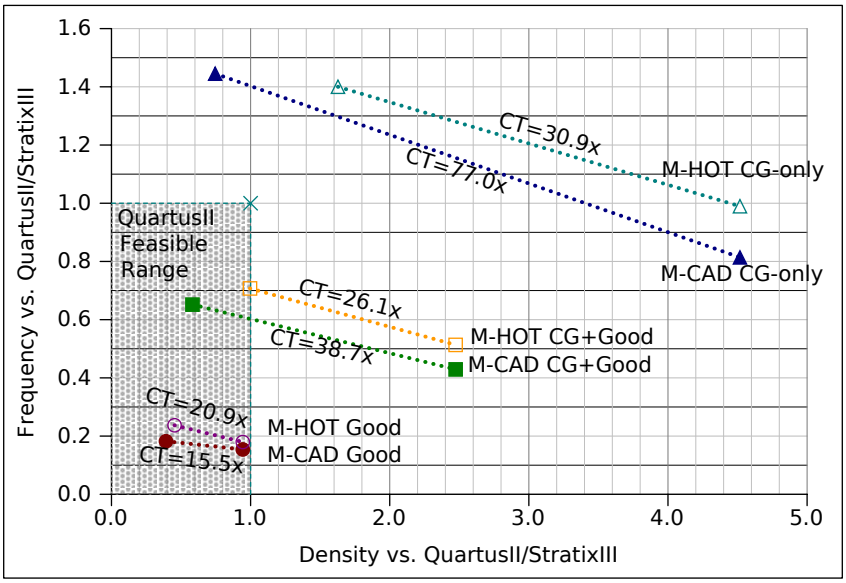


Figure 7.1: Malibu $W_f = 1$ results summary. The geometric mean of the frequency versus density relative to relative to QuartusII/StratixII is shown for maximum density and maximum performance. The CT value on each curve is the compile time speedup relative to QuartusII.

In Table 7.1, all the results are versus QuartusII synthesizing for a StratixIII FPGA. "0.1x" means $1/10^{th}$ the QuartusII/StratixIII result (slower for synthesis, larger for area), "1x" means the same result, and "10x" means ten times better. The table shows the average compile times, the performance at the maximum density, and the density at the maximum performance. M-HOT gives superior results compared to M-CAD, and $W_f = 1$ balances the loss in density (at maximum density) with the gains in performance.

Figure 7.1 shows the $W_f = 1$ results graphically. There are three curves for the *CG-only*, *Good*, and combined *CG-only*+*Good* benchmark sets. The maximum performance and maximum density is plotted for each benchmark set. The CT value on each curve is the compile time speedup relative to QuartusII. M-HOT achieves better performance and density results compared to M-CAD. For *CG-only*, both M-CAD and M-HOT achieve performance, density, and compile time results outside the range reachable by QuartusII/StratixIII. For *CG-only*+*Good*, both M-CAD and M-HOT achieve density and compile time results outside the range reachable by QuartusII/StratixIII.

It is difficult to compare the Malibu architecture and CAD to more mature work such as VPR and QuartusII, but the architecture and tools show promising results. Appendix A verified that it is not possible to achieve comparable compile times with VPR or QuartusII. The compile time in QuartusII and VPR was decreased by disabling features like timing-driven placement, and lowering the annealing placement effort. As placement time decreases, so does the quality of the placement solution, and that causes routing time to increase. The overall effect is that total compile time can be decreased by about 15% with a 13% reduction in F_{max}. In this thesis there is an average 18x to 57.5x decrease in compile time with a 33% to 66% reduction in F_{max}.

M-CAD and M-HOT sometimes have very slow runtimes, and the front-end synthesis sometimes produces more nodes than what seems necessary. However, both flows are new and written only by one person. In contrast, VPR and QuartusII are mature tools which have evolved for years with multiple developers, so they exhibit fewer special cases. Most impor-

tantly, the results presented in this thesis show it is possible to achieve fast synthesis, increased density, and reasonable performance on the Malibu architecture with a variety of circuits.

7.2 Future Directions

This work has shown that clock frequency results can be improved by no more than 2x or 1.6x by making simple adjustments to M-CAD and M-HOT, respectively. However, it may be possible to achieve further gains in compile time or density by tuning these tools.

For example, to decrease compile time, one possibility is to do away with placement entirely. The edge-centric modulo scheduler proposed by Park *et al.* [64] is an interesting approach to CGRA scheduling. Basically, if a route is created between an input (or register) and an output (or register), then the operations along that path must be implemented in the CLBs visited by the route. A routing algorithm like PathFinder could work to implement this approach for Malibu, and the time saved by not doing placement could allow for an iterative approach. Heavily used CLBs would be marked as congested, and the router would rip-up and re-route until all paths were satisfied, with the most critical paths being the highest priority.

There are also some limitations to the current tool that would merit future work regardless of any compile time or density improvements. We assume a single clock domain in this work, which could be an issue with complex industrial circuits. Multiple clock domains could be handled by either partitioning the CLBs into groups for each clock domain, or by unifying the domains into a single clock, e.g. if one clock is a multiple of the other, the code in one clock domain could be unrolled.

We have also avoided specifically handling large, user-instantiated memories. This could be handled in software by programming one CLB to fetch data from nearby CLBs which are, in turn, programmed specifically to act as memory fetch units. It could also be handled at the hardware level by building some mechanism into the Malibu architecture to aggregate memories automatically if required.

The work done on front-end synthesis in this thesis just scratches the surface of what is possible. A better front-end synthesis tool would generate a better-optimized circuit which would be smaller, faster, and take less time to compile. Using Verilator was an excellent start as it allowed the rest of the tool flow to be developed and evaluated. What is now needed is a new front-end synthesis tool that is suitable for both M-CAD and M-HOT. It must be both coarse-grained and fine-grained aware, and able to apply optimizations to both types of resources.

Due to time constraints, and the lack of an improved front-end synthesis tool tuned for Malibu, we have avoided a deep investigation into architectural enhancements. For example, the architecture could include long coarse-grained wires to reduce waiting time. It could include multiple ALUs per CLB to speed up computation and further reduce waiting time. It could have an improved FG/CG interface, or the ability to time-multiplex the fine-grained resources (which appear to be limiting the density).

As with all research, this thesis concludes by saying: there is much work to be done.

Bibliography

[1] *VPR and T-VPack User's Manual (Version 5.0)*. → pages 192, 206

[2] L. V. Agostini, R. C. Porto, S. Bampi, and I. S. Silva. A FPGA based design of a multiplierless and fully pipelined JPEG compressor. In *Proc. Euromicro Conference on Digital System Design*, pages 210–213, 2005. → pages 2

[3] Altera Corporation. Automated generation of hardware accelerators with direct memory access from ANSI/ISO standard C functions. whitepaper., May 2006. URL www.altera.com/literature/wp/wp-aghrdwr.pdf. → pages 2

[4] Altera Corporation. *Stratix III Device Handbook*, 2007. URL http://www.altera.com/literature/hb/stx3/stratix3_handbook.pdf. → pages 78

[5] Altera Corporation. *Quartus II Handbook*, 2010. URL http://www.altera.com/literature/hb/qts/quartusii_handbook.pdf. → pages 3, 16

[6] Altera Corporation. *SOPC Builder User Guide*, 2010. URL http://www.altera.com/literature/ug/ug_sopc_builder.pdf. → pages 2

[7] Altera Corporation. *Stratix V Device Handbook*, 2011. URL http://www.altera.com/literature/hb/stratix-v/stratix5_handbook.pdf. → pages 1

[8] R. Amerson, R. J. Carter, W. B. Culbertson, P. Kuekes, and G. Snider. Teramac–configurable custom computing. In *Proc. FPGAs for Custom Computing Machines (FCCM)*, pages 32–38, 1995. → pages 36

[9] R. Amerson, R. Carter, W. Culbertson, P. Kuekes, G. Snider, and L. Albertson. Plasma: An FPGA for million gate systems. In *Proc. Field-Programmable Gate Arrays (FPGA)*, pages 10–16, 1996. → pages 36

[10] ARM. Synthesizable ARM7TDMI^{TM} 32-bit RISC performance, retrieved Jan. 2010. URL www.arm.com/products/CPUs/ARM7TDMIS.html. → pages 62

[11] V. Betz and J. Rose. Directional bias and non-uniformity in FPGA global routing architectures. In *Proc. International Conference on Computer-Aided Design (ICCAD)*, ICCAD, pages 652–659, 1996. → pages 21

[12] V. Betz and J. Rose. Cluster-based logic blocks for FPGAs: Area-efficiency vs. input sharing and size. In *Proc. Custom Integrated Circuits Conference (CICC)*, pages 551–554, 1997. → pages 15, 17

[13] V. Betz and J. Rose. VPR: A new packing, placement and routing tool for FPGA research. In *Proc. Field Programmable Logic (FPL)*, pages 213–222, 1997. → pages 7, 19, 173

[14] V. Betz, J. Rose, and A. Marquardt, editors. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, 1999. → pages 45

[15] N. Bhat, K. Chaudhary, and E. S. Kuh. Performance-oriented fully routable dynamic architecture for a field programmable logic device. Technical Report UCB/ERL M93/42, EECS Department, University of California, Berkeley, 1993. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/1993/2355.html. → pages 12, 22, 23

[16] A. Boccardi, M. Gasior, R. Jones, K. Kasinski, and R. J. Steinhagen. The FPGA-based continuous FFT tune measurement system for the LHC and its test at the CERN SPS. Technical Report CERN-AB-2007-062, CERN, Geneva, 2007. → pages 1

[17] D. Brand, A. Drumm, S. Kundu, and P. Narain. Incremental synthesis. In *Proc. International Conference on Computer-Aided Design (ICCAD)*, pages 14–18, 1994. → pages 3

[18] R. Brayton and A. Mishchenko. ABC: An academic industrial-strength verification tool. In *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 24–40. 2010. URL http://dx.doi.org/10.1007/978-3-642-14295-6_5. → pages 7, 16, 173

[19] D. Carey. Under the hood: The MP3 that broke new ground, May 2007. URL http://www.eetimes.com/design/programmable-logic/4004667/Under-the-Hood-The-MP3-that-broke-new-ground. → pages 1, 2

[20] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek, and A. DeHon. Stream computations organized for reconfigurable execution (SCORE). In *Proc. Field Programmable Logic (FPL)*, pages 605–614, 2000. → pages 23, 32

[21] D. Cherepacha and D. Lewis. DP-FPGA: An FPGA architecture optimized for datapaths. *VLSI Design*, 4(4):329–343, 1996. → pages xi, 12, 23, 27

[22] K. Choi and M. Song. Design of a high performance 32x32-bit multiplier with a novel sign select booth encoder. In *Proc. International Symposium on Circuits and Systems*, pages 701–704, May 2001. → pages 45

[23] Christoph Albrecht. IWLS 2005 benchmarks. [Online]. Available: http://www.iwls.org/iwls2005/benchmarks.html, 2005. → pages 52

[24] E. G. Coffman. *Computer and Job Shop Scheduling Theory*. Wiley, New York, 1976. → pages 36

[25] J. Cong and K. Minkovich. Mapping for better than worst-case delays in LUT-based FPGA designs. In *Proc. Field-Programmable Gate Arrays (FPGA)*, pages 56–64, 2008. → pages 16

[26] D. C. Cronquist, P. Franklin, C. Fisher, M. Figueroa, and C. Ebeling. Architecture design of reconfigurable pipelined datapaths. In *Proc. Conference on Advanced Research in VLSI (ARVLSI)*, pages 23–40, 1999. → pages xi, 12, 23, 28, 29, 30

[27] A. DeHon. *Reconfigurable Architectures for General-Purpose Computing*. PhD thesis, Massachusetts Institute of Technology, 1996. → pages xi, 12, 23, 25, 26

[28] A. Donev, V. V. Bulatov, T. Oppelstrup, G. H. Gilmer, B. Sadigh, and M. H. Kalos. A first-passage kinetic Monte Carlo algorithm for complex diffusion-reaction systems. *J. Comput. Phys.*, 229(9):3214–3236, 2010. → pages 2

[29] P. Dorsey. Xilinx stacked silicon interconnect technology delivers breakthrough FPGA capacity, bandwidth, and power efficiency, Oct 2010. URL http://www.xilinx.com/support/documentation/white_papers/wp380_Stacked_Silicon_Interconnect_Technology.pdf. → pages 10, 170

[30] C. Ebeling, D. C. Cronquist, and P. Franklin. RaPiD - reconfigurable pipelined datapath. In *Proc. Field Programmable Logic (FPL)*, pages 126–135, 1996. → pages 28

[31] J. Fender and J. Rose. A high-speed ray tracing engine built on a field-programmable system. In *Proc. Field-Programmable Technology (FPT)*, pages 188 – 195, 15-17 2003. → pages 2

[32] S. Friedman, A. Carroll, B. Van Essen, B. Ylvisaker, C. Ebeling, and S. Hauck. SPR: an architecture-adaptive CGRA mapping tool. In *Proc. Field-Programmable Gate Arrays (FPGA)*, pages 191–200, 2009. → pages 6, 33, 36

[33] 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 *Proc. International Symposium on Computer Architectures (ISCA)*, pages 28–39, 1999. → pages 12, 23, 31

[34] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, R. R. Taylor, and R. Reed. PipeRench: A reconfigurable architecture and compiler. *Computer*, 33:70–77, 2000. → pages 31

[35] D. Grant and G. Lemieux. A spatial computing architecture for implementing computational circuits. In *Proc. Microsystems and Nanoelectronics Research Conference (MNRC)*, pages 41–44, Oct. 2008. → pages 52

[36] V. Granville, M. Krivanek, and J.-P. Rasson. Simulated annealing: A proof of convergence. In *Proc. Pattern Analysis and Machine Intelligence*, pages 652–656, 1994. → pages 18

