Optimization-based Models for Pruning the Design-space for Processors

By

Nilay Vaish

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN-MADISON

2017

Date of final oral examination: 04/05/2017

The dissertation is approved by the following members of the Final Oral Committee:

David A. Wood, Professor, Computer Sciences
Michael C. Ferris, Professor, Computer Sciences
Mark D. Hill, Professor, Computer Sciences
Jeffrey T. Linderoth, Professor, Industrial and Systems Engineering
Michael M. Swift, Professor, Computer Sciences

© Copyright by Nilay Vaish 2017
All Rights Reserved

Abstract

The growing number of transistors on a microprocessor chip are being used towards putting a large number of general purpose processing cores and several specialized processing units on the chip. This inflation in the number of on-chip components makes the space of possible processor designs far bigger than what it used to be. Typically, software-based simulation models have been used for exploring the space of processor designs. These simulation models are typically very detailed and hence are about 4-5 orders of magnitude slower than actual designs. These simulation models have not improved their performance significantly enough with the increase in the number of on-chip components. While designers may continue to use these models to come up with new designs, this would mean exploring smaller and smaller portions of the design space as we increase the number of on-chip components. Therefore, we need models that can help explore the design space in significantly less time. Designers would use such models to compute potential candidates for further exploration with software-based simulation models.

This dissertation focuses on mathematical models that can be explored using discrete or continuous optimization algorithms. We develop models for two different design problems related to manycore processors. The first design problem we tackle is that of designing a cache hierarchy. Given a set of resource and performance constraints, we develop a model for computing the performance of a given cache hierarchy for multiple objectives. We model the space of possible cache hierarchies as a discrete multi-dimensional space. Then we provide a dynamic programming and multi-dimensional divide and conquer based algorithm that solves the model and computes Pareto-optimal and nearby cache hierarchies. Simulation-based experiments show that cache hierarchies obtained through our proposed algorithm perform significantly better than those obtained using previously proposed continuous model.

The second design problem we tackle is that of distributing resources in an on-chip network. We first propose integer linear programs that maximize the bandwidth available for communi-

cation while distributing memory controllers and network resources like link widths and virtual channels across the on-chip network. After that, we propose a different set of integer programs that distribute these resources depending the expected utilization of the network. These new models strive for obtaining a good balance between latency and bandwidth performance of the on-chip network.

More broadly, this dissertation provides more examples for application of optimization-based models (both continuous and discrete) in problems related to design of processors. It also shows that such models can reduce the design space exploration time.

Acknowledgements

I wish to express gratitude to my thesis advisor, Prof. David Wood. He is an excellent computer architect and is really good at presenting and explaining research work in a lucid fashion. He gave me significant amount of freedom in choosing research topics and was very patient with me. I am thankful for all the time and effort he devoted in helping me complete my research work.

I am also very grateful to my dissertation committee, Prof. Michael Ferris, Prof. Mark Hill, Prof. Jeffrey Linderoth and Prof. Mike Swift. Prof. Ferris helped me in solving one of the optimization models presented in this thesis. He also introduced me to the area of derivative-free optimization and simulation-based optimization. I learnt a lot from Prof. Mark Hill and Prof. Mike Swift during the Multifacet meetings. Prof. Hill is brilliant at expressing things in concise and terse fashion. I find him witty as well. I think Prof. Swift is one of the most well-read persons I have ever met. On the top of it, he has a very good recall ability and many times pointed out valuable research papers. I learned a lot from Prof. Linderoth's course on techniques and tools for modeling and optimization. In fact, my work on designing cache hierarchies started as a project for the course. I would like to thank all the members of the committee.

I would like to thank all the people I interacted with at UW-Madison. I had many remarkable friends and colleagues here. I also had the opportunity of working at AMD and at Google. I am thankful to both the companies and the people I worked with.

My parents and my sisters have always been my best supporters. Thank you for all the love and care.

Contents

1 Introduction **1**

1.1 Research Questions Examined & Contributions 4

1.1.1 Exploring the Design Space of Cache Hierarchies. 4

1.1.2 How to place and distribute resources in On-chip Networks with bandwidth objective. 5

1.1.3 How to place and distribute resources in On-chip Networks with latency objective. 7

1.2 Overall Significance of the Thesis 8

1.3 Organization 8

2 Overview of Theoretical Techniques **10**

2.1 Dynamic Programming 10

2.1.1 A Different View of Dynamic Programming 11

2.1.2 Limitations of Dynamic Programming 12

2.2 Multi-Dimensional Divide-And-Conquer 13

2.3 Mathematical Optimization 14

2.3.1 Convexity 14

2.3.2 Types of Optimization Problems 16

2.3.3 Solving Optimization Problems 17

2.3.4 Challenges in Using Mathematical Optimization 17

3 Designing Cache Hierarchy **19**

3.1 Problem Definition and Proposed Solution 21

3.1.1 Structural Properties of an Optimal Memory Hierarchy 22

3.1.2 Proposed Algorithm 23

3.1.3 Example 25

3.1.4 Other Possible Approaches 26

3.2 Representing Workload Behavior 29

3.2.1 Shared Cache Behavior 30

3.3 Pruning Individual Cache Designs 30

3.3.1 Algorithmic Pruning of the Design Space 31

3.3.2 Further Pruning After Discretization 32

3.4 Computing Maximal Hierarchies 33

3.4.1 Comparing Two Hierarchies 34

3.4.2 Computing Expected Values for Private Levels 34

3.4.3 Computing Expected Values for Shared Levels 35

3.4.4 Estimating Shared Cache Behavior 37

3.4.5 Estimating Network Performance 39

3.4.6 Estimating Physical Memory Performance 43

3.5 Proof of Pareto-Optimality 44

3.5.1 Statement of the Problem 44

3.5.2 Proof 45

3.5.3 Why Work with Expected Values 46

3.6 Extending the Method 46

3.6.1 Pipelined Out-of-order Cores 46

3.6.2 Exclusive caches . . . 47
3.6.3 Cache replacement policies . . . 47
3.7 Comparison With a Continuous Model . . . 48
3.7.1 Variables . . . 48
3.7.2 Constraints . . . 49
3.7.3 Objective Function . . . 49
3.7.4 Comparison With the Dynamic Programming-based Algorithm . . . 49
3.8 How Effective is the Method . . . 51
3.8.1 Experimental Results . . . 53
3.8.2 Comparison With Prior Work . . . 56
3.9 Lessons Learnt . . . 57
3.10 Related Work . . . 58
3.11 Conclusion . . . 59

4 Resource Placement and Distribution **60**

4.1 Placement of Memory Controllers . . . 61
4.1.1 Assumptions . . . 61
4.1.2 Notation Used in the Model . . . 62
4.1.3 Description of the Model . . . 62
4.1.4 Solving the Model . . . 64
4.1.5 Evaluation of the Model . . . 64
4.1.6 Why `mc-opt` Performs Better . . . 67
4.1.7 Effect of Read and Write Traffic Ratio on Controller Placement . . . 68
4.2 Resource Allocation in On-chip Network . . . 69
4.2.1 Assumptions Made in the Model . . . 71
4.2.2 Notation Used in the Model . . . 71

4.2.3 Description of the Model . . . 72
4.2.4 Design Obtained from the Model . . . 73
4.2.5 Evaluation of the Design . . . 75
4.3 On-chip Network Design Combined With Placement of Controllers . . . 79
4.3.1 Analysis of the Model . . . 81
4.3.2 Solving the Model . . . 82
4.3.3 Linearized Model . . . 83
4.3.4 Optimal Design . . . 84
4.3.5 Sensitivity of the Optimal Design vis-à-vis the Objective Function . . . 85
4.3.6 Evaluation of the Designs . . . 86
4.3.7 Analysis of the Designs . . . 89
4.4 Why Use Mathematical Optimization . . . 90
4.4.1 Scalability . . . 90
4.4.2 Theoretical Bounds on Performance . . . 91
4.4.3 Flexibility . . . 91
4.5 Related Work . . . 92
4.6 Future Work and Extensions . . . 93
4.7 Conclusion . . . 94

