Brick and Mortar Chip Fabrication

Martha Allen Kim

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

2008

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Martha Allen Kim

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Chair of the Supervisory Committee:

_____

Mark Oskin

Reading Committee:

_____

Mark Oskin

_____

Susan Eggers

_____

Todd Austin

Date: _____

In presenting this dissertation in partial fulfillment of the requirements for the doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to Proquest Information and Learning, 300 North Zeeb Road, Ann Arbor, MI 48106-1346, 1-800-521-0600, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature_____

Date_____

University of Washington

Abstract

Brick and Mortar Chip Fabrication

Martha Allen Kim

Chair of the Supervisory Committee:
Professor Mark Oskin
Computer Science and Engineering

While Moore's Law has advanced the semiconductor and technology industries, it has simultaneously driven up the cost of engineering a chip in a modern silicon process. The result is that fewer and fewer chips are produced in larger and larger volumes, stifling hardware diversity.

This thesis introduces *brick and mortar* chips, which aim to obtain the benefits of Moore's Law without the financial side effects. Brick and mortar chips are made from small, pre-fabricated hardware components (called bricks) that are bonded in a designer-specified arrangement to a communication backbone chip which serves as the mortar (called the I/O cap).

Our research examines several aspects of this chip manufacturing system. We develop a family of functional bricks, demonstrating a methodology for developing families that make efficient use of physical computation and communication resources. For high-performance communication between arbitrary combinations of bricks we propose a polymorphic on-chip network. This network allows a single I/O cap to be configured to implement the ideal network for any particular application. We analyze a low-cost, physical component assembly technique called fluidic self-assembly, and find that the chip production rate is intertwined with the architectural design of the components. To minimize application execution time on these partitioned chips, we develop software partitioning and mapping techniques which balance communication costs against computational resource contention.

We close with a case study: an analysis of a brick and mortar implementation of a chip multiprocessor. Despite this being a highly latency sensitive design, our measurements indicate a worst case 36% average slowdown in application execution compared to a traditional, monolithic chip. Based on this, our cost analysis, and a survey of related technologies, we conclude that brick and mortar offers the best available performance for its price.

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I am very grateful for the support of Mark Oksin throughout my graduate career. Mark has been a reliable source of technical and professional guidance, with an astounding ability to cut to the quick of any issue. His advocacy, friendship, and easy manner – which guarantees me a laugh over my successes and failures alike – have been invaluable. Thank you also to Susan Eggers, who has provided mountains of thoughtful advice and feedback over the years, as an advisor, instructor, and mentor. Finally, many thanks to the other members of my committee, Todd Austin and John Sahr. They have graciously shared their time and thoughts with me and I am much obliged.

I have been fortunate to have a number of outstanding collaborators. John Davis has been at the ready as a sounding board, reader, and friend for some time now. In one short year Luis Ceze has discussed countless ideas with me, revitalizing the stagnant ones and refining the nascent ones. I was fortunate enough to join the architecture group soon after my arrival in graduate school. For many years now I have learned from and enjoyed the company of Steve Swanson, Andy Schwerin, Andrew Petersen, Andrew Putnam, and Lucas Kreger-Stickles. Thank you also, to Mojtaba Mehrara, for his perserverence in collecting the simulation and process variation data presented in this thesis.

I would like to thank the faculty and administrators that have made UW CSE such a congenial and supportive community for me. In particular, Hank Levy, Dan Grossman, Larry Snyder, David Notkin, Carl Ebeling and Ed Lazowska have been generous with their guidance and encouragement. Thank you also to Lindsay Michimoto for straightening the maze of university administrative policies before me.

I am unspeakably thankful for the support of my family. My parents have provided a steady stream of love and insight which has sustained me. My brother, Chris, is a model of determination. While I know I can never keep up with you, I would be happy to be the f who *icis*. Finally, thank you to my husband, Bryan, for many years of steadfast friendship. You are both cause and reward for all of those cross-country flights.

Chapter 1

INTRODUCTION

Technology scaling has produced a wealth of transistor resources and corresponding improvements in chip performance. However, these benefits come with an increasing price tag, due to rising design, engineering, and validation costs of modern chips [15]. The result has been a steady decline in unique application-specific integrated circuit (ASIC) designs that enter production [21]. This initiates a vicious cycle. Fewer unique chips means that fabs have fewer customers across which to amortize their costs, leading to even higher costs for those who do manufacture chips. The cycle completes as higher chip manufacturing costs exclude even more potential manufacturing customers.

1.1 The problem with Moore's Law

While Moore's Law has fueled the semiconductor industry, it has also fueled this spiral of increasing costs and shrinking fab customer bases. As transistors have shrunk, the cost of fabricating a semiconductor device has grown commensurately. While the fabrication cost per transistor has steadily declined [62], multiple other expenses have ballooned, contributing collectively to the growing total. For example, small features are more susceptible to process variation than larger ones, increasing the range of variation and the proportion of faulty chips. In addition, the smaller the transistor, the more of them that can fit in a given amount of silicon. The result is that circuit complexity has been increasingly outstripping designer productivity, in a phenomenon referred to as Moore's Law's corollary of "compound complexity" [143].

The industry has dealt with these challenges by increasing the engineering effort that goes into each chip. This effort manifests itself as larger design teams, or longer product cycles, and often both at once. The vast majority of this engineering effort is incurred once per chip design, and does not vary with the number of chips produced. Accordingly, this

expense is called the non-recurring engineering cost (NRE) of a chip. Industry analysts estimate that the NREs for a typical 90nm standard cell ASIC can range from $5M up to $50M [113].

Maintaining a particular price per chip in the face of skyrocketing NREs requires larger and larger batches of chips. This is because the single NRE is shared evenly across the population of chips produced. The larger the population, the smaller the impact of the NRE on individual chip cost, so chips produced in large batches cost less than chips produced in small batches. Growing NREs are pushing the line that divides "small" from "large" higher and higher. The result of this situation is that only high-volume chip manufacturers, or those who can sell smaller batches at high prices, can afford to be in the chip business.

Moreover, at the same time that complexity and engineering effort have been soaring, the commercial market has been demanding and rewarding short chip design cycles. This is due to shrinking product lifetimes and the increasing competitive importance of being the first to market with a new product. A technology that succeeds in reducing engineering effort will simultaneously attack the cost of chip preparation as well as its time to market. This research seeks to develop such a technology: one that reaps the benefits of Moore's Law (e.g., high clock speeds, integration) without incurring the downsides (e.g., high NRE costs, long time to market). A viable technology with these characteristics would serve markets that today are economically unreachable.

1.2 Brick and mortar overview

We propose a system, called *brick and mortar*, which is designed to allow fabricated ASICs to be used in many different chip designs. In this way, brick and mortar achieves high volume usage of the *individual* ASIC components while producing *small* batches of any given chip. The purpose is to reduce the non-recurring engineering costs as much as possible while maintaining, to the largest extent possible, the other benefits of ASICs.

1.2.1 Description

At the heart of the brick and mortar manufacturing technique are two architectural components: *bricks*, which are mass-produced pieces of silicon containing processor cores, memory

arrays, small gate arrays, DSPs, FFT engines, and other IP (intellectual property) blocks; and *mortar*, an *I/O cap*, that is a mass-produced silicon substrate containing inter-brick communication infrastructure and I/O support. In the brick and mortar process, engineers design chips by assembling an application-specific layout of bricks. This arrangement of bricks is then bonded, as illustrated in Figure 1.1, to the I/O cap that interconnects them.

Applications can execute on this chip exactly as they would on a traditional chip. Because this chip is constructed from discrete dies, the difference between local communication within a functional core and between functional cores is more pronounced. Thus, it is especially critical for performance on such a chip that an application be carefully partitioned and mapped to the functional cores.

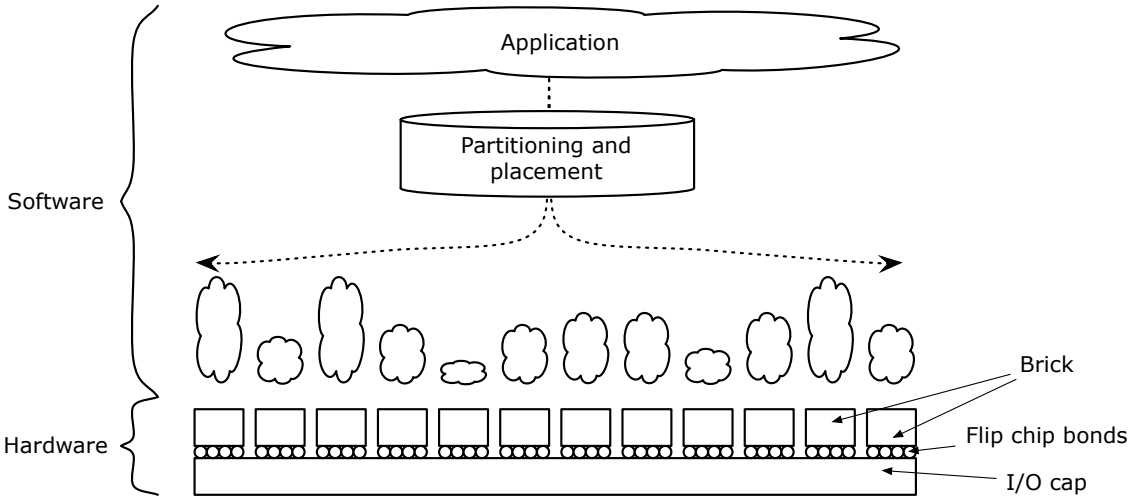

Figure 1.1: Cross-section of brick and mortar chip.

1.2.2 Key advantages

Before delving into the details of brick and mortar chip design, we highlight, qualitatively, the central reasons we pursue this research. The remainder of this thesis will revisit these issues with quantitative analysis.

- **Reduced cost.** This is the chief motivation for brick and mortar chips. The aim is to produce a low-cost alternative to ASIC chips. The cost savings of brick and mortar stems from mass-production of the constituent parts. Although the components themselves are ASICs, they are produced in bulk to be reused in a variety of end user designs. This reuse amortizes the design and verification cost of the components across multiple products. In addition, bricks are small, resulting in lower individual design and verification costs to begin with.

- **Compatible design flow.** Today ASIC designers employ significant amounts of existing IP to produce chips. This improves design reliability and saves design time. In such design flows, the IP blocks are provided as "gateware" netlists. The designer integrates these netlists into a complete design which is then manufactured. Brick and mortar is compatible with this design flow, merely moving the bricks from design modules, which fit into synthesis tool flows, to physical bricks, which fit into a manufacturing flow. The IP blocks are pre-manufactured physical entities which will be bonded to a general purpose communication substrate, the I/O cap. In many ways, bricks are the modern-day analogue of the 7400 series of logic, and the I/O cap is the modern wire-wrap board. Rather than spin custom ASICs for products, engineers could purchase these pre-fabricated components and bond them together as needed.

- **ASIC-like speed and power.** Because most of the logic of a brick and mortar chip exists within a single ASIC component, its performance, in speed and power, will tend to be closer to an ASIC than other custom logic implementation techniques, such as a field programmable gate array (FPGA). When a design calls for it, gate array bricks can implement any necessary custom logic.

- **Mixed process integration.** As we will show, bricks must comply with a standard physical and logical interface. They do not, however, have to be built from the same underlying technology. This makes it readily possible to mix and match bulk CMOS, SOI, DRAM and other process technologies into the same chip.

- **Improved speed.** A subtle positive effect with brick and mortar production is that under certain circumstances it can potentially produce higher-performing large chips than an ASIC process. This is because bricks can be partitioned according to speed grade, and chips then produced from parts with like grade.

- **Improved yield.** Large brick and mortar chips can have a higher yield than large ASICs. The advantage comes from assembling a large chip out of many smaller components. The smaller the component, the higher the yield. One can test component bricks before assembling them, ensuring only functional bricks are included in any assembly, and resulting in an extremely high overall yield.

1.2.3 Key challenges

These benefits will not come for free. Brick and mortar chips will achieve them only through careful design of the necessary hardware, software, and manufacturing subsystems:

- **Communication network.** The communication infrastructure to be implemented in the I/O cap presents a particular challenge. It must be fixed in silicon well before the brick and application layers that are to use it have been determined. Because of this, the communication infrastructure must be general purpose, while also endeavoring to maintain as much of the performance of an application-specific network as possible. This combination, high performance and general, is a very sensitive balance.

- **Brick family design.** The more applications implementable with brick and mortar the greater the cost savings the system can offer. It is important to keep this need for re-use in mind when designing a brick family. Specifically, the bricks must have appropriate sizes and useful functions.

- **Component assembly.** Brick and mortar chips introduce a new step to the chip fabrication pipeline: die assembly. It is essential that the assembly technique one uses not be so costly that it erases the savings gained elsewhere by brick and mortar. The assembly options that are available trade off assembly speed and cost.

- **Software.** Because they reside on different physical dies, there is a greater "distance" between cores in a brick and mortar chip than in a traditional chip. This makes the quality of the application's mapping to computational cores extremely important to overall performance.

1.3 Contributions of the thesis

The contribution of this thesis is the brick and mortar manufacturing system and an analysis of its subsystems.

- **An analysis of ASIC manufacturing costs** Our model can be used to define the bounds within which brick and mortar can save cost relative to an ASIC.

- **A polymorphic on-chip communication network architecture** Our on-chip network design space exploration indicates that there is no network that is universally optimal for varying workloads. One can always improve overall performance of a system with varying workloads if the communication network can be tailored to fit each workload. This configurable network can be adapted to the unknown workloads of a brick and mortar chip or to varying workloads on a monolithic chip.

- **A brick family design methodology** The methodology we present sizes bricks in order to best fit the communication and circuit area needs of the desired IP cores.

- **A sample system-on-chip brick family** We employ this brick family development methodology to construct a family of bricks for system-on-chip designs.

- **Quantification of interaction between component assembly and component architecture** Simulating the assembly of sample chip designs, we discover how the chip to be assembled affects assembly speed. We propose changes to the assembly process and component architecture based on these observations.

- **A parametrizable software partitioning algorithm** An application's mapping onto a highly partitioned chip such as brick and mortar significantly affects its perfor-

mance. We explore existing mapping algorithms and propose a hierarchical, parametrized algorithm which can be tuned to match the application partitioning to the application and to the hardware components.

- **A study of brick and mortar CMP implementation** We examine a brick and mortar implementation of a chip multiprocessor (CMP) from the perspectives of performance and process variation. We find the SPLASH2 [145] benchmarks execute an average of 36% slower on a brick and mortar chip. This slowdown is disproportionately small relative to the potential cost savings of the brick and mortar technique, and can be removed entirely in some chips by speed binning components prior to assembly.

1.4 Thesis outline

The following chapter presents an economic model of ASIC and brick and mortar chip manufacturing as well as an analysis of the economic opportunity for brick and mortar.

Chapters 3 through 6 examine each of the subsystems required to make brick and mortar a viable ASIC alternative. We work from the bottom of the stack to the top. Accordingly, Chapter 3 starts by presenting a network design for the I/O cap. Next, Chapter 4 examines brick family design and posits a system-on-chip brick family. Chapter 5 examines how to put these components together. In it, we examine manufacturing techniques for assembling and bonding bricks to an I/O cap and for packaging the completed chip. Finally, in Chapter 6 we study algorithms to map software onto tiled chips such as brick and mortar. We analyze existing algorithms and, based on our findings, present a novel, tunable placement algorithm.

In Chapter 7 we look at a brick and mortar case study: a chip multiprocessor design. We examine the performance of a CMP built with brick and mortar, as well as opportunities to exploit process variation amongst the components of this design.

Chapter 8 surveys existing chip manufacturing techniques and how they relate to brick and mortar, as well as technologies related to each of the subsystems examined in Chapters 3 through 6. In the concluding Chapter 9, we discuss potentially attractive variations of the basic brick and mortar proposal presented and explored in this thesis. We also indicate some veins of future research that build on this work.

Chapter 2

MANIPULATING THE ECONOMICS OF CHIP FABRICATION

To quantify the economic benefits of the proposed process we developed a model of chip production costs. This model has two parts. The first part approximates the cost of manufacturing dies in an ASIC process, and the second part calculates the cost of using those dies to produce a brick and mortar chip.

2.1 ASIC cost model

The cost of an ASIC has two components: the non-recurring engineering cost (NRE) that is incurred once for a given design and the unit cost that is incurred once for each chip produced. The top half of Table 2.1 lists the equations that make up this model.

2.1.1 Non-recurring engineering cost

The approximation of ASIC cost is based on several assumptions which are expressed to the model as input parameters. One of the inputs is the assumed NRE. The NRE includes all engineering effort encompassing circuit design, RTL implementation, functional verification, and physical verification of signal integrity, power, and timing. This can take large teams a long time, and with an engineer's time costing upwards of $380,000 per year [141], the engineering cost is nearly always a seven-figure number. While the engineering cost usually accounts for the largest share of the NRE, the sum includes several other expenses. NREs also encompass the cost of tools, IP licenses if necessary, and photolithographic masks. ASIC design tools typically cost more than $300,000 [146]. Mask cost has been roughly doubling every technology node, resulting in a complete set of 90nm masks costing between $1M and $3M [146].

2.1.2 Unit cost

The unit cost of a chip is the incremental cost of producing each individual chip. In our model we assume a constant unit cost (i.e., the millionth chip costs the same to produce as the first chip), but in reality this is not always the case. Fabs often offer an "economy size" option and discount especially large orders [71]. Also, actual unit costs can shrink, as fabs refine their processes and become more efficient over time.

Fabrication: Our approximation of unit cost starts by looking up published processed wafer costs from IC Knowledge [60]. We calculate how many whole dies of a given size fit on a wafer. Dividing the cost of a processed wafer by this number produces the cost of each die.

Test: In 2000, the cost to test each transistor was 10% of the cost to manufacture it. However, as transistors become cheaper and testing becomes more difficult, it is projected that by 2015 it will cost more to test a transistor than to make it [73]. To approximate the cost of testing the dies, we first approximate die test time by assuming a base, constant test time for maneuvering the die to and from the test harness. We then add to it some additional test time proportional to the die area. We compute the die test cost from this test time using an assumed test cost per hour. The manufacturing yield is the percentage of manufactured devices which pass post-manufacturer testing. Every die that fails testing still incurred the manufacturing and test expenses, driving up the effective cost of each working die. We estimate the yield based on an assumed defect rate per unit area and then apply it to the total die cost.

Given these two components, NRE, and $UnitCost$, the total cost of producing a batch of x chips is simply $NRE + x \cdot UnitCost$. This makes the price of one of those x chips $\frac{NRE+x \cdot UnitCost}{x}$.

2.2 Opportunity for brick and mortar

Using this model we can analyze the open window of opportunity for brick and mortar. In Figure 2.1 we have plotted the price of a chip (in the y dimension) produced at a given volume (in the x dimension). Note that this is a log-log plot. Each line indicates the chip

Table 2.1: ASIC and brick and mortar chip cost models.

ASIC cost		
Inputs		
$WaferDiameter$		200mm and 300mm wafers in production today
$FeatureSize$		This is the technology node: 130nm, 90nm, etc.
$MaskLayers$		Data available from IC Knowledge [60] for 24 and 26 layer processes
$TestCostPerHour$		Hundreds of dollars per hour [56]
$FixedTestTime$		Assumed no more than a minute
$TestTimePerArea$		Depends on the process technology, logic, and completeness of test; anywhere from .08 to 2.5 seconds per mm^2 [140, 114]
$DefectsPerArea$		Can range from 1.0 to 0.1 defects per cm^2 [60]
NRE		Varies between \$5M and \$50M [113]
$ProductionVolume$		
Equations		
$ProcessedWaferCost$	$=$	$Lookup_{ICKnowledge}(WaferDiameter, FeatureSize, MaskLayers)$
$WaferArea$	$=$	$\pi \cdot (\frac{WaferDiameter}{2})^2$
$NumberDies$	$=$	$\frac{WaferArea}{DieArea} - \frac{\pi \cdot WaferDiameter}{\sqrt{DieArea}}$
$TestTime$	$=$	$FixedTestTime + (TestTimePerArea \cdot DieArea)$
$TestCost$	$=$	$\frac{TestCostPerHour}{60\ cdot60} \cdot TestTime$
$DieYield$	$=$	$(\frac{1+DefectsPerArea \cdot DieArea}{\alpha})^{-\alpha}$
$UnitCost$	$=$	$\frac{\frac{WaferCost}{NumberDies} + TestCost}{DieYield}$
$ASICCost$	$=$	$\frac{NRE + ProductionVolume \cdot UnitCost}{ProductionVolume}$
Brick and mortar cost		
Inputs		
$IOCapCost$		From ASIC cost model
$NumBricks$		Can be anything, but we expect numbers in the 32 to 64 range
$AssemblyCost$		Low, simply the cost of the liquid and polymer for the shaker table
$TestTime$		Assumed the same as ASIC
$TestCostPerHour$		Assumed the same as ASIC
$Yield$		
Equations		
$TestCost$	$=$	$\frac{TestCostPerHour}{60 \cdot 60} \cdot TestTime$
$YieldedChipCost$	$=$	$\frac{(IOCapCost + NumBricks \cdot BrickCost) + AssemblyCost + TestCost}{Yield}$

price assuming a different NRE ranging from \$1M to \$50M.[1] As one can see from the y-intercepts, if only a single chip is produced, the effective price of that chip is dominated by the NRE. However, as more and more chips are produced, the NRE is amortized away and the price of a chip asymptotically approaches the unit price.

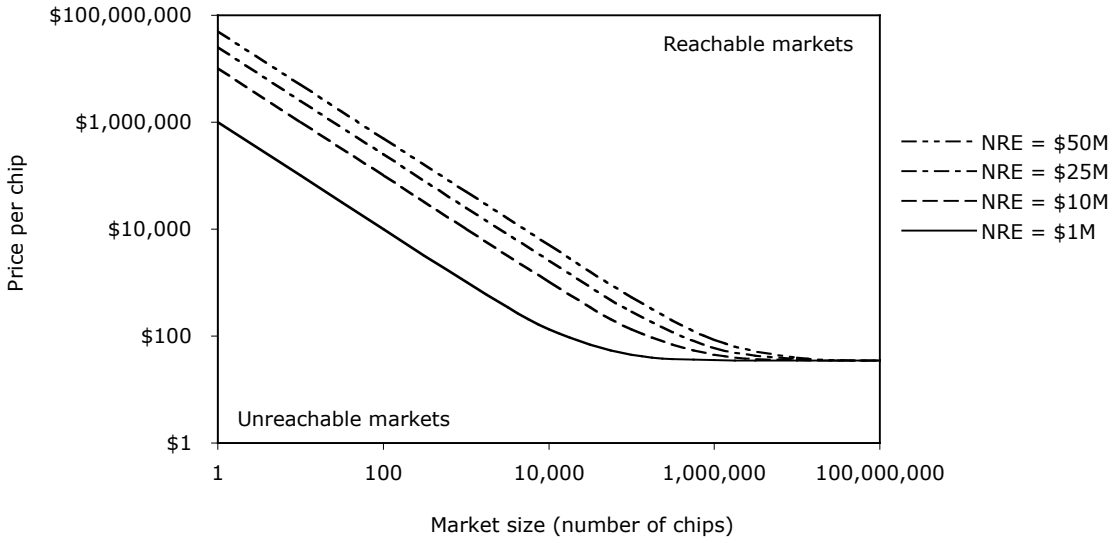

Figure 2.1: Effect of increased non-recurring engineering costs on chip production cost at different market sizes and price points.

There are multiple ways to analyze this data. The first is to track the price of a particular chip as NREs grow. For example, in order to maintain a price of \$100 per chip as the NRE grows from \$1M to \$50M, a market must increase in size from 15,298 to 764,877 units. Alternately, if one maintains a fixed market size of 1M units, the price of a chip will more than double from \$35.63 to \$84.63.

A second analysis takes the plot as a two dimensional market space that is split into two regions by the lines. In this view, one can plot any chip market, real or hypothetical, in the space using the size of the market (x-value) and the chip price it will support (y-value).

[1]The other ASIC cost model inputs that generated this plot are: $WaferDiameter = 300mm$, $FeatureSize = 90nm$, $MaskLayers = 26$, $TestCostPerHour =$ \$500, $FixedTestTime = 1$ minute, $TestTimePerArea = 1$ second, $DefectsPerArea = 0.5$ defects per cm^2.

Any markets that fall upwards of a given line are considered *reachable*. This means that one can produce chips and sell them to this market at a profit (however small). On the other hand, if a market falls below a line, it is considered *unreachable*, meaning that the market cannot support the cost of chip production. This plot illustrates the phenomenon alluded to in Chapter 1, that increasing NREs make small, low-cost markets unreachable. As NREs increase, the line separating reachable and unreachable markets creeps upward and to the right, away from inexpensive, low-volume markets. These are the markets that brick and mortar aims to serve.

2.3 Brick and mortar cost model

We estimate the cost of a brick and mortar chip by building on the ASIC cost model from Section 2.1. We use this model to compute the cost of the bricks and I/O cap. To the cost of the components, we add the cost of assembling them and testing the assembled chip. The yield then can be applied to this raw cost to produce an estimate of the yielded brick and mortar chip cost.

2.4 Quantifying the advantage of brick and mortar

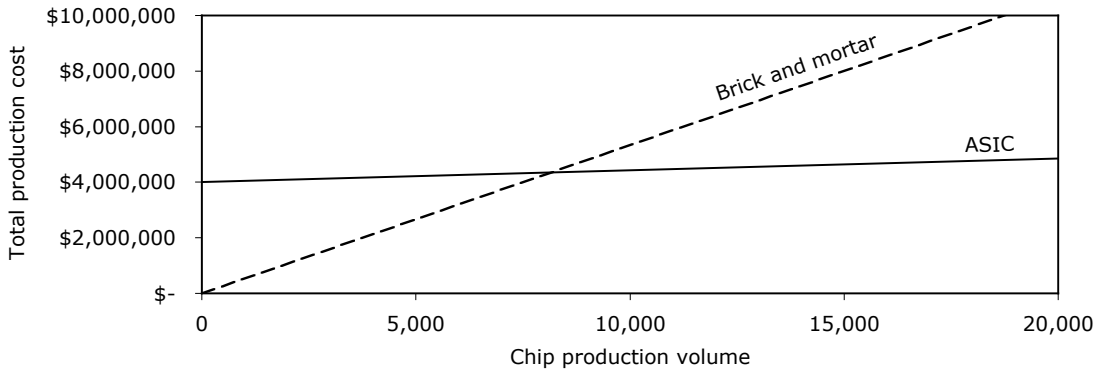

Figure 2.2: Comparison of total production cost for ASIC and brick and mortar chips as a function of batch size.

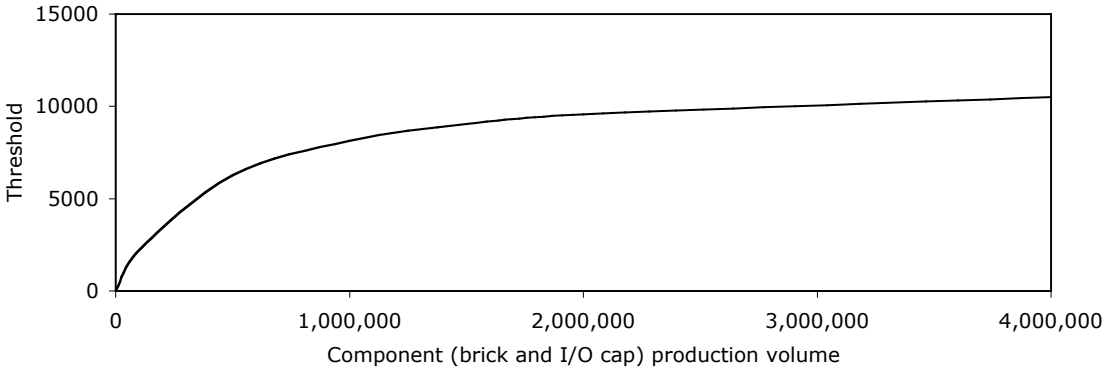

Figure 2.3: Brick and mortar threshold batch size as a function of brick and I/O cap production volume.