[37] T. R. Halfhill. Tabula's time machine. *Microprocessor Report*, Mar. 2010. → pages 12

[38] J. Jaeger. Virtually every ASIC ends up an FPGA, Dec. 2007. URL www.eetimes.com/showArticle.jhtml?articleID=204702700. → pages 2

[39] P. Jamieson, K. B. Kent, F. Gharibian, and L. Shannon. Odin II - an open-source verilog HDL synthesis tool for CAD research. In *Proc. Field-Programmable Custom Computing Machines (FCCM)*, pages 149–156, 2010. → pages 7, 66, 173

[40] D. Jones and D. Lewis. A time-multiplexed FPGA architecture for logic emulation. In *Proc. Custom Integrated Circuits*, pages 495–498, 1995. → pages xi, 12, 22, 23, 24, 118

[41] M.-E. Jones. 1T-SRAM-Q quad-density technology reins in spiraling memory requirements. http://csserver.evansville.edu/ ˜mr56/cs838/Paper16.pdf, retrieved Sept 2010. → pages 62

[42] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: applications in VLSI domain. *IEEE Trans. VLSI*, 7(1):69–79, Mar 1999. → pages 91

[43] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983. URL http://www.ncbi.nlm.nih.gov/pubmed/17813860. → pages 17

[44] G. Krishnamurthy, E. D. Granston, and E. J. Stotzer. Affinity-based cluster assignment for unrolled loops. In *Proc. ICS*, pages 107–116, 2002. → pages 35

[45] I. Kuon and J. Rose. Measuring the gap between FPGAs and ASICs. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, 2007. ISSN 0278-0070. → pages 1, 171

[46] I. Kuon and J. Rose. Area and delay trade-offs in the circuit and architecture design of FPGAs. In *Proc. Field Programmable Gate Arrays (FPGA)*, pages 149–158, 2008. → pages 5

[47] I. Kuon, R. Tessier, and J. Rose. FPGA architecture: Survey and challenges. *Foundations and Trends in Electronic Design Automation*, 2:135–253, February 2008. ISSN 1551-3076. → pages 19

[48] S. K. Lai. Flash memories: Successes and challenges. *IBM Journal of Research and Development*, 52(4.5):529 –535, Jul. 2008. → pages 62

[49] J.-E. Lee, K. Choi, and N. D. Dutt. Compilation approach for coarse-grained reconfigurable architectures. *IEEE Des. Test*, 20(1):26–33, 2003. → pages 33

[50] W. Lee, D. Puppin, S. Swenson, and S. Amarasinghe. Convergent scheduling. In *Proc. International Symposium on Microarchitecture (MICRO)*, pages 111–122, 2002. → pages 33, 35

[51] G. Lemieux and D. Lewis. *Design of Interconnection Networks for Programmable Logic*. Kluwer Academic Publishers, 2004. → pages 36

[52] V. Manohararajah. *Area Optimizations in FPGA Architecture and CAD*. PhD thesis, University of Toronto, 2005. → pages 12, 23, 24

[53] A. Marquardt, V. Betz, and J. Rose. Timing-driven placement for FPGAs. In *Proc. Field Programmable Gate Arrays (FPGA)*, pages 203–213, 2000. → pages 19, 20, 37, 92, 95

[54] L. McMurchie and C. Ebeling. Pathfinder: A negotiation-based performance-driven router for FPGAs. In *Proc. Field-Programmable Gate Arrays (FPGA)*, pages 111–117, 1995. → pages 21

[55] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins. DRESC: a retargetable compiler for coarse-grained reconfigurable architectures. *Proc. Field-Programmable Technology (FPT)*, pages 166–173, 2002. → pages 35

[56] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 255–261, 2003. → pages 33, 35

[57] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins. ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In *Proc. Field Programmable Logic (FPL)*, pages 61–70, 2003. → pages xi, 12, 23, 30, 165

[58] Mentor Graphics. Catapult C Synthesis, 2006. URL www.mentor.com/products/esl/high_level_synthesis/catapult_synthesis/upload/Catapult_DS_pdf.pdf. → pages 2

[59] C. Mulpuri and S. Hauck. Runtime and quality tradeoffs in FPGA placement and routing. In *Proc. Field Programmable Gate Arrays (FPGA)*, pages 29–36, 2001. → pages 5, 11, 37, 38

[60] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. CACTI 6.0: A tool to model large caches. Technical Report HPL-2009-85, HP Laboratories, 2009. → pages 61

[61] E. Nystrom and A. E. Eichenberger. Effective cluster assignment for modulo scheduling. In *Proc. International Symposium on Microarchitecture (MICRO)*, pages 103–114, 1998. → pages 36

[62] H. Park. *Polymorphic Pipeline Array: A Flexible Multicore Accelerator for Mobile Multimedia Applications*. PhD thesis, The University of Michigan, 2009. → pages 8, 34, 35, 133, 137, 138, 141, 165, 166, 174

[63] H. Park, K. Fan, M. Kudlur, and S. Mahlke. Modulo graph embedding: mapping applications onto coarse-grained reconfigurable architectures. In *Proc. Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 136–146, 2006. → pages 34, 37

[64] H. Park, K. Fan, S. A. Mahlke, T. Oh, H. Kim, and H.-S. Kim. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proc. Parallel Architectures and Compilation Techniques (PACT)*, pages 166–176, 2008. → pages 35, 177

[65] J. Sánchez and A. González. Modulo scheduling for a fully-distributed clustered VLIW architecture. In *Proc. International Symposium on Microarchitecture (MICRO)*, pages 124–133, 2000. → pages 36

[66] Y. Sankar and J. Rose. Trading quality for compile time: Ultra-fast placement for FPGAs. In *Proc. Field-Programmable Gate Arrays (FPGA)*, pages 157–166, 1999. → pages 5, 11, 37

[67] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. Keckler, and C. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proc. International Symposium on Computer Architecture (ISCA)*, pages 422–433, 2003. → pages 12, 23, 32

[68] D. P. Singh and S. D. Brown. Incremental placement for layout driven optimizations on FPGAs. In *Proc. Computer-Aided Design (ICCAD)*, pages 752–759, 2002. → pages 17

[69] H. Singh, M.-H. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh, and E. Chaves Filho. MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Trans. Computers*, 49(5):465–481, May 2000. → pages 12, 23, 31

[70] G. Smecher, S. Wilton, and G. G. Lemieux. Self-hosted placement for massively parallel processor arrays. In *Proc. Field-Programmable Technology (FPT)*, pages 159–166, Dec. 2009. → pages 124

[71] W. Snyder. Verilator-3.652, June 2007. URL www.veripool.com/verilator_doc.pdf. → pages 7, 69, 173

[72] M. B. Srivastava and M. Potkonjak. Optimum and heuristic transformation techniques for simultaneous optimization of latency and throughput. *IEEE Trans. VLSI*, 3(1):2–19, 1995. → pages 52

[73] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. WaveScalar. In *Proc. International Symposium on Microarchitecture (MICRO)*, page 291, 2003. → pages 12, 23, 32

[74] W. Swartz and C. Sechen. New algorithms for the placement and routing of macro cells. In *Proc. International Conference on Computer-Aided Design (ICCAD)*, pages 336–339, 1990. → pages 18

[75] Synopsys, Inc. *Synphony C Compiler*, 2010. URL http://www.synopsys.com/Systems/BlockDesign/HLS/Pages/SynphonyC-Compiler.aspx. → pages 2

[76] Tabula, Inc. *ABAX Product Family Overview*, 2010. → pages 23, 26

[77] Tabula, Inc. *Stylus Software Overview*, 2010. → pages 26

[78] Texas Instruments. *TMS320C6457 Communications Infrastructure Digital Signal Processor Data Manual*, 2010. URL http://www.ti.com/lit/gpn/tms320c6a8167. → pages 46

[79] X. Tian and K. Benkrid. American option pricing on reconfigurable hardware using least-squares Monte Carlo method. In *Proc. Field-Programmable Technology (FPT)*, pages 263 –270, 9-11 2009. → pages 2

[80] S. Trimberger. Moore's law, fpgas and computation. [Online]. Available:http://www.skatelescope.org/US_SKA_Technology_Day06/Trimberger.pdf, Mar 2006. → pages 1, 10, 170

[81] University of Massachusetts. Umass RCG HDL benchmark collection. [Online]. Available: http://www.ecs.umass.edu/ece/tessier/rcg/benchmarks, 2006. → pages 52

[82] University of Toronto. iFAR - intelligent FPGA architecture repository. [Online]. Available: http://www.eecg.utoronto.ca/vpr/architectures, 2008. → pages 97, 103

[83] B. C. Van Essen. *Improving the Energy Efficiency of Coarse-Grained Reconfigurable Arrays*. PhD thesis, The University of Washington, 2010. → pages 12, 23, 31, 36

[84] B. C. Van Essen, R. Panda, A. Wood, C. Ebeling, and S. Hauck. Energy-efficient specialization of functional units in a coarse-grained reconfigurable array. In *Proc. Field Programmable Gate Arrays (FPGA)*, pages 107–110, 2011. → pages 11

[85] G. Venkataramani, W. Najjar, F. Kurdahi, N. Bagherzadeh, and W. Bohm. A compiler framework for mapping applications to a coarse-grained reconfigurable computer architecture. In *Proc. Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 116–125, 2001. → pages 35

[86] K. Vorwerk. *On the Use of Directed Moves for Placement in VLSI CAD*. PhD thesis, The University of Waterloo, 2009. → pages 17

[87] C. C. Wang and G. G. Lemieux. Scalable and deterministic timing-driven parallel placement for FPGAs. In *Proc. Field Programmable Gate Arrays (FPGA)*, pages 153–162, 2011. → pages 124, 195

[88] S. J. E. Wilton, C. H. Ho, P. H. W. Leong, W. Luk, and B. Quinton. A synthesizable datapath-oriented embedded FPGA fabric. In *Proc. Field Programmable Gate Arrays (FPGA)*, pages 33–41, 2007. → pages 12, 23, 27

[89] M. G. Wrighton and A. M. DeHon. Hardware-assisted simulated annealing with application to fast FPGA placement. → pages 37, 206

[90] Xilinx. *7 Series FPGA Overview*, Mar 2011. URL http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf. → pages 1

[91] Xilinx, Inc. *EDK Concepts, Tools, and Techniques*, 2008. URL www.xilinx.com/support/documentation/sw_manuals/edk_ctt.pdf. → pages 2

[92] Xilinx, Inc. *ISE Design Suite Software Manuals and Help*, 2010. URL http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_3/manuals.pdf. → pages 16

[93] A. Yan and S. J. E. Wilton. Product-Term Based Synthesizable Embedded Programmable Logic Core. *IEEE Trans. VLSI*, 14(5):474–488, 2006. → pages 28

[94] A. Ye and J. Rose. Using bus-based connections to improve field-programmable gate array density for implementing datapath circuits. In *IEEE Trans. VLSI*, pages 3–13, 2005. → pages 12, 23, 27

[95] J. W. Yoon, A. Shrivastava, S. Park, M. Ahn, R. Jeyapaul, and Y. Paek. SPKM: A novel graph drawing based algorithm for application mapping onto coarse-grained reconfigurable architectures. In *Proc. Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 776–782, 2008. → pages 33, 34

[96] V. Zygouris, K. Karagianni, and T. Stouraitis. A Navier-Stokes processor for biomedical applications. In *Proc. Signal Processing Systems (SiPS)*, pages 368 – 372, 2-4 2005. → pages 2

Appendix A

Scaling Existing FPGA CAD Tools

A.1 Overview

This appendix presents an investigation into reducing the compile times in VPR and QuartusII. This investigation is to study whether existing tools can be made as fast as the M-CAD and M-HOT tools presented in this thesis. For QuartusII the compile time can be reduced by only 15% with a 13% loss in quality. For VPR the compile time can be reduced by 7% with a 25% loss in quality for about half the circuits (the other half are unroutable at this point). At this limit, the M-CAD and M-HOT approaches are still over 11x and 4.8x faster respectively.

As was done in the body of this thesis, all comparisons between the Malibu tools (M-CAD and M-HOT) to QuartusII and VPR exclude the *Impaired* benchmarks.

A.2 QuartusII

All the results in this thesis from the commercial QuartusII tool were generated using the default QuartusII settings. For the *CG-only* benchmark results in Chapters 5 and 6, M-CAD and M-HOT achieved a higher maximum frequency (geomean) result than QuartusII synthesizing for an StratixIII FPGA. However, for the *Good* benchmarks, which include some fine-grained

signals, the QuartusII frequency was up to 7.1x better than the result from the Malibu tools. For the *Impaired* circuits, the difference was even greater, favouring QuartusII.

The main advantage of Malibu CAD tools is very fast compile times, as shown in Tables 5.9 and 6.9. M-CAD is 16.9x to 57.5x (geomean) faster than QuartusII for the *CG-only* and *Good* benchmarks. M-HOT is 18.0x to 26.7x faster than QuartusII for the same benchmarks. In this section we investigate whether is possible for QuartusII to match the compile times of M-CAD or M-HOT.

The compile time in QuartusII is decreased in three ways:

1. **Lower Front-End Synthesis Effort** – The following setting can be used to disable many of the optimizations applied during front-end synthesis:

   ```
   set_global_assignment -name SYNTHESIS_EFFORT FAST
   ```

2. **Lower Placement and Routing Effort** – The following settings were used to speedup the placement and routing effort. Optimizations are turned off, timing-driven placement is disabled, and the router effort is set to the lowest permissible value (0.25).