5 Latency-optimized Models for On-chip Resource Distribution 95

5.1 Introduction . . . 95
5.1.1 Our Contributions . . . 96
5.1.2 Organization of the chapter . . . 97
5.2 Overview of Previous Work . . . 97
5.3 Min-Max Latency Optimization Model . . . 100
5.3.1 Notation Used in the Model . . . 100

5.3.2 Description of the Model . . . 102
5.3.3 Computational Results with the Non-linear Model . . . 103
5.3.4 Linearizing the Model . . . 104
5.3.5 Computational Results with Linearized Model . . . 108
5.3.6 Synthetic Simulation Results . . . 108
5.3.7 Discussion on the model and the designs . . . 109
5.4 Average Latency Optimization Model . . . 112
5.4.1 Nonlinear Model . . . 113
5.4.2 Linearized Model . . . 113
5.4.3 Discussion about the designs obtained . . . 117
5.5 Related Work . . . 118
5.6 Conclusion . . . 119

6 Conclusion **120**

6.1 Limitations of Our Work . . . 121
6.2 Future Work . . . 122

List of Figures

1.1 Graphs for improvement in Single-threaded CPU Performance for SPEC CPU Benchmark applications. The data was obtained from SPEC [2]. It was processed using scripts provided by Jeff Pershing [78]. The line segment going across the graphs shows that the rate of performance improvement decreased after about 2005. 2

2.1 A set of point in $\mathbb{R}^2$. Encircled points are maximal. 13

2.2 Example for Convex Underestimation. $\mathcal{NCP}$ is non-convex function and $\mathcal{CP}$ is its convex under-estimate. Suppose some algorithm outputs the local minimum (shown in red) for $\mathcal{NCP}$. The minimum for $\mathcal{CP}$ can be used to compute the maximum possible gap between the *unknown* global minimum for $\mathcal{NCP}$ and the known local minimum. 16

3.1 Miss Rate vs Cache Size for SPEC CPU2006 applications. All caches are 8-way set associative. 28

3.2 Graph for Dynamic Power vs $\sqrt{size}\times$ Bandwidth for different cache designs, represented by green dots, at 32nm process technology. The data was obtained using Cacti. 29

3.3 Results from synthetic simulation of the on-chip network. The graph shows the average queueing delay for different amounts of load (ρ) generated by the input sources. The graph also shows the theoretically predicted queueing delay for the same load using the *Pollaczek-Khinchin formula*. 42
3.4 Continuous Optimization Model for Hierarchy Design 50
3.5 Improvement in IPC for the best hierarchy computed using discrete model as percentage of the IPC for the best hierarchy computed using continuous model. The horizontal lines show the average improvement in IPC 52
3.6 Multicore Processor Layout 53
3.7 Graphs for homogeneous scenarios created from SPEC CPU2006 applications. . 55
3.8 Graphs for heterogeneous scenarios created from SPEC CPU2006 applications. . 56

4.1 MILP for Placing Memory Controllers 63
4.2 Different Controller Placements for an 8×8 Mesh/Torus Network. Tiles with black-colored sphere represent a memory port co-located with a core. 64
4.3 Average Latency versus Injection Rate (Uniform Random Traffic) 65
4.4 Weighted speedup for different application combinations and controller placements normalized to `row 0-7`. 67
4.5 Average Latency versus Injection Rate (Uniform Random Traffic). The figure on the right is for traffic with read and write requests being equi-probable. The figure on the left is for traffic with read requests twice as probable as write requests. Both figures show optimal designs obtained by solving the model. $opt-1:1$ is for the setting $R=1$ and $opt-2:1$ is for the setting $R=2$. 68

4.6 Average Latency versus Injection Rate (Uniform Random Traffic). The figure on the right is for traffic with read and write requests being equi-probable. The figure on the left is for traffic with read requests ten times as probable as write requests. Both figures show optimal designs obtained by solving the model. $opt-1:1$ is for the setting $R=1$ and $opt-10:1$ is for the setting $R=10$. . . 69
4.7 Traffic Distribution in a Mesh Network. 70
4.8 MILP for Distributing Network Resources 72
4.9 `net-opt` Network Design 74
4.10 Virtual Channel to Traffic Distribution. Each cell represents a router. Its color represents the ratio of virtual channels assigned to the router and the traffic that goes through that router. 75
4.11 Different Network Designs. Each box represents a router. Shaded routers are big. Wider links have been shown wider. 76
4.12 Average Latency vs Request Injection Rate for different request patterns 77
4.13 Sustained peak bandwidth as a function of the link resources used. 78
4.14 Weighted Speedup for different application combinations and network designs normalized to `base`. 79
4.15 Non-linear Program for the Combined Problem 80
4.16 `com-opt` design for the Combined Problem 82
4.17 Distribution of Memory Controllers, Buffers and Link Widths for `diagonal`. . 85
4.18 Graph for Average Flit Latency vs Request Injection Rate for synthetically generated uniform random traffic. 87
4.19 Graphs from experimental evaluation of designs for the combined problem. . . . 88
4.20 Traffic Distribution for the `diagonal` Design 89
4.21 Traffic Distribution for the `com-opt` Design 90

5.1 Processor Layout (not to scale). On the left is an $n \times n$-tiled processor. The tiles are connected using a mesh/torus network. The figure on the right shows a single tile. The caches are kept coherent using a directory-based coherence protocol. The memory controller is optional. Routers for different tiles may have different amounts of resources. The links in between tiles can be wide or narrow. 96

5.2 Non-linear Program for the combined problem in Chapter 4 98

5.3 Designs evaluated for the combined problem in Chapter 4. 99

5.4 *MinMax* Optimization Model for Latency 101

5.5 Graph showing the queueing delay of an $M/D/1$ model as function of the arrival rate. The service rate μ was assumed to be 1. Also shown are piece-wise linear approximations. 105

5.6 Linearized *MinMax* Optimization Model for Latency 107

5.7 Designs obtained for *Min-Max* Latency Optimization Model for different values of ρ. 109

5.8 Results from synthetic simulation of the designs obtained from the *Min-Max Latency Model*. The graph on the left shows maximum latency for different designs and the entire range of ρ, the node injection rate. The graph on the right shows the same data for a limited range of ρ. 110

5.9 Results from synthetic simulation of the designs obtained from the *Min-Max Latency Model*. The graph shows maximum load that a link has to service for different designs and different values of ρ, the node injection rate. 111

5.10 Results from synthetic simulation of the designs obtained from the *Min-Max Latency Model*. The graph shows the queueing delay observed for the link with maximum load for different designs and different values of λ, the link's traffic arrival rate. 112

5.11 *Average Latency* Optimization Model 114

5.12 Linearized *Average Latency* Optimization Model 115

5.13 Designs obtained for the Linearized *Average* Latency Optimization Model for different values of ρ. 116

5.14 Results from synthetic simulation of the designs obtained from the *Average Latency Model*. The graph on the left shows the average latency for different designs and for the entire range of ρ, the node injection rate. The graph on the right shows the same data for a smaller range of ρ. Note that in the graphs the following pairs overlap each other: (`diamond`, `average-0.010`) and (`com-opt`, `average-0.008`). 116

List of Tables

3.1 Memory Hierarchies in Commercial Products 20
3.2 Miss Ratio and Latency used in the example. 25
3.3 Miss Ratio and Average Latency for two-level Pareto-optimal designs. 26
3.4 Parameter Values used for solving the Continuous Model 51
3.5 Simulation Parameters 53