The purpose of brick and mortar is to provide a more economical option than ASICs for small batches of chips. Figure 2.2 plots the total production cost of a batch of chips as a function of the size of the batch for both ASIC and brick and mortar.[2] The NRE of each technology determines the y-intercept of these lines (the expense incurred before a single chip is produced) while the unit cost determines the slope (the cost to churn out each individual chip). We call the production volume at which these two lines cross the "threshold". If one's desired batch of chips exceeds the threshold, it is more cost-effective to build an ASIC. However, if one's batch is smaller than the threshold, brick and mortar will be more economical.

The value of this threshold indicates the utility of brick and mortar. The larger the threshold, the more markets brick and mortar can serve cost-effectively. Figure 2.3 plots the threshold as a function of the assumed component production volume (i.e., brick and I/O cap production volume). Note the positive correlation between these two values. The larger the batches of components, the less expensive they will be.[3] Less expensive components

[2]The ASIC model inputs for the brick and I/O cap were the same as Figure 2.1 including $NRE = \$4M$ and $ProductionVolume = 1,000,000$. The brick and mortar model assumes 32 bricks per chip.

[3]As Figure 2.1 illustrates, the price reduction is asymptotic towards the unit price of a chip.

means less expensive brick and mortar chips, which will push the threshold volume for building an ASIC even higher.

In our analysis, the component production volume is the independent variable (x), but in practice its value will depend on the threshold (y). The higher the threshold, the more scenarios in which brick and mortar under-prices an ASIC, making it feasible to produce components in larger volumes. In fact, we have a feedback loop in which increasing either variable (threshold or component volume) will increase the other. Thus in designing the brick and mortar process, maximizing either variable will work. Since component production volume is an input to our model, we will seek to maximize the utility of brick and mortar by maximizing the threshold.

Chapter 3

POLYMORPHIC ON-CHIP NETWORK

The I/O cap is a silicon die that has three primary functions: (1) to provide power, clock and ground to the bricks; (2) to house I/O pads for connectivity to external package pins; and (3) to provide connectivity between bricks. The first two offer little in the way of brick-and-mortar-specific architectural questions, so we focus on the third to drive the I/O cap design. Because each chip will have the same I/O cap but a unique brick arrangement and application, the interconnect in the I/O cap must be flexible. This chapter examines the communication requirements and proposes a suitably flexible network on-chip (NoC) architecture.

3.1 On-chip network design space exploration

To understand the requirements of the I/O cap's network, we conducted an exploration of the NoC design space, examining the performance of different network architectures operating under a range of traffic patterns.

3.1.1 Methodology

NoC designs. To explore a wide range of NoC designs, we vary not only the network parameters (e.g., queue capacities and packet sizes) but also the policies and structure of the network itself (e.g., the topology and arbitration algorithm). The first section of Table 3.1 defines the 360 NoCs in our design space.

We selected the topologies to cover a range of radices – from the highly-connected flattened butterfly, to the less-connected ring – and several widely-used topology families. For each topology, we used a deterministic, minimal, oblivious source routing algorithm. The design space includes two buffered arbitration policies: *store-and-forward* and *wormhole* [37] the latter of which reserves and preserves a connection at each switch until all packets in a

message have traversed the switch.

The topology, routing algorithm, and arbitration policy define the network. The performance of a network can vary a great deal based on its resources. Thus we include a range of queue capacities and packet sizes in the design space as well.

Cycle-level simulation. We measure the performance of each network using a software micro-architectural network simulator. The simulator is execution-driven and models the network on a cycle-by-cycle basis. We validated this simulator by successfully reproducing data from prior work [139].

Traffic patterns. The cores interact with the network using arbitrary-length messages which are converted to and from fixed-length packets by converters at the boundaries of the network. To drive the simulations, we use three synthetic workloads. The first of these, *uniform random* traffic, is a widely used traffic generation pattern, in which each node generates a message every cycle for a randomly chosen recipient node. This workload has traffic characteristics similar to an application with near-random data accesses running on a CMP (e.g., a breadth first graph traversal [152]). For this workload, and the others used in this study, the packets are injected via a Bernoulli process.

The other two workloads are permutation workloads, in which each node sends packets to exactly one other node. This is sometimes called an adversarial pattern, because, unlike in the uniform pattern, the communication load is unbalanced. We experiment with two workloads in this family: *random permutation*, in which the permutation of sender-to-receiver nodes is randomly generated, and *nearest neighbor*, in which each sender sends packets to a nearby receiver. The local adversarial pattern reasonably approximates a system-on-chip design in which the designer took care to place communicating blocks near one another to maximally exploit any locality in the application.

Area model. As on-chip interconnects are often designed under tight area budgets, it is unrealistic to explore interconnection options on the basis of performance alone. Thus, we adopt the methodology of previous work [14], and develop an area model for a network

design. To balance model accuracy with design space size, we synthesize designs for the base network components, including buffers, queues and crossbars. These base estimates target a 90nm process, using high-performance GT standard cell libraries from Taiwan Semiconductor Manufacturing Company (TSMC) for the memories, and a model of full-custom layout of crossbar interconnects.

Table 3.1 describes the network model in more detail. The inputs to the model are the parameter settings which define a particular network design. The synthesis-based section of the table specifies the base area components, including queue and crossbar areas. We then analytically combine the areas of the base components to estimate the total area of the network circuit. Because networks are wiring-heavy circuits, we conservatively assume a 20% wiring overhead [14] in the cell placement, in addition to the already-accounted-for crossbar.

3.1.2 Results

The graphs in Figure 3.1 come from 64-node (N from Table 3.1 = 64) network simulations. On the x-axis is average packet latency while on the y is the average throughput. There is one graph per traffic pattern, with each point in the graphs representing one network design. The circled points indicate the networks that are Pareto optimal for the given workload. These are the designs for which one dimension cannot be improved (latency reduced or throughput increased) without sacrificing in the other dimension (increasing latency or reducing throughput).

First, we examine the design trade-offs when implementing a network for a single traffic pattern. For example, the first graph shows that the Pareto optimal network designs for uniform random traffic include all four topologies. The fat tree and mesh offer the lowest-latency optimal designs, followed by the butterfly with double the latency, and lastly by the ring with quadruple. Over these optimal designs, increases in throughput come only with accompanying increases in latency. With both the fat tree and mesh, short queues (4 entries) and large packets (128 bits) give rise to the aggregate optimal performance. At the other end of the spectrum, the ring, when provisioned with the same large packets, has a

Table 3.1: On-chip network design space and hybrid synthesis-analytical area model

Network Design Space Parameters			
	Description	*Symbol*	*Range*
	Number Of Terminals	N	16,**64**,256,1024
	Topology	T	$\begin{cases} butterfly\ (k=2) \\ fat\ tree\ (k=2, levels=3) \\ flattened\ butterfly\ (k=2) \\ mesh \\ ring \end{cases}$
	Routing Algorithm	R	deterministic
	Arbitration Algorithm	A	store-and-forward, wormhole
	Message Size	M	256
	Packet Size	P	$32, 64, 128$
	Switch Packet Queue Capacity	SQ	$4, 16, 64$
	Converter Packet Queue Capacity	CPQ	$4, 16, 64$
	Converter Message Queue Capacity	CMQ	4
Synthesis-Analytical Area Model			
	Description	*Symbol*	*Value*
Synthesis	Queue area	$Queue_{area}$	$0.00002\ mm^2/bit$
	Wire pitch	χ	$0.00024mm$ [63]
	Crossbar area	$XBar_{area}$	$\chi^2 \times D_{in} \times D_{out} \times P^2$ [39]
	Wire overhead	W_{area}	20%
Analytical	Number Of Switches	S	$\begin{cases} \frac{N}{k} \times log(N) & \text{if } T \text{ is } butterfly \\ \frac{N}{k} & \text{if } T \text{ is } flattened\ butterfly \\ N + \frac{N}{k^2} + \frac{N}{k^4} & \text{if } T \text{ is } fat\ tree \\ N & \text{if } T \text{ is } mesh \\ N & \text{if } T \text{ is } ring \end{cases}$
	Switch Degree	D	$\begin{cases} k + log_k(\frac{N}{k}) & \text{if } T \text{ is } flattened\ butterfly \\ k & \text{if } T \text{ is } butterfly \\ \begin{cases} 2 & if\ leaf \\ k^2+1 & if\ internal \\ k^2+4 & if\ root \end{cases} & \text{if } T \text{ is } fat\ tree \\ 5 & \text{if } T \text{ is } mesh \\ 3 & \text{if } T \text{ is } ring \end{cases}$
	Switch Queue Area	SQ_{area}	$SQ \times P \times Queue_{area}$
	Switch Area	S_{area}	$SQ_{area} \times D + XBar_{area} + SQ_{area} \times D$
	Converter Message Queue Area	CMQ_{area}	$CMQ \times QPB$
	Converter Packet Queue Area	CPQ_{area}	$CPQ \times QPB$
	Converter Area	C_{area}	$2 \times CMQ_{area} + 2 \times CPQ_{area}$
	Network Component Area	N_{area}	$S \times S_{area} + N \times C_{area}$
	Total Network Area	I_{area}	$1.20 \times N_{area}$

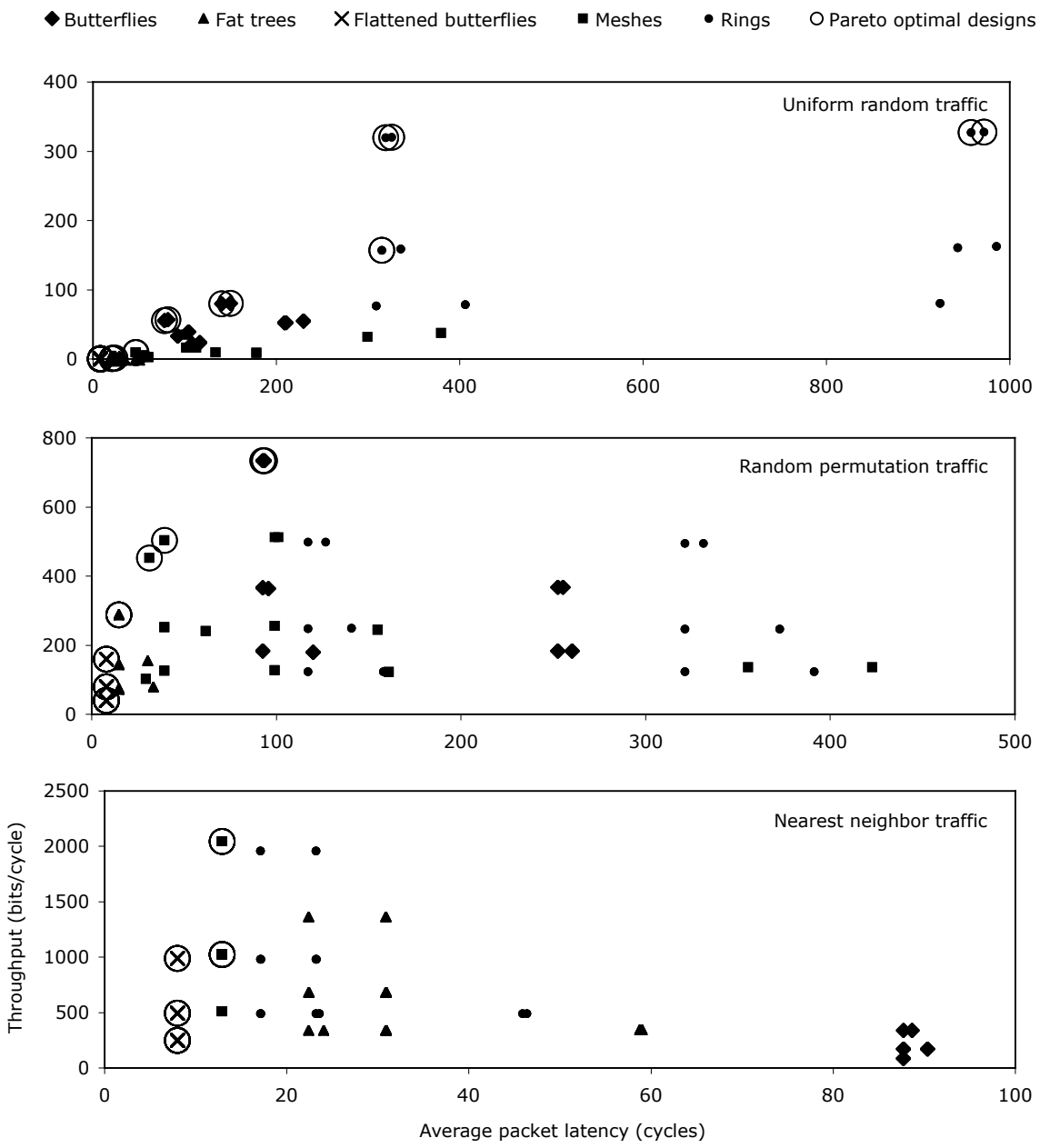

Figure 3.1: Latency and throughput of on-chip interconnect design space.

long transmission latency, but offers significantly higher throughput.

In the case of random adversarial traffic, Figure 3.1 indicates that a fat tree once again offers the lowest-latency communication option. As with uniform random communication, some communicating nodes are going to be at a distance in the network, and thus, the non-neighbor connections proffered by the higher levels of the tree speed that traversal. The differences in throughput amongst the fat tree designs on this workload are entirely attributable to packet size: the larger the packet the higher the throughput. This is feasible under the area budget, because the network does not require particularly deep queues on this workload. The same is true of the mesh network, for which the best designs incorporate large packets (128 bits) and short queues (4 entries) to maximize throughput under the area budget. However, on average, the packet latency through the mesh is slightly higher than the fat tree due to the neighbor-only links.

By contrast, the local adversarial traffic experiences exactly the opposite result. While random traffic latency *suffered* on neighbor-only topologies, the local adversarial traffic, which *is* neighbor-only, took good advantage of those topologies. Thus, for this workload, the mesh topologies are optimal, with the ring not far behind.

While we see that there is a network to fit each workload, there is no network to fit all workloads. In other words, no network in the design space is optimal across all three workloads. The optimal designs often differ in topology, and, when the topologies are the same, the resource provisioning is very different. Although these workloads are synthetic, it would not be far-fetched to encounter three similar patterns in the I/O cap, depending on the brick selection and the application or input data set that is running. Unfortunately, in selecting a single network, one will necessarily have to sacrifice performance on one or more workloads.

3.2 Polymorphic network microarchitecture

Having empirically realized that no single fixed-design on-chip network efficiently communicates different styles of traffic, we have designed the polymorphic on-chip network. This network can be configured at runtime to mimic traditional fixed-function networks. From a hardware standpoint, the network is built from a sea of resources, namely buffers and cross-

bars. Careful design allows post-fabrication or even runtime configuration of these resources to form an interconnect with a custom topology, buffer allocation, and packet size.

Such a network should be useful in a brick and mortar I/O cap where the brick configuration and application are unknown at the time the cap is fabricated. The polymorphic network would also be useful in a more standard monolithic device that will run multiple applications. The data, presented in Section 3.1.2, indicate that, even with a fixed hardware configuration, varying applications alone are sufficient to produce network workloads which benefit significantly from a customized network. The discussion of the polymorphic network will continue with the I/O cap use in mind, but will comment on this potential alternate use when appropriate. First we will describe the microarchitecture of the configurable fabric, followed by some examples of how to take advantage of the polymorphism.

3.2.1 Configurable switches

A switch in the polymorphic network, like a classic switch, consists of input and output packet queues, routers, arbiters and a crossbar of connections. As illustrated in Figure 3.2, we call the set of input queue, router, and arbiter a *slice*. These slices are clustered into *regions*. Although Figure 3.2 illustrates eight slices per region, in the following section we will explore different parametrizations of this underlying fabric structure.

The crossbar connecting the slices in a region is a double crossbar that allows redefinition of the input connections as well as the usual dynamic definition of output connections. The diamond-shaped intersections on the incoming packet lines in Figure 3.2's slice detail denote the static connections, while the circular intersections indicate the dynamically switching output connections. Although Figure 3.2 shows these crossbars as bidirectional, they in fact consist of two unidirectional wires, because drive buffers can be employed only on directed wires. This makes these wires physically segmentable, with a potential segmentation point between each of the slices.

Each slice supports two virtual channels, which can share the available buffer space, using a flexible sharing scheme [38]. Networks requiring more than two channels, must aggregate multiple slices together.

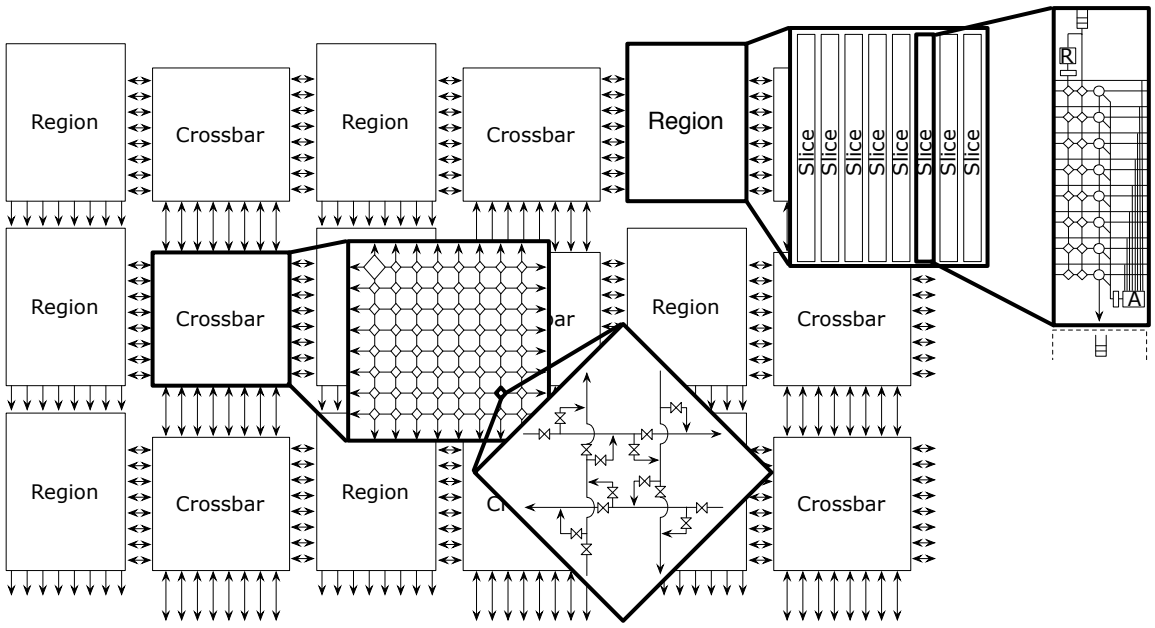

Figure 3.2: Microarchitecture of the polymorphic on-chip network.

The configurable connections between input wires and crossbars and the ability to segment a crossbar make it possible to construct switches out of slices from *different* regions in the fabric. This ability to aggregate slices from different regions into one logical switch turns out to be very valuable, as it permits a dense packing of switches in the fabric. This has two benefits. First, as the input and output queues consume most of the area of this fabric, it is important to waste as few of them as possible when configuring the network. Second, the closer the switches are physically, the shorter the routed links that connect them must be, leading to less link routing congestion and less capacitive load on those links.

Imagine if one had an 8-slice region configured as two 4×4 switches consuming all of the eight crossbars in the region. The ability to segment a crossbar allows us to make one of those switches larger if we desire, increasing it to a 5×5, by pulling in a slice from a neighboring region. This new slice must connect its input to a crossbar that is not already used within its own switch, meaning it must connect to a crossbar that the other switch is using, which is possible only if the wires of the crossbar can be physically disconnected.

3.2.2 Packet routing and arbitration

Each network slice contains routing and arbitration logic. These two pieces of logic serve the same function as the router and arbiter in their classic switch counterparts. A router is associated with an input queue and determines to which output to route the first packet in the queue. Meanwhile each output queue has an associated arbiter which arbitrates write access to the output queue amongst the requesting inputs.

The polymorphic network uses source routing, where each packet carries its pre-computed route with it, rather than making routing decision at each point in the network. This allows the network to support any static routing scheme, such as the one used in the design space exploration presented in Section 3.1. However, this adds bits to the packet header, reducing the payload of a fixed-size packet. One way to work around this, which we employ, is to implement wormhole arbitration [36], allowing a lead packet to carry the route and establish necessary connections, that are used by the subsequent packets belonging to the same message.

Configurable link resources. The crossbars and any wires unemployed in switches can implement connections between the switches. These connections are statically routed by an FPGA-like wire routing algorithm. The connections in these crossbars are statically configurable. They will implement one fixed topology per configuration.

Interface to cores. The polymorphic fabric will be connected to cores in much the same way as a normal network-on-chip. One common network technique is to *concentrate* network traffic, essentially gathering traffic from multiple cores and injecting it into one network node. In a classic network, this resource sharing technique reduces the total size of the network. If concentration or a similar technique would effect the workload in such a way as to improve performance, the polymorphic fabric can readily be configured to concentrate traffic upon injection into the network.

Configuration mechanism. Like an FPGA, the polymorphic fabric is configured via bitstream. This bitstream can originate either from an on-chip memory or from an external

source. In the first case, the OS or runtime system can configure the network by writing to memory-mapped bits containing the configuration as it would any other I/O device. This configuration technique would also work on a non-brick and mortar chip. The device could also include a default network configuration to support applications which do not specify their own custom interconnect. The question of *when* and *how* to reconfigure the polymorphic fabric in such a chip is an open and future question that is elaborated on at the end of this thesis in Section 9.2.

3.3 Example configurations

The fabric microarchitecture we have described makes it possible to customize the implemented networks. The customization can happen in several ways including custom topology, custom buffering, and custom packet size.

3.3.1 Custom topologies

Configuration of the fabric to implement a particular topology is a matter of forming appropriately-sized switches and connecting them according to the desired topology. Figure 3.3 illustrates the mapping of a segment of a fat tree onto the fabric. It is a four-way fat tree with three levels (the same topology used in the design space exploration in Section 3.1.1). For legibility, we show the sub-tree of only one root node. In a full tree, the unused routing resources on the north, south and east edges of this diagram are used to connect multiple nodes together. Note that in the regions occupied by this design, only 8 of the 128 queues are not part of a switch and have gone unused. They are indicated by "x" in Figure 3.3. The ability to form switches out of slices in adjacent regions enables this high queue utilization. We have demonstrated the instantiation of a fat tree, because, with highly-connected, high-degree switches, it demands a careful configuration of the fabric.

3.3.2 Custom buffer allocation

One can use this fabric to increase the size of the logical buffers in the switches. By configuring a single slice to connect the input queue directly to the output queue, one forms

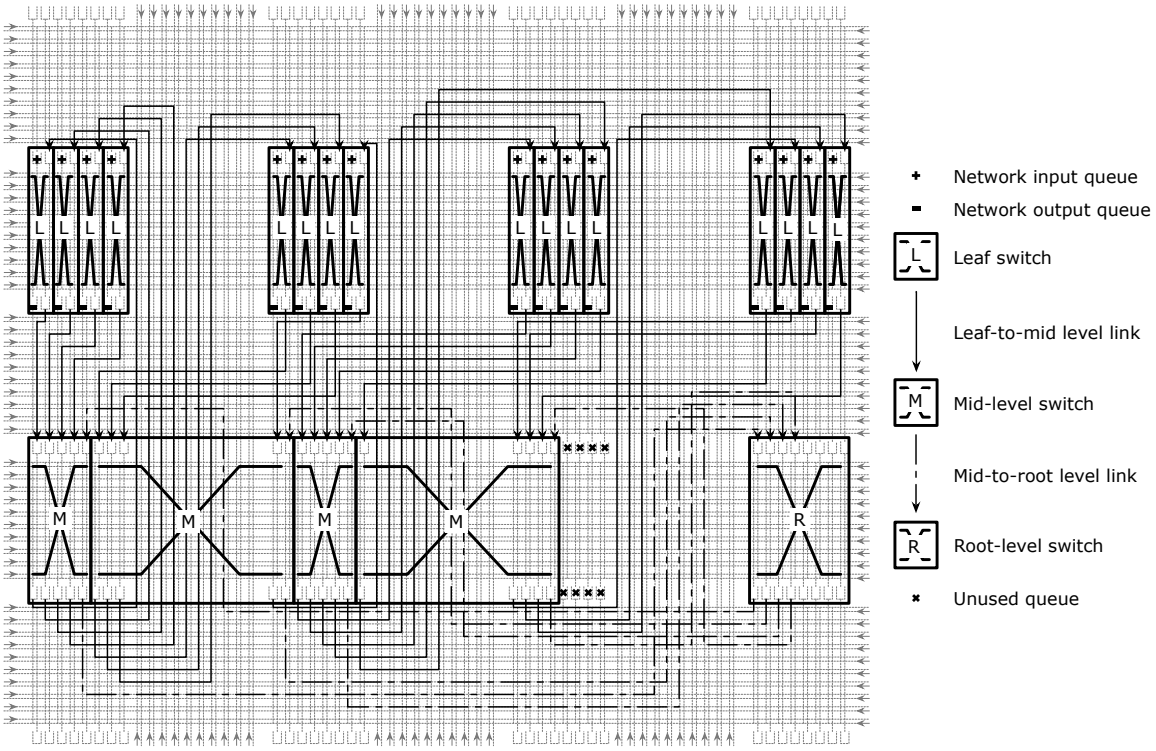

Figure 3.3: Instantiation of a fat tree network in polymorphic fabric. The links that form the mesh connections between root nodes are not shown.

a single logical buffer from two physically separate ones. In this buffer, packets advance from the back half of the buffer to the front half automatically, as necessary. Another way to view such a configuration is as a 1×1 switch, which does not route or switch, but simply advances packets forward as a queue would.

Switch buffer resources can be increased individually. One can selectively increase buffers on certain switches or ports as the application may demand. This capability dovetails nicely with other research that develops tools to identify the ideal network configuration, including buffer allocation, for an application [64, 90, 57].

3.3.3 Custom packet size

Finally, one can also increase the network packet size. Instead of aggregating sequential buffers, as in the previous example, one aggregates parallel buffers and links. In effect, this increases the network's packet size by increasing the datapath width. Both of these queue enlargement techniques increase the per-switch, or per-link resource usage requirements of the fabric. However, with the polymorphic fabric, the increased resource usage is commensurate with the analogous increase in resources of a classic ASIC interconnect.

Note that the polymorphic network configuration need not be restricted solely to general-purpose interconnects. It can support nearly arbitrary application-by-application tailoring. The performance benefits of such customization are well-documented in the literature [57, 53, 90], but were previously unattainable on a single hardware design.

3.4 Polymorphic fabric parametrization

The microarchitecture presented in the previous section outlines a schema of a polymorphic fabric for implementing on-chip networks. Several parameters remains unspecified. These parameters must be defined in order to actually implement a polymorphic fabric and are listed in the first section of Table 3.2.