   ```
   set_global_assignment -name SYNTH_TIMING_DRIVEN_SYNTHESIS OFF
   set_global_assignment -name ROUTER_EFFORT_MULTIPLIER 0.25
   set_global_assignment -name OPTIMIZE_HOLD_TIMING OFF
   set_global_assignment -name FITTER_EFFORT "FAST FIT"
   set_global_assignment -name FINAL_PLACEMENT_OPTIMIZATION NEVER
   set_global_assignment -name FITTER_AGGRESSIVE_ROUTABILITY_OPTIMIZATION
   NEVER
   ```

3. **Reduce the Placement Effort Multiplier** – The placement effort multiplier is the same as the inner-loop multiplier in VPR annealing algorithm. Reducing this value decreases the time spent in the inner-loop of the QuartusII annealer and speeds up placement. However, speeding up placement results in a poorer placement, and that means router has to work harder to find a solution. The default placement effort multiplier is 1.

```
set_global_assignment -name PLACEMENT_EFFORT_MULTIPLIER 1
... down to ...
set_global_assignment -name PLACEMENT_EFFORT_MULTIPLIER 0.00001
```

Table A.1 shows the result of changing only the front-end synthesis effort in QuartusII. "Reg" columns are results where the placement and routing effort is lowered, and the placement effort multiplier is set to 1. The "Fast" columns are results with same settings but with the front-end synthesis effort lowered as well. Enabling fast front-end synthesis reduces the front-end synthesis time by about 25%, but increases the placement and routing time by 2%. Overall there is less than a 3% reduction in total compile time, a 1.4% reduction in frequency, and a 1.6% increase in area.

Since the overall impact of setting is small, and to be consistent with the QuartusII front-end synthesis used for VPR, we have chosen to not lower the front-end synthesis effort for the results in this section and in the next section with VPR. As we will demonstrate in this section, a further 3% reduction in compile time is negligible when compared to M-CAD and M-HOT.

Figure A.1 shows the geometric mean of the maximum frequency and compile time for a sweep of the QuartusII placement effort multiplier. The rightmost point on the x-axis is the *default* value where QuartusII is run with the default settings. This is the QuartusII data used throughout this thesis. For the other data points along the x-axis, the two methods above are used to speed up the synthesis. The individual graphs for each benchmark are in Section A.5.

Disabling optimizations (default vs. 1 on the x-axis) causes about a 15% reduction in compile time for placement effort multiplier values down to 0.01 compared to the default settings. Across this range, there is also up to a 13% decrease in F_{max}. Below 0.01, the compile time increases, presumably because the routing is taking longer. Unfortunately, QuartusII does not give accurate separate runtime statistics for placement and routing (however in the next section we can see that routing time increases as placement time decreases for VPR). At the far left of the graph, with a 0.00001 multiplier, there is a 67% reduction in F_{max} and the compile time

Table A.1: QuartusII results with fast front-end synthesis optimization. Enabling fast front-end synthesis reduces the quality of the circuit, so front-end synthesis is faster, but place and route is slower. Also with fast front-end synthesis, the frequency is decreased and area is increased.

		Front-End Synthesis (s)			Place and Route (s)			Front-End + P&R	Frequency (MHz)			Area (ALMs)		
	Circuit	**Reg**	**Fast**	**Fast/Reg**	**Reg**	**Fast**	**Fast/Reg**	**Fast/Reg**	**Reg**	**Fast**	**Fast/Reg**	**Reg**	**Fast**	**Fast/Reg**
CG-only	fft16	26	22	0.857	126	130	1.029	0.999	143.0	156.3	1.093	4558	4561	1.001
	me	18	14	0.786	96	97	1.013	0.977	203.8	203.8	1.000	5575	5575	1.000
	chem	23	23	1.000	182	186	1.020	1.017	12.0	12.0	1.000	3589	3589	1.000
	fft8	10	8	0.778	73	78	1.065	1.030	166.3	163.8	0.985	1715	1719	1.002
	honda	19	19	1.000	55	56	1.021	1.016	18.3	12.0	0.659	1216	1251	1.029
	mcm	12	12	1.000	57	57	1.000	1.000	23.3	24.8	1.065	1057	1008	0.954
	wang	12	12	1.000	53	54	1.022	1.018	19.3	19.9	1.034	797	789	0.990
	pr	9	9	1.000	53	54	1.022	1.019	23.7	23.1	0.975	646	641	0.992
	Geo. Mean (CG-only)			**0.922**			**1.024**	**1.009**			**0.966**			**0.996**
Good	ac97_ctrl	9	8	0.857	60	60	1.000	0.981	236.2	243.6	1.031	1275	1287	1.009
	aes_core	16	5	0.286	64	70	1.089	0.929	185.0	187.9	1.016	1156	1157	1.001
	dir	22	13	0.611	75	81	1.077	0.971	79.7	78.7	0.987	1326	1523	1.149
	spi	7	5	0.667	51	51	1.000	0.960	124.1	115.9	0.934	497	531	1.068
	pci_master	5	5	1.000	50	49	0.977	0.979	211.8	218.4	1.031	134	133	0.993
	Geo. Mean (Good)			**0.631**			**1.028**	**0.964**			**0.999**			**1.042**
Geo. Mean (CG-only and Good)				**0.797**			**1.025**	**0.992**			**0.979**			**1.013**
Impaired	ethernet	57	42	0.744	152	154	1.015	0.941	152.7	156.1	1.022	7194	7378	1.026
	wb_conmax													
	dma	23	16	0.684	87	82	0.947	0.892	81.2	94.8	1.168	1813	1776	0.980
	tv80	25	13	0.522	55	56	1.021	0.865	107.1	97.8	0.913	886	943	1.064
	jpeg_enc	17	17	1.000	57	58	1.021	1.016	182.9	180.1	0.985	1185	1218	1.028
	systemcaes	23	13	0.579	65	65	1.000	0.890	116.5	115.8	0.994	811	925	1.141
	des	10	6	0.625	63	64	1.019	0.965	247.8	247.8	1.000	1316	1316	1.000
	systemcdes	7	5	0.667	52	54	1.047	1.001	172.2	159.2	0.924	340	315	0.926
	Geo. Mean (Impaired)			**0.675**			**1.010**	**0.937**			**0.998**			**1.022**
Geo. Mean (All)				**0.752**			**1.020**	**0.972**			**0.986**			**1.016**

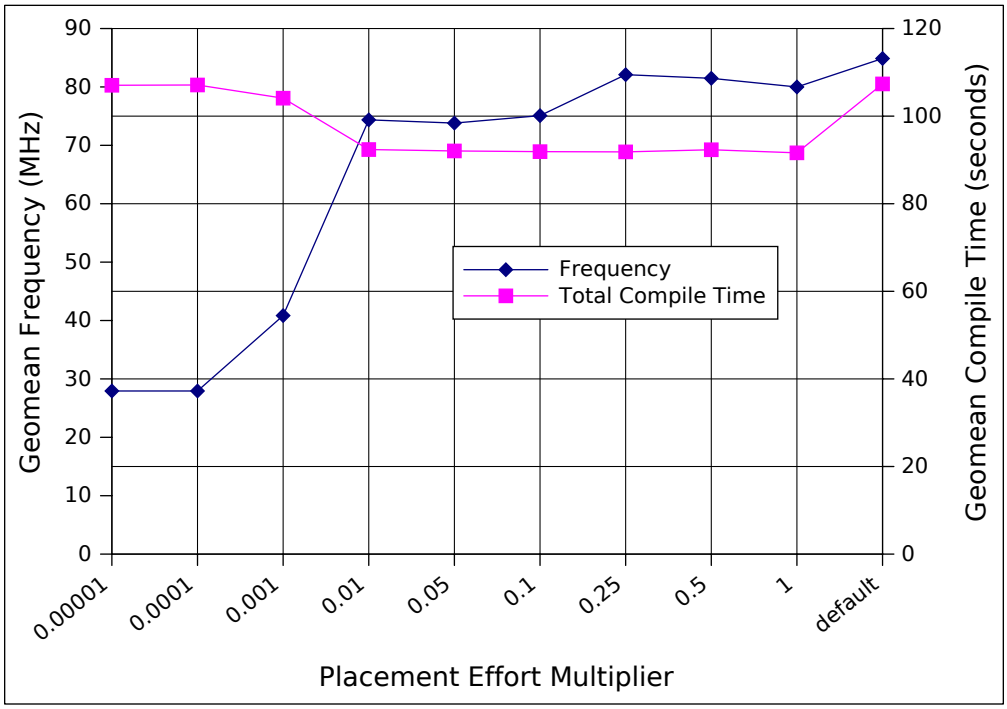


Figure A.1: QuartusII compile time and frequency for various values of the placement effort multiplier.

increases almost to the original value.

The area of these results (number of ALMs) reported by QuartusII is unchanged. Regardless of the optimizations on placement and routing, the number of logic entities is the same. This is because we have chosen to leave the front-end synthesis optimizations at their default settings.

Compared to QuartusII run with the default settings, it is possible to speed up QuartusII by 15% at most. Also compared to QuartusII with the default settings, the worst M-CAD speedup is 16.9x ($W_f = 4$, excluding the *Impaired* benchmarks, Table 5.9), and the worst M-HOT speedup of 18x ($W_f = 0$, excluding the *Imparied* benchmarks, Table 6.9). The Malibu approaches offer significant compile time savings.

A.3 VPR

This section looks at the effect of reducing the compile time in VPR. To reduce compile time, the "-fast" option is used with both the VPR placer and router. This option sets the following five parameters:

- Reduces *bb_factor* to 0 (default = 3). This parameter allows the router to use channels outside the bounding box. The bounding box is defined by the source/sink pin locations. Larger values cause the router to be slower, but perform a more extensive search for possible routes [1].

- Reduces *max_router_iterations* to 10 (default = 50). This parameter specifies the number of iterations the router will try before giving up and returning that the circuit is unroutable. This option does not speed up finding a successful routing solution [1].

- Increases *first_iter_pres_fac* to 10000 (default = 0.5). This parameter sets the penalty for overusing routing channels for the first routing iteration. Larger values will clear overused resources faster and cause the router to run faster, but will use more tracks in the final solution [1].

- Increases *initial_pres_fac* to 10000 (default = 0.5). This parameter sets the penalty for overusing routing channels for the second routing iteration. Larger values will clear overused resources faster and cause the router to run faster, but will use more tracks in the final solution [1].

- Reduces the *inner_num* multiplier to 1 (default = 10). This parameter controls the number of moves at each annealing temperature in placement $\left(moves = inner\_num \times num\_blocks^{\frac{4}{3}}\right)$. Smaller values cause the placer to run faster at the expense of placement quality [1].

In addition to using -fast, the *inner_num* is further decreased to speed up placement even more. Each benchmark was placed and routed using the same iFAR architecture file (n10k04l04.fc15.area1delay1.cmos65nm) as all the other VPR results in this thesis. The number of channels was set to 100 in all cases too, as recommended by the architecture file. So the density results are the same as those reported in Chapters 5 and 6.

Decreasing the *inner_num* reduces the placement quality and that, in turn, increases routing time. Figure A.2 shows this trend. On the x-axis is a sweep of the *inner_num* down to 0.05. About half the circuits failed to route with *inner_num*=0.01, and all the circuits except *jpeg_enc*, *spi*, and *pci_master* failed to route with *inner_num*=0.001. No circuits successfully routed with *inner_num* below 0.001. The *chem* benchmark failed to route with any value of *inner_num* below 1, so it has been excluded from all calculations in this section. The geometric mean reported in Figure A.2 is computed for all benchmarks (except chem), which is why it does not extend below 0.05. The same placement and routing trend can be seen in the individual graphs of each benchmark, which are in Section A.6.

Figure A.3 plots only the benchmarks which built at *inner_num*=0.01, and the trend is the same; as placement time decreases with smaller *inner_num* values, routing time increases. This figure also shows the rapid loss in frequency as *inner_num* reaches low values, with no reduction in compile time. There is a 7% reduction in compile time for a 25% loss in quality. This trend is also visible in the individual graphs of each benchmark (Section A.6).

To better compare the compile time of M-CAD and M-HOT versus VPR, Table A.2 shows the minimum VPR compile time for each benchmark circuit, the value of *inner_num* where minimum compile time is achieved, and the frequency result at minimum synthesis time. It then shows the compile time and performance compared to M-CAD and M-HOT. For the compile time, the value is the Malibu compile time speedup computed by taking the VPR compile time column and dividing it by the values in Table 5.9 for M-CAD or Table 6.9 for M-HOT. Thus, a value of 2.00 means that Malibu synthesis is 2x faster than VPR. For the frequency results,

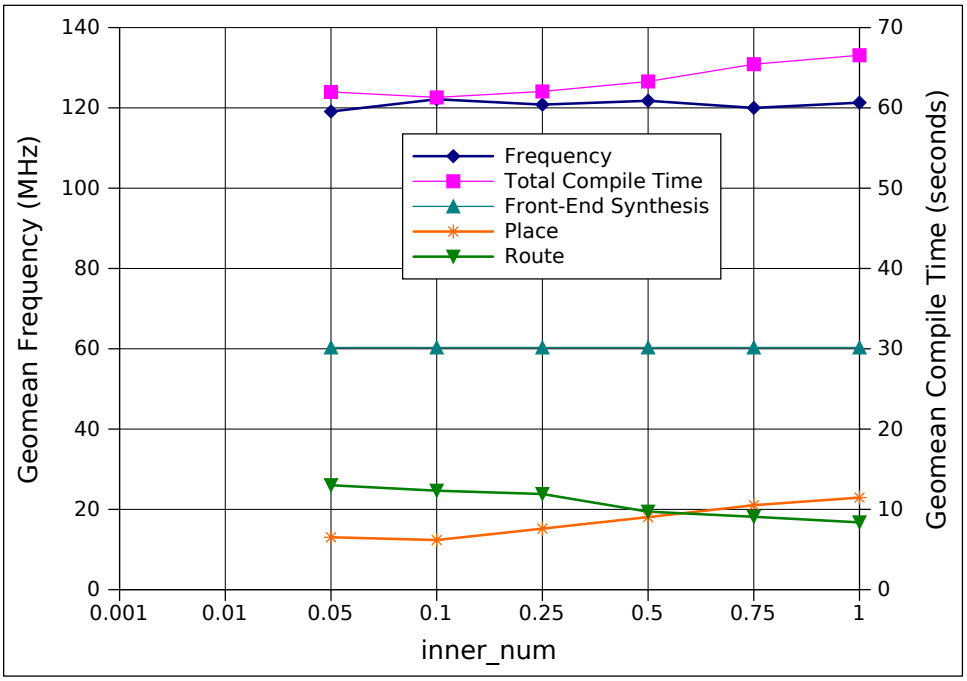


Figure A.2: VPR compile time and frequency versus *inner_num* for all benchmarks. As *inner_num* is decreased, placement time decreases, but routing time increases due to reduced placement quality.