4.1 Processor and Cache Simulation Parameters 66

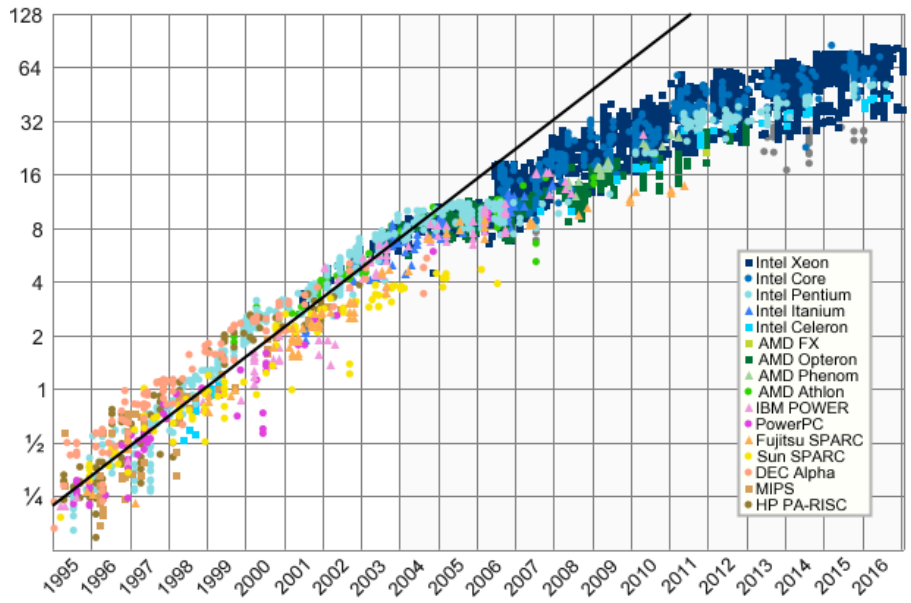
5.1 Value of the objective function for the optimal designs for the Linearized Min-Max Latency Model. 109
5.2 Average Latency for Different Designs on the Linearized Average Latency Model. Integrality gap for the model solution is shown in parentheses. ∞ means that a design cannot sustain that value of ρ. 116

Chapter 1

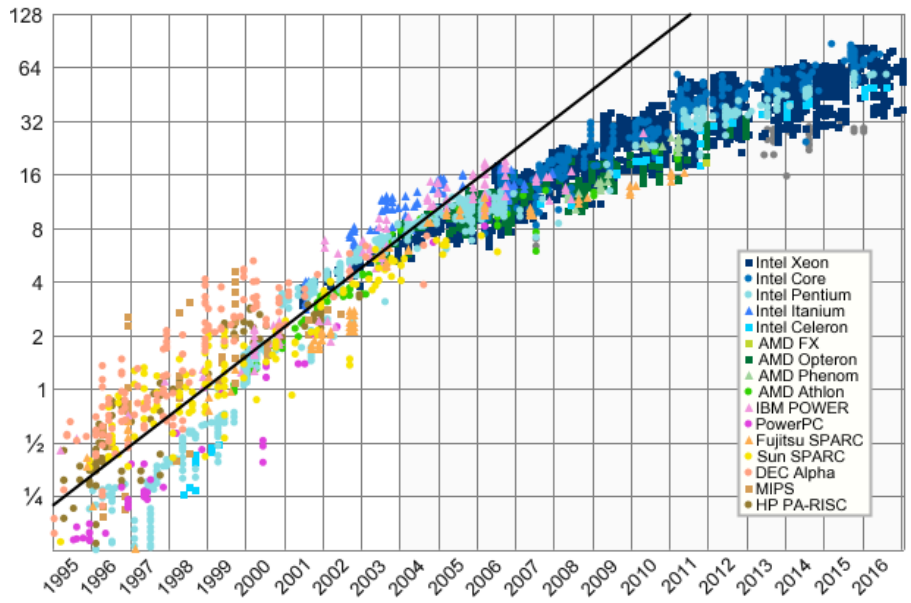
Introduction

The Information Technology revolution has brought about a sea change in the way we create, synthesize, dissect and disseminate information. Microprocessors are at the heart of this revolution as their performance is a major factor in deciding the rate at which these steps can be carried out. These microprocessors can be abstracted as integrated circuits made from transistors. Transistors were invented in 1947 and integrated circuits in 1958. Initially integrated circuits with only a few transistors (< 10) could be manufactured. But scientific advancement over many years allowed us to scale down the size of the transistors and integrate more and more of them cheaply and reliably. This scaling down of the transistors, predicted by Moore [71], came to known as *Moore's law*.

Since late 1960s, computer architects have focused on designing processors using integrated circuits. Over years, they developed various hardware and software techniques for improving performance of these microprocessors. These techniques, coupled with Moore's law, resulted in tremendous improvement in performance of microprocessors, especially those that executed one task at a time (single-threaded). While Moore's law is likely to continue for several more generations [52], the single-threaded performance of the processors started tapering off towards the beginning of the 21^{st} century. This tapering can observed from the data presented in Fig-



(a) Single-Threaded Integer Performance. Based on adjusted SPECint results.



(b) Single-Threaded Floating-Point Performance. Based on adjusted SPECfp results

Figure 1.1: Graphs for improvement in Single-threaded CPU Performance for SPEC CPU Benchmark applications. The data was obtained from SPEC [2]. It was processed using scripts provided by Jeff Pershing [78]. The line segment going across the graphs shows that the rate of performance improvement decreased after about 2005.

ure 1.1. The figure shows graphs for the single threaded integer and floating-point performance for a number of processors from the past and present. The data for these graphs was obtained from SPEC [2]. This tapering in performance led to the introduction of multi-core processors as they allowed doubling the performance each generation without significant increase in power consumption [73, 14]. We have now moved towards designs with large number of cores (possibly with heterogeneity in the designs of different cores) and specialized computational units (for performing specific tasks, like processing images) [20]. Hereafter, we refer to these designs as manycore processors.

Increased number of components on a chip has led to combinatorial explosion in the design space. There are an enormous number of ways to allocate on-chip resources among cores, caches, accelerators, on-chip interconnect, and memory controllers. Typically the design space is explored using architectural simulators. Depending on the complexity of the design and the workload being simulated, a single simulation may require several seconds to weeks to run. Therefore, we can only explore about several thousand to a million different design configurations. But manycore design problems result in far more configurations. Prior work has had some success in using brute force search (solve the same design problem, but on a smaller scale), randomized search, genetic algorithms and regression models for solving manycore design problems. This thesis is an effort towards developing better tools for designing processors. We suggest algorithmic and modelling approaches that can help prune the set of possible designs, thus keeping the design time and costs under control. In this dissertation, we focused on optimization-based approaches only. In particular, we worked on two problems related to the design of the memory hierarchy and the on-chip network. These two problems and our solution approach are described next.

1.1 Research Questions Examined & Contributions

In this section, we summarize the specific research questions this dissertation tackles and the contributions made in the process:

1.1.1 Exploring the Design Space of Cache Hierarchies.

With the increase in the number of cores, the hierarchy of caches that feeds these cores needs to keep pace. A good cache hierarchy not only reduces the time required to access data, it also reduces the energy/power required for the accesses. Therefore, it is important to carefully decide the number of levels in the hierarchy and various parameters like the size, the associativity, the latency and the bandwidth of each level.

The design space for cache hierarchies has a huge number of designs (quantified in Chapter 3). Given the rate at which architectural simulation can be carried out, the design problem is not tractable when approached using simulation. Therefore, the design space first needs to be pruned using analytical techniques. Prior work has explored analytical models that model the design space using continuous functions while the actual design space is discrete. Computing the optimal solution using these continuous functional forms results in designs which are not physically realizable. Hence, the obtained design(s) need to be *rounded* to some discrete design. This process effectively does away with the guarantees on the performance of designs obtained. We are not aware of any work that shows that the process of rounding would retain the performance guarantees. Another limitation of continuous modeling is that it makes it harder to model certain important on-chip functionality like the on-chip network and shared caches. It also makes it harder to obtain designs that perform well along several different directions, and not just one. Finally, none of the previously proposed techniques cater to the diversity in the application space. The techniques typically assume a particular model of applications that would be executed on the many core processor.

Contribution 1: A Method for Pruning the Design Space of Cache Hierarchies. We initially thought of modeling the problem of designing cache hierarchies using continuous functions and using global optimization techniques for obtaining possibly good designs. But over the course of time, we realized that such an approach is not the way to go due to the limitations mentioned above. As we improved our understanding of the problem and the solution space, we decided to frame the problem as a multi-objective discrete optimization problem.

We show that the discrete model has two properties that enable efficient solution procedures. Using these structural properties, we propose a method that combines dynamic programming and multi-dimensional divide and conquer for solving the discrete model. Our method, under the modelling assumptions, computes hierarchies that are on or are close to the Pareto-optimal frontier. In practice, due to modeling errors, hierarchies that are off the computed Pareto-frontier may actually lie on the Pareto-frontier when detailed simulations are carried out. Through simulation-based experiments, we show that such hierarchies do not lie too far away from the computed Pareto-optimal frontier, thus providing evidence that our method can be used for pruning the design space. We further show that our method computes designs that perform better compared to the designs computed as solutions to a continuous model based on prior work.