3.4.1 Parameter space

As Table 3.2 states, there are N slices to a region, each containing a queue that is W bits wide and D entries deep. Each region has two $N \times H$ crossbars where H (the number of horizontal bars) must be at least N (the number of slices in each region). Similarly, each crossbar outside the regions is a $V \times H$ crossbar. We explore the parameter space of polymorphic fabrics defined in the first section of Table 3.2, totaling 108 polymorphic fabrics in all.

Based on the parameters, we can calculate the area of the *base element* of the fabric. The base element consists of a region and a crossbar and is the minimal unit of fabric. Multiple base elements can be produced to form larger swaths of fabric. The second section of Table 3.2 details how the area of a base element is computed. In a region, there are N

Table 3.2: Polymorphic fabric design space and area and configuration models.

Polymorphic network fabric design space parameters			
	Description	*Symbol*	*Range*
	Number of slices per region	N	2,4,8,16
	Queue width	W	32,64,128
	Queue depth	D	4,16,64
	Width of horizontal configurable link tracks	H	N, $2 \times N$
	Width of vertical configurable link tracks	V	N,$2 \times N$
	Network configuration	C	each network in first section of Table 3.1
Polymorphic fabric area model			
	Description	*Symbol*	*Value*
Synthesis	Queue area	$Queue_{area}$	$0.00002\ mm^2/bit$
	Wire pitch	χ	$0.00024mm$ [63]
	Crossbar area	$XBar_{area}(D_{in}, D_{out})$	$\chi^2 \times D_{in} \times D_{out} \times W$
Analytical	Region area	$PolyRegion_{area}$	$N \times W \times D \times Queue_{area}+$ $XBar_{area}(N, H)+$ $XBar_{area}(H, N)$
	Crossbar area	$PolyXBar_{area}$	$2 \times XBar_{area}(H, 2 \times V)+$ $2 \times XBar_{area}(V, 2 \times H)$
	Total area of base element	$BaseElement_{area}$	$PolyRegion_{area} + PolyXBar_{area}$
Polymorphic fabric configuration model			
	Description	*Symbol*	*Value*
	Base number of polymorphic queues	$PQueues_{base}$	manual switch configuration
	Queue depth adjustment	$Adjustment_{depth}$	$\begin{cases} \frac{C_{QueueDepth}}{D} & \text{if } C_{QueueDepth} > D \\ 1 & \text{otherwise} \end{cases}$
	Queue width adjustment	$Adjustment_{width}$	$\begin{cases} \frac{C_{QueueWidth}}{W} & \text{if } C_{QueueWidth} > W \\ 1 & \text{otherwise} \end{cases}$
	Adjusted number of polymorphic queues	$PQueues_{adjusted}$	$PQueues_{base} \times$ $Adjustment_{depth} \times$ $Adjustment_{width}$
	Number of base elements needed	$BaseElements_{needed}$	$PQueues_{adjusted}/N$
	Area of polymorphic fabric to implement C	$Fabric_{area}$	$BaseElements_{needed} \times$ $BaseElement_{area}$

queues and two crossbars. One crossbar from the input lines to the horizontal bars which is $N \times H$, and one crossbar from the horizontal bars to the output queues which is $H \times N$. Between regions is a second crossbar. It consists of four crossbars, two of size $H \times 2V$ for packets entering in the horizontal direction, from the left or right, and two of size $V \times 2H$ for packets entering in a vertical direction, from above or below.

The number of base elements needed to implement a particular network, as well as the resulting latency and throughput of the network, depend on how the polymorphic fabric is configured. This is where the last parameter, the network configuration C, comes in. To determine how much polymorphic fabric a particular network configuration will require, we calculate how many queues each of its switches will require. This calculation was performed manually for 2×2 switches up to 8×8 switches in each polymorphic fabric. These mappings account for the spatial blowup that occurs when a switch requires more bi-sectional bandwidth than is available horizontally in a single region. These initial queue counts are increased proportionally if a particular network configuration demands deeper or wider queues than those provided by the polymorphic fabric. Note that in the reverse situation, when a configuration calls for queues that are smaller than those provided by the fabric, the queues in the polymorphic fabric cannot be subdivided, and the extra queue capacity is simply wasted. Finally, we convert this queue count into a base element count by dividing the number of queues by the number of queues per region (N).

3.4.2 Parameter selection

We explore the parameter space by varying the parameters outlined in Table 3.2 labeled "Polymorphic network fabric design space parameters". In each fabric, we instantiate each of the networks from the initial network design space (from Section 3.1), counting how many base elements are required to implement each network according to the model in the section of Table 3.2 labeled "Polymorphic fabric configuration model". Finally, we translate these base element counts to area with the "Polymorphic fabric area model" from Table 3.2. Figure 3.4 plots the average area expansion of each possible polymorphic fabric design across all of the network configurations. Each bar corresponds to one polymorphic fabric design.

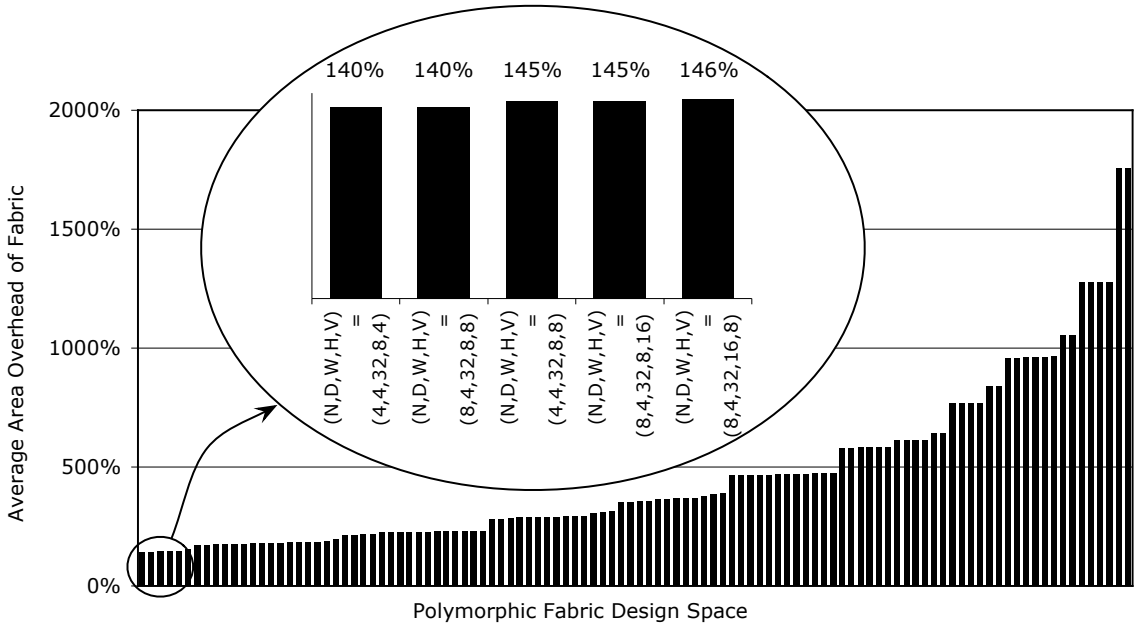

Figure 3.4: Average area overhead of each polymorphic fabric across instantiations of all members of the on-chip network design space.

The height of the bar represents the area of the polymorphic fabric relative to the ASIC area of the network it is implementing. There are several overriding trends which govern the quality of a polymorphic fabric:

- The "worst" fabrics, those that incur the greatest increases in area consumption on the right side of Figure 3.4, are those which over-provision the polymorphic fabric queues, either in depth or width. When configuring these fabrics to implement a network that requires smaller queues, the excess queue capacity ends up costing area and offering no benefit.

- For a fixed queue size, the more constrained the horizontal routing resources, the greater the area overhead. When a switches input count exceeds the number of horizontal routing wires, one must add extra stages to the switch to achieve the necessary bisection bandwidth. This consumes unnecessary queues and with them, area.

- The number of slices per region and the amount of vertical routing do not influence area overhead with any notable pattern. Vertical routing resources are less significant than horizontal routing resources because vertical wires are dedicated exclusively to routing.

On the left hand side of Figure 3.4 we highlight the fabrics that incur, on average, the smallest area overhead. In keeping with the overall trends, the best polymorphic fabrics provide narrow and shallow queues, with generous routing resources in the horizontal and vertical directions. These fabrics minimize waste, because queues are aggregated only when the instantiated network design requires it. The most area-efficient polymorphic fabric incurs a 40% area overhead compared to the average size of the ASIC network designs.

3.5 Evaluation of fabric flexibility

We continue our evaluation of the most efficient polymorphic fabric, one with four slices per region, four, 32-bit packets per queue, eight horizontal and four vertical routing tracks ($N = 4, D = 4, W = 32, H = 8, V = 4$). We now examine the amount of this fabric required to implement each Pareto optimal network design from Section 3.1.1. Figure 3.5 plots these results. Each pair of bars corresponds to an optimal network design. The height of the black bar indicates the ASIC area of the network, and the height of the white indicates the amount of polymorphic fabric required to implement the same network.

One can use this information to examine the flexibility of the polymorphic fabric. Once one has committed to spending a certain amount of area on a polymorphic fabric (corresponding to some point on the y axis of Figure 3.5), one can implement *any* network for which the corresponding white bar does not exceed that area budget. Moreover, provided it fits in the fabric, the network configuration needs not be constrained to be one of these designs.

Figure 3.6 plots this flexibility as a function of the selected area budget. For small area budgets, the number of Pareto optimal networks that fit inside the fabric is much smaller, by approximately half, than the total number of Pareto optimal networks. This indicates that when network area is scant, one is better off implementing a fixed-function

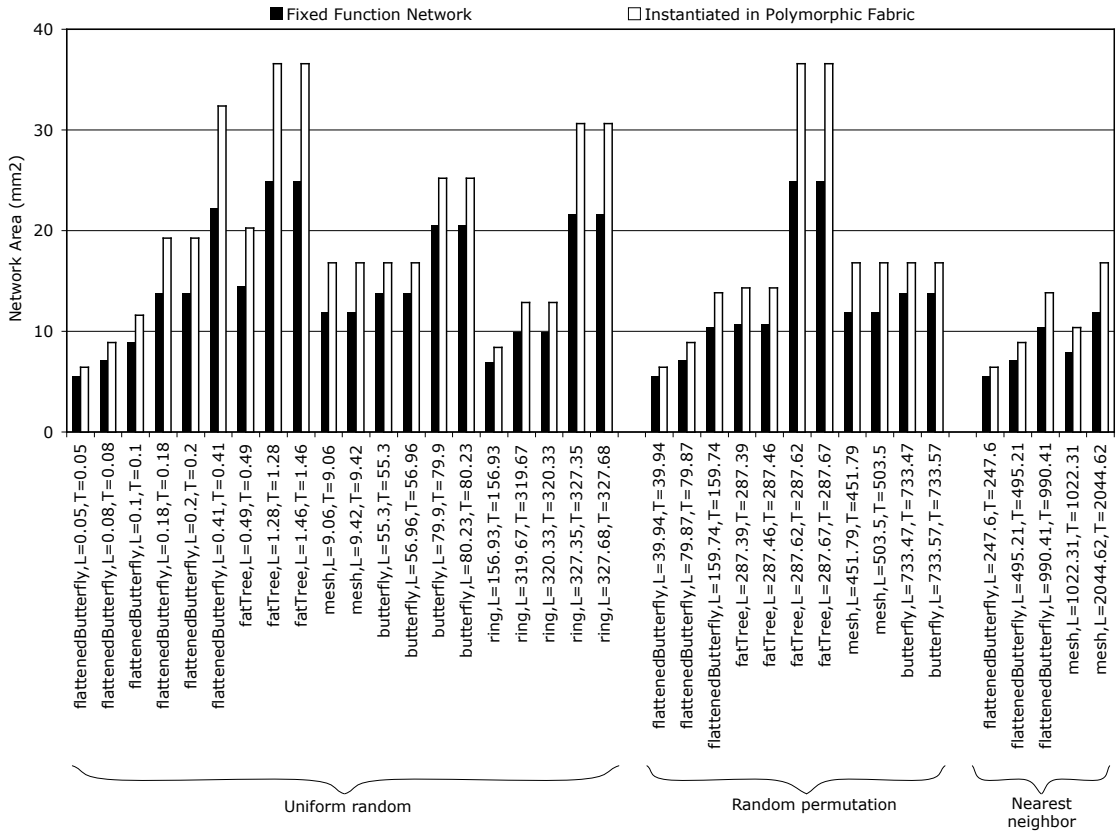

Figure 3.5: Area overhead of most efficient polymorphic fabric when configured with Pareto optimal networks.

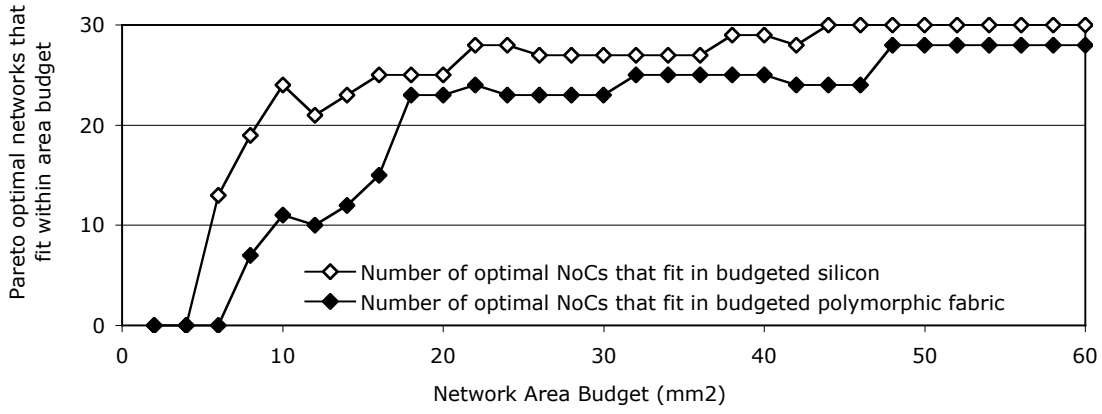

Figure 3.6: Flexibility of polymorphic fabric as a function of the network area budget.

network, because the flexibility of a polymorphic network is compromised. However, once approximately $18mm^2$ has been allocated to the interconnect, a polymorphic fabric of a given size can implement the overwhelming majority (90%, on average) of the viable (with respect to size) optimal fixed-function networks. As a reasonably large chip, the I/O cap clearly falls into the latter category.

Thus, this data indicate that there are only two cases in which one should *not* build a polymorphic network: (1) when the application is fixed and well known ahead of time (e.g., embedded fixed-function devices); or (2) if the interconnect area budget is less than approximately $18mm^2$. This apparently-broad potential use opens a host of questions which we discuss at the end of this thesis in Section 9.2.

Chapter 4

EMPIRICAL BRICK FAMILY DESIGN

When it comes to bricks, there are several important architectural questions to address. How do bricks communicate? How large is a brick? What is the appropriate functionality for bricks to provide? We investigate these questions by examining how the physical constraints placed on bricks influence the architectural decisions.

4.1 Brick form factor

We begin by discussing some of the physical constraints the brick and mortar chip system imposes on a brick.

4.1.1 Brick size

A brick, naturally, cannot be smaller in area than the functional circuit it contains. This circuit consists of an IP core, any local memory it requires, and logic to send and receive messages over the physical interface to the I/O cap.

Bricks must communicate with other bricks through the I/O cap. Large bricks can integrate more logic and/or memory on a single brick, thereby increasing performance via the decreased amount of inter-brick communication. Larger bricks also mean those bricks are more specialized, reducing their potential re-use across designs; hence, they are more expensive. In addition, large bricks will have lower fabrication yields, making them more expensive than small bricks. In contrast, small bricks are likely offer more design flexibility and, because they are less specialized, more potential re-use across designs.

This trade-off in the architecture of a brick is best summarized as follows: dividing a design into multiple bricks provides the benefits – brick re-use, design flexibility, etc. – but also the costs – increased latency between components on separate bricks, and constricted bandwidth as communication is routed through the I/O cap. For this technology, it is

important to find a suitable balance between integration and generality in brick function.

Bricks will also have a maximum useful size. Early VLSI engineers observed a phenomenon dubbed "Rent's rule" [74]. Rent's rule states that a circuit's required I/O is proportional to an exponential of its area ($IO \propto Area^{\beta}$ where $\beta < 1$). While the precise constant β used in the rule changes depending on the type of circuit, the structure of the rule does not [17]. Prior work suggests that $\beta = 0.45$ for processors and memory, and $\beta = 0.6$ for less structured logic [17]. Because the I/O of a brick is proportional to its size, beyond some brick size the circuit contained inside the brick will not be able to utilize all of the I/O available to it. One should design bricks that use the available I/O, because it is through this I/O that fixed, inflexible bricks are connected in unique ways to produce unique chip designs. Beyond this maximum brick size, designers no longer save any I/O by integrating functions onto a large brick.

Finally, there can be multiple brick sizes. The more brick sizes offered, the better the area and I/O offering of the bricks can match the true area and I/O requirements of the circuit. However, we will require that the bricks conform to standard sizes, because it is very difficult to design an I/O cap to interconnect arbitrarily-sized bricks.

4.1.2 Brick shape

Ultimately the bricks will be assembled into a two-dimensional array to be bonded to the I/O cap. A simple, locality-aware placement algorithm could readily produce a suitable 2D brick arrangement. To use space efficiently, it is important that the bricks be shaped such that they can be packed together. While there are ddvantages to different brick shapes, such as rectangles with the golden ratio, we will use square bricks.

4.2 Function selection

The applications for which we envision using brick and mortar manufacturing are those which currently employ traditional ASICs, for example, wireless transceivers, media encoding/decoding, system-on-chip (SoC) integrations, etc. In these realms, the functional blocks forming a design are fairly large (e.g., FFT engines, JPEG compressors, embedded microprocessors).

The cost savings of brick and mortar stems from re-use of the component parts. If a particular brick is useful to only a small number of users, it is likely not worth including in the family. The cost to produce it would be amortized over only a small number of chips making it a very expensive brick. Fortunately, there is evidence of strong functional component re-use in several design domains. For example, video and audio codecs that implement different standards still share many blocks in common.

To develop this brick family, we used freely available IP cores to define bricks, collecting Verilog source code from `opencores.org` [94] and other sources of publicly available IP [131, 120, 124]. Chapter 9 describes several other possible brick families that might be useful, as well as ways to enhance the family designed here. All in all, there were 23 cores, which we list in Table 4.1.

4.3 Technology selection

4.3.1 Standard cell libraries

We compiled the designs with the Synopsys DC Ultra design flow [130], targeting the 90nm TSMC [135] ASIC standard cell library. We used a commercial memory compiler [12] to generate optimized memory IP blocks.

4.3.2 Brick to I/O cap bonding

Flip chip bonds connect the I/O pads of each brick to the I/O pads in the cap. Studies indicate that each bonding bump requires $25\mu m \times 25\mu m$ in surface area and can provide at least 2.5Gbps bandwidth [47]. We reserve 15% of these I/O pads for power, ground and clock, but the remainder are available for data communication.

There are other bonding technologies that one could use here. Proximity communication is one alternative [43]. In proximity communication the communication vias are left dangling on the metal surface of the die. When dies are aligned, dangling via to dangling via, they can communicate via capacitive coupling. This method has several advantages. The pads are significantly smaller than flip chip pads, because they track die-internal feature sizes, as opposed to the larger die-external features of flip chip pads. Proximity communication

also makes it possible to execute real applications on the chip without physically bonding bricks to the I/O cap. While flip chip bonding allows only a built-in self-test (BIST) of the parts prior to component bonding, proximity would allow more complete, whole chip testing prior to permanent component bonding. We chose to use flip chip bonding in spite of these advantages, because it is a more mature process that is already common in fabs today.

4.4 Brick assignments

Just as Rent's rule was originally developed through empirical observation, we too determine the appropriate brick size through empirical data. We use synthesis to determine the area and I/O requirements of the desired IP cores and then select the most reasonable brick sizes based on this data.

Synthesizing a functional Verilog module determines its area, I/O pin count, and maximum operating frequency. We compute an upper bound on the bandwidth requirements of the module by multiplying the pin count by the maximum operating frequency.

Based on this data and the constraints outlined above, we conclude that three brick sizes are reasonable: small ($0.25mm^2$), medium ($1.0mm^2$) and large ($4.0mm^2$). The bricks are square shaped with four small bricks packing into the footprint of a medium, and four mediums packing into the footprint of a large.

Table 4.1 shows the assignments of IP core to brick. Each brick size offers a fixed I/O bandwidth based on the brick area. We have converted these bandwidth limitations into upper bounds on the brick clock speed by assuming the 2.5Gbps per pin from prior work [47]. This upper bound is also restricted by the speed at which the IP block can operate in a 90nm TSMC standard cell process. When present, the lower frequency bound indicates the minimum speed required to meet application requirements (e.g., an ethernet device must process data at the line rate).

The IP cores listed in Table 4.1 are sorted according to their sizes. We assign IP blocks to the smallest brick size which could meet their area and application bandwidth needs. Note that none of the medium bricks benefits from increasing the brick size, indicating that none of them is I/O constrained. This is a direct effect of Rent's rule. The data do

Table 4.1: IP block synthesis results and brick assignments.

Function	Cite	Circuit Area (um^2)	Max Circuit Freq. (MHz)	Min Perf. (Mbps)	$0.25mm^2$ brick freq. range (MHz)	$1.0mm^2$ brick freq. range (MHz)	$4.0mm^2$ brick freq. range (MHz)
Small Bricks							
USB 1.1 physical layer	[94]	2,201	2941	12	**2 - 2941**	No benefit	No benefit
Viterbi	[124]	2,614	1961	-	**N/A - 1961**	No benefit	No benefit
VGA/LCD controller	[94]	4,301	1219	-	**N/A - 1046**	N/A -1219	No benefit
WB DMA	[94]	13,684	1163	-	**N/A - 521**	N/A - 1163	No benefit
Memory controller	[94]	29,338	952	-	**N/A - 843**	N/A - 952	No benefit
Tri-mode ethernet	[94]	32,009	893	1000	**125 - 893**	No benefit	No benefit
PCI bridge	[94]	76,905	1042	-	**N/A - 610**	N/A - 1042	No benefit
WB Switch (8 master, 16 slave)	[94]	81,073	1087	-	**N/A - 88**	N/A - 353	N/A - 1087
FPU	[94]	85,250	1515	-	**N/A - 505**	N/A - 1515	No benefit
DES	[94]	85,758	1370	1000	**16 - 1203**	16 - 1370	No benefit
16K SRAM (Singleport)	[12]	195,360	2481	-	**N/A - 2481**	No benefit	No benefit
Aho-Corasick string match	[131]	201,553	2481	-	**N/A - 1331**	N/A - 2481	No benefit
RISC Core (no FPU,8K cache)	[94, 12]	219,971	1087	-	**N/A - 1087**	No benefit	No benefit
8K SRAM (Dualport)	[12]	230,580	1988	-	**N/A - 1988**	No benefit	No benefit
Medium Bricks							
Triple DES	[94]	294,075	1282	1000	No space	**16 - 1282**	No benefit
FFT	[120]	390,145	1220	-	No space	**N/A - 1220**	No benefit
JPEG decoder	[94]	625,457	629	-	No space	**N/A - 629**	No benefit
64K SRAM (Singleport)	[12]	682,336	2315	-	No space	**N/A - 2315**	No benefit
32K SRAM (Dualport)	[12]	733,954	1842	-	No space	**N/A - 1842**	No benefit
RISC core (64K cache)	[94, 12]	864,017	1087	-	No space	**N/A - 1087**	No benefit
Large Bricks							
256K SRAM (Singleport)	[12]	2,729,344	2315	-	No space	No space	**N/A - 2315**
128K SRAM (Dualport)	[12]	2,935,817	2882	-	No space	No space	**N/A - 2882**
RISC core (256K cache)	[94, 12]	3,111,025	1087	-	No space	No space	**N/A - 1087**

indicate, however, that five of the thirteen small bricks could take significant advantage of the increased I/O bandwidth that a larger brick affords. In these cases, we envision brick builders will do one of two things: (1) provide two different brick sizes, with the smaller brick supporting only lower frequency designs, or (2) more likely, redesign the bricks to take advantage of the added area of a larger brick. We did not investigate this aspect of brick design in this study, but one option would be to group blocks of similar functionality (e.g., an ethernet and USB controller on the same "general purpose computing I/O" brick). Another option is to tune buffer sizes on the design. For example, the Aho-Corasick [131] block can use additional buffer space to support more complex matching patterns.

Chapter 5

HARDWARE COMPONENT ASSEMBLY AND PACKAGING

Previous chapters have examined the physical building blocks of a brick and mortar chip: the bricks and the I/O cap. In this chapter we explore techniques to assemble and align the bricks in the desired arrangement and to physically bond them to the I/O cap.

The most direct approach to brick arrangement is robotic assembly. Manipulating and aligning devices at approximately the $1\mu m$ scale that is required for brick and mortar is a solved mechanical engineering problem, and commercial systems already exist to do this [156]. The drawback of these devices is cost. As with all non-recurring manufacturing costs, the greater the expense, the greater the impact on the final price of the chip, particularly small batches of them. As the principle goal of brick and mortar chips is to provide low volume manufacturing at low cost, we seek a lower-cost manufacturing technique.

5.1 Fluidic self-assembly

One such technique is fluidic self-assembly (FSA) [151]. With FSA, the objects to be assembled, bricks in our case, are suspended in water in a closed container above a template. This template contains an array of indentations corresponding in shape and size to the bricks. The template/water/brick system is then carefully shaken, allowing the bricks to move about and slip into the indentations in the template.

Figure 5.1 illustrates this process. To accommodate multiple brick sizes, FSA begins with one assembly template per brick size. A single FSA process cannot assemble bricks of differing sizes, because different brick sizes require different agitation forces. Simply running at the maximum agitation force will work fine for the larger bricks, but the small bricks will never settle into the template. This means that the assembly process would require one FSA system per brick size.

Bricks begin loosely and randomly placed en masse in this water-filled compartment. The

compartment is then gently shaken, causing the bricks to move around. As the bricks move about, they fall into the brick-sized holes in the template at the bottom of the compartment. Eventually all holes in the template will fill, completing the arrangement of bricks. Previous work [48] has demonstrated this technique using components that have approximately the same size as the bricks from Chapter 4.