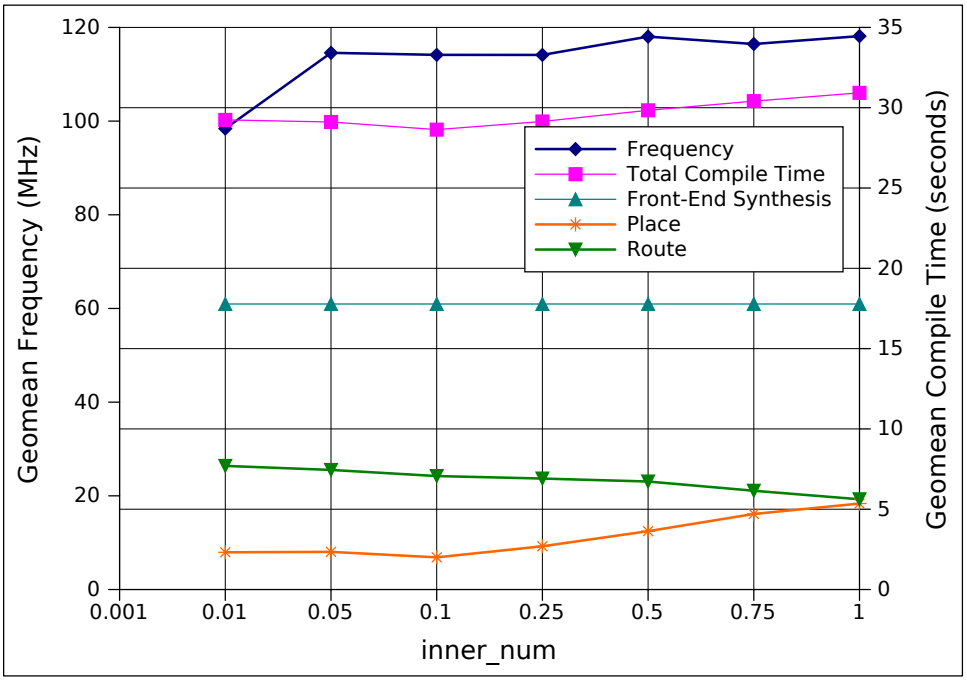


Figure A.3: VPR compile time and frequency versus *inner_num* for benchmarks which build at *inner_num*=0.01. Benchmarks: *dir*,*honda*, *des*, *pci_master*, *spi*, *system-cdes*, *jpeg_enc*, *mcm*, *pr*, and *wang*. The same placement and routing trend as Figure A.2 is visible.

Tables 5.1 and 6.1 are used, and a value of 2.00 means that the Malibu circuit is 2x faster than the one produced by VPR.

The results for the *CG-only* and *Good* benchmarks show that M-CAD is over 11x faster than VPR and achieving about half the performance. Comparing these results to Table 5.10 (compile time of M-CAD versus VPR-fast), Malibu was 17.97x faster than VPR and is now just over 15x faster for $W_f = 0$. The frequency results in Table 5.2 (frequency of M-CAD versus VPR-fast) show that the quality has stayed about the same.

The M-HOT results are similar to the M-CAD results, although the change in compile time is not as drastic (see Tables 6.2 and 6.10). M-HOT is still faster than VPR, and the performance is about the same.

Other methods of reducing placement time exist, such as parallel placement [87]. However, the front-end synthesis time (QuartusII BLIF generation) and routing dominate the compile time; so even if placement time could be reduced to zero, that would only speed up VPR by less than 2x (see Figure A.2). Because of this, the Malibu approach would still be faster.

A.4 Conclusions

It is possible to speed up both QuartusII and VPR to achieve faster synthesis and reduced quality. However, the compile time cannot be reduced to the times reported by M-CAD or M-HOT. Part of the reason is the interplay between placement and routing: as placement time decreases, the quality of the result also decreases, and that increases routing time. If the placement quality is reduced too much, the router simply fails. Even if placement time could be reduced to zero, the front-end synthesis and routing time would still exist, and the Malibu approach would still be faster.

There are likely more options for both tools which could be explored in future work to further reduce compile times. However, we feel that the results in this appendix demonstrate the virtues the Malibu coarse-grained synthesis approach and the underlying coarse-grained archi-

Table A.2: M-CAD and M-HOT results compared to the fastest VPR compile time. The *inner_num* parameter was decreased to find the fastest VPR synthesis (where the circuit still routed).

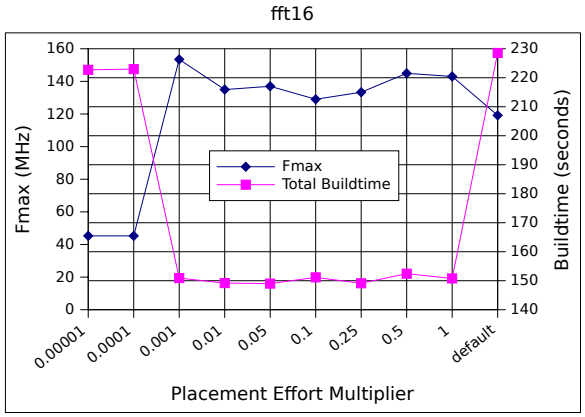
		Fastest VPR Compile Time			M-CAD						M-HOT					
					Compile Time Speedup			Performance Factor			Compile Time Speedup			Performance Factor		
	Circuit	*inner_num*	Synth. Time (s)	Freq. (MHz)	$W_f = 0$	$W_f = 1$	$W_f = 4$	$W_f = 0$	$W_f = 1$	$W_f = 4$	$W_f = 0$	$W_f = 1$	$W_f = 4$	$W_f = 0$	$W_f = 1$	$W_f = 4$
CG-only	fft16	0.05	541.6	104.7	98.07	–	–	0.43	–	–	7.08	–	–	0.33	–	–
	me chem	0.25	142.7	71.8	10.54	–	–	0.77	–	–	0.37	–	–	0.36	–	–
	fft8	0.25	93.9	118.0	32.46	–	–	0.61	–	–	11.23	–	–	0.47	–	–
	honda	0.1	44.9	57.4	67.65	–	–	0.79	–	–	74.13	–	–	0.97	–	–
	mcm	0.1	33.6	88.9	43.93	–	–	0.80	–	–	43.06	–	–	0.94	–	–
	wang	0.05	26.9	71.8	2.53	–	–	1.16	–	–	2.55	–	–	1.55	–	–
	pr	0.1	23.7	85.3	18.42	–	–	0.98	–	–	27.56	–	–	1.17	–	–
	Geo. Mean (CG-only)				**24.05**	**24.05**	**24.05**	**0.76**	**0.76**	**0.76**	**9.41**	**9.41**	**9.41**	**0.71**	**0.71**	**0.71**
Good	ac97_ctrl	0.1	26.0	306.4	1.86	0.54	0.41	0.05	0.13	0.13	0.11	0.69	0.76	0.06	0.14	0.17
	aes_core	0.1	159.8	174.0	26.92	13.08	16.18	0.16	0.14	0.15	3.02	8.83	8.47	0.17	0.17	0.17
	dir	0.1	104.8	57.3	43.30	12.43	18.30	0.51	0.47	0.47	15.62	31.41	34.53	0.58	0.60	0.65
	spi	1	18.0	144.0	4.61	5.56	5.36	0.18	0.22	0.27	3.31	3.86	3.75	0.19	0.37	0.37
	pci_master	0.001	10.9	154.3	5.48	1.62	1.91	0.15	0.19	0.50	1.36	4.85	4.58	0.19	0.26	0.81
	Geo. Mean (Good)				**8.87**	**3.80**	**4.15**	**0.16**	**0.21**	**0.26**	**1.87**	**5.13**	**5.21**	**0.18**	**0.27**	**0.35**
Geo. Mean (CG-only and Good)					**15.87**	**11.15**	**11.57**	**0.40**	**0.44**	**0.49**	**4.80**	**7.31**	**7.35**	**0.40**	**0.47**	**0.53**
Impaired	ethernet	0.1	226.0	107.4	3.28	0.42	0.42	0.06	0.11	0.19	0.06	0.54	0.60	0.09	0.12	0.30
	wb_conmax	0.5	1069.7	72.3	11.08	0.05	0.11	0.18	0.27	0.25	0.10	0.04	0.55	0.18	0.31	0.28
	dma	0.25	75.4	133.6	0.48	0.04	0.010	0.04	0.08	0.08	0.007	0.02	0.009	0.09	0.12	0.13
	tv80	0.25	46.4	112.0	2.92	0.06	0.06	0.06	0.04	0.57	0.03	0.33	0.59	0.04	0.05	0.30
	jpeg_enc	0.1	34.9	159.2	0.38	0.25	0.28	0.03	0.03	0.03	0.13	0.56	0.36	0.15	0.18	0.17
	systemcaes	0.05	33.6	153.5	0.28	2.05	0.02	0.10	0.10	0.10	0.18	1.37	0.02	0.20	0.26	0.33
	des	0.25	22.9	162.2	2.96	0.11	0.10	0.02	0.03	0.03	0.25	1.15	1.17	0.04	0.05	0.06
	systemcdes	0.01	24.7	191.7	10.19	1.74	1.48	0.11	0.11	0.10	1.66	3.98	3.64	0.13	0.14	0.16
	Geo. Mean (Impaired)				**1.89**	**0.21**	**0.11**	**0.06**	**0.08**	**0.11**	**0.10**	**0.39**	**0.30**	**0.10**	**0.13**	**0.19**
Geo. Mean (All)					**6.77**	**2.28**	**1.80**	**0.19**	**0.22**	**0.27**	**1.02**	**2.27**	**2.04**	**0.23**	**0.28**	**0.35**

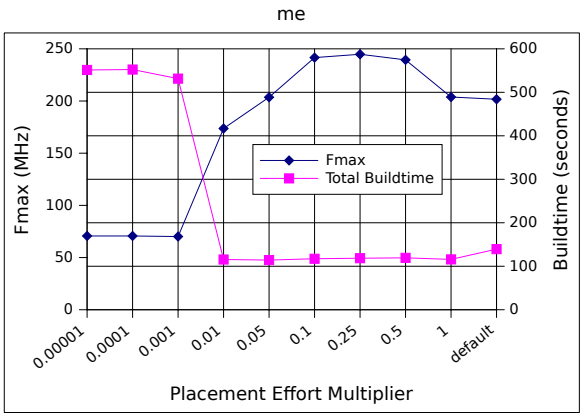
tecture. For coarse-grained circuits (which are the likely candidates for synthesis on Malibu) the Malibu approach is faster than available academic and commercial tools, and has reasonably good quality given the compile time.

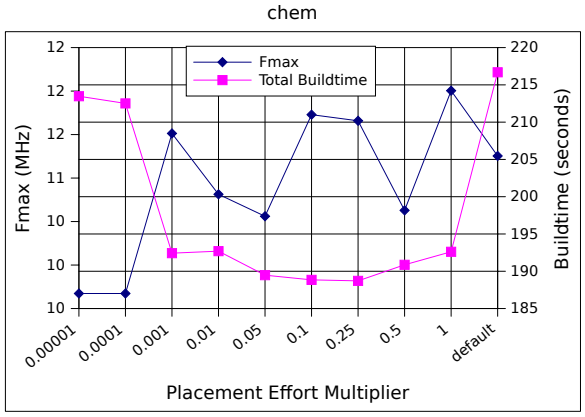
A.5 Individual QuartusII Graphs

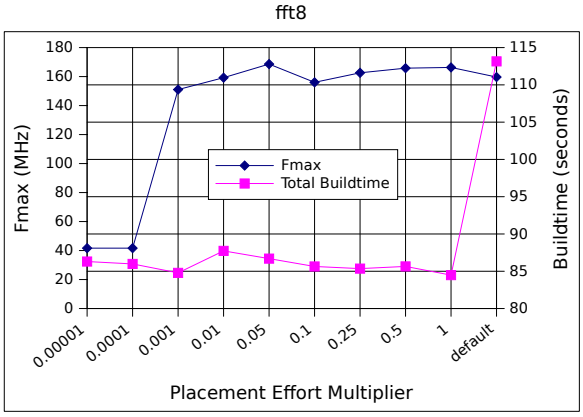
The graphs in this section were used to generate Figure A.1. For each graph, the default QuartusII settings are on the right, and the placement effort multiplier is swept, with optimizations disabled, from 1 to 0.00001.