1.1.2 How to place and distribute resources in On-chip Networks with bandwidth objective.

As mentioned before, the increased number of transistors are being used to put a larger number of processing cores on the chip. To keep these cores gainfully occupied, large amounts of data needs to moved between the on-chip caches and the off-chip memory. But the bandwidth between the on-chip and off-chip components has not been increasing proportionately. Because of this, the on-chip caches need to share the limited bandwidth as efficiently as possible. The controllers responsible for managing accesses to the off-chip memory, hereafter referred to as

memory controllers, thus need to be placed strategically. Different placements yield different latencies and bandwidth utilization. Thus coming up with an optimal strategy for placement of memory controllers is necessary for a good processor design.

On-chip communication between the processor cores themselves also becomes more important with more cores on the chip. This is because a typical use case of manycore processors is executing multi-threaded applications. Such applications share data amongst the threads, which is kept up-to-date by communicating over the on-chip network. Moreover, the shared resources on the chip, like memory controllers and shared caches, are accessed via the on-chip network. Therefore, even single-threaded applications need a well designed on-chip network. Allocating resources to network components depends on the traffic distribution(s) the network is being designed for. A good design would perform well for multiple different distributions.

These design problems described above are inter-related. The placement of the memory controllers plays an important role in the on-chip traffic distribution. Similarly, the on-chip network design plays a role in deciding how well the memory controller can utilize the available off-chip bandwidth. While prior research work has looked at this two problems independently, solving them together may potentially yield a better overall design.

Contribution 2: Bandwidth-focused Optimization Models for Resource Placement and Distribution in On-chip Networks. A major challenge in placing and distributing resources in on-chip network is efficiently searching through the enormous number (quantified later) of possible designs. Prior work has explored different techniques like *genetic algorithms*, *randomized search*, *extrapolation* and *divide and conquer* for navigating through the design space. These techniques may yield good designs, but they do not provide any evidence on how on how much better other feasible designs might be. This lack of evidence made us wonder whether the designs proposed in prior work are actually the best designs that we can obtain. We turned to *mathematical optimization* for gathering evidence on this point.

We formulated mathematical optimization based models that can help us in solving the

resource placement and distribution problems described above. In particular, we formulated two *mixed integer linear programs* and one *mixed integer non-linear program* for these problems. We linearized the mixed integer non-linear program to provide optimality guarantees. Via simulation-based experiments, we show that these models achieve better designs compared to the ones presented in prior work. We also show that optimization-based design approach can significantly cut down the time required for exploring the design space.

1.1.3 How to place and distribute resources in On-chip Networks with latency objective.

As described above, the optimization models we framed for distributing resources for on-chip networks are based on maximizing the bandwidth available for communication. Typically, maximizing the available communication bandwidth increases the latency of communication. But performance for many applications is not bound by the communication bandwidth provided by the on-chip network, but by the latency of communication. Therefore, we need models for exploring on-chip network designs that minimize communication latency.

Contribution 3: Latency-optimized Models for On-chip Network Design. We present two optimization models: one for minimizing the maximum latency of communication, and another for minimizing the average latency of communication. The designer is provided with a single parameter—traffic input rate—that they can set and explore the design space. Depending on the value of this parameter, the models choose designs that strike a balance between bandwidth utilization and communication latency.

We solve the models for tiled processors (shown in Figure 5.1) and present optimal designs for a range of traffic input rate. We think these designs are new and have not appeared in the literature before. We evaluate the designs using synthetic simulations. Our simulations show that these models work well across the entire range of the parameter value. This means irrespective

of the application space (bandwidth-sensitive or latency-sensitive), designers can use the same model to explore the design space and compute designs that are likely to perform well.

1.2 Overall Significance of the Thesis

Apart from providing improved solutions over prior research work, we think the thesis is significant because of the following reasons. First, this thesis provides more examples for application of optimization-based models (both continuous and discrete) in problems related to design of processors. As shown by simulation-based experiments, our models provide solutions that perform better than previously proposed solutions. This helps in building confidence that optimization-based models, whether continuous or discrete, can be useful in designing processors.

Second, our proposed models typically require from a few minutes to a few hours of computational time for exploring the design space. This, in general, is a major reduction over brute force, randomized or genetic algorithm-based search procedures. This reduction in exploration time brings down overall design time and more time can be devoted in detailed simulation of the candidate designs.

Third, even if these models fail to compute an optimal design, they can provide information to guide a designer towards good designs. The efficiency of the algorithms and solvers used to compute on these models may allow designers to solve problems of larger scale and complexity.

1.3 Organization

The rest of the thesis is organized as follows. Chapter 2 discusses theoretical concepts applied in rest of the chapters. It mainly goes over *continuous optimization*, *dynamic programming*, *multi-dimension divide and conquer* and *queueing theory*. The following three chapters provide details of three problems related to design of manycore processors that we tackled. Chapter 3 deals

with designing of cache hierarchies. Chapter 4 deals with distribution of resources in on-chip networks and placement of memory controllers. Chapter 5 deals with latency-optimized models for designing on-chip networks. We conclude our thesis in chapter 6.

Chapter 2

Overview of Theoretical Techniques

This chapter describes the theoretical techniques we use in solving architectural problems related to manycore processors. Section 2.1 describes *dynamic programming*, an algorithm design technique that is commonly used in solving combinatorial optimization problems. Section 2.2 describes *multi- dimensional divide-and-conquer*, an algorithmic technique for solving geometric problems in multi-dimensional spaces. Section 2.3 presents the idea behind *mathematical optimization*, some basic definitions and an overview on different types of problem classes in optimization.

While our description of these techniques is not very comprehensive, we hope the reader would find it enough for the purpose of perusing through this thesis. With the architectural examples that appear in later chapters, the potential these techniques have will become much more apparent.

2.1 Dynamic Programming

Dynamic programming [26] is a general technique for solving problems by combining solutions to subproblems of the same form. As opposed to the divide-and-conquer approach, the same

subproblems (or the smaller problems that these subproblems are composed of) arise more than once. Therefore, solutions to these smaller problems are stored to avoid computing them when needed again. The solutions to the subproblems can be stored since in a dynamic programming algorithm a discrete state is associated with each subproblem being solved. This state is used as a key for looking up the data structure used for storing the solutions. Typically a tabular form is used for storing the solutions. This led to the word "programming" being used to describe the method. The use of a tabular form can be used to convert a recursive dynamic programming algorithm to an iterative one by computing the solution associated with each cell of the table.

Cormen *et al.* [26] suggest that a problem necessarily needs to have two properties for a dynamic programming based solution to be applicable to the problem. These properties are:

- *Optimal substructure*: an optimal solution to the problem is composed of optimal solutions to subproblems.

- *Overlapping subproblems*: the same subproblems are generated again and again. This is in contrast with the divide-and-conquer approach where different subproblems are generated in each step.

2.1.1 A Different View of Dynamic Programming

In this section, we take a slightly different look at dynamic programming that might be more appropriate from some practitioners. We base this discussion on Powell's work [79]. In this view, dynamic programming is seen as a technique for solving optimization problems in which decisions are made over multiple stages. In each stage, a decision is made, some reward is given, and possibly new information about the system under consideration is provided. The aim of the problem is maximize the sum total of all the rewards received. More formally, we are interested

in solving the following problem:

$$\underset{\pi \in \Pi}{\text{maximize}} \quad \sum_{t=0}^{T} \psi(S_t, a_t)) \tag{2.1}$$

$$\text{subject to } S_{t+1} = \phi(S_t, a_t, W_{t+1}) \tag{2.2}$$

$$a_t = \pi(S_t) \tag{2.3}$$

Here S_t represents the state of the system at time t. ψ represents the time separable reward function. ϕ is the transition function. a_t is the decision made at time t using the policy π. Lastly, W_{t+1} represents the new information released after the decision a_t has been made. T represents the time horizon over which the problem is solved. Typically, such problems are solved by solving the recurrence:

$$V_t(S_t) = \underset{a_t}{max}(\psi(S_t, a_t) + V_{t+1}(S_{t+1})) \tag{2.4}$$

This recurrence is known as Bellman's equation. In words, the equation says that we should step backward in time to solve the optimization problem. We first evaluate $V_t(S_t) = \underset{a_t}{max} \ \psi(S_t, a_t)$ for $t = T$ and for all S_t. Then, we iterate over t and for each S_t, compute the value $V_t(S_t)$. We stop at $t = 0$.