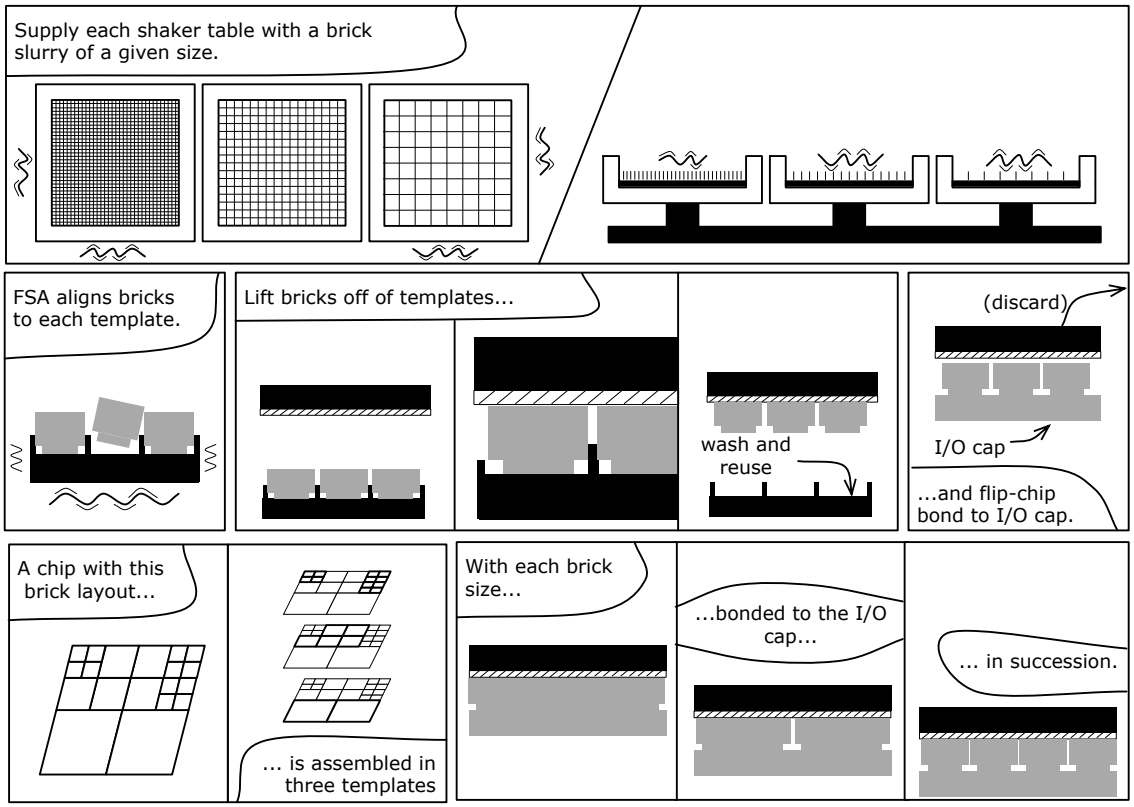

Figure 5.1: Fluidic self-assembly of a brick and mortar chip.

We propose the following process for inexpensive mass-production of brick and mortar chips using FSA.

5.1.1 Brick modification

We modify the basic architecture of a brick to include a simple AC coupled power and communication device that is capable of transmitting a unique identification tag. This tag

identifies the type of the brick to the substrate. This technology is well-established and in wide use in RFID tags [43, 66, 87]. A simple version is suitable here, as the communication of the identification tag need travel only a few micrometers, instead of a few feet.

5.1.2 Chemically-directed assembly

The specific FSA process on which we base our experiments is a semi-dry one [151, 99, 107, 108]. In this process, bricks of the same size are mixed with a small amount of water and poured over an assembly substrate with correspondingly-sized binding sites. Parts arrive at binding sites by random motion, thanks to the shaking of the substrate. When a brick nears a binding site, there are three forces that determine how it falls into the site. First, bricks are not exact squares, but rather modified in shape such that they can fit into binding sites only when aligned properly [29, 48]. Second, they fall due to gravity. Third, capillary forces assist gravity. This sort of capillary force-driven self-assembly [128] relies on the minimization of inter-facial energy.

The substrate on which bricks are assembled can be coated with a polymer pNIPAM (poly-N-isopropylacrylamide [55, 96, 58, 148]) which can be reversibly switched between hydrophobic and hydrophilic states through a small change in the local temperature of the binding site. This allows bricks to be attracted to, or repelled from, any site. As bricks fall into binding sites, the template queries the brick, using the AC-coupled communication, to determine the brick's type. The substrate can then reject bricks that fall into the wrong location by setting the polymer's state to hydrophobic.

The ability to control the ejection or retention of a brick at a certain location allows the logic embedded in the template to steer an otherwise random assembly process to the desired final brick arrangement. Bricks that are the wrong type for a given location, or simply do not pass the testing as described below, can be ejected from the template while all others can be retained.

5.1.3 Brick testing

Because they are bare, unpackaged silicon dies, bricks are not trivial to test. This includes functional testing at speed and burn in. Historically, one had either to package a die or bond it to a board in order to perform these tests. Packaging is not an option for a technology such as brick and mortar or multi-chip modules (MCMs), the field which gave this particular problem its name: the *known good die* problem. This threatens overall brick and mortar chip yield, because an inability to test component parts until after assembly means that a single bad component will ruin a whole assembly.

There are several solutions currently available, including temporary bonding [16] and communication technologies that allow power and at-speed communication with a bare die [150, 110, 4]. Any one of these options could work to weed out the bad bricks from brick and mortar chip assemblies. We have selected capacitive coupling because it does not require maneuvering the bricks in any way other than what the described assembly system already requires.

5.1.4 Composite assembly

Once a complete template of same-size bricks has been formed, it is robotically lifted off the substrate and placed onto the I/O cap, as shown in Figure 5.1. At a minimum, one must perform this step once for each size brick used in the design. However, one could repeat this step more than once for each size brick if so desired. The following section will outline circumstances under which this might be the case.

5.2 Interaction with architecture

An interesting fact of brick and mortar manufacturing is that there is an extremely close interaction between *what* one manufactures and *how* one manufactures it. In this section we explore how the architecture of the chip affects and should be affected by the FSA process.

For this study we simulated the FSA process itself. The simulator matches production capabilities of current experimental FSA systems [48]. It models assembly of bricks onto the substrate, programmatic discard if they are the wrong type, and accidental discard (a

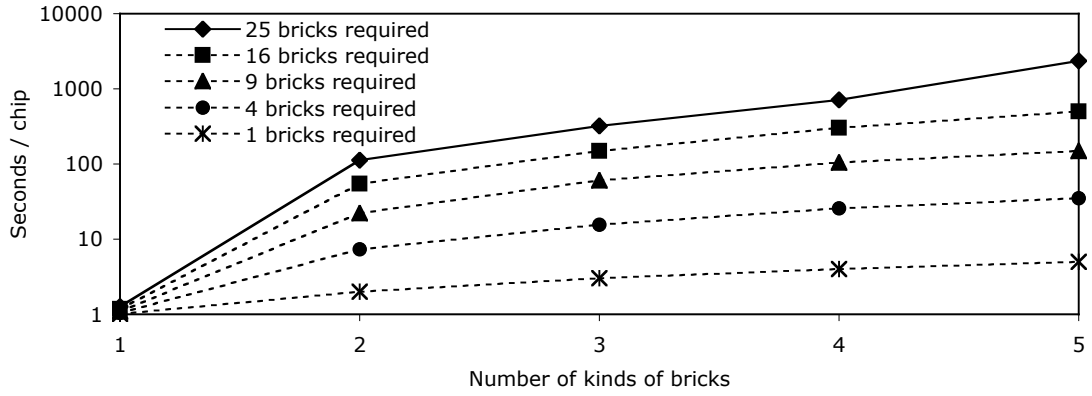

Figure 5.2: Brick and mortar chip assembly time as a function of brick heterogeneity and design size.

feature of the FSA process which, in a well-tuned system, happens 5% as often as bricks bind to sites [48]). For this study, we utilized synthetic chip designs to examine assembly times as the number of bricks and the number of kinds of bricks in a design varies.

The data in Figure 5.2 show that increasing the number of kinds of bricks (their heterogeneity) or the number of bricks required increases the assembly time exponentially. This means that there are strategic decisions one can make at multiple stages – from brick architecture, to chip architecture, to how one uses the assembly process – to control assembly time. In the sections below we explore some of those decisions.

5.2.1 Brick design

The data in Figure 5.2 indicate that assembly time grows exponentially with the number of kinds of bricks present in an FSA process. This means that one can reap significant gains in assembly time by offering one single, slightly reconfigurable brick instead of two different brick types with largely similar functionality. Brick functions where this might be possible include byte- vs. word-aligned memories and bus interface standards.

5.2.2 Interconnect design

Another interesting interaction between architecture and manufacturing is in the placement of bricks. Specifically, if we can architect the system such that the placement of bricks need not be identical for every instance of a design that is produced, the assembly time will shorten dramatically. The data in Figure 5.3 indicate that by relaxing constraints on brick placement, the assembly process will complete significantly more quickly. One way to accomplish this is to carve the grid of bricks into square regions of *slackRadius* bricks on a side. A brick that falls *anywhere* within a region can satisfy any of the bricks required by that region (provided a brick of that type was indeed required by the region). The original, strict, brick placement is simply the special case of a 1×1 region ($slackRadius = 1$).

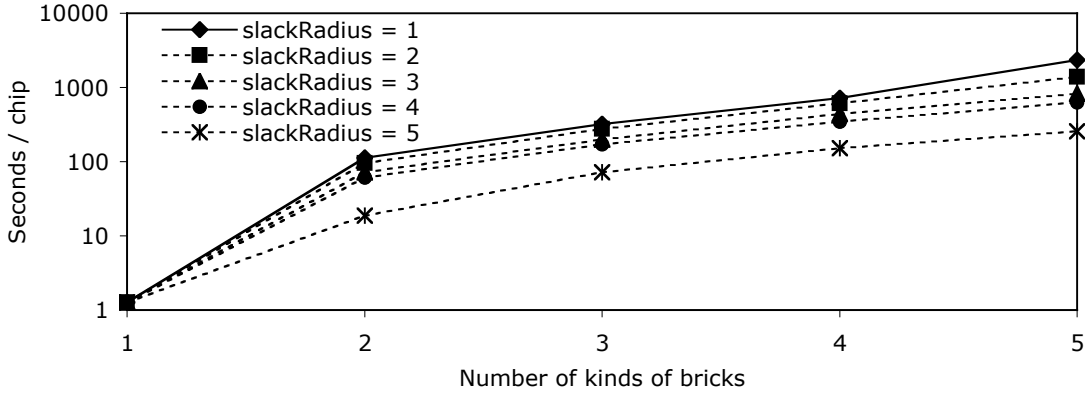

Figure 5.3: Brick and mortar chip assembly time as a function of brick heterogeneity and brick placement relaxation.

Note that the solid lines in Figures 5.2 and 5.3 are identical. The 25-brick line in Figure 5.2 is the same as the $slackRadius = 1$ line in Figure 5.3, because the data in Figure 5.3 was collected using 25-brick designs and the data in Figure 5.2 with $slackRadius = 1$. These graphs indicate that one can save approximately the same amount of time introducing a *slackRadius* of 5 to a 25-brick chip (Figure 5.3), as by reducing the size of the design from 25 to 9 bricks (Figure 5.2).

If placement slack is allowed, a single chip design could result in a different brick layout

for each chip that is assembled. In order to accelerate the assembly process in this way, the communication infrastructure of the I/O cap must be sufficiently general or programmable to handle the unpredictability in placement. Therefore the interconnect must be flexible not only on a per-design basis, but a per-chip basis. It is up to the designer to determine how much slack to introduce. He or she will have to make a decision based on the trade-offs between assembly time, desired performance, and the latency tolerance of the design itself.

5.2.3 Assembly management

Thus far we have described the brick and mortar assembly process as one that utilizes different FSA stations to assemble bricks of the same size. These partial-chip assemblies are then robotically assembled onto an I/O cap. However, nothing requires that multiple assemblies be used *only* for bricks of different sizes. Given some number of template assembly stations, what mix of bricks should each station assemble?

Taken to the limit, the data in Figure 5.2 indicate that, in theory, one should use a separate self-assembly process for each brick type. In practice, that would require one assembly substrate per brick type. This might not be feasible, as it essentially requires purchasing sufficient equipment for the most heterogeneous chip design one might ever make. Instead, one could conserve equipment and either (1) pair up brick types (of the same size) to assemble at the same time on the same template, or (2) assemble the two brick types in sequence, first the first brick type, then the second, on the same table. The data in Figure 5.2 indicates that both options will result in a faster assembly.

5.3 Component binning

There will be some natural variation among all of the bricks that are produced. Brick and mortar affords a unique opportunity to pre-select bricks based on their process variation characteristics prior to production. With brick and mortar, it is possible for the chip designer to "cherry pick" the high-performance bricks and build a chip entirely out of them. This is similar to how Seymour Cray separated the high- and low-performing gates when he built the Cray-1 [118]. The exact benefit gained by such binning will depend on the particular

chip in question. Thus, we will quantify the benefits of binning in the context of our case study in Chapter 7.

5.4 External communication

Like any other chip, brick and mortar chips must be packaged for inclusion in a larger system. One function of a package is to transfer external signals from the die to the external pins of the package. Packages accomplish this by using a carrier. A carrier is a small laminate board which sits underneath the chip. It is responsible for receiving signals from the upper metal layers of the chip and routing them to the pins of the package.

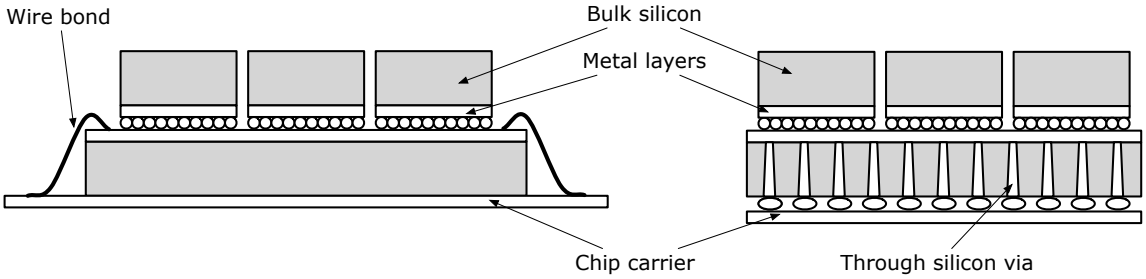

Figure 5.4: Two alternatives for brick-and-mortar-chip-to-carrier communication: wire bonds and through silicon vias.

Because brick and mortar chips are constructed with flip chip bonding, the metal layers of both the bricks and the I/O caps are sandwiched in the middle of the silicon wafers. There are two ways to retrieve signals from those metal layers, illustrated in Figure 5.4. The first technique uses wire bonds, a well-established form of chip-to-carrier connections. This is illustrated on the left side of Figure 5.4. Wire bonds originate from the perimeter of a chip, so the I/O cap must be slightly larger to accommodate a fringe of wire bond pads. In addition to this increased I/O cap size, wire bonds themselves require larger carriers than other options, thereby further increasing the size of the whole package.

The second technique, through silicon vias, is illustrated on the right side of Figure 5.4. Through silicon vias (TSVs) are a newer technology in which metal vias are bored through the silicon layers of a chip. This could be applied to the I/O cap in order to bring signals

to an external surface of the chip, which could then be flip chip bonded to the carrier. Unlike wire bonds, TSVs do not increase the size of the I/O cap or carrier. However, as the picture shows, the vias themselves eat into available silicon of the I/O cap, leaving less space for the interconnect memory and logic. Most importantly, however, since TSVs are more expensive, the wire bond would probably be a better option for these low-cost chips.

Chapter 6

SPATIAL APPLICATION SCHEDULING

An essential characteristic of brick and mortar chips is that the functional cores are *disintegrated* relative to their monolithic, single-die counterparts. Because this disintegration places computational cores on separate dies, it becomes very important that all software be *intelligently* mapped onto the chip. In order to maximize application performance, the mapping must be aware of and respect the marked difference between local communication within a brick and remote communication between bricks. As much as the I/O cap tries to speed inter-brick communication, the difference between remote and local communication will always be more pronounced in brick and mortar than in a monolithic chip because the essence of brick and mortar requires inter-die communication.

6.1 Tiled architectures

Brick and mortar chips represent an extreme design point in the space of *tiled architectures*. Tiled architectures consist of multiple processing elements (PEs) connected by an on-chip interconnect. In addition to brick and mortar, Raw [138], SmartMemories [84], TRIPS [92] and WaveScalar [127] are all examples. These designs address several emerging, critical problems in monolithic processor design, including design complexity, wire delay, and fabrication reliability. A simple PE decreases both design and verification time; PE replication provides robustness in the face of fabrication errors; and the combination reduces wire delay for both data and control signal transmission. The result is a scalable architecture that allows a chip designer to target different levels of performance, with different area budgets [126].

To maximize application performance, it is essential that software be intelligently mapped onto the computational tiles. Our results comparing several algorithms show over an order of magnitude difference in their performance when executing single-threaded applications.

For unicore (non-tiled) designs, instruction schedulers focus exclusively on deciding when an instruction should be fetched. For example, scheduling a load instructions early helps hide its latency. This is called *temporal scheduling*. On a tiled architecture however, the scheduler also decides where an instruction will execute. For example, placing dependent instructions on the same or adjacent tiles reduces producer-to-consumer operand latency. Instruction scheduling for a tiled architecture also includes this second element of *spatial scheduling*.

Our goal is to find a practical scheduling algorithm that generates efficient code schedules for brick and mortar chips. In order to design an algorithm that generalizes to arbitrary combinations of bricks, we will make no assumptions about the type of cores to which instructions are to be mapped. The temporal scheduling algorithm that will perform best will depend on the particular combination of bricks in a chip, so that will not be our focus. However, all sets of bricks will require a good spatial scheduling algorithm. Because the WaveScalar architecture [127] employs the dataflow model of computation, instruction scheduling for this architecture is exclusively a question of spatial scheduling. Thus, it is an ideal architecture on which to explore and evaluate exclusively spatial instruction scheduling techniques.

6.2 Overview of WaveScalar

We begin by describing WaveScalar, the target architecture for the scheduling techniques presented here. We confine our description to a high level, except when specific details are relevant to instruction scheduling. A more exhaustive description of the architecture appears in [127].

6.2.1 Dataflow instruction set architecture

WaveScalar is a dataflow architecture. As with all dataflow architectures (e.g. [41, 40, 70, 117, 54, 98, 109, 52, 97, 35, 13]), an application is represented as a dataflow graph (DFG), with control dependencies converted into data dependencies. Nodes in the graph are instructions, and directed edges between them represent operand dependencies. Unlike traditional processors, dataflow machines do not have a program counter; instead instruction fetch,

like instruction execution, is data-driven. Also, instead of a register file, they have a token store which associates tokens, comprised of operand values and instruction-identification tags, with the appropriate instruction. WaveScalar's token store is distributed across the processor, with the processing elements. When all of the operands for a particular instruction have arrived at a PE's token store, the instruction can be executed. This is known as the *dataflow firing rule* [41, 40].

Unlike previous dataflow ISAs, WaveScalar supports a memory model which commits memory accesses in program order. This enables it to execute applications composed in imperative languages, such as C. In WaveScalar, the store buffer implements this memory model, the details of which are not relevant to our work here, but can be found in related work [127].

6.2.2 Microarchitecture

WaveScalar's microarchitecture consists of a grid of simple, 5-stage pipelined dataflow processing elements (PEs). Each static instruction in a WaveScalar program executes in a PE. Each PE contains a small, local instruction cache, which can hold up to 64 static instructions at a time. The microarchitecture swaps instructions in and out of these caches, as program execution requires.

To reduce communication costs within the grid, we organize PEs hierarchically, as depicted in Figure 6.1. Two PEs are first coupled, forming a *pod*; within a pod, instructions can execute and send their results to their partner PE in a single cycle. Four PE pods are grouped into a *domain*, within which producer-consumer latency is five cycles. Four domains form a *cluster*, which also contains a store buffer and a traditional L1 data cache. A single cluster, combined with an L2 cache and main memory, suffices to run any WaveScalar program. To build larger machines, a dynamically-routed on-chip packet network connects multiple clusters. Communication latency between clusters depends upon how far apart they are on the chip. A directory-based coherence protocol maintains data cache coherence. The coherence directory and the L2 cache are distributed around the edge of the grid of clusters. Table 6.1 depicts the micro-architectural parameters used for this study.

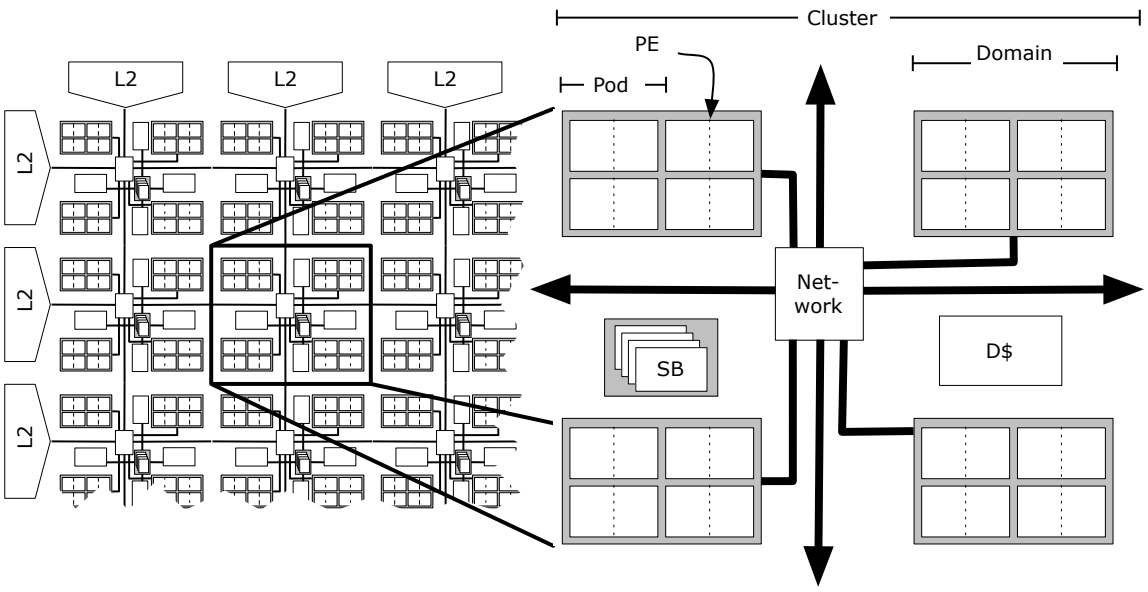

Figure 6.1: Microarchitecture of the WaveScalar processor.

6.2.3 Instruction loading

The microarchitecture and the runtime system collaborate to load instructions into the WaveScalar processor. When a token's consumer instruction is not resident in the processor, the processor signals the runtime. The runtime then decides where to place the instruction, either by checking a statically constructed table (produced by the compiler) that maps instructions to PEs, or by using an online algorithm to create a new mapping. It also notifies the microarchitecture of the location of the consumer instruction, to bypass the runtime system for future operands. Eventually, the entire working set of instructions will have been loaded into the PE grid in this manner and the runtime loading mechanism will be largely out of the way of execution.

Each PE contains a functional unit, specialized memories to hold operands, and logic to control instruction execution and communication, as shown in Figure 6.1. The PEs contain storage for several different static instructions, although only one can execute per cycle. Operand matching is handled locally at each PE.

Table 6.1: Micro-architectural parameters of the WaveScalar processor.

Processing element parameters	
PE input queue	16 entries, 4 banks
PE output queue	8 entries, 4 ports (2r,2w)
PE pipeline depth	5 stages
Computational hierarchy	
PEs per domain	8 (4 pods)
Domains / cluster	4
Memory hierarchy	
L1 caches	32KB, 4-way set associative, 128B line, 4 accesses per cycle
L2 Cache	16 MB shared, 1024B line, 4-way set associative, 20 cycle access
Main RAM	1000 cycle latency
On-chip communication network	
Switch	4-port, bidirectional
Latency within pod	1 cycle
Latency within domain	4 cycles
Latency within cluster	7 cycles
Latency between clusters	7 + cluster distance

6.3 Hierarchical approach to scheduling

We propose to schedule instructions hierarchically in two passes. The first pass, which we call *coarse scheduling* allocates instructions to domains. The second pass, *fine scheduling*, further refines this initial domain assignment by designating instructions to individual processing elements. Figure 6.2 illustrates the process.

We broke the scheduling problem at the domain level, instead of some other place in the micro-architectural hierarchy, for two reasons. First, the network designs within and between domains are quite different. Within a domain, all communication occurs via a full crossbar interconnect, which has a fixed and relatively short latency. However, between domains communication latency is variable and relatively long, as it traverses a buffered packet-switching network. Second, a domain holds 512 static instructions. This size captures a large enough instruction working-set that a coarse scheduling algorithm can make gross decisions based upon overall program graph structure, without concerning itself about optimizing every intra-domain micro-architectural trade-off.

The hierarchical approach to scheduling also helps to manage the large instruction scheduling problem size. Breaking up the problem into sub-problems allows the sched-

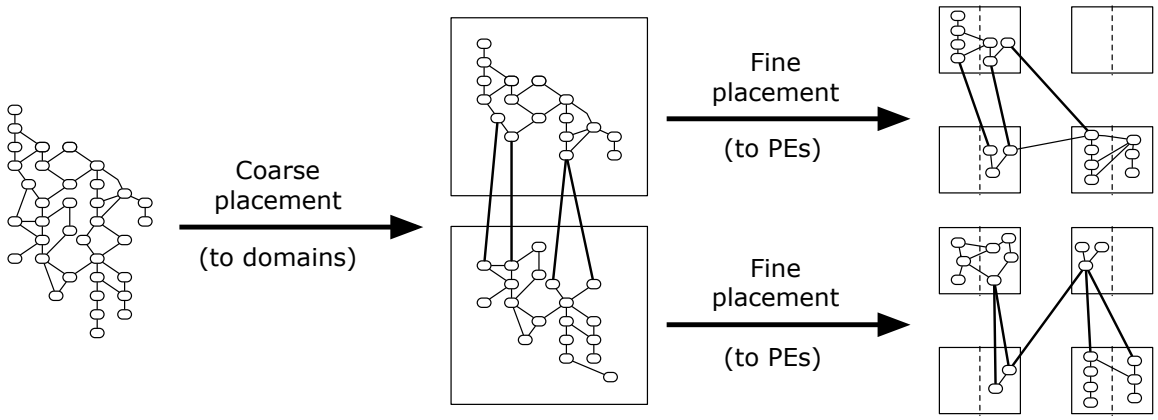

Figure 6.2: Coarse and fine stages of hierarchical instruction placement on WaveScalar.

uler to tackle the problem in smaller pieces. WaveScalar has a large number of processing elements (2048 in our target design). With the hierarchical approach, instead of considering all instructions and all processing elements at once, the coarse scheduler need only consider all instructions and the 64 possible domains. Then the fine scheduler is left with some fraction of the instructions (those assigned to a domain) and only 8 possible PE locations for each of those instructions.

In the next two sections we explain the coarse- and fine-grained scheduling algorithms we use in this study.

6.3.1 Coarse scheduling algorithms

The coarse scheduler's job is to partition instructions into large groups and to assign each group to a domain. In this study we examine three approaches. We keep our descriptions of the algorithms to a high level, as the mechanics of applying them to any architecture are simple.

Coarse-By-Function. Coarse-By-Function allocates all instructions in a function to the same domain. For each function, it cycles through the domains, assigning the function to the first domain that has room for it. If no domain qualifies, it selects the domain which

currently contains the fewest instructions, in order to balance the instruction load across domains.

Coarse-By-Topology. Apart from function boundaries, Coarse-By-Function does not examine the topology of the application dataflow graph. Coarse-By-Topology inspects the topology in more detail, placing chains of dependent instructions in the same domain. In particular, Coarse-By-Topology performs a depth-first traversal of the dataflow graph, filling domains with instructions in the order in which it encounters them. Thus the algorithm tends to map long chains of producer-consumer instructions to the same domain, thereby localizing instruction communication within a domain.

Coarse-By-Exe-Order. Like Coarse-By-Function and Coarse-By-Topology, Coarse-By-Exe-Order fills domains with instructions in sequence, but it does so based on an actual execution profile. As program execution demands each instruction, Coarse-By-Exe-Order assigns it to the current domain. Once this domain is full, Coarse-By-Exe-Order moves on to the next domain. Implementing Coarse-By-Exe-Order requires that the instruction scheduler have access to an execution profile.