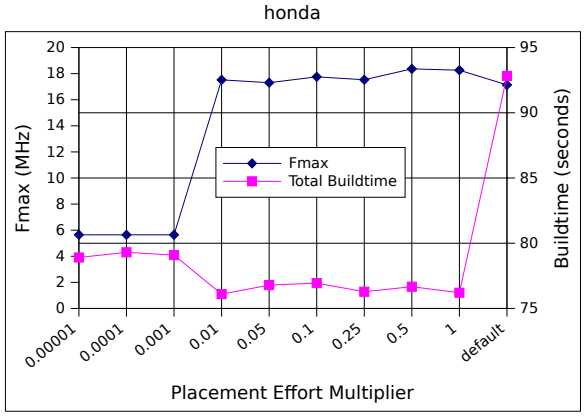
CG-only Benchmarks

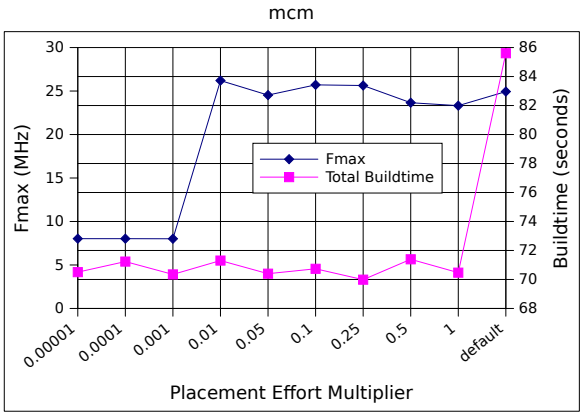


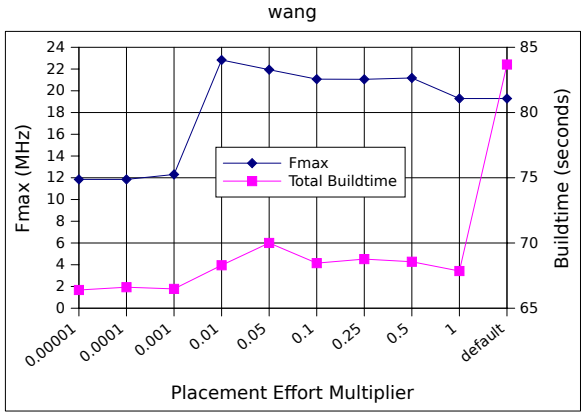


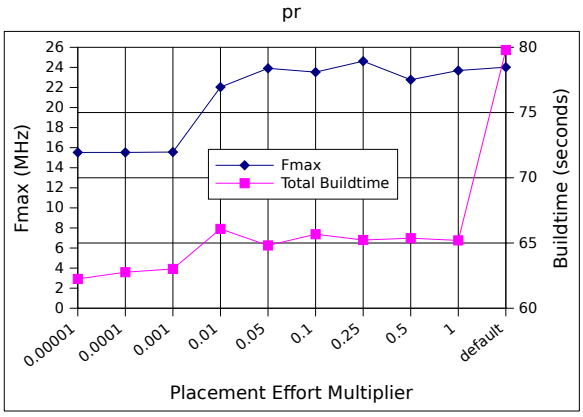




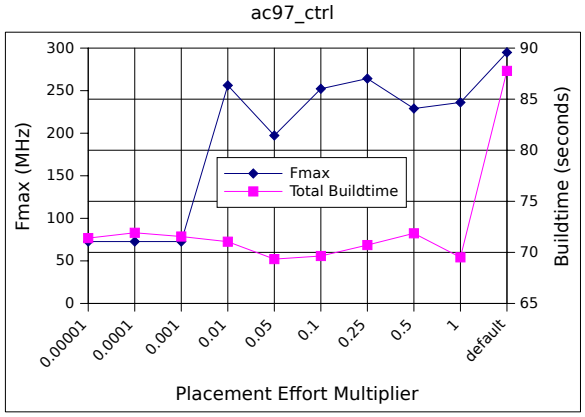


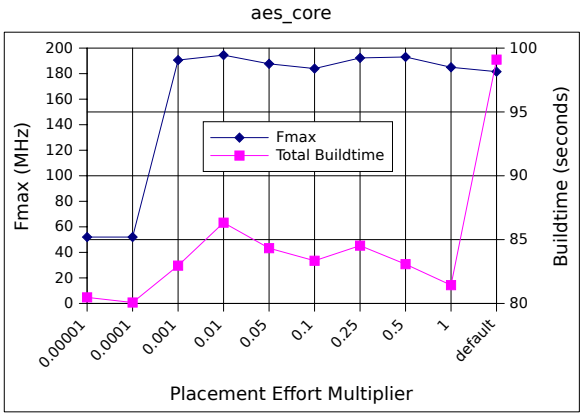


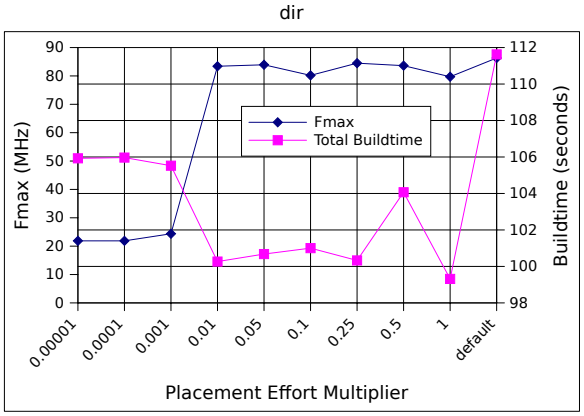


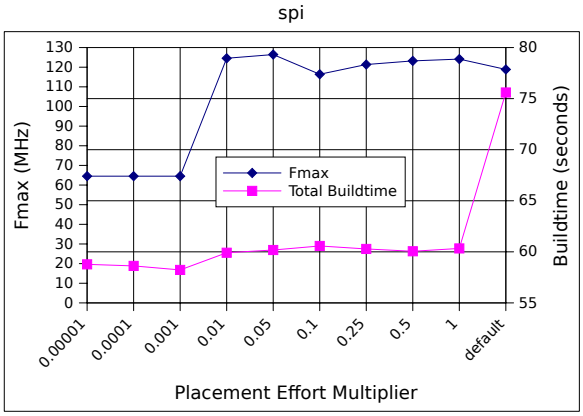


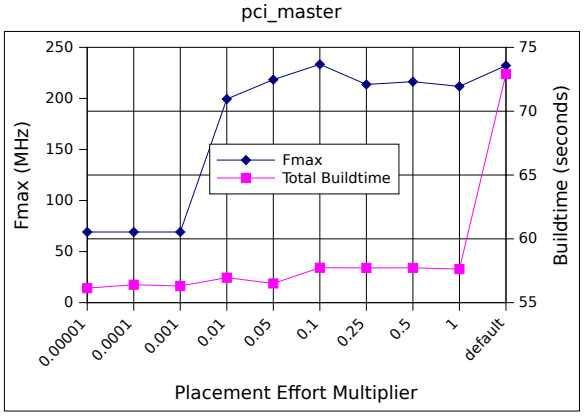
Good Benchmarks



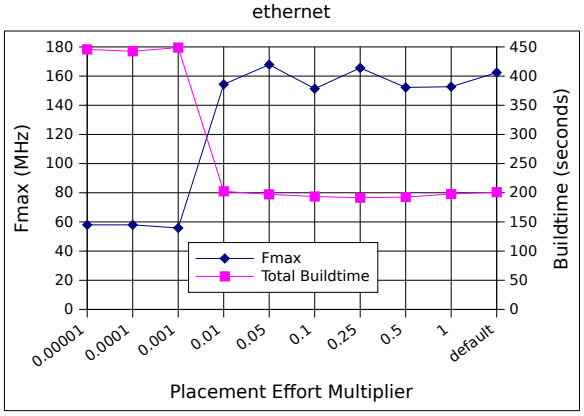




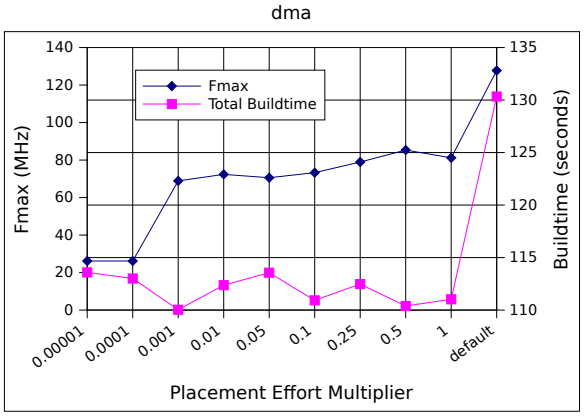


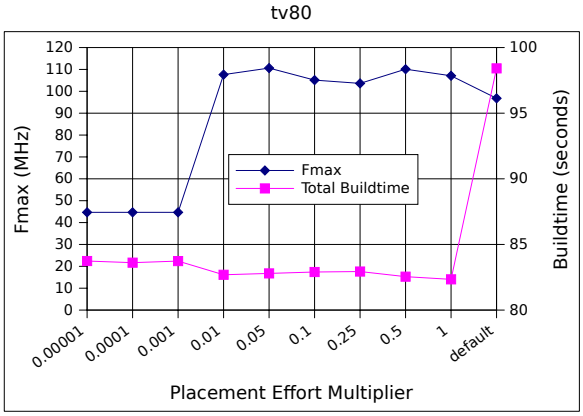


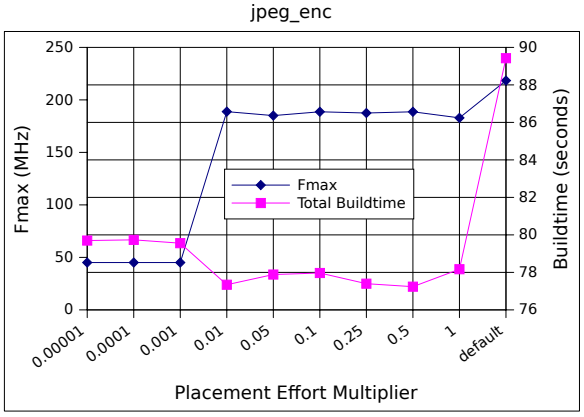
Impaired Benchmarks

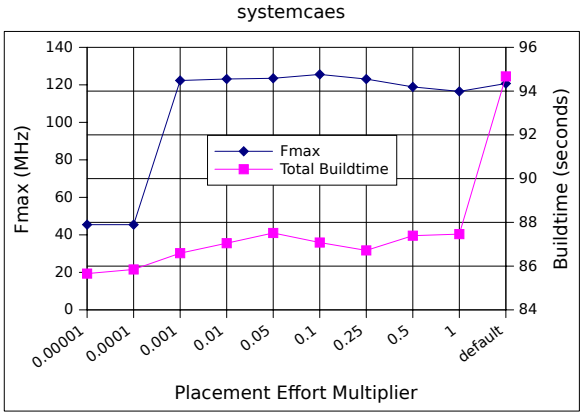


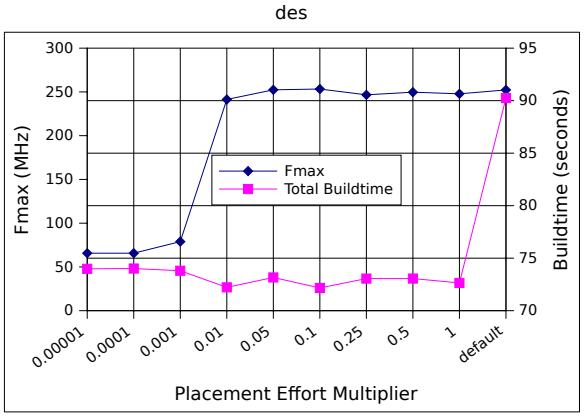
wb_conmax failed to build with any optimizations enabled, so it has been excluded from this analysis.

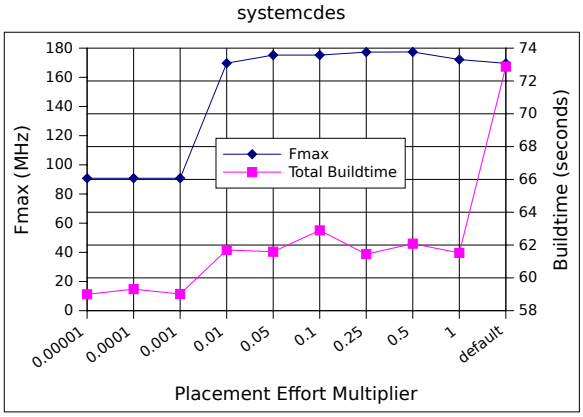

dma
Fmax
Total Buildtime
Fmax (MHz)
Buildtime (seconds)
Placement Effort Multiplier
0.00001
0.0001
0.001
0.01
0.05
0.1
0.25
0.5
1
default


tv80
Fmax
Total Buildtime
Fmax (MHz)
Buildtime (seconds)
Placement Effort Multiplier
0.00001
0.0001
0.001
0.01
0.05
0.1
0.25
0.5
1
default


jpeg_enc
Fmax
Total Buildtime
Fmax (MHz)
Buildtime (seconds)
Placement Effort Multiplier
0.00001
0.0001
0.001
0.01
0.05
0.1
0.25
0.5
1
default


systemcaes
Fmax
Total Buildtime
Fmax (MHz)
Buildtime (seconds)
Placement Effort Multiplier
0.00001
0.0001
0.001
0.01
0.05
0.1
0.25
0.5
1
default


des
Fmax
Total Buildtime
Fmax (MHz)
Buildtime (seconds)
Placement Effort Multiplier
0.00001
0.0001
0.001
0.01
0.05
0.1
0.25
0.5
1
default


systemcdes
Fmax
Total Buildtime
Fmax (MHz)
Buildtime (seconds)
Placement Effort Multiplier
0.00001
0.0001
0.001
0.01
0.05
0.1
0.25
0.5
1
default

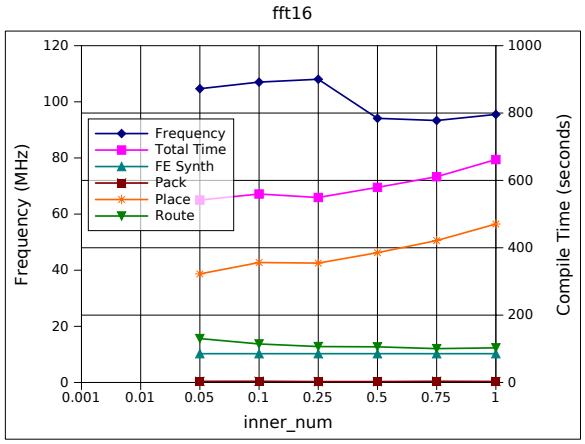
A.6 Individual VPR Graphs

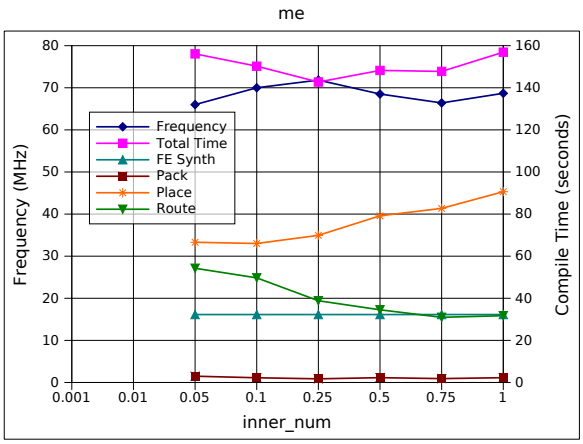
The data from the graphs in this section was used to generate Figure A.2 and Figure A.3. For each graph, the VPR -fast option was used for both the placer and the router. On the x-axis, *inner_num* is swept from 1 (the value used in the thesis, and the default setting for VPR-fast), down to 0.001. The curves stops on each graph where the circuit failed to route. About half the circuits failed to route with *inner_num*=0.01, and all the circuits except *jpeg_enc*, *spi*, and *pci_master* failed to route with *inner_num*=0.001.