2.1.2 Limitations of Dynamic Programming

The most important drawback of solving problems using dynamic programming is that the amount of computation required grows exponentially with the number of bits required to represent the state space, the action space and the information space. This makes the algorithm described for solving equation (2.4) difficult to work with in many situations. This explosion in the size of the problem is referred to as the *curse of dimensionality*, a phrase coined by Bellman.

Thus, only problems which can be modeled using small enough state, action and information space, can be solved using dynamic programming. For problems that do suffer from the

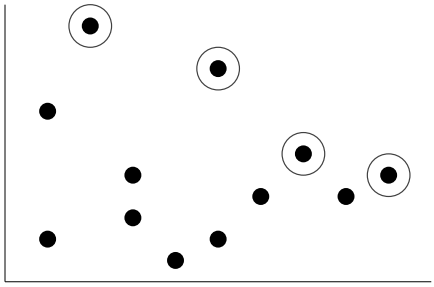


Figure 2.1: A set of point in $\mathbb{R}^2$. Encircled points are maximal.

curse, Powell [79] suggests stepping forward in time by approximating the function V_t instead of computing it exactly as was being done while stepping backward in time.

2.2 Multi-Dimensional Divide-And-Conquer

As is common in computer architecture, many problems involve optimizing a multi-variate cost function over a finite set. If a dynamic programming algorithm is used for solving such a problem, then more than one solution may have to be retained for the subproblems. This is because in a multi-dimensional space, points are only partially ordered, which means no single solution may dominate the rest. In such a case, all the points that are not dominated by other points of the set need to be retained. Such points are called *maxima*, *maximal*, or *Pareto-optimal* points of that set. An example illustration appears in Figure 2.1.

Bentley [16] gave a multi-dimensional divide and conquer algorithm for computing the maximal points of a given finite set. The algorithm solves a problem of N points in $\mathbb{R}^k$ by first recursively solving two problems of size $N/2$ in $\mathbb{R}^k$ and then combining their solutions by recursively solving a problem of at most N points in $\mathbb{R}^{k-1}$. The resulting algorithm has the time complexity $O(N \log^{k-1} N)$. Due to its exponential dependence on the dimensionality of the space, the algorithm works well only for small values of k.

2.3 Mathematical Optimization

Mathematical Optimization deals with the problem of making the best possible choice from a set of feasible choices. Choices are expressed as the different values of a set of *variables*. *Feasible* choices are specified by constraints on the values of these variables. A designated function that evaluates a particular criterion decides the best choice. Mathematical optimization helps us make the *best possible* choice efficiently using the power of calculus, logic and mathematics.

Formally, an optimization problem has the form [22] –

$$\begin{aligned} &minimize && f_0(x) \\ &subject\ to && f_i(x) \leq 0, \quad i = 1, \cdots, m \end{aligned}$$

Here $x = (x_1, \cdots, x_n)$ is the vector of *optimization variables*, the function $f_0 : \mathbb{R}^n \to \mathbb{R}$ is the *objective function*, the functions $f_i : \mathbb{R}^n \to \mathbb{R}$ are the *constraint functions*. A vector x^* is called *optimal* if it has the smallest objective function value among all vectors $z \in \mathbb{R}^n$ that satisfy the constraints i.e. $f_0(x^*) \leq f_0(z)$ for any z for which $f_i(z) \leq 0,\ i = 1, \cdots, m$.

2.3.1 Convexity

Certain mathematical concepts facilitate more efficient solution of these problems. We use notions related to convexity throughout this paper; interested readers may go through any standard text on mathematical optimization for a detailed discussion, e.g. [22]. The core concepts we use are:

- *Convex Set*: A set $C \subseteq \mathbb{R}^n$ is said to be convex if for any $x_1, x_2 \in C$ and any θ with $0 \leq \theta \leq 1$, we have $\theta x_1 + (1 - \theta)x_2 \in C$.

- *Convex Function*: A function $f : D \to \mathbb{R}$ is called convex if its **domain** $D \subset \mathbb{R}^n$ is a convex set and for all $x, y \in D$, and any θ with $0 \leq \theta \leq 1$, we have $f(\theta x + (1 - \theta)y) \leq$

$\theta f(x) + (1-\theta) f(y)$.

- An optimization problem is called convex if it is of the form

$$\begin{aligned} &minimize && f_0(x) \\ &subject\ to && f_i(x) \leq 0, \quad i = 1, \cdots, m \\ &&& a_i^T x = b_i, \quad i = 1, \cdots, p \end{aligned}$$

  where $f_0, \cdots, f_m$ are convex functions.

A point $x^* \in \mathbb{R}^n$ is locally optimal if it is feasible and its objective value is no worse than the objective value at all neighboring feasible points. It is globally optimal if the neighborhood is the whole space. For a convex optimization problem, any locally optimal solution is also globally optimal. This property makes solving convex problems relatively easy and efficient methods exist for solving them.

Non-convex optimization problems, on the other hand, do not have efficient solution procedures in general. Methods exist that can provide solutions that are locally optimal but such a solution might be very far from the global optimum. In particular cases, it is possible to use convex problems to approximate or even exactly solve non-convex problems. For example, suppose we need to a solve a non-convex minimization problem $\mathcal{NCP}$. One may construct a convex problem, $\mathcal{CP}$, such that the objective function value for $\mathcal{CP}$ is at most the objective function value for $\mathcal{NCP}$ at all feasible points i.e. $\mathcal{CP}$ underestimates $\mathcal{NCP}$. Then, the optimal objective function value for $\mathcal{CP}$ gives us a lower bound on the optimal objective function value for $\mathcal{NCP}$. We show this pictorially in Figure 2.2. For integer linear programs, such bounds can be obtained by removing the integrality constraints. The bound can be used to compare the quality of any given feasible solution with the *unknown* optimal solution.

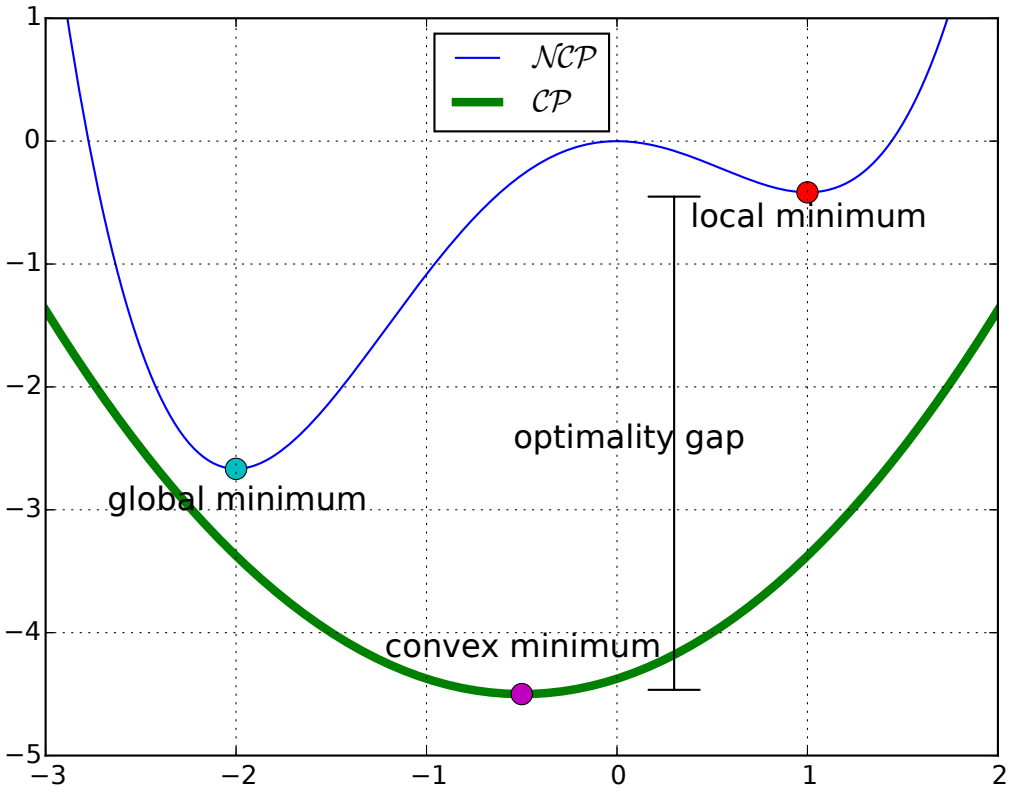


Figure 2.2: Example for Convex Underestimation. $\mathcal{NCP}$ is non-convex function and $\mathcal{CP}$ is its convex under-estimate. Suppose some algorithm outputs the local minimum (shown in red) for $\mathcal{NCP}$. The minimum for $\mathcal{CP}$ can be used to compute the maximum possible gap between the *unknown* global minimum for $\mathcal{NCP}$ and the known local minimum.