6.3.2 Fine scheduling algorithms

The job of a fine scheduling algorithm is to take the output of the coarse scheduling phase, i.e., a mapping of instructions to domains, and to assign each instruction to a specific PE in its domain. Each of the three fine scheduling algorithms described in this section processes domains one at a time.

Fine-BUG. Fine-BUG adapts the Bottom-Up-Greedy (BUG) algorithm, first used in the Bulldog VLIW compiler[46]. BUG was the first phase of a two-phase scheduling strategy. Processing instructions in a bottom-up, breadth-first order, it divided instructions into groups, such that one instruction from each group would form a very long instruction word. The second phase then produced the temporal schedule for each group. Originally, BUG assumed zero communication latency between instructions in different groups. A

second version of BUG that appeared in the Multiflow compiler [80] for clustered VLIWs, differentiates between local and remote operand latency. It still processes instructions in a bottom-up, breadth-first order, but attempts to place dependent instructions in the same group, in order to reduce operand latency. This second version of BUG is best-suited to WaveScalar, because it is aware of the non-uniform operand latency within a domain.

Our version of BUG, Fine-BUG, also schedules instructions with a bottom-up, breadth-first traversal of the dataflow graph.[1] However, to place an instruction, Fine-BUG first calculates, for each PE, the number of operands the instruction shares with any of its successors that have already been assigned to the PE. Fine-BUG assigns the instruction to the PE with the largest number of communicating operands. Ties between PEs are broken by round-robin priority.

Fine-UAS. Fine-BUG ignores the resource conflicts that arise when two instructions at the same PE can execute at the same time. When this occurs, one instruction must wait an extra cycle (or more) for the other to complete. Unified Assign and Schedule (UAS) [95] accounts for stalling due to execution conflicts, as well as operand latency. It unifies into a single heuristic both the spatial assignment of instructions to execution locations (to minimize operand latency) and the temporal scheduling of instructions at each location (to minimize execution resource conflicts).

Our implementation of UAS, Fine-UAS, uses a heuristic that reflects WaveScalar's domain resources and latencies. Fine-UAS processes instructions in a top-down, breadth-first order. For each instruction it greedily chooses the best PE, according to a heuristic that estimates when the instruction will execute, based on input operand communication latency and projected resource conflicts.

Because WaveScalar fetches instructions for execution dynamically, it is particularly difficult to determine when two instructions will conflict at execution. We use a simple, but

[1]Some implementations of BUG involve two passes through the dataflow graph. The first, bottom-up pass collects "candidate assignments" for each instruction, and then the second pass, top-down, makes the final assignment based on both the locations of the predecessors and candidate locations of the successors. We experimented with this version, but found that the single pass implementation described above was equally effective.

intuitive, heuristic: if instructions are at the same depth in the dataflow graph, we assume they will conflict, otherwise we assume they will execute at different times.

Fine-By-Exe-Order. The third fine scheduling algorithm we consider is a profile-driven scheduler. Like Coarse-By-Exe-Order, Fine-By-Exe-Order uses the dynamic execution order to choose PEs for instructions. As the program executes and instructions are loaded, Fine-By-Exe-Order begins with one PE, filling it to capacity (64 instructions). It then moves on to the other PE in the pod and then to another pod in the domain.

6.4 Experimental evaluation

To evaluate the three coarse and three fine scheduling algorithms from Sections 6.3.1 and 6.3.2, we produced schedules using each combination of coarse and fine algorithm. We scheduled nine sample applications from the Spec2000 [122] and SPLASH2 [145] benchmark suites (art, equake, gzip, mcf, radix, twolf and fft, lu, ocean, respectively).[2] The cycle-level simulator used for this study is tuned to match the latencies, resources, and restrictions of an RTL implementation [126] of the architecture.

Table 6.2 shows the average performance of each of these nine schedules. The IPC for each is normalized to the IPC of Coarse-By-Topology with Fine-BUG. In this section we discuss only the results in the top part of Table 6.2. We will describe the experiments that produced the last two rows of data in Section 6.5.

6.4.1 Coarse scheduler evaluation

To evaluate the quality of the coarse scheduling algorithms, we compare the normalized IPCs in each of the first four rows in Table 6.2. Each row enables us to evaluate: for a given fine scheduling algorithm, how do the coarse schedulers compare? The data show that Coarse-By-Exe-Order has the best overall performance with two of the three fine schedulers.

[2]We use these particular applications because both our binary translator-based compiler and WaveScalar simulator can compile and simulate them.

Table 6.2: Application performance by instruction scheduler on WaveScalar.

	Coarse-By-Function	Coarse-By-Topology	Coarse-By-Exe-Order	Average
Fine-BUG	100%	110%	123%	111%
Fine-UAS	79%	94%	110%	94%
Fine-By-Exe-Order	61%	77%	66%	68%
Average	80%	94%	100%	
Fine-DAWG-Max	108%	120%	141%	123%
Fine-DAWG-Same	96%	112%	130%	113%

Two factors account for the better performance. First, Coarse-By-Exe-Order uses execution-order information to pack the subset of static instructions that are actually executed more compactly. (Rarely used paths are placed elsewhere in the processor.) This reduces average operand latency.

Second, Coarse-By-Exe-Order-generated schedules incur cheaper inter-domain operand traffic. The coarse schedulers determine what share of operand traffic crosses domain boundaries. As Figure 6.3 shows, Coarse-By-Exe-Order and Coarse-By-Function each incur approximately the same amount of this traffic (6% and 7%, respectively). However, for Coarse-By-Exe-Order the majority of the inter-domain traffic (85%) is local to the cluster. In contrast, only 14% of Coarse-By-Function's inter-domain traffic remains in the cluster, with the rest traversing the more costly inter-cluster routing network. Thus, while each of these two coarse algorithms produces the same proportion of inter-domain traffic, this traffic generally travels farther, with increased latency, using Coarse-By-Function.

The one case in which Coarse-By-Exe-Order is not the best strategy is when combined with Fine-By-Exe-Order. In this situation Coarse-By-Topology produces bet-

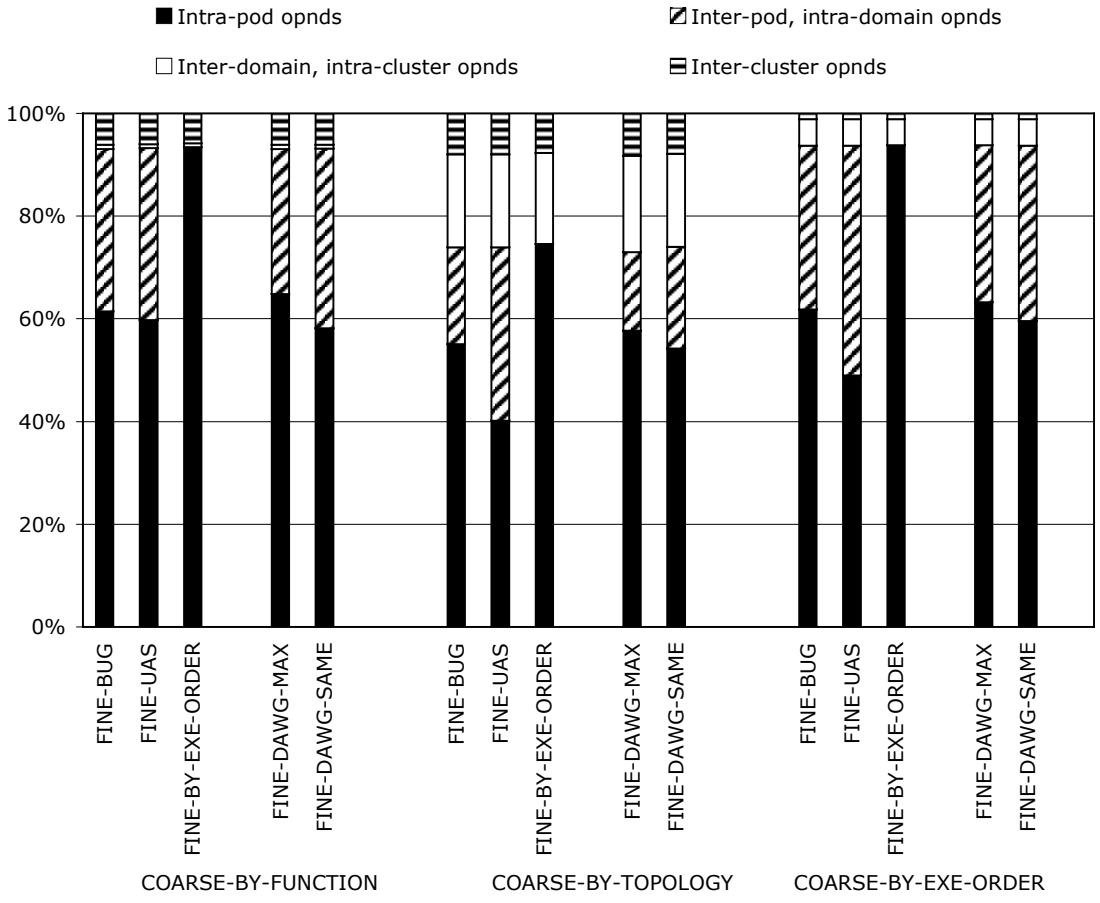

Figure 6.3: WaveScalar operand traffic breakdown by instruction scheduler.

ter performance. Judging only by the data in Figure 6.3, this is somewhat surprising, because Coarse-By-Topology incurs the *smallest* amount of intra-domain traffic of the three coarse schedulers. However, when paired with a terrible fine scheduling algorithm (which, as we'll see in the following Section Fine-By-Exe-Order turns out to be), the best coarse strategy is to keep operand traffic *outside* of the domain and out of the hands of the fine scheduler.

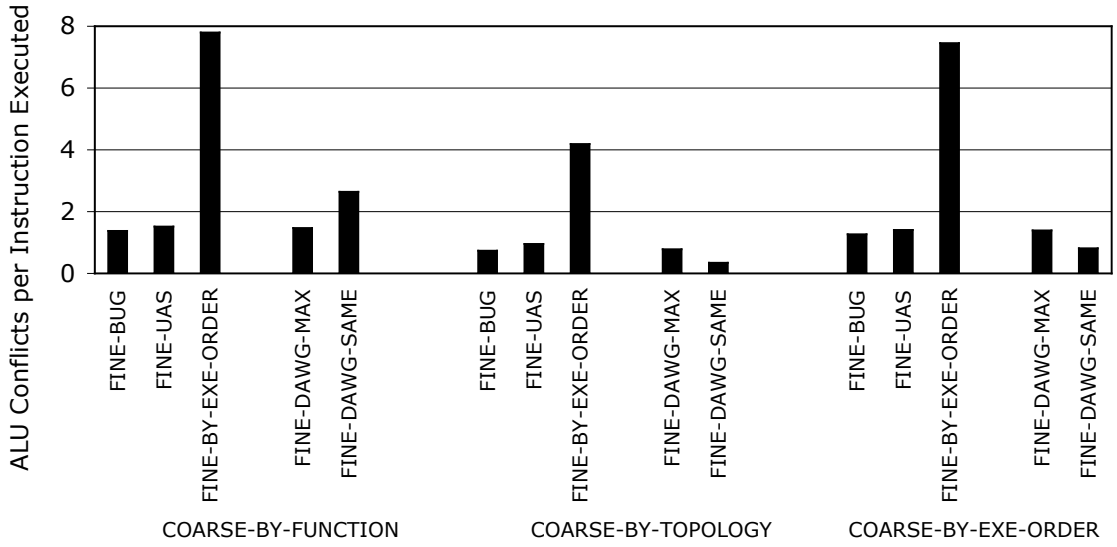

Figure 6.4: WaveScalar ALU conflicts by instruction scheduler.

6.4.2 Fine scheduler evaluation

To compare the fine schedulers, we examine the values in each column of Table 6.2. No matter the coarse scheduler, Fine-BUG readily outperforms both Fine-By-Exe-Order and Fine-UAS. We first use simulator-generated data to examine the reasons why Fine-By-Exe-Order and Fine-UAS fall short, and then analyze Fine-BUG's success.

Based only on the distribution of intra-domain traffic shown in Figure 6.3, Fine-By-Exe-Order's poor performance is somewhat unexpected. Of the three fine scheduling algorithms, it confines almost all of its intra-domain operand traffic to the cheaper intra-pod communication. The preponderance of very localized traffic resulted in average operand latencies that were 68% lower than that of the other fine schedulers. Operand traffic is not the whole story, however, as execution resource conflicts also contribute to performance. Figure 6.4 shows that Fine-By-Exe-Order incurs on average 5.3 times the number of ALU conflicts/instruction as the other fine schedulers. Thus, despite near-perfect communication locality, its performance is hampered by its aggressive PE-packing approach to fine scheduling.

As an aside, this validates the intuition described in Section 6.3 that coarse and fine placement are qualitatively two different problems. The scheduling strategy that produced the best performance at the coarse level, Coarse-By-Exe-Order, resulted in the worst performance at the fine level, Fine-By-Exe-Order. Coarse schedulers can focus single-mindedly on reducing operand latency and capturing instruction set working locality. Fine schedulers, however, must carefully balance latency against resource contention.

Fine-UAS was designed to optimize the operand latency-resource contention trade-off for processors that schedule instructions statically (e.g., VLIW processors). To incorporate the notion of out-of-order execution, we augmented it with a simple heuristic to estimate execution resource conflicts that occur from dynamic dispatch. (Recall that according to this heuristic, two instructions will conflict if they are at the same depth in the dataflow graph). The data in Table 6.2 indicates that Fine-UAS performs more poorly than Fine-BUG which does not explicitly consider resource conflicts. Fine-UAS attempts to predict and avoid execution conflicts. However, the prediction mechanism is imperfect. Not only did it fail to reduce execution conflicts (Figure 6.4), the conflict avoidance measures it took – splitting those instructions across pods – served only to increase operand latency (Figure 6.3).

The failure of the conflict prediction heuristic can be traced to two factors, both of which relate to the topology of the dataflow graph. First, it is far more likely that two instructions that are "nearby" in the dataflow graph will conflict. This is because they are more likely to execute around the same time, even if they lie at different depths. Conversely, two instructions in entirely separate regions of the application, but at the same depth of the graph, are more likely to belong to different temporal phases of execution and therefore will likely not conflict. Second, not all anticipated resource conflicts should carry equal weight. When a dataflow graph is about to fan out, increasing parallelism, it is best to exploit that parallelism by dividing the instructions between multiple PEs; thus, these instructions should be weighted toward separate PEs. However, if potentially conflicting instructions lie on paths which are about to merge together, the scheduler should weigh those conflicts as less likely to slow down execution.

Although more sophisticated heuristics, such as those based on the discussion above, may

improve the quality of FINE-UAS schedules, we opted to pursue a more general strategy. Instead, we have developed an algorithm that allows us to explore a range of schedules in the latency-contention trade-off. We will explain this algorithm in Section 6.5.

FINE-BUG does not use an explicit estimate of operand latency or resource capacity. Nevertheless, its simple approach to balancing these two factors appears successful. The algorithm generally tries to disperse instructions across parallel execution units, except when overridden by operand locality concerns. The resulting middle ground between latency and conflicts is reflected by the data in Figures 6.3 and 6.4. Coupled with FINE-BUG's strong IPC results, this indicates that some middle point, trading off communication locality and execution conflicts, is a good design target. This observation motivates the development of FINE-DAWG.

6.5 Fine-DAWG *scheduler*

In this section we describe FINE-DAWG, a new scheduling algorithm for placement of instructions within domains. It is designed both to explore the latency-contention design space and to generate quality code for dynamically scheduled processors. FINE-DAWG operates in two phases, first bundling instructions into groups, and then assigning each of these groups to a PE. Before delving into the mechanics of the algorithm, we first describe these two phases informally.

Phase one forms groups based on the topology of the dataflow graph. It takes two parameters, $MaxDepth$ and $MaxWidth$, and carves the dataflow graph into groups of instructions which have dependence chains at most $MaxDepth$ instructions long, with each instruction having at most $MaxWidth$ potentially parallel successors. The $MaxDepth$ parameter controls how aggressively FINE-DAWG confines operand communication to within a single PE: a high value reduces operand latency, as more dependent instructions are placed at the same PE; alternatively, a low setting increases operand latency by assigning instructions to different PEs. $MaxWidth$ limits the amount of parallelism within a single PE: a high value increases ALU contention, as more parallel instructions are scheduled to the same PE, and a low value has the opposite effect. While all of these values can be selected independently, wise choices will take care to observe the capacity at each PE (64 instructions). Settings

where $MaxWidth \times MaxDepth$ vastly exceed this number will likely cause the WaveScalar processor to thrash.

After phase one has assigned all of the instructions in the dataflow graph to groups, phase two maps these groups to PEs using a single parameter, *DepDegree*. *DepDegree* is a value between zero and one, and determines how aggressively inter-group operand dependencies are used to choose PE locations for groups. A value close to zero raises the probability of dependent groups being placed on the same PE; a value close to one separates them.

We will make this brief description of FINE-DAWG more precise by walking through the algorithm's pseudocode.

6.5.1 FINE-DAWG *phase one*

Algorithm 1 FINE-DAWG: $CreateGroup(i)$ (a portion of phase one)

```
1: GRP = i
2: LEVEL = successors(i)
3: for depth = 1 to MaxDepth do
4:    SELECTED = MaxWidth nodes from LEVEL
5:    GRP = GRP ⋃ SELECTED
6:    GROUPED = GROUPED ⋃ SELECTED
7:    LEVEL = successors(SELECTED)
8: end for
9: return GRP
```

Phase one gathers instructions into groups based on the topology of the dataflow graph. The algorithm maintains a list of nodes that have no predecessors. To create a group (GRP), FINE-DAWG first calls $CreateGroup(i)$ on an instruction i from this list. Then, after the group is created, it removes the instructions in the group from the graph (adding them to $GROUPED$) and updates the list of predecessor-free nodes. The process repeats until all instructions belong to a group.

Algorithm 1 details how to form a group, beginning from some instruction i. $CreateGroup(i)$ traverses levels of the graph iteratively from instruction i, up to $MaxDepth$ levels deep (line 3). At most $MaxWidth$ instructions from each level are added to the group (lines 4 and 5).

6.5.2 FINE-DAWG *phase two*

Algorithm 2 FINE-DAWG: phase two (applied to the output of phase one, $GROUPS$)

```
1: for all GRP ∈ GROUPS do
2:    r = rand()
3:    if r < DepDegree then
4:       assign GRP to the PE with which it has the most communication
5:    else
6:       assign GRP to the PE with which it has the least communication
7:    end if
8: end for
```

Phase two accepts a single tuning parameter, $DepDegree$, and assigns each of the groups produced by phase one ($GROUPS$) to a PE. Initially, all of the PEs contain no instructions. For each group, GRP, FINE-DAWG identifies the two PEs, $pe_{\max}$ and $pe_{\min}$, with whose instructions GRP has most and fewest dependences, respectively. With probability $DepDegree$, it will assign GRP to $pe_{\max}$; otherwise it assigns it to $pe_{\min}$.

6.6 Fine-DAWG *evaluation*

We explore the trade-off between execution resource conflicts and operand latency by exploring the parameter space of FINE-DAWG. Beginning with the domain assignments produced by COARSE-BY-FUNCTION, COARSE-BY-TOPOLOGY, and COARSE-BY-EXE-ORDER, we produced PE assignments using FINE-DAWG, parametrized by each combination of $MaxDepth \in \{2, 4, 8, 12, 16, 32, 50, 64, 128\}$, $MaxWidth \in \{1, 2, 3, 4, 6, 10\}$, and $DepDegree \in \{.1, .5, .9\}$.

We begin our analysis of FINE-DAWG by observing how much of the latency-contention design space it explores. Using COARSE-BY-EXE-ORDER as the coarse scheduler, along with FINE-DAWG with all 162 parameter settings, we scheduled all of the applications in our workload. Figure 6.5 depicts the average latency-contention for the benchmarks for each parameter configuration. The X-axis measures the percentage of inter-pod, intra-domain operand traffic within total traffic; the Y-axis measures ALU conflicts per executed instruction. Schedules that land down and to the left in this plot are desirable. The graph

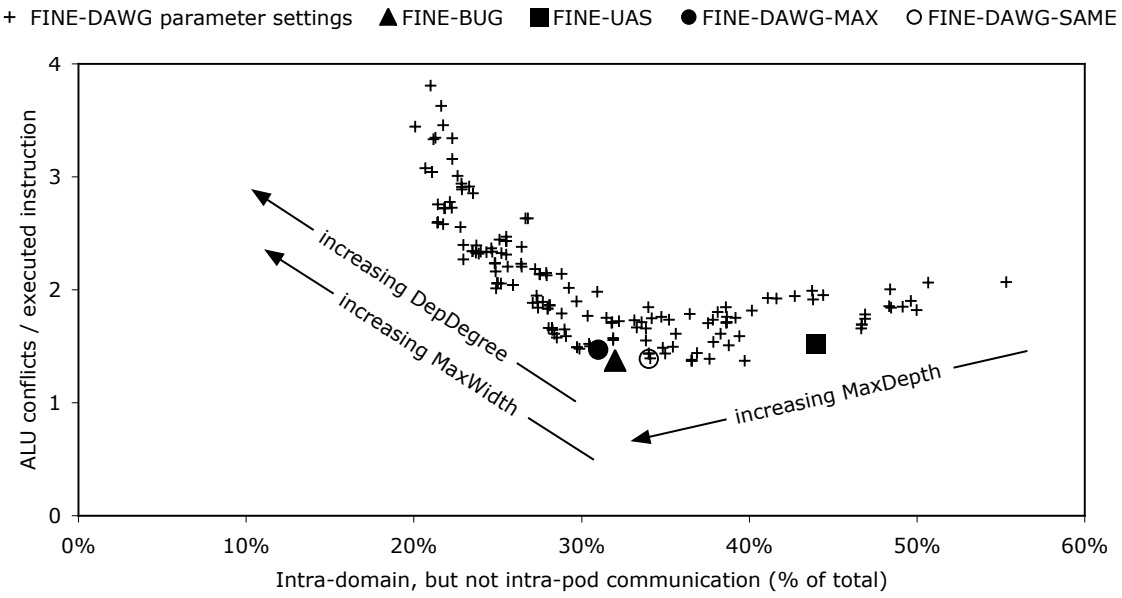

Figure 6.5: Communication latency and resource conflict trade-off achieved by Fine-DAWG.

also shows three additional labeled axes. These depict how changes in Fine-DAWG-Max's input parameters effect the resulting schedule output. Finally, we have added points for Fine-BUG, Fine-UAS, Fine-DAWG-Same, and Fine-DAWG-Max (the latter two are explained below). (Fine-By-Exe-Order is an outlier, with low latency and extremely high conflicts, and is not shown on the graph.)

The data provide several insights about Fine-DAWG. First, it indicates that varying the parameters of Fine-DAWG successfully manipulates both operand latency and ALU conflicts, providing a complete exploration of that trade-off space. Second, it indicates the intra-domain, inter-pod operand traffic threshold below which ALU conflicts rise dramatically falls at 32%. This gives code schedulers an indication of the end point for PE instruction occupancy. Note that Fine-UAS lies at a non-Pareto optimal point; as discussed in Section 6.4.2, there are schedules, such as those produced by Fine-BUG, which have both lower latency and lower ALU contention. One would not expect schedules produced

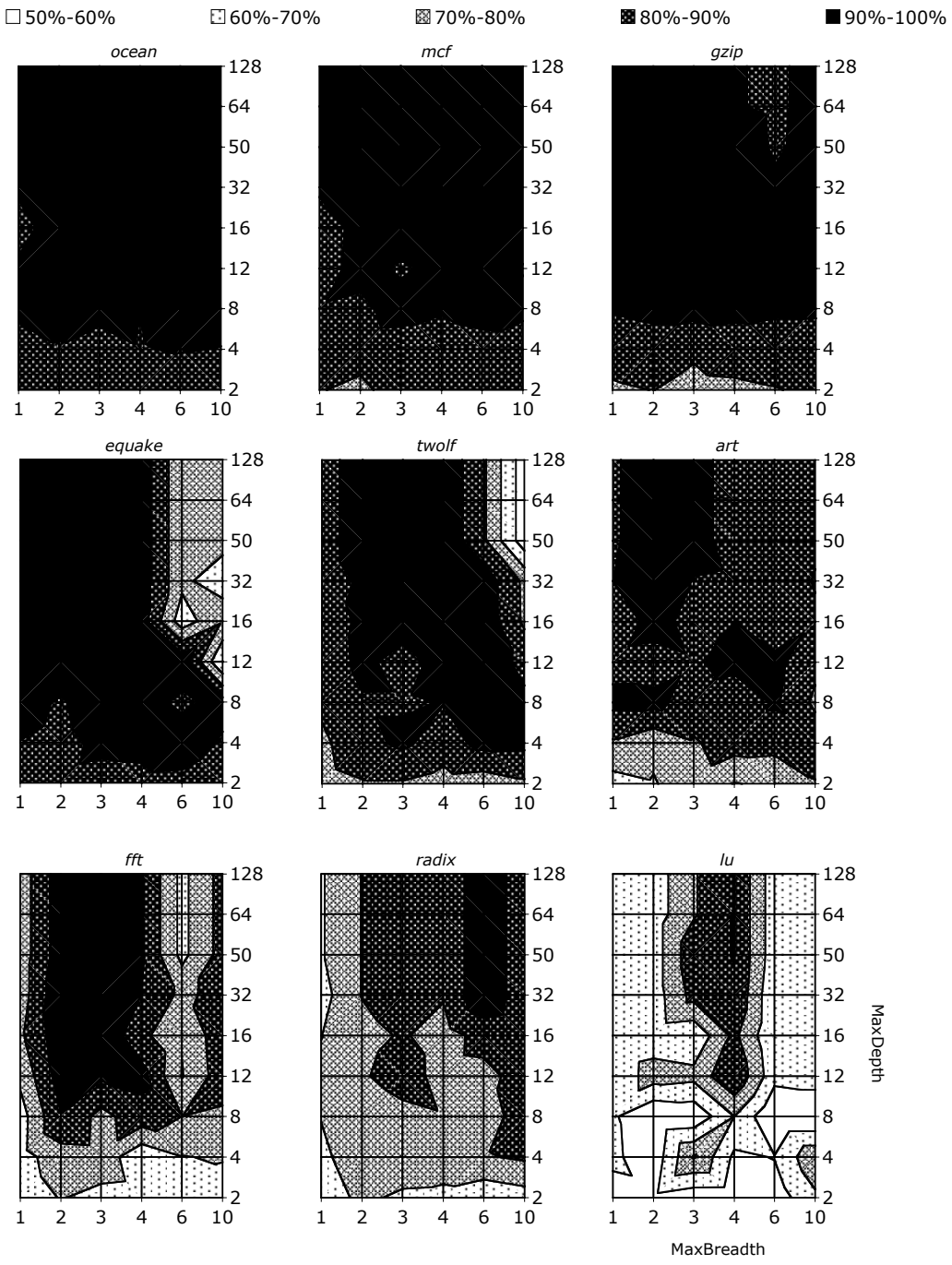

Figure 6.6: Application performance sensitivity to FINE-DAWG parameters.

by FINE-UAS to perform well on WaveScalar, as indeed they do not.

The charts in Figure 6.6 show how the performance of FINE-DAWG varies from application to application. The X and Y axes in the chart show the parameter space $MaxWidth$ and $MaxDepth$, respectively, for $DepDegree = 0.1$. The shading gradient indicates the performance of each parameter setting relative to the performance for the best parameter setting for that application. The darker the shading, the stronger the performance.