Of note on these graphs is that:

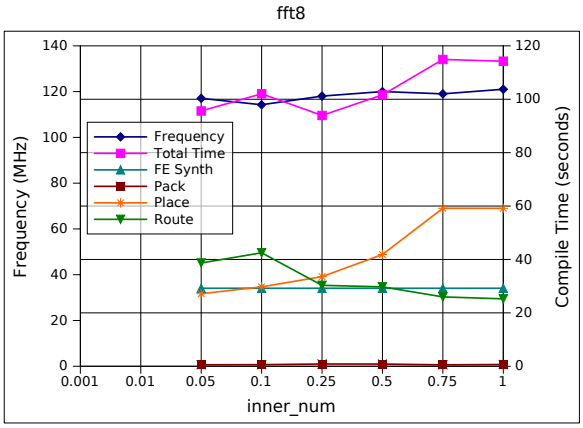
- As placement time decreases the routing time increases. Overall, there is a small reduction in the total compile time.
- On the left of these graphs (low values of inner_num) routing dominates the compile time.
- The frequency falls rapidly for low values of *inner_num*, without any further decrease in compile time.

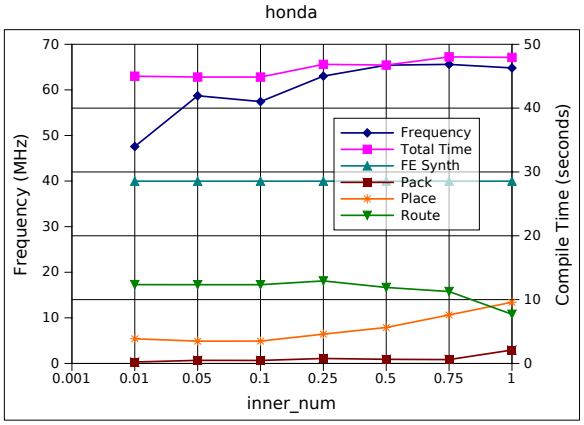
CG-only Benchmarks

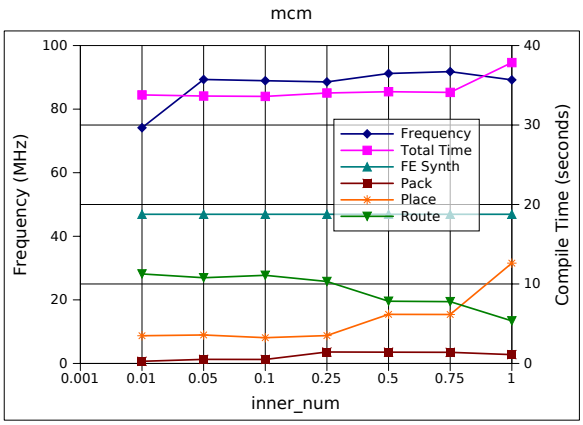


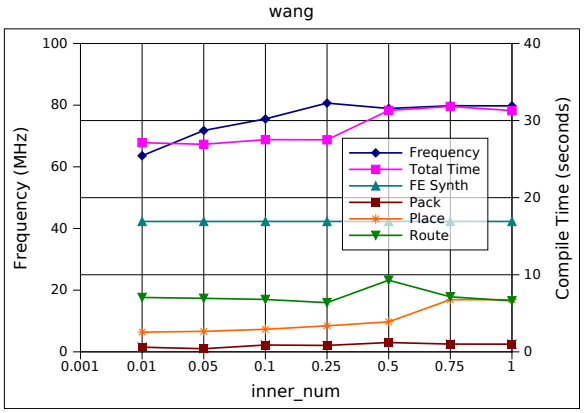


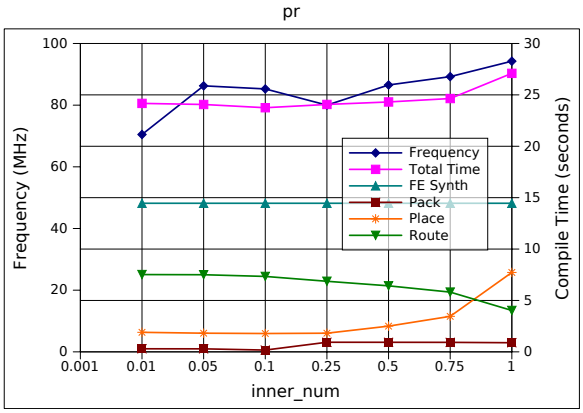
chem failed to synthesize values of inner_num < 1 so it has been excluded from this analysis.