2.3.2 Types of Optimization Problems

Optimization problems can be divided in to several different classes depending on the particular forms of the objective and the constraint functions. In our work, we make use of the following classes:

Mixed Integer Linear Program (MILP). In an MILP, the objective and the constraint functions are linear. Also, one or more variables in the program take integer-only values.

Mixed Integer Non-Linear Program (MINLP). This class includes problems that have at least one function (objective or constraint) that is not linear. Also, one or more variables take integer-only values.

2.3.3 Solving Optimization Problems

Theoretically, problems in these classes are known to be NP-hard [77]. But over the years, researchers have developed algorithms and tools that can solve these problems efficiently in many instances. We briefly describe the tools used in our work. General Algebraic Modeling System (GAMS) [1] is a language for representing optimization models. The representation is compiled using the GAMS compiler, which is then passed on to a solver for obtaining an optimal solution. A solver is a tool that implements algorithms for solving optimization models. For example, Gurobi [42] is a solver for linear and quadratic optimization problems, and Baron [96] is a solver for optimization problems defined with nonlinear functions f_i. While a solver may not always be successful in finding an optimal solution for the model due to NP-hardness, the above solvers give solution guarantees relating the values found to the best possible ones.

2.3.4 Challenges in Using Mathematical Optimization

Despite the benefits, mathematical optimization may not be applicable to a large set of computer architecture problems. Optimization requires that the problem be represented using algebraic functions, which is not always possible. Many design problems require detailed models which may not be suitable for optimization techniques.

Secondly, optimization-based models may also face computational challenges in exploring the design space. MILPs are known to be NP-hard [77], so it is not always possible to obtain an optimal solution in an acceptable time frame, limiting its use on time constrained operational problems. No practical algorithms are known for solving problems involving non-linear, and/or non-convex functions to global optimality [22]. Available solvers use different heuristics which might result in high computational effort or sub-optimal solutions.

Moreover, many problems in computer architecture require performing a trade-off amongst multiple objectives. For example, one may need to design an on-chip network where both the

network bandwidth and the network latency need to be optimized. For such problems, one may have to solve multiple instances of the optimization model to trace out a Pareto frontier. This is only possible if solving individual instances is fast enough. Otherwise, one may have to weight the different objectives based on expert opinion.

Lastly, a given solution is only optimal for the specific instance of the model. Changes to inputs or the model's structure may make the solution suboptimal or infeasible. One may have to use techniques like stochastic optimization if only probability distributions are known for the input parameters.

Chapter 3

Designing Cache Hierarchy

Designing a cache hierarchy for a multicore processor requires deciding the number of levels in the hierarchy and for each level, the parameters like size and associativity. Given the diversity in applications and in cache designs, it is unlikely that a particular hierarchy works well for all the applications. Therefore, designers search for designs that work well in general.

In Table 3.1, we show the memory hierarchies available in different manycore processors available in the market. Product briefs and technical papers for these processors claim that these processors are for high performance computing. There is significant disparity amongst these hierarchies. This raises some questions. Why would designers arrive at significantly different hierarchies? Are these hierarchies optimal (or close to optimal)? As we show later in the chapter, the design space of cache hierarchies is large. But designers rely on *accurate* but considerably slow software simulation models for making design decisions. These models simulate processor designs at rates about $5 - 7$ orders of magnitude lower compared to native execution (from experience with the gem5 simulator). Hence, designers only partly explore the design space. Depending on the portion of space explored, different designers come up with different designs.

Similarly, people who deploy these processors for their tasks, are faced with plethora of choices. This abundance makes it hard for them to make the optimal (or near optimal) choice of

Processor	Cores	Level 1		Level 2	Level 3	Memory Controllers
		Instruction	Data			
Cavium's ThunderX [23]	48	78KB	32KB	16 MB shared	-	6 DDR3/4
Cavium's ThunderX2 [24]	54	64KB	40KB	32 MB shared	-	6 DDR3/4
Intel's Xeon E7-8890 v4	24	32KB	32KB	256KB	60MB	4 DDR4 channels
Intel's Xeon Phi [90]	72	32KB	32KB	1 MB unified shared by two cores	16GB (HBM)	2 DDR4, 6 channels
Mellanox's Tile-Gx72 [68]	72	32KB	32KB	256KB per core	18MB	4 DDR3
Oracle's M7 [88, 8]	32	16KB	16KB	256KB I$, shared by four cores 256KB D$ shared by two cores	8MB shared by four cores	4 DDR4, 16 channels

Table 3.1: Memory Hierarchies in Commercial Products

processor to use.

The above discussion suggests need for an algorithm that can prune the design space of cache hierarchies. There have been prior attempts at exploring this design space using mathematical models [54, 80, 76, 95, 99, 29]. To keep the models and the search methods simple, prior techniques [54, 80, 76, 95] represent the design space using continuous variables and search the best design for a single objective (either access time or energy). They also make simplifying assumptions and omit some of the design parameters so that the resulting models involve functions with differentiable algebraic forms. These functions are then differentiated and the maxima / minima are obtained without using any sophisticated algorithm.

Our Contribution. We model the problem of pruning design space for memory hierarchies in a discrete fashion. We show that the discrete model has two properties that enable efficient solution procedures. Using these structural properties, we propose a method combines dynamic programming and multi-dimensional divide and conquer. Our method:

- optimizes for multiple different objectives and obtain hierarchies that are on or are close to the Pareto-optimal frontier.

- incorporates cache design tools like Cacti [75] directly in the process of designing the hierarchy. This ensures that designs obtained are physically realizable.
- uses prior work on mathematical models for shared caches for designing shared caches. It uses queueing theory to tie up the design of shared caches and on-chip network.

In section 3.5, we justify our method with a formal proof that hierarchies computed using the method are optimal under the assumptions of our model. In practice, due to modeling errors, hierarchies that are off the computed Pareto-frontier may actually lie on the Pareto-frontier when detailed simulations are carried out. In section 3.8, we show that such hierarchies do not lie too far away from the computed Pareto-optimal frontier, thus providing evidence that our method can be used for pruning the design space.

In section 3.1.4, we present arguments on why continuous modeling of the design space prevents modeling of important design requirements and makes it hard to optimize for multiple different objectives. In section 3.7, we present a continuous model for designing cache hierarchy for single core processors. We show that our method computes better designs than the designs computed as solutions to the continuous model. In section 3.8, we show that our proposed method performs better than the continuous modelling-based method proposed by Sun *et al.* [95].

3.1 Problem Definition and Proposed Solution

Define a *feasible* memory hierarchy as one which satisfies the specified performance and budget constraints. A *Pareto-optimal* cache hierarchy is a feasible hierarchy not dominated by any other hierarchy on all the metrics being considered. We seek an algorithm for pruning the design space for the following problem: given a set of cache designs, a set of applications and a many-core processor's performance and budget constraints, design a memory hierarchy that is Pareto-optimal with respect to several performance metrics: *access latency*, *dynamic energy*, *leakage*

power and *chip area*. The problem may have multiple solutions since a multi-dimension space can have points that are not comparable to each other.

Why Multi-Objective. Trading-off resources used and performance obtained is an important part of the design process. Access to the set of Pareto-optimal hierarchies would help in making better decisions. Therefore, instead of just optimizing for a particular metric, we need a method that can handle multiple metrics.