The applications interact with FINE-DAWG's parameters in a variety of ways. In the top row are ocean, mcf, and gzip, for which the vast majority of the parameter settings fall in the 90-100% gradient. This consistency is arises when forming deeper and deeper sub-graphs increasingly failed to yield any more improvement in performance. By contrast the applications in the bottom row, fft, radix, and lu, are all extremely sensitive to $MaxWidth$ and $MaxDepth$, and each prefers a slightly different parameter range. This is because kernels drive the performance of these three benchmarks. When the topology of FINE-DAWG's sub-graphs fits the kernel well, performance is strong; but if the fit is not good, it quickly drops off. This implies that it may be fruitful to identify and use hot spots to set FINE-DAWG's parameters.

In practice, it is not practical to explore the entire parameter space as we have. As an alternative, we have identified the single parameter set which produced the best overall performance. The analysis in Figure 6.7 is the basis for this selection. The plot depicts the cumulative distribution functions of the different parameter settings applied to our set of applications. Each line in the graph corresponds to one FINE-DAWG parameter assignment. The x-axis is the performance loss relative to the best performing schedule (expressed as a percentage decrease in maximal IPC). The y-axis indicates the number of applications for which the given parameter setting will produce performance at least as high as that specified by the x value. For a particular x-value, the y-value indicates the number of applications whose performance is at the x-value level or faster. The y-value can be interpreted as the likelihood that an application will perform within x% of maximal performance using a particular parameter assignment.

In order to display the information in Figure 6.7 more clearly, we have shown only five of the total 162 different parameter combinations used in this study, the two best settings

and three others that are representative of the range of results. The data shows that $(MaxDepth, MaxWidth, DepDegree) = (50, 3, .1)$ consistently produced better schedules than all other settings, coming within 17% of maximal for all applications.

Returning to Table 6.2, we see that the combination of Fine-DAWG with Coarse-By-Exe-Order produces the best performing schedules. We show two data points for Fine-DAWG: Fine-DAWG-Same and Fine-DAWG-Max. Fine-DAWG-Same is the performance when using the strongest consistent parameter set, $(50, 3, .1)$, across all applications. This exceeds the previous best combination of Coarse-By-Exe-Order and Fine-BUG by 14%. Fine-DAWG-Max is the performance when using the best parameters for each application, which exceeds Coarse-By-Exe-Order and Fine-BUG by 28%.

While at a coarse level, scheduling entirely to minimize operand communication latency works well, the strategy fails at the fine level. At this level, a scheduler must carefully balance operand latency with execution resource conflicts. The algorithm we have developed, Fine-DAWG, can be used to efficiently identify and produce schedules at the an optimal latency-conflict trade-off point.

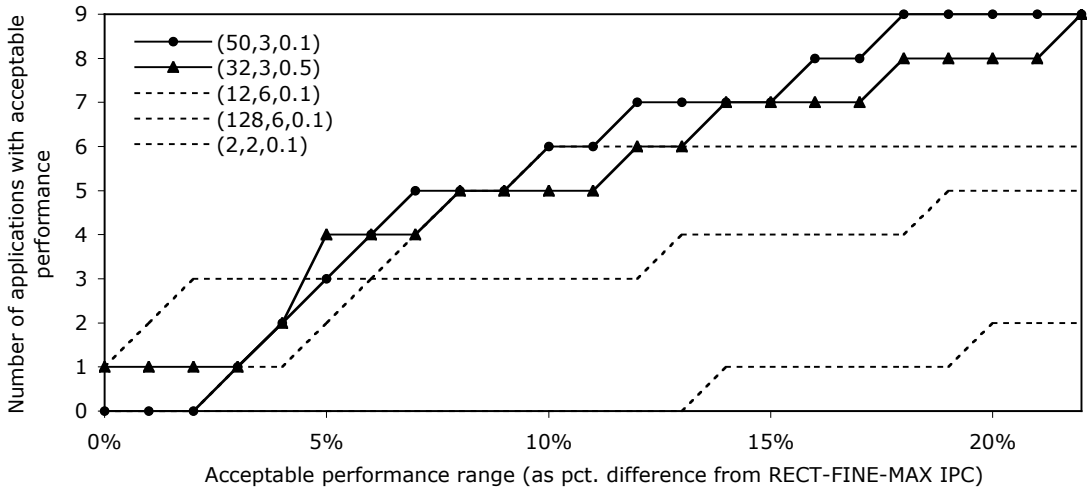

Figure 6.7: Cumulative distribution function of selected Fine-DAWG parameters: the best two parameter settings (solid lines) and three representing the range of 160 others (dashed lines).

Chapter 7

EXPERIMENTAL EVALUATION OF MADE-TO-ORDER CHIP MULTIPROCESSORS

In this chapter we examine and quantify the costs and benefits of the brick and mortar assembly process with a top to bottom case study. The subject of our study is a chip multiprocessor.

7.1 Methodology

We rely on several analysis tools to understand the characteristics of a brick and mortar implementation of a CMP.

7.1.1 CMP simulator

To time the execution of applications on the CMP we used the Virtutech Simics [82] simulation framework and GEMS [85] tool set. We configured the simulator to match the CMP design configurations outlined in Table 7.1. We ran the SPLASH2 [145] suite of multi-threaded benchmarks. We also used Simics to simulate the increased communication latency across brick boundaries to estimate an application's runtime on a brick and mortar chip.

7.1.2 I/O cap model

We estimate the latency of the polymorphic network in order to provide appropriate CMP interconnect delays to Simics. Because the interconnect in a CMP is a delay-sensitive component, we model a latency-minimizing network configuration of the CMP. According to our results in Section 3.1.2, fat trees most effectively minimize packet latency.

Synopsys DesignCompiler Ultra [129] provided delay information, while bandwidth data came from the design itself (e.g., the packet-switched network supports 64-bit data words)

Table 7.1: Comparison of brick and mortar and monolithic implementations of three CMP designs.

Chip multiprocessor designs						
	CMP-L		CMP-M		CMP-S	
Total Area mm^2	193.5		177.5		200.5	
Chip composition						
	Count	% Area	Count	% Area	Count	% Area
Small bricks ($0.5 \times 0.5\ mm^2$*)*						
RISC Core (no FPU,8K cache)	-	N/A	-	N/A	16	1.99%
FPU	-	N/A	-	N/A	16	1.99%
Tri-mode ethernet	1	0.13%	1	0.14%	1	0.12%
Memory controller	1	0.13%	1	0.14%	1	0.12%
USB 1.1 physical layer	1	0.13%	1	0.14%	1	0.12%
WB DMA	1	0.13%	1	0.14%	1	0.12%
PCI bridge	1	0.13%	1	0.14%	1	0.12%
VGA/LCD controller	1	0.13%	1	0.14%	1	0.12%
Medium bricks ($1.0 \times 1.0\ mm^2$*)*						
RISC core (64K cache)	-	N/A	16	9.01%	-	N/A
Large bricks ($2.0 \times 2.0\ mm^2$*)*						
RISC core (256K cache)	16	33.07%	-	N/A	-	N/A
256K SRAM	32	66.15%	40	90.14%	48	95.29%
Simics/GEMS performance simulation						
	brick & mortar	*ASIC*	*brick & mortar*	*ASIC*	*brick & mortar*	*ASIC*
Number of cores	16	16	16	16	16	16
L1 cache / core (KB)	256	256	128	128	8	8
L2 cache size (MB)	8	8	10	10	12	12
L2 associativity	4	4	5	5	6	6
L2 block size (B)	64	64	64	64	64	64
L2 set size (KB)	32	32	32	32	32	32
Processor cycles to L2	31	22	41	22	50	22
Exe. time (avg.)	108%	100%	120%	100%	136%	100%

and the physical pin constraints [47]. Each "hop" in this network requires one 800MHz clock cycle. Each level up or down the fat tree requires one hop, and moving laterally in the grid at the root can require 1-4 hops, for which we have conservatively accounted as always requiring 4 hops.

7.1.3 Process variation model

We model the effect of process variation on brick performance using the FMAX model [20]. This model assumes that as die size increases, the number of critical paths increases. This leads to an increase in the mean delay and a decrease in the standard deviation of that delay. We use this delay variation model to explore how individual brick speeds effect overall chip speed in the brick and mortar process.

7.2 Performance results

Large customers of computing systems recognize that their application requirements do not always match the one-size-fits all processors available today. For example, network servers need throughput on task-based parallelism, while scientific computing requires extensive floating point capabilities. Thanks to the low overhead associated with starting and producing a run of chips with brick and mortar assembly, it is possible to produce made-to-order CMPs.

We examine three different 16-way CMPs, built from the bricks in Chapter 4, all of which fit within $200mm^2$. Each design consists of 16 combined processor and L1 cache bricks. These bricks differ in size (small, medium, and large), cache capacity (6K, 64K, and 256K), and whether or not the FPU is on the same brick (no, yes, and yes). After some general-purpose I/O bricks, the L2 cache fills out the remainder of the $200mm^2$ area budget. Table 7.1 summarizes the designs.

Across the three designs we study, three significant factors change. First, the L1 caches are constrained by the choice of brick size to implement the processor. Second the network delay between the L1 and L2 depends on brick layout. Third, the CMP built from smallest RISC core brick (CMP-S) requires an additional FPU brick per processor. The configurable mesh interconnect connects the processor and FPU bricks.

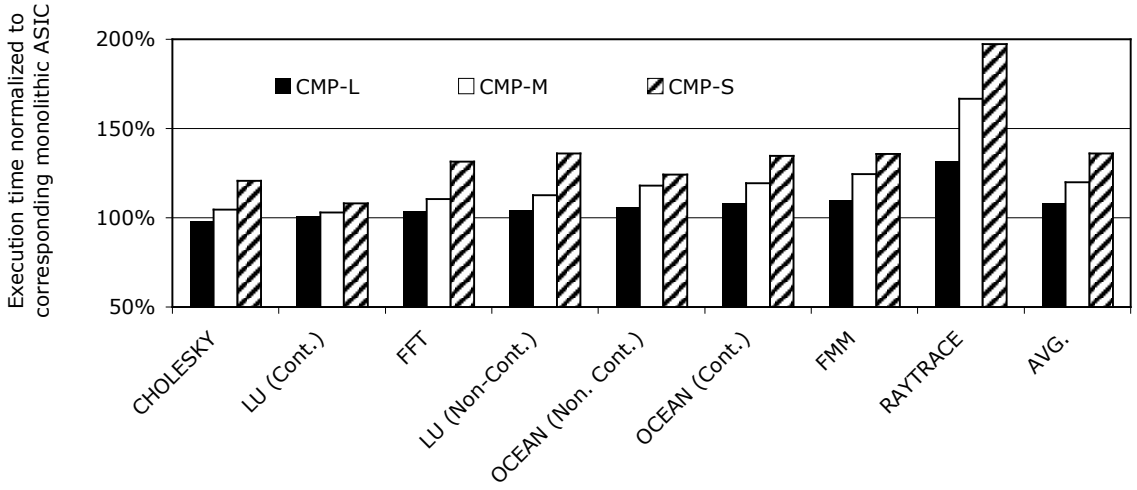

Figure 7.1: Application runtime on brick and mortar CMP compared to a monolithic ASIC CMP.

For each CMP, we used the I/O cap network model to calculate the additional latency between bricks introduced by the I/O cap. We convert network cycles (one per hop from the processor to each cache brick and back) into processor cycles based on the the relative network and processor speeds. The L2 cache will have non-uniform access times depending on how close the processor brick happens to be to the brick containing the cache line. However, because we are trying to bound the performance degradation due to using the I/O cap instead of an on-chip interconnect, we have conservatively assumed the worst case access time for all accesses. We introduce delay cycles into the simulation accordingly to account for this.

For each brick and mortar design we modeled a corresponding, equally-provisioned ASIC version of the CMP for comparison. Between the ASIC and brick and mortar designs the primary difference is in the component latencies. The brick and mortar inter-brick latencies come from the I/O cap network model, while the ASIC latencies come from the published latencies of the UltraSparc T1 [125] on-chip interconnect. The "Simics/GEMS Performance Simulation" section of Table 7.1 gives the specifics of each pairwise comparison.

Figure 7.1 shows the performance results from the simulations. The performance of each

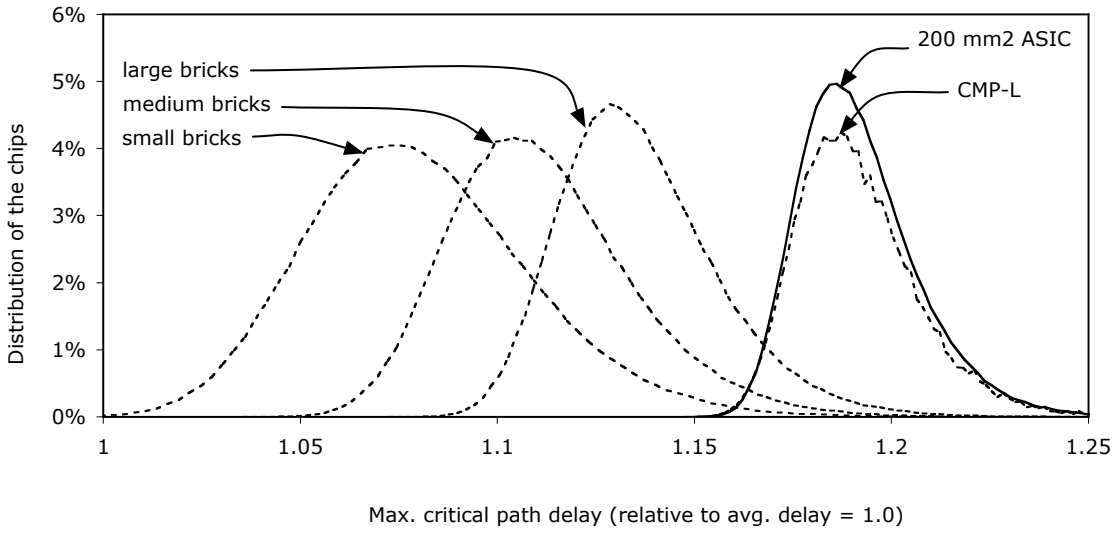

Figure 7.2: CMP and component clock speed variation derived from FMAX models.

brick and mortar chip was normalized to the performance of the corresponding ASIC design. On average, the benchmarks took 36%, 20%, and 8% more time to complete on the three brick and mortar CMPs than on the corresponding ASICs. This was due to the increased interconnect latency that the I/O cap introduces. The principle difference between the three CMP designs was the size of the L1 cache in the processor brick. Naturally, the smallest (8KB cache in CMP-S) incurred the most L1 misses while the largest (256KB cache in CMP-L) incurred the fewest. Each L1 miss sends a request to the L2, which on the brick and mortar designs required communication via the I/O cap. Thus, the L1 miss rate in the brick and mortar CMP designs largely determined performance.

7.3 Exploiting process variation

To understand how process variation effects brick and mortar chip performance we used the variation model to characterize the critical path delay for each brick size. The results are depicted by the three leftmost dashed curves in Figure 7.2. The data are scaled to an average critical path delay of 1.0 and show the distribution of maximum critical path

length for each component. We also used this model to calculate the expected distribution of maximum critical path delay in a $200mm^2$ ASIC chip. Because a smaller chip (such as a brick) has fewer critical paths, the longest critical path is likely to be shorter than the longest critical path on a larger chip (such as the $200mm^2$ ASIC). However, when one assembles $200mm^2$ worth of small bricks (as in CMP-L) it results in the same distribution of worst case critical paths as a monolithic $200mm^2$ circuit. For clarity, we have plotted variation date for CMP-L only, but the principle applies to CMP-M and CMP-S as well, with even more pronounced effects.

We experimented with speed-binning the bricks prior to assembly, into 1 (no binning), 2, 4, and 8 speed grades. The resulting variation in brick and mortar chips is plotted in the solid curves in Figure 7.3. Note that with only two speed bins, the number of high performance chips is significantly improved (the spike on the left). The reason this happens is that the speed of the chip is set by the brick with the highest delay. By speed grading bricks prior to assembly, we can focus on building high-performance chips from high-performance bricks. Without speed grading, since a chip contains tens of bricks (54 total for CMP-L), a low performing brick often slips onto the design and constrains the entire chip's performance. Speed grading in this way brings further improvements in high-performance chip production as the number of grades is increased.

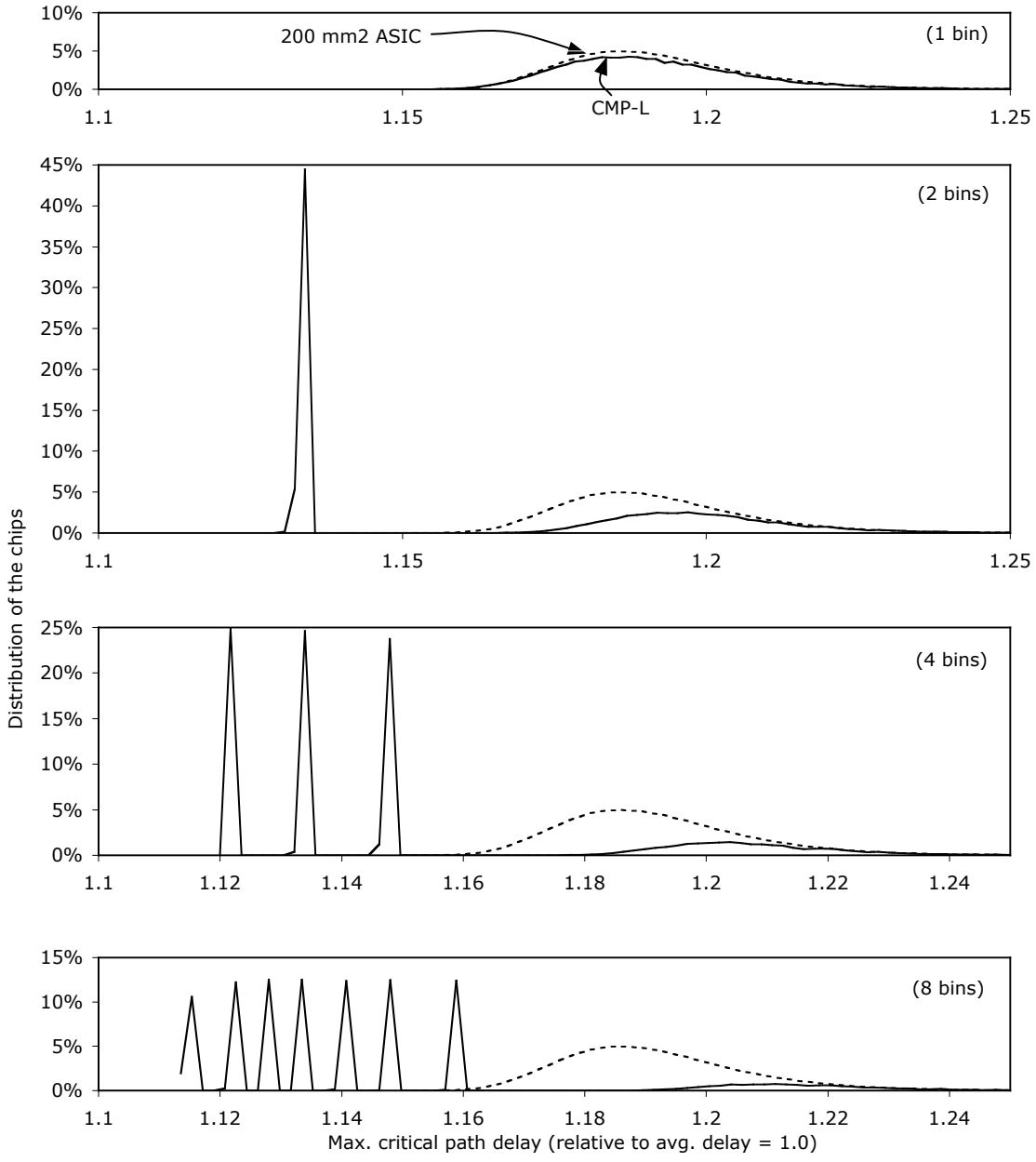

Figure 7.3: Effects on chip speed of pre-assembly component speed binning.

Chapter 8

RELATED TECHNOLOGIES

This chapter looks at custom chip implementation technologies and how they relate to brick and mortar. We also survey technologies related to each of the brick and mortar subsystems discussed in Chapters 3 through 6. Because those subsystems touch on disparate subjects, from on-chip networks to code scheduling, so too do the contents of this chapter.

8.1 Custom circuit implementation

To implement a design, engineers typically choose between two options. Either they must face the high initial cost of ASIC production, and hope to amortize it over a large batch of units, or they use an FPGA with low fixed costs but high unit cost. The trade-offs are not just financial. ASICs provide significant speed (3-4x) and power (up to 12x) savings [73], compared to FPGAs. The requirements of certain applications in these areas (e.g., cell phones) will demand an ASIC. However, FPGAs offer in-field reconfigurability, which is useful for bug fixes and other updates. On an FPGA, these updates require just 1-4 weeks compared to 2-5 months for ASICs [155]. This gap drives the need for a manufacturing technology that provides the key advantages of FPGAs – low, non-recurring costs and quick turnaround on designs – coupled with the key advantages of ASICs – low unit cost, high performance and low power.

Before we get into specific technologies, we outline the factors a designer typically weighs when comparing different implementation options. We will then classify and describe the implementation options spanning the range from ASICs and FPGAs.

8.1.1 Implementation considerations

Choosing a platform on which to implement a hardware design is a multifaceted decision. Each designer's choice is unique and often driven by multiple factors. For example, each

designer has his or her own unique hardware design, unique economic constraints, uniquely academic or commercial goals. Thus, there is no universal method for judging the factors outlined below.

Production cost. Many elements influence the ultimate price of a chip, from the size of the design to the size of the silicon wafers. Generally speaking, all of the costs fall into two categories: *non-recurring engineering* costs and *unit* costs. Non-recurring engineering costs, or NREs, are the one time costs which one must pay, no matter how many chips are produced. Unit cost, on the other hand, is the cost per chip, and thereby depends on the number of chips to be manufactured. The determinants of these two cost components were discussed and modeled in depth in Chapter 2. FPGAs avoid many of the non-recurring costs of ASICs and are thus significantly less expensive in this respect. While it depends upon the design, an FPGA-based design's NRE is typically 10-25% of that of the corresponding ASIC [113].

Performance. Depending on the application in question, the goal will be either to maximize circuit performance (e.g., a microprocessor design) or to simply meet a performance target (e.g., a network processor design). One must eliminate any technology that does not meet required performance targets. The choice between the remaining technologies will come down to a balance of the other considerations discussed in this section. A good rule of thumb, with respect to performance is that going from an ASIC implementation to an FPGA implementation incurs an order of magnitude slowdown. To put this in perspective, going from an FPGA to a software implementation incurs another two orders of magnitude slowdown [68]. Studies of the performance gap between ASICs and FPGAs indicate that the critical path in an FPGA is three to four times longer on average than in an ASIC [73].

Power. Generally speaking, the smaller the die, the lower the power consumption. This is due to static energy leakage, the power lost from powered-but-otherwise-inactive transistors. Studies have shown that the energy per gate in an FPGA design is 7-10x the energy per gate in the equivalent ASIC, largely as a result of the increase in area [104].

One technique to minimize power consumption is to operate the circuit at the lowest possible voltage and frequency. This is accomplished by exploiting parallelism in the application. At some point, however, the power savings is outweighed by the power spent on the extra parallel hardware [22].

Design flexibility. An ASIC design represents a much larger investment of time and resources. Once fabricated, one usually cannot apply any design fixes or modifications to an ASIC. If one knows that updates will be needed, for example to comply with an evolving communication standard, this particular design will have a limited lifetime. Therefore, there is some limited potential sales volume before the product becomes obsolete. In such circumstances, one must carefully consider whether an investment in an ASIC makes strategic sense. By contrast, an FPGA is designed for functional updates. Updating the "hardware" in an FPGA is a comparatively simple and inexpensive matter of reconfiguring the device.

8.1.2 Implementation technology classification

We propose the following classification of the hardware implementation technologies proposed by industry and academia. This classification system divides circuit implementation technologies into five classes based on how the custom circuit is implemented: in custom silicon, in custom metal, or in a custom hardware configuration. Figure 8.1 illustrates the five classes. The solid arrows indicate, for each of the five classes, which layers of the stack are customized while the dashed lines indicate those which are standard. While members of all classes require both silicon and metal layers (either standard or custom), this is not the case for a hardware configuration. The presence of this configuration is what gives three classes the name "configurable". The "ASIC" classes all require fabrication of custom silicon or metal layers. Finally, the "structured" classes are those for which some hardware (silicon) is standard while the rest (metal) is custom.

What follows is a description and discussion of members of the five classes defined above.

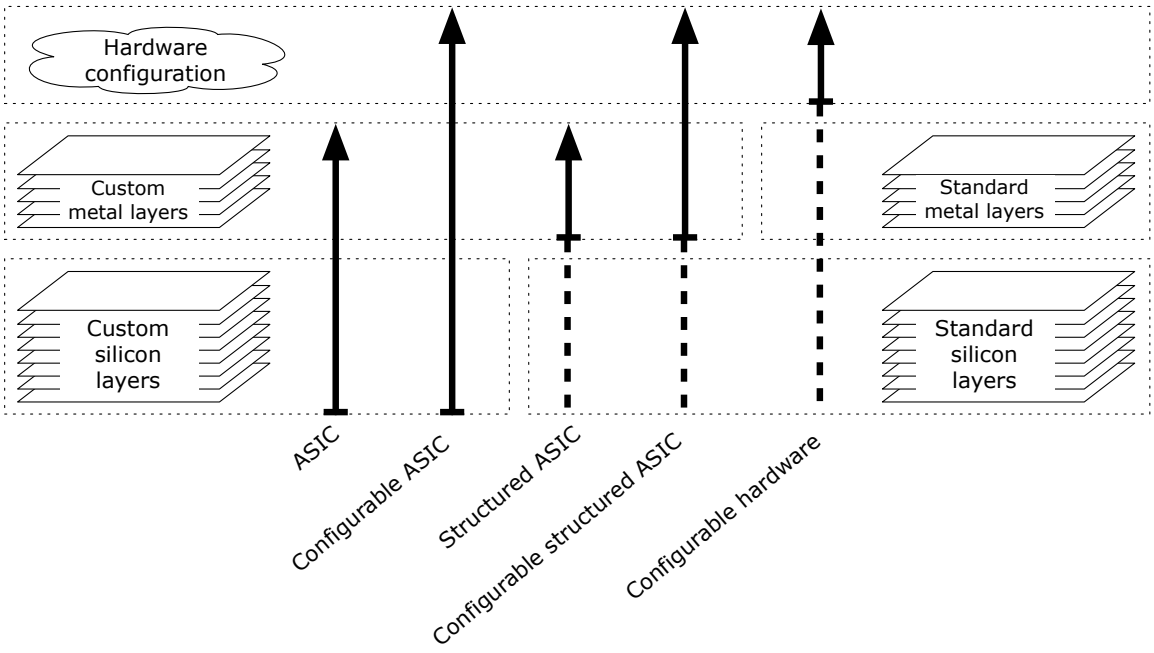

Figure 8.1: Classification of circuit implementation technologies.

8.1.3 Application-specific integrated circuits

Application-specific integrated circuits, or ASICs, come in two flavors: full-custom and standard cell.