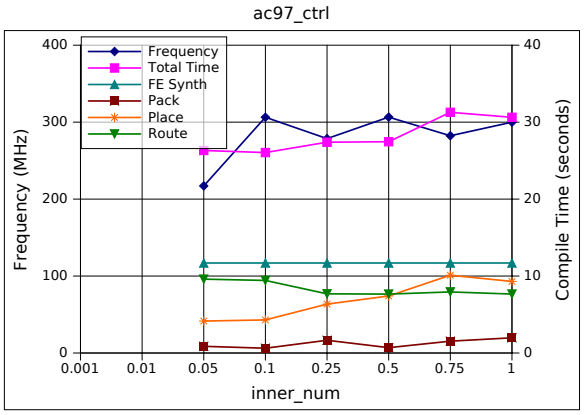


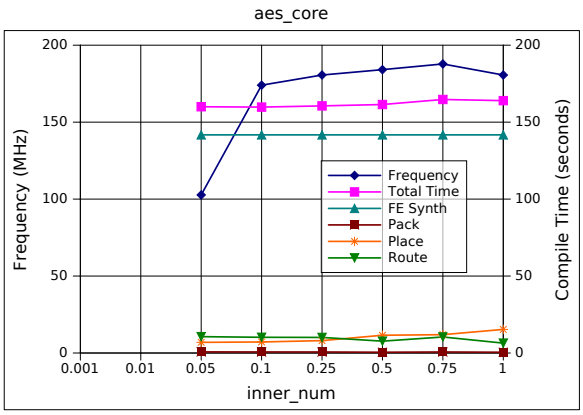


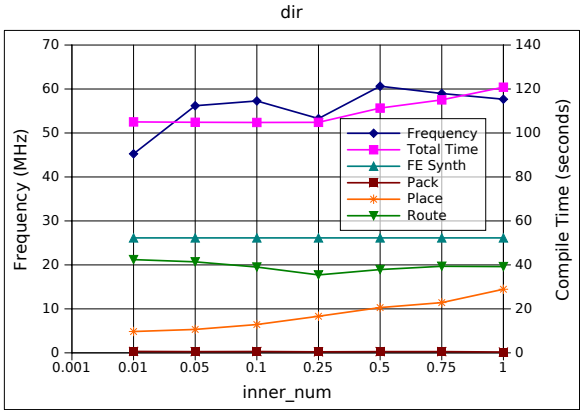


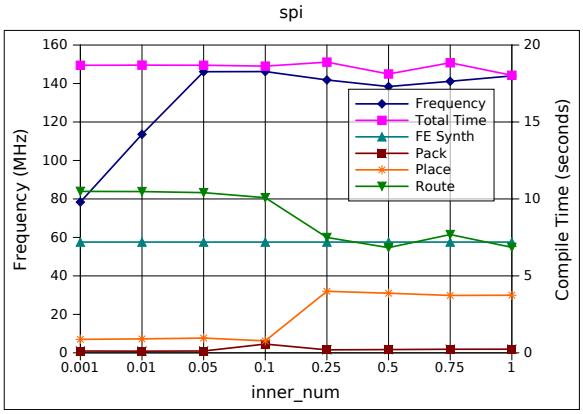


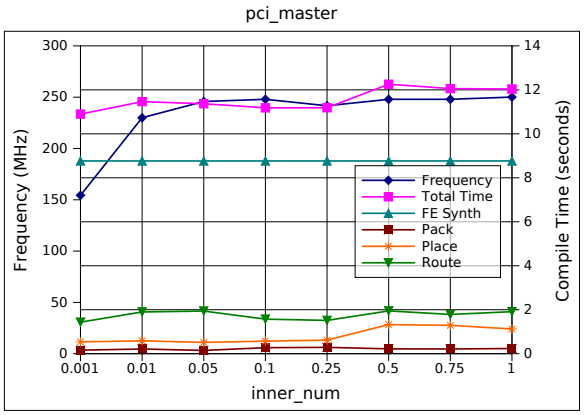
Good Benchmarks



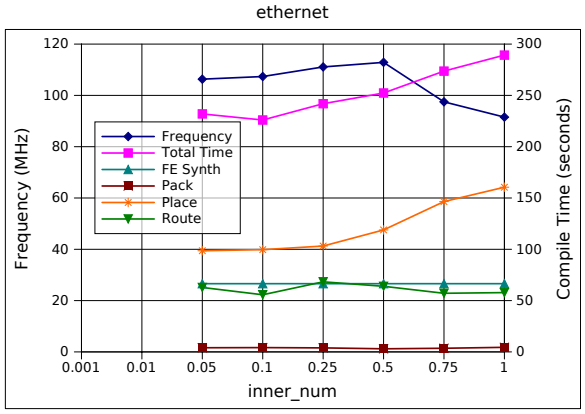


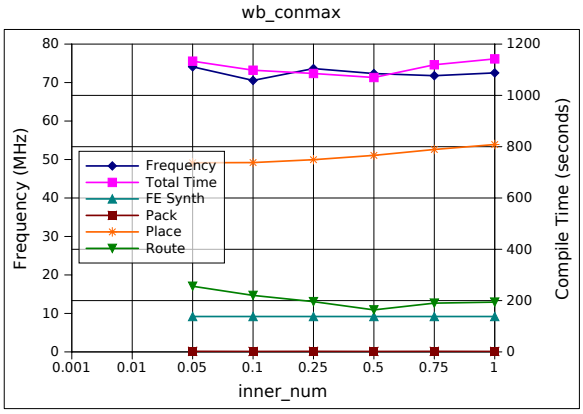


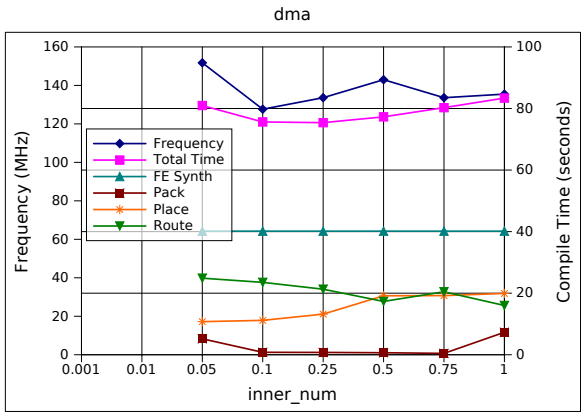


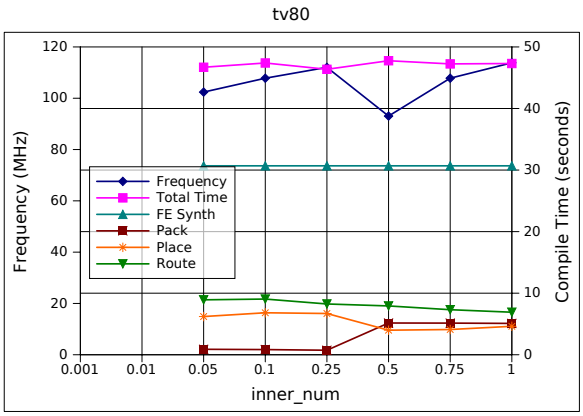


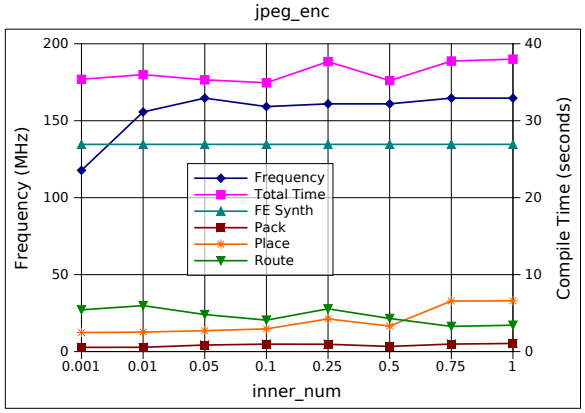
Impaired Benchmarks

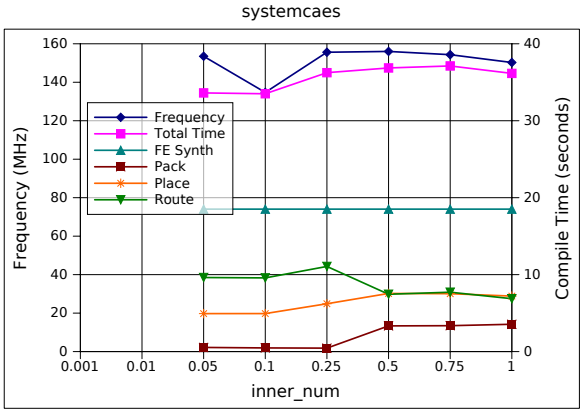


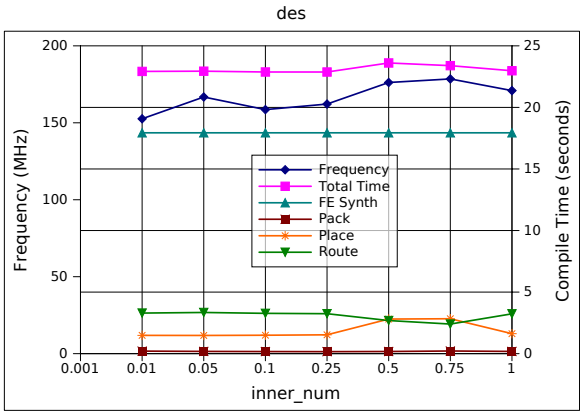


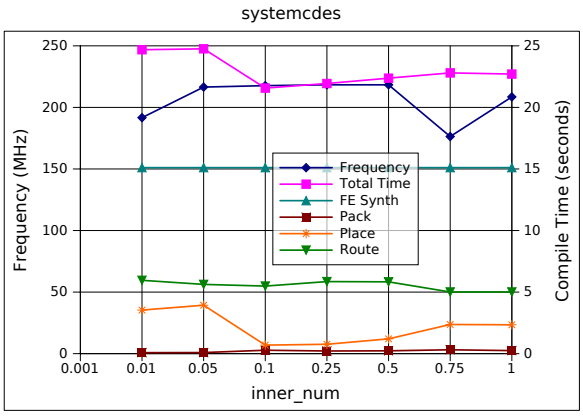









des
Frequency (MHz)
Compile Time (seconds)
inner_num
Frequency
Total Time
FE Synth
Pack
Place
Route
0.001
0.01
0.05
0.1
0.25
0.5
0.75
1


systemcdes
Frequency (MHz)
Compile Time (seconds)
inner_num
Frequency
Total Time
FE Synth
Pack
Place
Route
0.001
0.01
0.05
0.1
0.25
0.5
0.75
1

Appendix B

VPR Without "-fast"

All the VPR results in this thesis were generated using the "-fast" option (referred to as "VPR-fast"). VPR-fast reduces the number of inner-loop iterations in the VPR annealer by 10x. According to the VPR manual [1], this reduces placement time by about 10x (only placement time, not routing too) with a maximum 10% loss in quality. Table B.1 compares VPR-fast to VPR using the benchmarks used in this thesis. The placer is over 5x faster with the "-fast" option, but when routing time is added, the overall speedup is 3.6x. This data does not include the time for BLIF generation.

The channel width for VPR-fast is about 5% worse than VPR. This is expected since VPR has more iterations to find a higher quality solution.

The critical path delay results for VPR-fast are about 7% faster than VPR. This a somewhat surprising result, but it is consistent with previous tests [89].

Table B.1: VPR versus VPR-fast results.

		Synthesis Time								Channel Width			Critical Path Delay		
		VPR -fast			VPR			Ratio		VPR -fast	VPR		VPR		
		Pack	Place	Route	Pack	Place	Route	Place	Total			Ratio	-fast	VPR	Ratio
CG-only	fft16	1.23	318.37	49.07	1.06	1239.85	47.90	3.89	3.50	110	70	1.57	10.17	10.41	0.98
	me	0.71	62.76	16.16	0.60	285.14	21.36	4.54	3.86	51	50	1.02	15.14	16.41	0.92
	chem	1.00	189.38	2960.49	0.64	1102.91	3176.31	5.82	1.36	110	106	1.04	36.84	40.68	0.91
	fft8	0.24	34.19	10.06	0.22	159.73	11.78	4.67	3.86	50	46	1.09	8.53	8.68	0.98
	honda	0.09	5.45	3.10	0.09	39.04	4.51	7.17	5.05	47	46	1.01	15.68	16.33	0.96
	mcm	0.07	4.54	2.52	0.07	29.34	3.61	6.47	4.64	46	44	1.05	10.67	11.35	0.94
	wang	0.05	3.26	1.95	0.06	19.31	2.74	5.93	4.20	44	40	1.10	12.55	13.59	0.92
	pr	0.05	2.63	1.52	0.05	13.99	2.14	5.31	3.85	47	50	0.93	11.41	11.80	0.97
	Geo. Mean (CG-only)							**5.38**	**3.58**			**1.09**			**0.95**
Good	ac97_ctrl	0.11	5.27	3.11	0.10	29.27	4.21	5.56	3.96	40	40	1.01	3.59	3.83	0.94
	aes_core	0.14	8.88	4.20	0.12	59.00	5.59	6.64	4.90	39	36	1.08	5.62	6.76	0.83
	dir	0.19	21.87	19.41	0.17	141.22	11.49	6.46	3.69	81	80	1.01	16.77	17.49	0.96
	spi	0.03	1.27	1.05	0.03	7.56	1.34	5.96	3.81	44	44	0.99	7.15	7.32	0.98
	pci_master	0.01	0.63	0.56	0.02	3.40	0.43	5.37	3.21	29	26	1.13	4.03	4.68	0.86
	Geo. Mean (Good)							**5.98**	**3.87**			**1.04**			**0.91**
Geo. Mean (CG-only and Good)								**5.60**	**3.69**			**1.07**			**0.93**
Impaired	ethernet	1.40	107.80	31.47	1.13	438.97	33.30	4.07	3.37	66	68	0.97	9.80	10.46	0.94
	wb_conmax	0.51	582.71	83.49	0.43	1045.97	56.20	1.80	1.65	70	66	1.06	13.82	14.48	0.95
	dma	0.16	16.38	7.56	0.14	101.92	7.93	6.22	4.56	59	62	0.95	7.64	7.86	0.97
	tv80	0.06	3.09	2.80	0.06	20.52	3.29	6.63	4.01	52	50	1.03	9.50	9.53	1.00
	jpeg_enc	0.14	2.38	1.88	0.12	10.49	1.95	4.41	2.86	30	30	1.00	6.14	6.21	0.99
	systemcaes	0.06	4.04	2.18	0.07	28.14	3.06	6.96	4.97	51	52	0.99	6.47	7.34	0.88
	des	0.02	1.02	0.81	0.03	5.17	0.84	5.06	3.25	32	32	0.99	5.70	6.38	0.89
	systemcdes	0.02	0.90	0.65	0.03	6.75	0.87	7.50	4.86	33	30	1.09	4.55	5.22	0.87
	Geo. Mean (Impaired)							**4.93**	**3.51**			**1.01**			**0.94**
Geo. Mean (All)								**5.34**	**3.62**			**1.05**			**0.93**

Appendix C

Verilator Node Mapping

Table C.1: Verilator to Malibu node mapping.

Verilator Node	Malibu Op	Notes
ALWAYS	–	Indicates a clocked region of the code, Malibu supports only a single clock.
ARRAYSEL	LOAD/STORE	User-instantiated array operations. Either a load or a store depending on usage.
ASSIGNDLY	(deleted)	Delayed assignment to a variable. Deleted after ensuring all child nodes are *registered*.
ASSIGNDLY_IF	MUX	Marked as *registered* and renamed if a single IF, mapped into a decision tree if multiple IFs write to the same value.
ASSIGNW	(deleted)	Assignment to a variable.
ACTIVE	–	Indicates a combinational region of the code.
ADD	ADD	
AND	AND	
ASSIGN	(deleted)	Assignment to a variable
ASSIGNPRE	(deleted)	Delayed assignment (from previous cycle). Deleted after ensuring all child nodes are *registered*

Continued on next page...

Table A.1: Verilator to Malibu node mapping. *Continued from previous page.*

Verilator Node	Malibu Op	Notes
ASSIGNPOST	(deleted)	Delayed assignment to a variable. Deleted after ensuring all child nodes are *registered*
CAST	MOV	Change the width of a signal.
CONCAT	CONCAT	
COND	MUX	Conditional assignment
CONDBOUND	MUX	Conditional assignment with a bounded output.
CONST	CONST	Kept until scheduling, then deleted after the scheduler maps the values to the R memory.
EQ	EQ	
EXTEND	MOV	Zero-extension. Implemented as a MOV operation with the output truncated.
EXTENDS	EXTS	Sign extension.
IF	MUX	Renamed if a single IF, mapped into a decision tree if multiple IFs write to the same value.
LOGNOT	EQ	NOT reduction to a single bit, really a test for zero.
LT	LT	
LTE	LTE	
LTS	LTES	
GT	LTE,NOT	Operands swapped and LTE is used instead. If an operand is immediate, then they cannot be swapped. LTE is still used but the value is written to a temp. variable and a NOT operation is inserted.
GTE	LT,NOT	Same as GT but using the LT operation.
GTS	LTES,NOT	Same as GT but using the LTES operation.
GTES	LTS,NOT	Same as GT but using the LTS operation.
MUL	MULU	
MULS	MULS	
NEQ	NEQ	
NOT	NOT	

Continued on next page...

Table A.1: Verilator to Malibu node mapping. *Continued from previous page.*

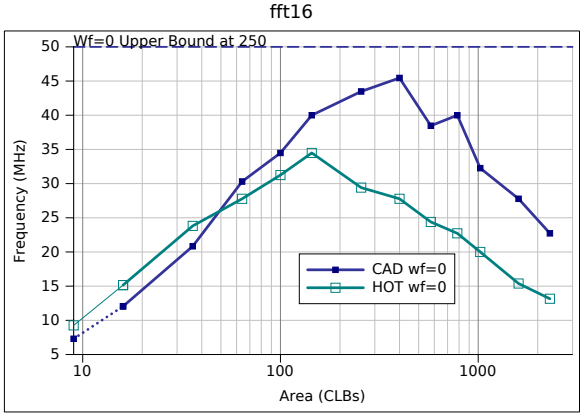
Verilator Node	Malibu Op	Notes
OR	OR	
REDAND	REDAND	
REDNOT	EQ	NOT-reduction is the same as a comparison to zero
REDOR	REDOR	
REDXOR	REDXOR	
REPLICATE	CONCAT	Mapped into a tree of concatenate operations to replicate the required bits.
SHIFTL	LSL	Renamed
SHIFTR	LSR	Renamed
SHIFTRS	LSR,EXTS	Mapped to an unsigned shift plus a sign extension.
SEL	AND,LSR	Bit selection. Converted into a bitmask and a shift.
SUB	SUB	
UNARYMIN	SUB	2's complement NOT, mapped to subtract: $0 - value$.
VAR:BLOCKTEMP	(deleted)	Intermediate assignment, not needed.
VAR:IN	INPUT	Renamed. A placeholder for input, not a real operation.
VAR:MODULETEMP	(deleted)	Intermediate assignment, not needed.
VAR:OUT	OUTPUT	Renamed. A placeholder for output, not a real operation.
VAR:REG	(deleted)	Deleted after ensuring the parent node is *registered*.
VAR:WIRE	(deleted)	Intermediate assignment, not needed.
XOR	XOR	

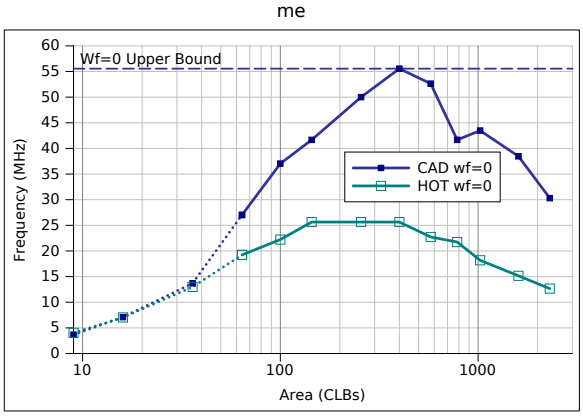
Appendix D

Area Versus Performance Graphs

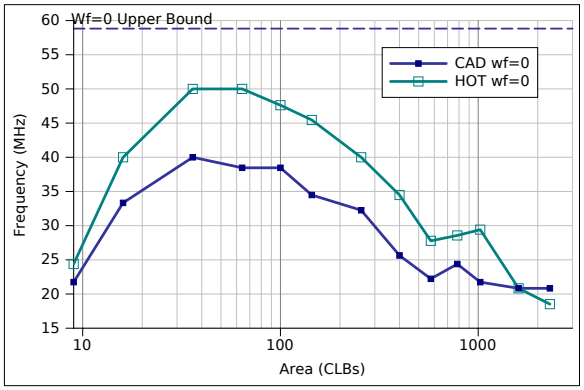
This appendix contains the area versus performance graphs for each benchmark circuit. For the *CG-only* benchmark there is only a $W_f = 0$ curve because these circuits do not require fine-grained resources, and the synthesis solution is the same for any value of W_f.