Assumptions Made. Before we start describing our solution method for the problem described above, we list the assumptions that we make. In section 3.6 we discuss how we may relax these assumptions. We assume that the processing core is un-pipelined and executes instructions in program order. The memory hierarchy is assumed to be *inclusive* and the replacement policy for caches is assumed to be *random*.

3.1.1 Structural Properties of an Optimal Memory Hierarchy

Had we been designing a single-level memory hierarchy, we could have simulated all the feasible cache designs one-by-one and figured out which ones perform the best on the given metrics. We could have then chosen one of these best ones using some other design criteria. But with multiple levels in the hierarchy, the design space becomes much bigger and one cannot explore the space just through simulation-based model. Hence one needs a more tractable model.

We impose a discrete model (explained later) on the design space. We noticed that solutions under the discrete model exhibit two structural properties which can be used to explore the design space much more efficiently. We discuss these two properties next. Let's work with a simplified version of the problem to demonstrate these properties. Assume we are trying to design the hierarchy which satisfies all requests and has the least average access time.

- **Optimal Substructure**: Suppose the optimal hierarchy H_n has n levels: $(d_1, d_2, ..., d_n)$ where each d_i is some cache design. Consider the hierarchy H_{n-1}: $(d_1, d_2, ..., d_{n-1})$ i.e.

H_{n-1} is composed of the first $n-1$ designs from H_n. We claim that it is not possible that there is some hierarchy H' such that H' has better average access time and lower miss ratio than H_{n-1}. Assume otherwise and let H'_m be $(d'_1, d'_2, ..., d'_m)$ be such a hierarchy. Then the hierarchy H'_{m+1} given by: $(d'_1, d'_2, ..., d'_m, d_n)$ will have a better average access time compared to H_n and would also satisfy all requests. This is a contradiction since H_n was assumed to be the optimal hierarchy. Thus, the optimal hierarchy H_n is composed of hierarchies that are also optimal.

- **Overlapping Subproblems**: Again consider the problem of computing the hierarchy that has the least average access time and satisfies all memory requests. Assume we are exploring all the hierarchies with exactly n levels. Lots of these n-level hierarchies will share the first $n-1$ levels and only their n^{th} level would look different. Thus we may compute all the $(n-1)$-level hierarchies once and use them again and again while computing n-level hierarchies. These helps in saving computational time.

3.1.2 Proposed Algorithm

The properties we described above are essential elements for dynamic programming [26] to be applicable as a solution for a discrete-optimization problem. We next propose a dynamic programming-based approach for pruning the design space of memory hierarchies for manycore processors. We design the memory hierarchy one level at a time. At each level, we use a multi-dimensional divide-and-conquer approach to compute memory hierarchies that are *feasible* and are on the Pareto-optimal frontier for that level. Our method involves the following steps:

1. Choose a set $\mathbb{A}$ of applications and a set $\mathbb{S}$ of multi-programmed workloads from these applications. Each workload consists of one application for each core. Different cores may execute the same application, or different threads from a single application.

2. Collect data on miss ratios for caches with different sizes and associativities for all the

applications. The data can be represented algebraically like the $2 - to - \sqrt{2}$ rule [44] or as a table of values. This step is discussed in more detail in section 3.2.

3. Enumerate all possible individual cache designs. For example, Cacti [75] can be used to generate all possible cache designs for a given set of parameters like size and associativity. From these individual designs, select designs that are feasible and Pareto-optimal with respect to the chosen performance metrics. For this study, we use access time, dynamic energy, leakage power and on-chip area as metrics for cache performance. This step is discussed in more detail in section 3.3.

4. Design the private levels of the hierarchy by adding one level at a time. At each level, prune the design space by only keeping the hierarchies that are both feasible and Pareto-optimal till the current level. This step is discussed in more detail in section 3.4.

5. For the shared levels, analyze how the applications share the cache space, rates at which cache controllers process requests, and rates at which network requests and responses are sent. Use the analysis to estimate the values of different performance metrics for the shared levels of the hierarchy. Again select hierarchies that are both feasible and Pareto-optimal. This step is discussed in more detail in sections 3.4.4 and 3.4.5.

6. Stop adding new levels if at a particular level no feasible design exists or the performance of each feasible design is inferior to designs obtained at previous levels.

7. Lastly, account for latency and dynamic energy incurred in accessing the physical memory. This step is discussed in more detail in section 3.4.6.

Mapping the algorithm to a formal dynamic programming framework

In section 2.1.1, we described a formal framework which might be used to reason about the method of dynamic programming. We next describe how we can map quantities involved in

Cache Size	16KB	32KB	256KB	1024KB	Physical Memory
Miss Ratio	10%	8%	2%	1%	0%
Access Latency (Cycles)	1	2	10	20	100

Table 3.2: Miss Ratio and Latency used in the example.

the algorithm described above to the framework. In our algorithm, each level of the hierarchy constitutes a time step. S_t, the state of the system at time t, is the value of the performance metrics for a t-level hierarchy. ψ represents the function that weighs all the performance metrics and computes a reward. For our algorithm, ψ outputs a tuple of values instead of one single number since we are working with a multi-dimensional objective function. Π is the set of cache designs and π is an individual cache design. ϕ is the function for computing the new values of the performance metrics from the newly added level to the hierarchy.

3.1.3 Example

We next give an example illustrating how the algorithm proceeds. Assume we are designing a memory hierarchy for a single core processor. Further, we have a single application whose global cache miss ratios for different sizes of caches are as shown in Table 3.2. In the same table, we also show the latencies of the cache designs we can choose from.

At each level of the hierarchy, we can choose from any of the five designs presented in Table 3.2. This means we have 5 possible one-level hierarchies, 25 two-level and 125 three-level hierarchies. Note the exponential growth in the number of possible hierarchies as the number of levels in the hierarchy go up. We later quantify the size of the design space using cache designs from Cacti.

Consider the first level of the hierarchy. All five designs presented in Table 3.2 are Pareto-optimal since miss-ratio goes down with size, but the latency goes up. So, we retain all the five designs as possibilities.

Next consider memory hierarchies with two levels. After considering all possible two-level

Level 1	16KB	16KB	16B	32KB
Level 2	32B	256KB	1024KB	Physical Memory
Miss Ratio	8%	2%	1%	0%
Average Access Latency (Cycles)	1.2	2	3	10

Table 3.3: Miss Ratio and Average Latency for two-level Pareto-optimal designs.

combinations of the Pareto-optimal designs from Table 3.2, we arrive at two-level Pareto-optimal designs shown in Table 3.3. We can continue this process further computing Pareto-optimal designs with three levels and so on. We stop when in a given iteration no more Pareto-optimal memory hierarchies are found. In our example, the hierarchy with three levels of caching: (16KB, 256KB, 1024KB) followed by physical memory turns out to be the best memory hierarchy with zero miss ratio.

3.1.4 Other Possible Approaches

Next we argue against other methods we considered for solving the problem.

Exhaustive Simulation. As discussed later in sections 3.3 and 3.4, there are lots of choices for individual cache designs. The number of memory hierarchies is approximately n^l where n: number of cache designs, and l: number of levels. Even with about 100 choices for cache designs and 3-level hierarchies, it is computationally infeasible to simulate the million possibilities for memory hierarchies. It is better to first prune the design space to a manageable size and then use simulation or some other technique to make the final design decision.

Greedy Approach. Since a hierarchy has multiple levels, one may make design decisions for each level individually, without keeping the overall hierarchy in perspective. This *greedy* approach would yield a good design for a particular level, but unlikely to yield a good hierarchy design. For example, a small cache with low access latency and dynamic energy consumption may seem optimal when looking only at the first level of the hierarchy. But because of its high miss rate, the small cache may not be a good choice for the overall hierarchy design. In fact, this

choice of first level may lead to a situation in which the performance/resource constraints cannot be satisfied.

Scalarize The Objective. A designer may use a scalar objective function that combines all the metrics together. We believe significant experience is required to design such a function. Such function would change with changes in fabrication technology, processor architecture and application requirements. If such a function is available, then our solution would not be required. But it seems more likely that designers would fail to properly trade-off performance and cost [53] when using such a function.