Full-custom ASIC. Full-custom ASICs represent the gold standard of integrated circuits: shortest critical path delay, smallest circuit area, and lowest power consumption. They are produced at a great cost, however, as each feature and wire is constructed and laid out manually by a designer. This increases cost both directly, through the increased engineering effort, and indirectly, through a product's time to market. Once fabricated, a full-custom ASIC is not configurable or updatable.

Standard cell ASIC. While still completely specialized and application-specific, standard cell ASICs are constructed, not manually, but out of components found in a *standard*

cell library produced by a fab. The cells in such a library consist of transistors and gates. All cells in a library have the same technology node but vary with respect to their electrical and delay characteristics. This library is used analogously to the machine-dependent back end of a software compiler. There, late in compilation, the compiler identifies the specific machine instructions which will execute the until-then generically described program. At the end of hardware compilation, the tools translate a generic circuit description to the language of a particular fab's gates and wires.

Because of the automated translation, standard cell ASICs require significantly less design effort than full-custom. This reduces the NRE of a standard cell design. All the same, the remaining expenses are often still prohibitive.

On average, a standard cell ASIC runs at one sixth to one eighth the speed of a full-custom implementation [26]. This gap is approximately equivalent to the progress made over five process generations or nearly a full decade. Power consumption can range from three to ten times that of a custom circuit [23]. This performance degradation is usually seen commercially as the cost of doing business in a world where time to market, and thus short turnaround time of chips, is vitally important to success. A number of vendors offer standard cell libraries, including IBM [59], TSMC [134], Fujitsu [51], LSI Logic [81], MOSIS [88], NEC [93], Samsung [111], Texas Instruments [132], ST Microelectronics [123], and Infineon Technologies [61].

8.1.4 Configurable ASICs

Configurable ASICs are characterized by fabrication of both metal and silicon, followed by a post-fabrication hardware configuration to complete the implementation. The only technologies that fall into this class are academic proposals.

ASIC-FPGA hybrid. As ASIC-FPGA hybrid is an obvious middle ground between ASICs and FPGAs. Depending on one's perspective, such hybrids may consist of an array of configurable gates embedded in an ASIC chip, or, conversely, ASIC components embedded in an FPGA. Either way, however, the final chip consists of some custom components and some configured logic.

The appeal of such chips is that the FPGA logic allows updates, iterations and bug fixes without incurring a costly re-spin. Even though these chips require custom masks, their proponents argue that the savings from avoiding re-spins greatly outweighs the mask expense. In fact, the higher the mask cost, the more advantageous it becomes to avoid re-spins [155]. Such hybrid chips are particularly suitable for implementation of applications for which there is a base design that might be re-used with additional changing features (e.g., printers, faxes, or designs supporting multiple or emerging standards).

This technology is not especially mature, and it faces a number of open challenges. The major design question an ASIC-FPGA hybrid raises, is how to partition a design's functionality between the ASIC and FPGA. Natural candidates for the FPGA fabric are any circuits that are expected to change. It is more difficult to use the FPGA logic to fix or avoid bugs, as this requires predicting where and what sort of bugs will arise. Certain types of logic are known to be bug-prone [155] and are therefore also reasonable candidates. Finally, studies show that the percentage of newly-designed, and hence more bug-prone, logic in SoCs has been shrinking, down from 65% in 1999 to 15% in 2005 [62]. While this trend is promising, the logic remaining in that 15% sill exceeds the capacity of today's embedded FPGA fabrics.

Even if a suitable FPGA-ASIC partition can be identified, the design tools present a challenge. ASIC and FPGA design flows do not have a lot in common. As Section 8.1.5 discusses in more detail, ASIC tool chains typically manipulate the circuit as a flat entity while FPGA tools treat it hierarchically.

Even if one succeeds in implementing a hybrid design, more challenges remain. Testing a hybrid ASIC-FPGA chip poses a problem, because the FPGA configuration is not known at fabrication time. It is possible to test the FPGA fabric itself, and perhaps the ASIC components in isolation, but it is not possible to test the entire design consisting of the circuit to be implemented in the FPGA and its interaction with the ASIC part of the chip.

In light of these challenges, why not address them with a two-die solution, keeping the configurable and fixed circuits on separate dies? First, the single chip approach avoids the performance and power penalty of off-chip communication. Second, the single chip approach eliminates the cost of assembly, test and packaging of a second chip. Recent developments

in packaging technology, discussed later in Section 8.2, are undermining this argument, and thus a two chip solution might become viable in the future.

8.1.5 Structured ASICs

Structured ASICs are a form of custom chip in which the silicon die is prepared with an array of logic components, sometimes called *cells*, *modules*, or *tiles*. The components in these arrays are typically classified as either coarse-, medium- or fine-grained. A custom metalization step, in which custom metal layers are applied to standard silicon layers, completes the chip.

- **Fine-grained:** In existence since the 1980s, components in these arrays are transistors or logic gates. These fine-grained structured arrays are sometimes called "gate array ASICs".

- **Medium-grained:** Medium-grained structured ASICs are built of components approximately the size of a mux.

- **Coarse-grained:** Elements of these arrays are LUT-based, and are frequently paired with a corresponding FPGA, allowing designs initially implemented on the FPGA to be readily migrated to the structured ASIC fabric.

We refer to such devices as "structured ASICs," although they are also alternately called "gate array ASICs", "platform ASICs" and "mask-programmable gate arrays" or MPGAs. Structured ASICs are currently commercially available at the 180 and 250 nm nodes[15] through companies such as AMI Semiconductor [9], ChipX [27], eASIC [44], Faraday [50], Fujitsu [51] and NEC [93].

With respect to performance and power, structured ASICs are more ASICs than FPGAs. They can achieve nearly 70-80% of standard cell performance, compared to 10-20% for FPGAs [153]. For power, structured ASICs require 2-3x the power of ASICs compared with 10-15x for FPGAs [153]. Standard cell ASICs NRE cost, including design, is rarely less than $5M, while for FPGAs this expense is closer to $100,000. Structured ASICs' NREs are in

the neighborhood of $500,000 and fall much closer to FPGAs than ASICs [146]. Similarly, with turnaround times in the days to weeks [113], they are much more FPGA-like, 1-4 weeks, than ASIC-like, 2-5 months [155].

Structured ASICs have their downsides. Toolchains targeting ASICs and FPGAs differ in many respects. Most fundamentally, they differ in their circuit representation. While FPGA tools view the circuit hierarchically, or are *block aware*, ASIC tools see the circuit as flat. Current tools that target structured ASIC platforms are based on ASIC tools and thus also use flat circuits. Studies have shown that developing a hierarchy aware structured ASIC toolchain could improve the quality of the resulting circuit by as much as 25% [153]. There has been some effort to translate FPGA-compiled designs to structured ASIC designs, by preserving the netlist and placement and simply re-routing signals [136]. However, this technique applies only to coarse-grained structured ASICs for which there is a corresponding FPGA.

In addition to the poor tool support, the architectural design space of the platform itself is largely unexplored. For example, in contrast, 3-, 4-, and 5-input LUTs were examined both in academia and industry for FPGA platforms [153]. Even though ultimate circuit performance is much closer to that of standard cell ASICs than that of FPGAs, the circuit density of the chip can still be quite low [116]. This is least severe for fine-grained structured ASICs, and most severe for coarse-grained versions, where the logic density is almost exactly that of an FPGA.

In a loose way, brick and mortar resembles structured ASICs. While a structured ASIC provides a fixed array of logic blocks with a custom interconnect on top, brick and mortar offers a custom array of bricks with a one-size-fits-all I/O cap. However, brick and mortar's bricks offer larger, more complex functions than the lookup tables and RAMs typical of structured ASICs. Also, because brick and mortar requires that bricks fit a standard form factor to interact with the I/O cap, some logic area might go to waste. Because structured ASICS have a custom interconnect, they are therefore not subject to this restriction on the logic blocks.

8.1.6 Configurable Structured ASICs

Members of the configurable structured ASIC class resemble members of the configurable ASIC class described in Section 8.1.4 with the exception that the silicon layers of these chips are standard and not custom. The Embedded Array from ChipX [27, 28] is a member of this class.

8.1.7 Configurable Hardware

The final implementation technology class is configurable hardware, which is customized entirely via post-fabrication configuration. Of the chip classes outlined, these offer the fastest turnaround time (including design, validation and production) because they alone do not require any physical implementation at a fab. This allows the designer to avoid many of the growing challenges associated with implementation in a deep sub-micron silicon process.

Within the class of configurable hardware, the hardware may or may not be reconfigurable, and the configuration may or may not be volatile.

Field-programmable gate arrays. The most famous member of the configurable hardware class is the FPGA. One large selling point for FPGAs, is that, as architectures, they are "future proof" [19] in that they are regular, parallel, highly pipelined, distributed and testable. Xilinx [147] and Altera [6] are the market leaders. Lattice Semiconductor [75] manufactures non-volatile FPGAs, while QuickLogic [102] and Actel [3] offer anti-fuse-based, non-reconfigurable and the devices. While these devices are also non-volatile, the loss of reconfigurability and accompanying inability to make design updates costs them a principle advantage of reconfigurable hardware.

FPGAs with hard IP cores. For years, FPGA manufacturers have provided complex, fixed-logic cores inside their FPGA fabrics. For example, Virtex2Pro [1] provides both fixed multipliers, SRAM blocks, and entire PowerPC cores. Recent products from Xilinx[147] and Altera[6] have specialized further, with specific FPGAs targeting different market segments (e.g., the Xilinx "FX" series targets embedded processing, and the "SX" series aims for signal

processing). The advantage to having these cores is that, if a design requires them, they incur little area/delay/power overhead relative to an ASIC. The disadvantage is that the core selection is set by the FPGA manufacturers and the product offerings that are necessarily limited. Brick and mortar's ability to synthesize different combinations of complex logic functions cheaply into the same chip is a potential advantage over these domain-specific FPGAs.

Coarse-grained, reconfigurable devices. These chips consist of relatively large, reconfigurable "objects", which are configurably connected in an FPGA style. Many coarse-grained, reconfigurable devices tend to target specific application domains [100, 24] or types of computation [45, 115, 86, 119, 2], starting even with products from FPGA vendors, which incorporate larger fixed blocks used in embedded and signal processing domains. FPGAs are distinguished from this group because their structure assumes nothing, down to the bit-level, about the configured circuit.

In some devices these objects are targeted towards a specific class of applications. For example, the picoArray from picoChip [100] targets wireless signal processing. In other cases, such as QuickSilver [103], Abric [8], and Cradle Technologies [33] offerings, the compute nodes are more general. In still other cases, the entire device operates as a single reconfigurable processor [45]. One startup, MathStar, Inc. [86], recently introduced its second generation field-programmable object array (FPOA) family called Arrix, which supports 400 individually configured 16-bit objects connected via a 1GHz programmable interconnect. A second startup, CSwitch [34], has announced an architecture consisting of configurable control, compute and switch nodes, connected via a 20-bit wide, 2 GHz interconnect fabric.

Certain applications, such as HDTV decoding, map well onto these devices. Applications that map well generally contain significant amounts of traditional data parallelism and operate on word-size chunks of data. Applications that do not map well are those that require specialized bit-level operations and those with specific circuit requirements (e.g., analog to digital converters). There is a synergy between brick and mortar and these technologies. Brick and mortar fabrication can produce coarse-grained, configurable devices. It also offers the opportunity to mix and match part types and process technologies to

produce a wider variety of these coarse-grained, configurable chips.

8.2 Chip carriers and the I/O cap

There are a number of trends driving packaging innovation today, including thermal management, interconnect density, and integration. All but one of these trends originates with the integrated circuit itself, an indication sign that circuit and packaging technology are becoming increasingly interwoven. Soon it will be impossible to develop one while ignoring the other. The major thrusts of packaging research outlined in this section will further illustrate this inter-relatedness.

- As a result of Moore's Law, integrated circuit features are shrinking, resulting in more potential I/Os per die. This pushes packaging to accommodate more densely packed I/O signals.

- Another consequence of Moore's Law is increasing integrated circuit speed, which makes off-chip components, relatively, many more cycles away. This compels package designers (1) to try to reduce the penalty of crossing the package boundary, and (2) to reduce the need for package boundary crossings by bringing other system components within the boundaries of a single package.

- As die feature density and speed increase, the power density of the die also increases, meaning that packages must work even harder to supply power and dissipate the resulting heat.

- Ever-shrinking consumer devices are pushing packages to continue to shrink and improve packaging space efficiency.

The result of these trends are the following three major avenues of packaging development.

8.2.1 High-performance carriers

The last decade has seen a surge of research into high-performance chip carriers. The carrier is the part of the package which is responsible for redistributing signals from the chip I/O

pads to the package I/O pads. Historically wire bonds have been the "workhorse technology" [18] of this redistribution with the carrier's role restricted to physically "carrying" the die. However, with the advent and widespread adoption of flip chip die-to-substrate bonding, the carrier itself has provided the signal redistribution, via a layer-based interconnect.

Recently, a number of technologies have been proposed to improve carrier signal redistribution performance and density:

- **Sequential build-up (SBU) laminate substrate [18]** First invented in 1988, these carriers are essentially sophisticated printed circuit boards (PCBs). In 1997 Intel chose an SBU carrier for use in a flip chip package, and they have been in wide use ever since. They are significantly lower cost and higher performance than earlier multilayer ceramic substrates, and to this day development of denser designs and more-reliable SBU manufacturing techniques represent a large segment of PCB research and development.

- **Bumpless build-up layer (BBUL) [83, 133]** BBUL is related to SBU laminate substrates, but is characterized by its lack of controlled collapse chip connection (C4) bumps between chip and carrier. Unlike SBU laminates, BBUL bypasses C4 bumps and manufactures the carrier directly onto the surface of the chip, resulting not only in a thinner unit, but also in minimal signal discontinuity which reduces signal delay and improves electrical performance. As of 2003, BBUL had yet appear in any commercial products.

- **Silicon Carriers [47]** Silicon carriers were developed in recent years at IBM. Enabled by through silicon vias, silicon carriers offer unprecedentedly high I/O and wiring density. As of 2005, silicon carriers offered 10 to 100 μm I/O pitch and 1 to 10 μm wiring pitch, compared with 150 μm I/O pitch and 40 μm wiring pitch in contemporary laminate substrates. In addition, the silicon carrier and silicon die have similar thermal expansion properties, thus reducing the mechanical stress on the chip-carrier joint and improving reliability. The less agitation there is at this joint, the smaller the contacts can be. IBM has developed micro C4 bumps (μC4) with 25 μm diameters on a 50

μm pitch compared with typical 100 μm solder bumps on 200 or 225 μm pitch. This reduction in size represents a 16-fold improvement in die I/O area density. This technology might be an ideal technology in which to implement the I/O cap.

8.2.2 Thermal management

Heat transfer poses the second great challenge to modern package designers. Currently, packages cannot dissipate more than 40 watts without the use of bulky fans [105]. In 2005, researchers began to make the transition from air-based cooling mechanisms (which can dissipate up to 90 watts/cm^2) to liquid cooling (which can dissipate up to 200 watts/cm^2).

Complicating the cooling problem is the fact that any heat dissipation technique is less effective against an uneven heat source, caused, for example, by hot spots on the chip. To be effective, a cooling system must use a heat spreader to distribute the heat evenly before dissipation and cooling. In addition, various pieces of the package respond differently to heat. In particular the region between the chip and the seal of the package lid is extremely susceptible to heat.

8.2.3 Deeper integration and shrinking form factor

Finally, driven by the enormous mobile device market, package designers are constantly striving to reduce the size of packaged chips. This is not a new battle, and past innovations, such as surface mount technology and flip chip, have reduced the footprint of packaged devices. Today this battle is largely pushing in the direction of multiple-die packages.

Multi-chip modules (MCMs) consists of multiple silicon integrated circuits which share a single package. MCMs have been in commercial use for over 30 years, with packages as large as 10cm on a side in use [65]. Integrating multiple dies into a single package not only reduces the collective package overhead, but can also improve the cross-die communication bandwidth and latency. Just like single-chip packages, multiple-chip packages must interconnect, power, cool, and protect their contents. As multiple dies are integrated via ceramic then organic (plastic) then silicon carriers, there are orders of magnitude improvements to be had in inter-die communication performance. These improvements approach, but do

not quite meet, fully integrated, 3D silicon performance [10, 47]. Depending on how one integrates system components into a package, it might be called a system-on-package (SoP) or a System-in-package (SiP). An SoP integrates components into the carrier itself while an SiP integrates multiple dies into a single package.

A second school of thought attempts to reduce the package footprint, not by building die-on-die stacks, but instead stacking package-on-package. While both performance and space efficiency suffer under this scheme, it proffers a number of advantages. First, by standardizing the pin connections, one gains package pluggability and its accompanying flexibility not unlike how a standard interface enables mixing and matching of bricks. By contrast, each die-on-die stack constitutes a completely unique design. The second advantage relates to testing. To maximize the yield of any multi-chip design, die-on-die or package-on-package, one should test each individual chip before including it in an assembly, and it is significantly easier to effectively test a packaged chip than a bare die.

Ultimately the number of dies, space and economic constraints of specific applications will determine which of these two 3D integration approaches is the more suitable space saving scheme.

8.3 On-chip communication networks and the polymorphic fabric

On-chip networks are an extremely rich research domain, covering both breadth and depth. Here we outline the research context most helpful in framing our polymorphic network design.

8.3.1 Network designs

The history of on-chip networks begins with off-chip, large-scale system interconnects. This previous art used a variety of topologies, amongst them those included in our study: fat tree [78], flattened butterfly [69], mesh [37] and ring [37]. Present and future multi-core designs demand much more than a simple integration of earlier large-scale system interconnects onto a single die. For example, while mapping meshes and rings is relatively easy, mapping a butterfly network onto a single chip presents more challenges. In addition, new topologies, specifically designed for on-chip interconnects have arisen [69, 32].

Meanwhile, the switches that are connected to form these topologies have also been refined to meet the stringent power, latency, and fault tolerance requirements of an on-chip network [30, 72, 139, 89]. The result is that today, we have a rich and expanding array of options when implementing an on-chip interconnect.

8.3.2 Tailored NoC designs

There are a number of published techniques for determining the appropriate network design for a specific application. In 2006, Hu et al. [57] presented evidence that non-uniform network input buffers offer significant performance improvements. The work presents a pre-fabrication buffer allocation algorithm that can be applied to find the appropriate buffer sizings to best improve performance and economize resources. Other research develops tools to synthesize a custom network on chip for a particular application [53, 90]. Each of these projects has demonstrated a significant improvement in performance when the interconnect was tailored to an application.

8.3.3 Relationship to FPGAs

Field-programmable gate arrays (FPGAs) already contain configurable networks. While certainly the concept of "island style" routing and FPGAs have inspired our polymorphic network, the details and underlying purpose are entirely different. The ability to configure routes at the bit level, means the overhead is quite high. Consequently, the long history of research into coarse-grained FPGAs, outlined in Section 8.1.7. This work reduces the overhead caused by bit-level configurability, but still provides interconnect structures geared towards fully statically routed designs, as would be required for emulating circuits. A polymorphic network is a tailored configurable device designed for emulating on-chip networks, in the same way that an FPGA is a tailored configurable device designed for emulating circuits.

8.4 Multi-die assemblies

Both system-on-package (SoP) and multi-chip modules (MCMs) package multiple silicon dies together in a single package. Those technologies were discussed in some detail in Section 8.2.3. What distinguishes brick and mortar from these devices is the architectural work. SoPs are a way to lower package costs, but still rely upon users to design and pay for fabrication of the constituent ASICs that are bonded together. Our goal, instead, is to develop a market of pre-fabricated ASIC bricks that interconnect in a standard way.

8.5 Self-assembly techniques

Self-assembly has been studied extensively. The goal of this research has always been similar to our motivation for proposing FSA as a brick and mortar assembly technique. It offers a low-cost alternative for bulk manufacturing that would otherwise require robotic assembly. To the best of our knowledge, only one commercial company has attempted to employ FSA, Alien Technologies [5]. Alien intended to use FSA to bond the antenna to the processing device for RFID tag production.

8.6 Spatial application scheduling

Instruction scheduling is a classic compiler optimization problem, which has been widely studied [101, 142, 79, 67]. We focus our discussion of related work on algorithms designed for tiled architectures which have a spatial component.

8.6.1 Scheduling on clustered microarchitectures

Recent processor designs partition hardware resources to counteract increasingly long communication latencies (relative to computation latencies). Instruction scheduling is used, in part, to minimize the additional cycle spent accessing the remote resources. Several scheduling algorithms attempt to balance operand locality with instruction-level parallelism. We opted to begin our scheduling work by implementing Bottom-Up-Greedy [46], because it is the canonical solution for this type of problem. Unified Assign and Schedule [95] was also chosen, because it is, in essence, a general scheduling framework into which a compiler writer

inserts a custom heuristic for the target architecture. The original developers examined several heuristics for clustered microarchitectures [95]. In adapting UAS for WaveScalar, we developed the WaveScalar-specific heuristic described in Section 6.3.2.

Other efforts have focused specifically on clustered VLIW architectures, including modulo scheduling [112], which, like UAS, performs both instruction placement and instruction ordering, and in later versions, register spill code insertion [154], in a single pass. The work by Zalamea et al. [154] studies the effects of program transformations on the success of these scheduling algorithms.

The developers of Lx [49], a clustered VLIW microarchitecture, also studied the spatial aspect of instruction scheduling, and developed another heuristic-based approach to instruction placement [42]. This method employs a pared-down list scheduler to evaluate the ultimate schedule length for a given placement.

8.6.2 Scheduling for Raw

Despite several fundamental differences, Raw [138] and WaveScalar share a grid topology. Raw tiles are arranged in a 2D mesh, with distance-dependent communication latencies between tiles. Raw employs a four-phase instruction scheduler [76, 77]: clustering instructions using dominant sequence clustering [149], merging clusters to match the smaller number of tiles, assigning the clusters to tiles, and finally producing a temporal schedule for the instructions at each tile using a list scheduler.

8.6.3 Scheduling for TRIPS

The TRIPS compiler [91, 31] assigns instructions to locations in its grid of processing elements based on the estimated length of the critical path. The compiler applies this algorithm to map each instruction in a hyperblock of up to 128 instructions to one of execution tiles. This smaller problem size, coupled with additional mapping constraints such as the location of the register file and memory interface, significantly reduces the space of possible mappings.

8.6.4 Partitioning and scheduling for data cache locality

Data cache-conscious instruction scheduling has seen two primary thrusts. One approach has been to schedule instructions so as to improve the locality of the data stream for a single processor [121, 25, 106, 144]. The second approach, applied to shared memory multiprocessors, selects and assigns tasks to processors to minimize data sharing between them [137, 7, 11]. Neither of these techniques is particularly well-suited to the application scheduling problem as it pertains to brick and mortar. The former deals with temporal schedules which we do not. The latter deals with cache coherence issues due to distributed memory which should be subsumed by a good application schedule.

Chapter 9

VARIATIONS, FUTURE RESEARCH, AND CONCLUSIONS

In this chapter we examine some modifications to the basic brick and mortar system presented in this thesis. We also discuss avenues for future research that build on the work described here.

9.1 Variations on brick and mortar

Chip manufacturing involves complex trade-offs between performance, function, cost, time and power. Because we set out to attack chip manufacturing costs, when faced with one of these trade-offs, we always opted to save cost. However, we believe that making different design choices could lead to other useful points in the chip manufacturing design space. These variations employ the basic brick and mortar technique, but use it slightly differently from the way we have presented. We list the variations in order of increasing deviation from the system presented here.

9.1.1 Multiple I/O caps

While a single I/O cap design is the least expensive way to implement many different chips, one might consider offering users a choice of multiple I/O caps. There would be a cost associated with this, but it would allow users to better fit their inter-brick and external communication resources to their application's needs.

9.1.2 A custom component

If the user has some functionality that cannot be implemented with the desired performance using the standard bricks or I/O cap, it would be possible for the user to fabricate a custom component for use in their chips. This would be significantly more costly than using exclusively standard components. However, because it uses *some* standard components, it

would be significantly less expensive than implementing the entire system from scratch. In this scenario, the user could save some expense by fabbing his or her component in a slightly older process.

9.1.3 Brick re-use over time

We have proposed achieving brick re-use across multiple different chip designs. An alternate option would be to re-use bricks to implement a single design that evolves over time. If features of a chip could be designed to map onto individual bricks, this would be a very efficient way to incrementally add new features to a design. One could evolve a design by adding a brick or bricks supporting the newest features and while continuing to use existing bricks for the remainder of the design.

9.1.4 Breaking the function-communication abstraction

The brick and mortar system presented here has maintained a delineation between computation (implemented in bricks) and communication (implemented in the I/O cap). This physical distinction is, in fact, unnecessary. What truly distinguishes the bricks from the I/O cap is how they are used in the chips. Brick and mortar chips automatically include whatever circuitry is contained in the I/O cap, whereas the brick layer is customizable.

There is no reason that computation *has* to be contained in bricks. If there is some computational core that turns out to be useful for all chips (for example, something very general, such as a processor or SRAM), then there is no reason it could not be placed in the I/O cap. Similarly the brick layer which can be customized per chip can be used to augment the basic communication network for a particular chip.

9.2 Future research

There are a number of interesting veins of research that build on what has been presented here.

9.2.1 Polymorphic network

Reconfiguration. As mentioned in Chapter 3, the polymorphic network has potential uses beyond the brick and mortar I/O cap. A monolithic chip in which the workload changes dramatically may be able to take advantage of a configurable network. This use raises an interesting question of when to reconfigure a network. Reconfiguration is likely to be fast compared to an FPGA (as the configuration file itself is significantly smaller). If an FPGA takes roughly tens of milliseconds, polymorphic network configuration probably takes a handful of milliseconds. Determining when and how to reconfigure the network is an open question. Reconfiguration is analogous in many ways to a context switch in an OS. Extending the analogy, the program in this case is the configuration itself. One would essentially be looking to design an operating system for spatial (rather than the traditional temporal) applications.

Adaptation to communication needs. This online network reconfiguration framework could be augmented to support evolution of the network configuration. Such a network would adapt to the changing communication characteristics of applications, application phases and input data sets. Research questions include not only when to reconfigure the network, as mentioned above, but how much of the network to change and in what way. This task might be well-suited to self-organization or online learning algorithms.

Support for on-chip isolation. In addition to connecting communicating cores, the polymorphic network could be used to isolate cores that should *never* communicate. When we want to maintain functional and performance isolation between software entities (such as processes or virtual machines) that share the same chip, this could be especially useful.

Microarchitectural optimization. The microarchitecture of the polymorphic network presented here is relatively simple, and there is room to push some of the recent advances made by the on-chip network community into the fabric.

9.2.2 Software systems

Per-chip API construction. It would be very useful to add an API to the system stack in Figure 1.1 between the hardware and software. One could envision constructing such an API in building block fashion along with the hardware system. As the hardware designer adds bricks, modules could automatically be inserted in the API.

Modular OS. Brick and mortar makes it inexpensive to produce a family of related hardware devices. Construction of a per-device operating system might negate much if not all of the cost savings. How might one design an operating system that can run on varied hardware devices? Perhaps the OS should be modular like the chip itself. More generally, how far up the application stack should the software know about brick and mortar?

Variation in programming model. One might use brick and mortar to support not only variations in functional cores, but variations in other aspects for the system. For example, one might offer two I/O caps, one that supports distributed points of control within the bricks, and a second that coordinates global synchronization.