CG-only Benchmarks

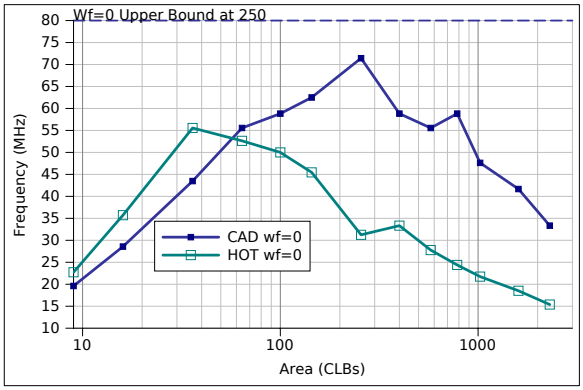




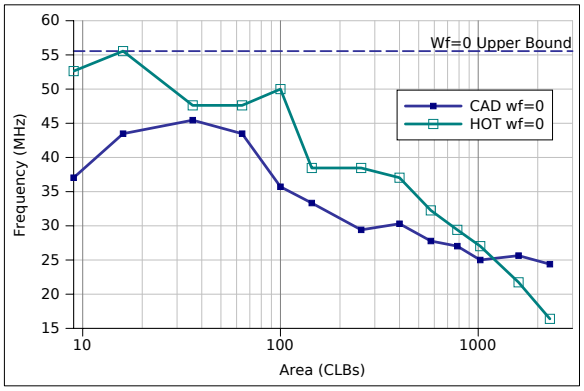
chem



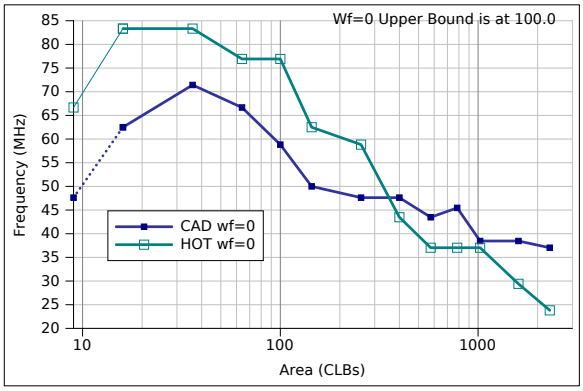
fft8



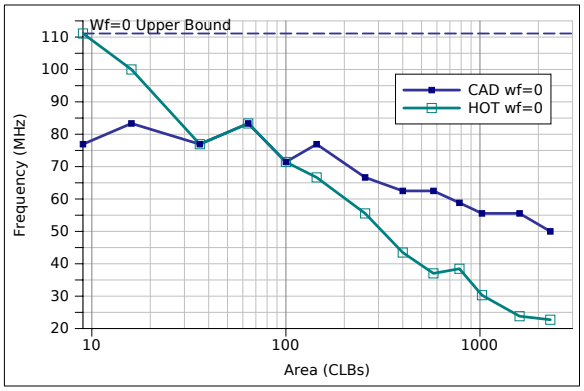
honda



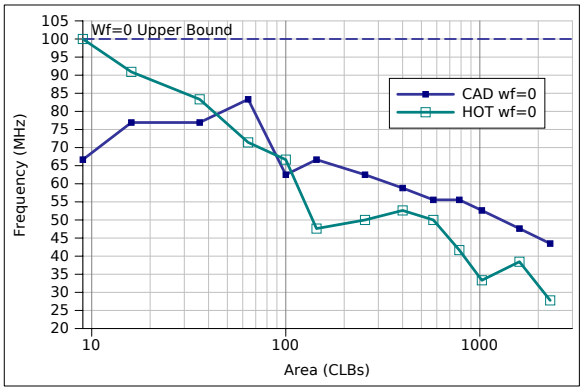
mcm



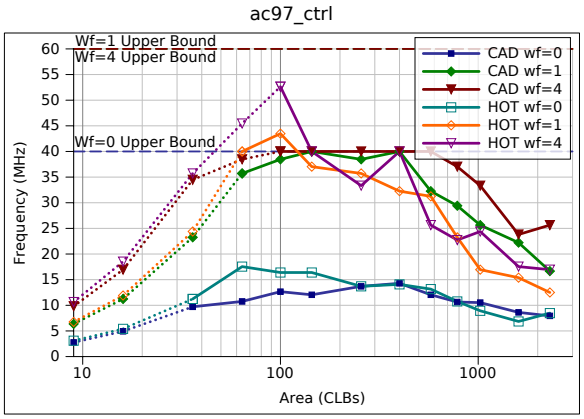
wang

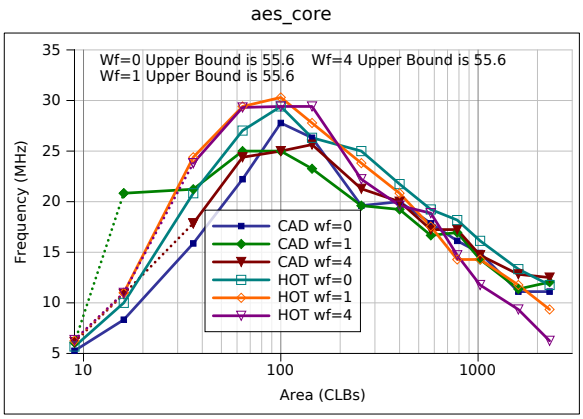


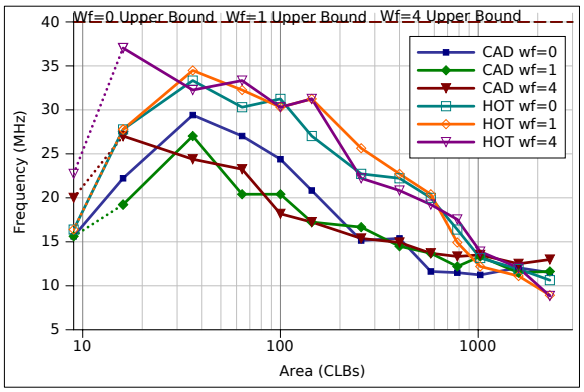
pr

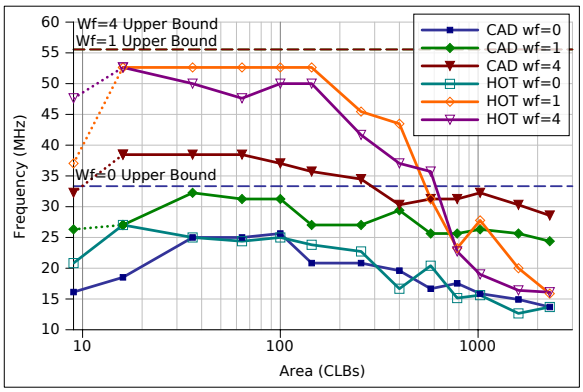


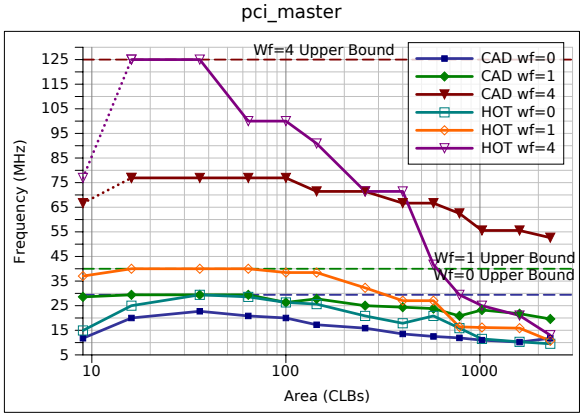
Good Benchmarks





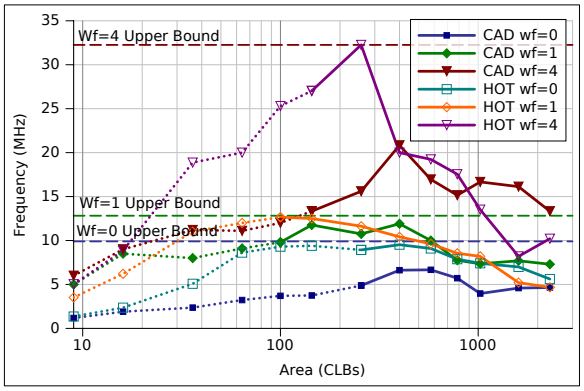




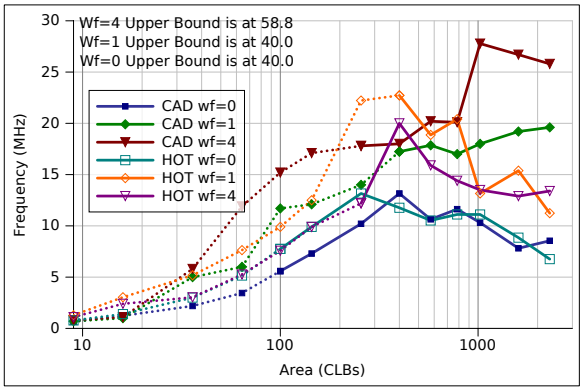


Impaired Benchmarks

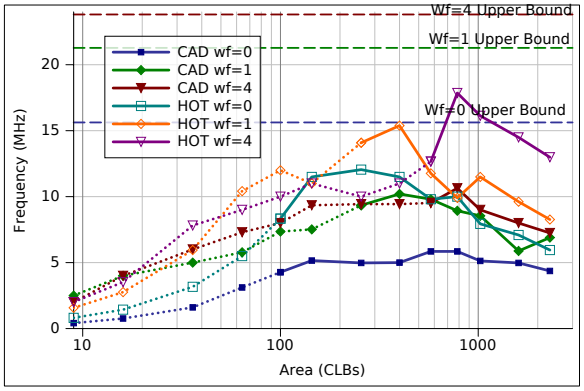
ethernet



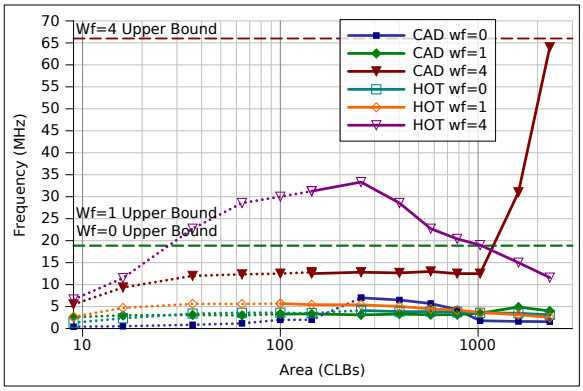
wb_conmax



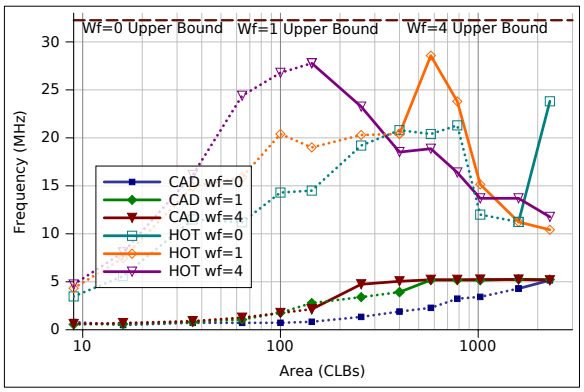
dma



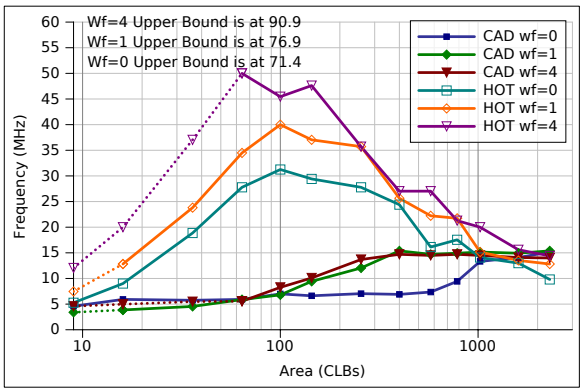
tv80

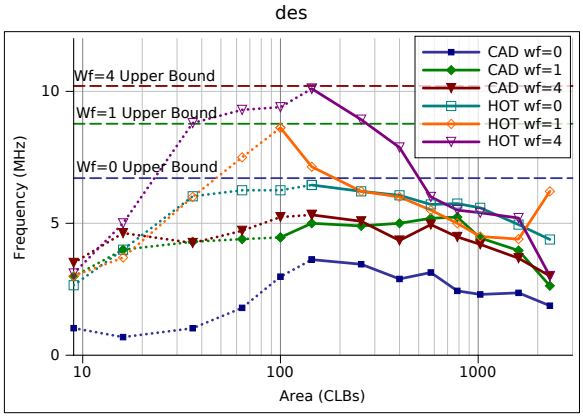


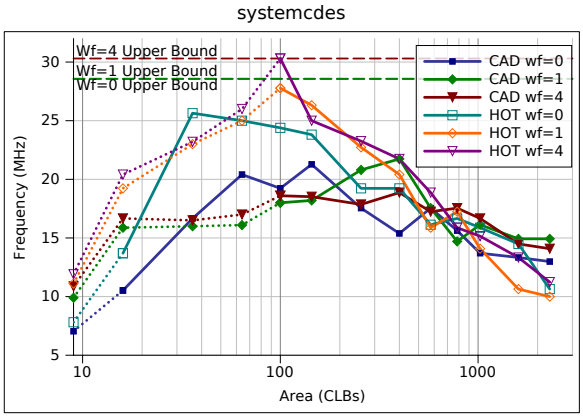
jpeg_enc



systemcaes




des
Wf=4 Upper Bound
Wf=1 Upper Bound
Wf=0 Upper Bound
CAD wf=0
CAD wf=1
CAD wf=4
HOT wf=0
HOT wf=1
HOT wf=4
Frequency (MHz)
Area (CLBs)
0
5
10
10
100
1000


systemcdes
Wf=4 Upper Bound
Wf=1 Upper Bound
Wf=0 Upper Bound
CAD wf=0
CAD wf=1
CAD wf=4
HOT wf=0
HOT wf=1
HOT wf=4
Frequency (MHz)
Area (CLBs)
5
10
15
20
25
30
10
100
1000