Mix and match fine scheduling algorithms. The hierarchical application scheduling algorithm presented in Chapter 6 makes it possible to select tailored fine scheduling algorithms on a per-core basis. While all WaveScalar cores are equivalent and thus would not benefit, this code scheduling scheduling framework could incorporate a variety of per-brick tailored fine scheduling algorithms to improve application performance on brick and mortar chips.

9.2.3 Refinement of brick and mortar design

Physical and architectural design space co-exploration. Brick and mortar chips incorporate two very large design spaces: the architectural and the physical. The architectural space concerns the functionality implemented within the walls of any individual piece of silicon. The physical space concerns the specific method of assembling dies and the technology used to bond them together. As we sought to optimize cost, we selected

the least costly points in the physical space and then explored the architectural space. In actuality this is a co-design problem, where the physical design point selected will influence your choice of architectural design and vice versa. This thesis leaves open a comprehensive exploration of these interactions.

Power savings. As presented in this thesis a brick and mortar implementation of a design is likely to consume more power than a monolithic implementation due to the addition of inter-die signaling. There is opportunity to recoup some of that power by taking advantage of asynchronous communication between bricks to support voltage scaling. One could also push some asynchronous design into the I/O cap, which as the largest single die that stands to benefit the most.

Optimizing design across die boundaries. It would be possible to break the current delineation between communication (supported in the I/O cap) and computation (supported in the bricks). Breaking this distinction opens up the possibility of "network booster bricks" which can be added, on a per-application basis to improve communication performance. Conversely one could provide generic computation hardware to all designs by implementing it in the I/O cap.

9.3 Conclusion

Brick and mortar chips reduce the engineering effort required to implement a custom circuit in a modern silicon process. Reducing this effort significantly reduces the total cost of the chip. We achieved this by designing a system in which silicon components can be re-used across many custom designs, thereby amortizing the cost of creating the components. We developed and employed a model of chip manufacturing cost and used it to examine the economic constraints on brick manufacturing and to determine under what circumstances brick and mortar is economically beneficial.

This thesis presents and explores several technical aspects of brick and mortar chips. We presented a methodology to develop a re-usable family of bricks, which makes efficient use of the available silicon area and inter-brick communication resources. For communication

between the bricks, we developed a polymorphic network architecture. This network can be configured on a per-design basis to function as the network that best suits an application's needs. We explored the interaction between the brick and mortar system architecture and the physical technique by which the components are assembled. We found that planning for some flexibility in the architecture can be used to dramatically increase manufacturing production rate. Finally, to maximize application performance while executing on a highly partitioned substrate like a brick and mortar chip, we developed a software partitioning algorithm that carefully balances communication penalties and computational resource conflicts.

In closing, we examined how one might use brick and mortar to assemble made-to-order CMPs. We found that such chips perform comparably to their monolithic ASIC counterparts. Modularity and re-use have long histories in engineering, and we hope that their application to hardware via brick and mortar chips will put custom hardware within reach of more applications.

BIBLIOGRAPHY

[1] Virtex-II Pro and Virtex-II Pro X FPGA user guide. www.xilinx.com.

[2] Arthur Abnous, Hui Zhang, Marlene Wan, George Varghese, Vandana Prabhu, and Jan Rabaey. The pleiades architecture. *The Application of Programmable DSPs in Mobile Communications*, pages 327–360, 2002.

[3] Actel website. www.actel.com.

[4] Fariborz Agahdel, Chung Ho, and Randal Roebuck. Known good die: A practical solution. In *Proceedings of the International Conference and Exhibition on Multichip Modules*, pages 177–182, 1993.

[5] Alien technology website. www.alientechnology.com.

[6] Altera website. www.altera.com.

[7] Saman P. Amarasinghe and Monica S. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 126–138, 1993.

[8] Ambric, Inc. website. www.ambric.com.

[9] AMI Semiconductor website. www.amis.com.

[10] Amkor technology website. www.amkor.com.

[11] Jennifer M. Anderson and Monica S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 112–125, 1993.

[12] Artisan website. www.artisan.org.

[13] Arvind and Rishiyur S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3):300–318, March 1990.

[14] James Balfour and William J. Dally. Design tradeoffs for tiled CMP on-chip networks. In *Proceedings of the International Conference on Supercomputing*, pages 187–198, 2006.

[15] Richard Ball. The promise of structured ASIC. *Electronics Weekly*, October 2004. www.electronicsweekly.com.

[16] Alan Barber, Ken Lee, and Hannsjorg Obermaier. A bare-chip probe for high I/O, high speed testing. Technical Report HPL-94-18, Hewlett Packard, March 1994.

[17] Roy L. Russo Bernard S. Landman. On a pin versus block relationship for partitions of logic graphs. *IEEE Transactions on Computers*, 20(12):1469–1479, December 1971.

[18] Edmund D. Blackshear, Moises Cases, Erich Klink, Stephen R. Engle, Ronald S. Malfatt, Daniel N. de Araujo, Stefano Oggioni, Luke D. LaCroix, Jamil A. Wakil, Gareth G. Hougham, Nam H. Pham, and David J. Russell. The evolution of build-up package technology and its design challenges. *IBM Journal of Research and Development (POWER5 and Packaging)*, 49(4/5), 2005.

[19] Ivo Bolsens. Challenges and opportunities for FPGA platforms. *Lecture Notes in Computer Science*, 2438, 2002.

[20] Keith A. Bowman, Steven G. Duvall, and James D. Meindel. Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration. *Journal of Solid-State Circuits*, 37(2):183–190, February 2002.

[21] Karen Brown. Economic challenges on the path to 22 nm. *Future Fab International*, June 2004. www.future-fab.com.

[22] Anantha P. Chandrakasan. *Low Power Digital CMOS Design, Chapter 4*. Kluwer Academic Publishers, 1995.

[23] Andrew Chang and William J. Dally. Explaining the gap between ASIC and custom power: a custom perspective. In *Proceedings of the Conference on Design Automation*, pages 281 – 284, 2005.

[24] Devereaux C. Chen. *Programmable Arithmetic Devices for High Speed Digital Signal Processing*. PhD thesis, University of California, Berkeley, 1992.

[25] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 1999.

[26] David G. Chinnery and Kurt Keutzer. Closing the gap between ASIC and custom: An ASIC perspective. In *Proceedings of the Conference on Design Automation*, pages 637–642, 2000.

[27] ChipX product offerings. www.chipx.com.

[28] ChipX introduces embedded array products as alternatives to standard cell ASICs, February 2007. www.soccentral.com.

[29] Thomas D. Clark, Rosaria Ferrigno, Joe Tien, Kateri E. Paul, and George M. Whitesides. Template-directed self-assembly of 10-micron-sized hexagonal plates. *Journal of the American Chemical Society*, 124:5419–5426, 2002.

[30] Kypros Constantinides, Stephen Plaza, Jason Blome, Bin Zhang, Valeria Bertacco, Scott Mahlke, Todd Austin, and Michael Orshansky. Bulletproof: A defecttolerant CMP switch architecture. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, pages 5–16, 2006.

[31] Katherine Coons, Xia Chen, Sundeep Kushwaha, Kathryn McKinley, and Doug Burger. A spatial path scheduling algorithm for EDGE architectures. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 129 – 140, 2006.

[32] Marcello Coppola, Riccardo Locatelli, Giuseppe Maruccia, Lorem Pieralisi, and Alberto Scandurra. Spidergon: a novel on-chip communication network. In *Proceedings of the International Symposium on System-on-Chip*, page 15, 2004.

[33] Cradle technologies website. www.cradle.com.

[34] CSwitch website. www.cswitch.com.

[35] David E. Culler, Anurag Sah, Klaus E. Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164 – 175, 1991.

[36] W. J. Dally and Charles L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, 36(5):547–553, 1987.

[37] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[38] William J. Dally. Virtual-channel flow control. In *Proceedings of the International Symposium on Computer Architecture*, pages 60–68, 1990.

[39] William J. Dally and John W. Poulton. *Digital Systems Engineering*. Cambridge University Press, New York, NY, USA, 1998.

[40] Alan L. Davis. The architecure and system method of DDM1: A recursively structured data driven machine. In *Proceedings of the International Symposium on Computer Architecture*, pages 210–215, 1978.

[41] Jack B. Dennis. A preliminary architecture for a basic dataflow processor. In *Proceedings of the International Symposium on Computer Architecture*, pages 126–132, 1975.

[42] Giuseppe Desoli. Instruction assignment for clustered VLIW DSP compilers: A new approach. Technical Report HPL-98-13, Hewlett-Packard Laboratories, January 1998.

[43] Robert J. Drost, Robert David Hopkins, Ron Ho, and Ivan E. Sutherland. Proximity communication. *IEEE Journal of Solid State Circuits*, 39(9), September 2004.

[44] eASIC website. www.easic.com.

[45] Carl Ebeling, Darren C. Cronquist, Paul Franklin, Jason Secosky, and Stefan G. Berg. Mapping applications to the RaPiD configurable architecture. In *Symposium on FPGAs for Custom Computing Machines*, page 106, 1997.

[46] J.R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. PhD thesis, Massachusetts Institute of Technology, 1986.

[47] John U. Knickerbocker et. al. Development of next-generation system-on-package (SOP) technology based on silicon carriers with fine-pitch chip interconnection. *IBM Journal of Research and Development*, 49:725, 2005.

[48] Jiandong Fang and Karl F. Böhringer. Wafer level packaging based on uniquely orienting self-assembly (the DUO-SPASS processes). *Journal of Microelectromechancial Systems*, 15(3):531–540, June 2006.

[49] Paolo Faraboschi, Geoffrey Brown, Joseph A. Fisher, Giuseppe Desoli, and Fred Homewood. Lx: A technology platform for customizable VLIW embedded processing. In *Proceedings of the International Symposium on Computer Architecture*, pages 203–213, 2000.

[50] Faraday electronics website. www.faradayelectronics.com.

[51] Fujitsu website. www.fujitsu.com.

[52] Victor G. Grafe, George S. Davidson, James E. Hoch, and Victor P. Holmes. The ϵpsilon dataflow processor. In *Proceedings of the International Symposium on Computer Architecture*, pages 36 – 45, 1989.

[53] Cristian Grecu and Michael Jones. Performance evaluation and design trade-offs for network-on-chip interconnect architectures. *IEEE Transactions on Computers*, 54(8):1025–1040, 2005.

[54] John R. Gurd, Chris C. Kirkham, and Ian Watson. The Manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52, 1985.

[55] Michael Heskins and James E. Guillet. Solution properties of poly(N-isopropylacrylamide). *Journal of Macromolecular Science*, A2(8):1441–1455, 1968.

[56] Rob Hilkes. Under the hood: Uncovering hidden chip costs. *EE Times*, October 2007. www.eetimes.com.

[57] Jingcao Hu, Umit Y. Ogras, and Radu Marculescu. System-level buffer allocation for application-specific networks-on-chip router design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(12):2919–2933, December 2006.

[58] Dale L. Huber, Ronald P. Manginell, Michael A. Samara, Byung-Il Kim, and Bruce C. Bunker. Programmed adsorption and release of proteins in a microfluidic device. *Science*, 301(5631):352–354, 2003.

[59] IBM website. www.ibm.com.

[60] IC Knowledge LLC website. www.icknowledge.com.

[61] Infineon Technologies website. www.infineon.com.

[62] International technology roadmap for semiconductors, executive summary, 2005. www.itrs.net.

[63] International technology roadmap for semiconductors, executive summary, 2006. www.itrs.net.

[64] Antoine Jalabert, Srinivasan Murali, Luca Benini, and Giovanni De Micheli. xpipesCompiler: A tool for instantiating application specific networks on chip. In *Proceedings of the Conference on Design, Automation and Test in Europe*, page 20884, 2004.

[65] Ron Kalla, Balaram Sinharoy, and Joel M. Tendler. IBM Power5 chip: a dual-core multithreaded processor. *IEEE Micro*, 24:40–47, Mar-Apr 2004.

[66] Kouichi Kanda, Danardono Dwi Antono, Koichi Ishida, Hiroshi Kawaguchi, Tadahiro Kuroda, and Takayasu Sakurai. 1.27gb/s/pin 3mw/pin wireless superconnect (wsc) interface scheme. In *Proceedings of the International Solid State Circuits Conference*, pages 186–187, 2003.

[67] Daniel R. Kerns and Susan J. Eggers. Balanced scheduling: Instruction scheduling when memory latency is uncertain. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 515 – 527, 1993.

[68] Kurt Keutzer, Sharad Malik, and A. Richard Newton. From ASIC to ASIP: the next design discontinuity. In *Proceedings of the IEEE International Conference on VLSI in Computers and Processors*, pages 84–90, 2002.

[69] John Kim, William J. Dally, and Dennis Abts. Flattened butterfly topology for on-chip networks. In *Proceedings of the International Symposium on Microarchitecture*, pages 172–182, 2007.

[70] Masasuke Kishi, Hiroshi Yasuhara, and Yasusuke Kawamura. DDDP-A distributed data driven processor. In *Proceedings of the International Symposium on Computer Architecture*, pages 236–242, 1983.

[71] Arun Kottolli. The economics of structured- and standard-cell-ASIC designs. *EDN Magazine*, 2006. www.edn.com.

[72] Amit Kumar, Li-Shiuan Peh, Partha Kundu, and Niraj K. Jha. Express virtual channels: towards the ideal interconnection fabric. In *Proceedings of the International Symposium on Computer Architecture*, pages 150–161, 2007.

[73] Ian Kuon and Jonathan Rose. Measuring the gap between FPGAs and ASICs. In *International Symposium on Field Programmable Gate Arrays*, pages 21–30, 2006.

[74] Bernard S. Landman and Roy L. Russo. On a pin versus block relationship for partitions of logic graphs. *IEEE Transactions on Computers*, C-20(12):1469 – 1479, December 1971.

[75] Lattice Semiconductor website. www.latticesemi.com.

[76] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-time scheduling of instruction-level parallelism on a Raw machine. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 46 – 57, 1998.

[77] Walter Lee, Diego Puppin, Shane Swenson, and Saman Amarasinghe. Convergent scheduling. In *Proceedings of the International Symposium on Microarchitecture*, pages 111–122, 2002.

[78] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The network architecture of the connection machine CM-5 (extended abstract). In *Proceedings of the Symposium on Parallel Algorithms and Architectures*, pages 272–285, 1992.

[79] Jack L. Lo and Susan J. Eggers. Improving balanced scheduling with compiler optimizations that increase instruction-level parallelism. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 151–162, 1995.

[80] P. Geoffrey Lowney, Stefan M. Freudenberger, Thomas J. Karzes, W. D. Lichtenstein, Robert P. Nix, John S. O'Donnell, and John Ruttenberg. The multiflow trace scheduling compiler. *Journal of Supercomputing*, 7:51–142, 1993.

[81] LSI Logic website. www.lsilogic.com.

[82] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[83] Ravi Mahajan, Raj Nair, Vijay Wakharkar, Johanna Swan, John Tang, and Gilroy Vandentop. Emerging directions for packaging technologies. *Intel Technology Journal (Semiconductor Technology and Manufacturing)*, 3(2), 2002.

[84] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, and Mark Horowitz. Smart Memories: A modular reconfigurable architecture. In *Proceedings of the International Symposium on Computer Architecture*, pages 161–171, 2002.

[85] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Computer Architecture News*, 33(4):92–99, 2005.

[86] MathStar website. www.mathstar.com.

[87] Stephen Mick, John Wilson, and Paul Franzon. 4Gbps high-density AC coupled interconnection. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 133–140, 2002.

[88] MOSIS website. www.mosis.org.

[89] Robert Mullins, Andrew West, and Simon Moore. Low-latency virtual-channel routers for on-chip networks. In *Proceedings of the International Symposium on Computer Architecture*, page 188, 2004.

[90] Srinivasan Murali, Paolo Meloni, Federico Angiolini, David Atienza, Salvatore Carta, Luca Benini, Giovanni De Micheli, and Luigi Raffo. Designing application-specific networks on chips with floorplan information. In *Proceedings of the International Conference on Computer-Aided Design*, pages 355–362, 2006.

[91] Ramadass Nagarajan, Sundeep K. Kushwaha, Doug Burger, Kathryn S. McKinley, Calvin Lin, and Stephen W. Keckler. Static placement, dynamic issue (SPDI) scheduling for EDGE architectures. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 74–84, 2004.

[92] Ramadass Nagarajan, Karthikeyan Sankaralingam, Doug Burger, and Stephen W. Keckler. A design space evaluation of grid processor architectures. In *Proceedings of the International Symposium on Microarchitecture*, pages 40–51, 2001.

[93] NEC website. www.nec.com.

[94] Opencores.org website. www.opencores.org.

[95] Emre Özer, Sanjeev Banerjia, and Thomas M. Conte. Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures. In *Proceedings of the International Symposium on Microarchitecture*, pages 308–315, 1998.

[96] Y. Vickie Pan, Roger A. Wesley, Reto Luginbuhl, Denice D. Denton, and Buddy D. Ratner. Plasma polymerized N-Isopropylacrylamide: Synthesis and characterization of a smart thermally responsive coating. *Biomacromolecules*, 2(1):32–36, 2001.

[97] Gregory M. Papadopoulos and David E. Culler. Monsoon: An explicit token-store architecture. In *Proceedings of the International Symposium on Computer Architecture*, pages 82–91, 1990.

[98] Gregory M. Papadopoulos and Kenneth R. Traub. Multithreading: A revisionist view of dataflow architectures. In *Proceedings of the International Symposium on Computer Architecture*, pages 342–351, 1991.

[99] James M. Perkins. Magnetically assisted statistical assembly of III-V heterostructures on silicon: Initial process and technology development. Master's thesis, Massachusetts Institute of Technology, 2002.

[100] picoChip website. www.picochip.com.

[101] Todd A. Proebsting and Charles N. Fischer. Linear-time, optimal code scheduling for delayed-load architectures. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 256 – 267, 1991.

[102] QuickLogic website. www.quicklogic.com.

[103] Quicksilver technology website. www.qstech.com.

[104] Jan Rabaey. Reconfigurable processing: The solution to low-power programmable DSP. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 275–278, 1997.

[105] N.J. Rao and G. Ananda Rao. Introduction to electronic packaging: Semiconductor packaging. www.prc.gatech.edu.

[106] Gabriel Rivera and Chau-Wen Tseng. Data transformations for eliminating conflict misses. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 38–49, 1998.

[107] Joseph Rumpler. Optoelectronic integration using the magnetically assisted statistical assembly technique: Initial magnetic characterization and process development. Master's thesis, Massachusetts Institute of Technology, 2002.

[108] Joseph Rumpler, James M. Perkins, and Clifton G. Fonstad. Optoelectronic integration using statistical assembly and magnetic retention of heterostructure pills. In *Proceedings of the Conference on Lasers and Electro-Optics*, page 2, 2004.

[109] Shuichi Sakai, Yoshinori Yamaguchi, Kei Hiraki, Yuetsu Kodama, and Toshitsugu Yuba. An architecture of a dataflow single chip processor. In *Proceedings of the International Symposium on Computer Architecture*, pages 46–53, 1989.

[110] David Saltzman and Thomas T. Knight Jr. Capacitive coupling solves the known good die problem. In *Proceedings of the Multi-Chip Module Conference*, pages 95–100, 1994.

[111] Samsung website. www.samsung.com.

[112] Jesus Sanchez and Antonio Gonzalez. Instruction scheduling for clustered VLIW architectures. In *Proceedings of the International Symposium on System Synthesis*, pages 41–46, 2000.

[113] Michael Santarini. Structured ASICs deserve serious attention at 90nm. *EDN Magazine*, 2005. www.edn.com.

[114] T. Sanuki, Y. Sogo, A. Oishi, Y. Okayama, R. Hasumi, Y. Morimasa, T. Kinoshita, T. Komoda, H. Tanaka, K. Hiyama, T. Komoguchi, T. Matsumoto, K. Oota, T. Yokoyama, K. Fukasaku, R. Katsumata, M. Kido, M. Tamura, Y. Takegawa, H. Yoshimura, K. Kasai, K. Ohno, M. Saito, H. Aochi, M. Iwai, N. Nagashima, F. Matsuoka, Y. Okamoto, and T. Noguchi. High density and fully compatible embedded DRAM cell with 45nm cmos technology (cmos6). In *Proceedings of the Symposium on VLSI Technology*, pages 14–15, 2005.

[115] Herman Schmit, David Whelihan, Andrew Tsai, Matthew Moe, Benjamin Levine, and R. Reed Taylor. PipeRench: A virtualized programmable datapath in 0.18 micron technology. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 63–66, 2002.

[116] Deepak D. Sherlekar. Design considerations for regular fabrics. In *Proceedings of the International Symposium on Physical Design*, pages 97 – 102, 2004.

[117] Toshio Shimada, Kei Hiraki, Kenji Nishida, and Satoshi Sekiguchi. Evaluation of a prototype data flow processor of the SIGMA-1 for scientific computations. In *Proceedings of the International Symposium on Computer Architecture*, pages 226 – 234, 1986.

[118] Daniel P. Siewiorek, C. Gordon Bell, and Allen C. Newell. *Computer Structures: Principles and Examples*. McGraw-Hill, Inc., New York, NY, USA, 1982.

[119] Hartej Singh, Guangming Lu, Eliseu Filho, Rafael Maestre, Ming-Hau Lee, Fadi Kurdahi, and Nader Bagherzadeh. MorphoSys: case study of a reconfigurable computing system targeting multimedia applications. In *Proceedings of the Conference on Design Automation*, pages 573–578, 2000.

[120] Software/hardware generation for dsp algorithms. ttp://www.spiral.net.

[121] Yonghong Song and Zhiyuan Li. New tiling techniques to improve cache temporal locality. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 215 – 228, 1999.

[122] SPEC. Spec CPU 2000 benchmark specifications. SPEC2000 Benchmark Release, 2000.

[123] ST Microelectronics website. www.st.com.

[124] http://www-vlsi.stanford.edu/ee272/proj99/babyviterbi/verilogcode.html.

[125] Sun UltraSparc-T1. http://www.sun.com/processors/UltraSPARC-T1/.

[126] Steve Swanson, Andrew Putnam, Martha Mercaldi, Ken Michelson, Andrew Petersen, Andrew Schwerin, Mark Oskin, and Susan J. Eggers. Area-performance trade-offs in tiled dataflow architectures. In *Proceedings of the International Symposium on Computer Architecture*, pages 314 – 326, 2006.

[127] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. WaveScalar. In *Proceedings of the International Symposium on Microarchitecture*, 2003.

[128] Richard R. A. Syms, Eric M. Yeatman, Victor M. Bright, and George M. Whitesides. Surface tension-powered self-assembly of microstructures: The state-of-the-art. *Journal of Microelectromechanical Systems*, 12(4):387–416, 2003.

[129] Synopsys website. http://www.synopsys.com.

[130] Synopsys Corporation. *Power Product Reference Manual*. http://www.synopsys.com.

[131] Lin Tan, Brett Brotherton, and Timothy Sherwood. Bit-split string-matching engines for intrusion detection and prevention. *ACM Transactions on Architecture and Code Optimization*, 3(1):3–34, 2006.

[132] Texas Instruments website. www.ti.com.

[133] Steven N. Towle, Henning Braunisch, Chuan Hu, Richard D. Emery, , and Gilroy J. Vandentop. Bumpless build-up layer packaging. www.intel.com.

[134] Taiwan Semiconductor Manufacturing Company website. www.tsmc.com.

[135] TSMC 90nm technology platform. http://www.tsmc.com/download/english/a05_literature/90nm_Brochure.pdf.

[136] Francisco-Javier Veredas, Michael Scheppler, and Hans-Joerg Pfleiderer. Automated conversion from a LUT-based FPGA to a LUT-based MPGA with fast turnaround time. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 36–41, 2006.

[137] Reinhard von Hanxleden and Ken Kennedy. GIVE-N-TAKE - a balanced code placement framework. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 107–120, 1994.

[138] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring it all to software: Raw machines. *Computer*, 30(9):86–93, September 1997.

[139] Hangsheng Wang, Li-Shiuan Peh, and Sharad Malik. Power-driven design of router microarchitectures in on-chip networks. In *Proceedings of the International Symposium on Microarchitecture*, pages 105–115, 2003.

[140] Laung-Terng Wang, Cheng-Wen Wu, and Xiaoqing Wen. *VLSI Test Principles and Architectures: Design for Testability*. Morgan Kaufmann, San Francisco, CA, USA, 2006.

[141] John Weekley. Modeling total cost of ownership for semiconductor. *Design & Reuse Industry Articles*, 2004. www.us.design-reuse.com.

[142] Kent Wilken, Jack Liu, and Mark Heffernan. Optimal instruction scheduling using integer programming. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 121–133, 2000.

[143] Ron Wilson, Joe Gianelli, Chris Hamlin, Steve Leibson, Rich Tobias, Ken McElvain, Ivo Bolsen, and Raul Camposano. Panel position statements: Structured/platform ASIC apprentices: Which platform will survive your board room? In *Proceedings of the Conference on Design Automation*, 2005.

[144] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 30–44, 1991.

[145] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the International Symposium on Computer Architecture*, pages 24–36, 1995.

[146] Kun-Cheng Wu and Yu-Wen Tsai. Invited paper: Structured ASIC, evolution or revolution? In *Proceedings of the International Symposium on Physical Design*, 2004.

[147] Xilinx website. www.xilinx.com.

[148] Xiaorong Xiong, Yael Hanein, Jiadong Fang, Weihua Wang, Daniel T. Schwartz, and Karl F. Böhringer. Controlled multibatch self-assembly of microdevices. *Journal of Microelectromechanical Systems*, 12(2):117–127, April 2003.

[149] Tao Yang and Apostolos Gerasoulis. PYRROS: static task scheduling and code generation for message passing multiprocessors. In *Proceedings of the International Conference on Supercomputing*, pages 428–437, 1992.

[150] Zao Yang, K.-T. Cheng, and K.L. Tai. A new bare die test methodology. In *Proceedings of the VLSI Test Symposium*, page 290, 1999.

[151] Hsi-Jen J. Yeh and John S. Smith. Fluidic assembly for the integration of GaAs light-emitting diodes on Si substrates. *Photonics Technology Letters*, 6(6):706–708, 1994.

[152] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Umit Catalyurek. A scalable distributed parallel breadth-first search algorithm on BlueGene/L. In *Proceedings of the International Conference on Supercomputing*, page 25, 2005.

[153] Behrooz Zahiri. Structured ASICs: Opportunities and challenges. In *Proceedings of the International Conference on Computer Design*, page 404, 2003.

[154] Javier Zalamea, Josep Llosa, Eduard Ayguade, and Mateo Valero. Modulo scheduling with integrated register spilling for clustered VLIW architectures. In *Proceedings of the International Symposium on Microarchitecture*, pages 160–169, 2001.

[155] Paul S. Zuchowski, Christopher B. Reynolds, Richard J. Grupp, Shelly G. Davis, Brendan Cremen, and Bill Troxel. A hybrid ASIC and FPGA architecture. In *Proceedings of the International Conference on Computer-Aided Design*, pages 187–194, 2002.

[156] Zyvex Nanotechnology website. http://www.zyvex.com.

VITA

Martha Allen Kim, née Mercaldi, grew up in Massachusetts, where she earned her A.B. from Harvard University in Computer Science in 2002. She moved to Lugano, Switzerland where she studied Embedded Systems Design at the University of Lugano's Advanced Learning and Research Institute. After earning an M. Eng. in 2003 she returned to the United States to begin her doctorate at the University of Washington's department of Computer Science and Engineering. Her research interests are in computer architecture, particularly its boundaries: in the hardware/software interaction, as well as the architecture/circuit interaction. She earned her doctorate in December 2008.