Continuous Optimization We next discuss aspects of continuous optimization-based methods which may make them less favorable compared to a discrete optimization-based method.

- **Program Behavior.** Jacob *et al.* [54] assumed that a cache of size x has a hit rate of $P(x) = 1 - \frac{1}{(\frac{x}{\beta}+1)^{\alpha}}$. The form of $P(x)$ is similar to the $2 - to - \sqrt{2}$ rule which relates the cache miss rate (M) with the cache size in the following manner: $M = M_0 x^{-\alpha}$. This rule was examined in detail by Hartstein *et al* [44]. Sun *et al.* [95] use the $2 - to - \sqrt{2}$ rule in their work.

  This rule works well as an approximation. But Hartstein *et al.* show that the exponent α typically varies between 0.3 and 0.7. We show this variation for applications from the SPEC CPU2006 benchmark suite in Figure 3.1.

- **Energy / Power / Cost Modeling.** Jacob *et al.* assumed the access time for a cache to be fixed, independent of its size. They also assumed the cost of a cache, in dollars, is linearly dependent on its size. Similarly, Sun *et al.* assumed the access power of a cache is given by: $\rho\sqrt{size} \times bandwidth$. Here ρ is constant determined using Cacti [75].

  These approximations may act as good thumb rules. But such simple functional forms may not correctly represent the different ways in which caches can be organized. We obtained data from Cacti for dynamic power and bandwidth for many different cache designs. The

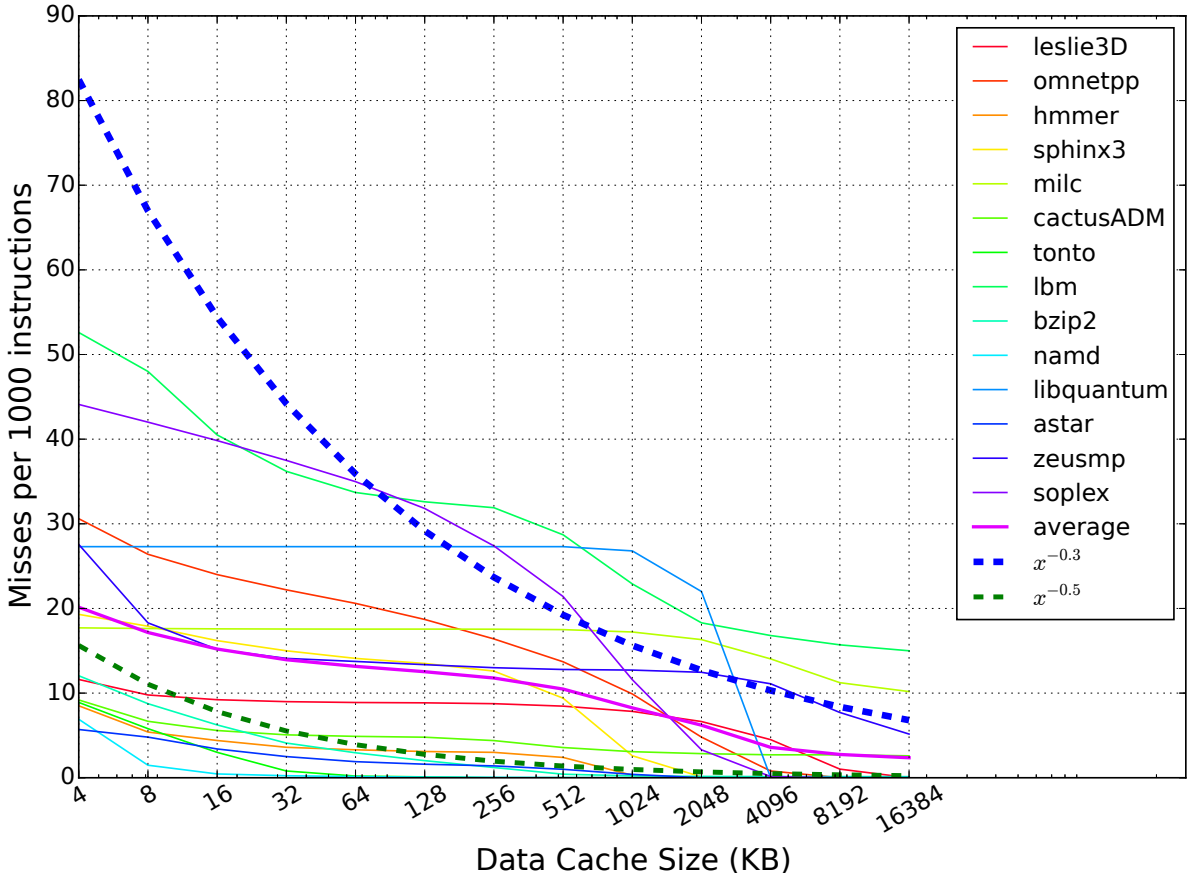


Figure 3.1: Miss Rate vs Cache Size for SPEC CPU2006 applications. All caches are 8-way set associative.

data has been plotted in Figure 3.2. As can be seen, there is no one value for ρ that can be used for the approximation suggested by Sun *et al.*

- **Physical Realization of Designs.** The solutions obtained with the continuous functional forms mentioned above are unlikely to be physically realizable. This is because the functions assume cache designs can be of arbitrary size. This is not true in reality and hence one will have to round off the values obtained for different levels.

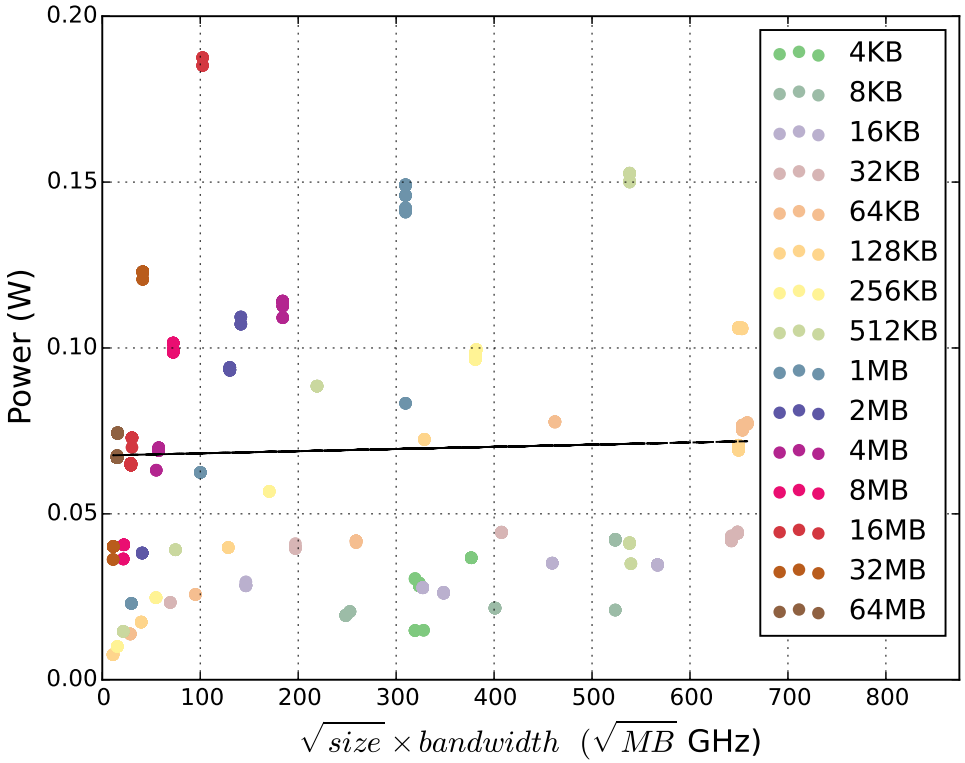


Figure 3.2: Graph for Dynamic Power vs $\sqrt{size}\times$ Bandwidth for different cache designs, represented by green dots, at 32nm process technology. The data was obtained using Cacti.

3.2 Representing Workload Behavior

For designing a cache hierarchy, we need to specify the behavior of different applications that the hierarchy needs to cater to. By behavior, we mean how the probability of a hit / miss in the cache varies with parameters like cache size and associativity. In this section, we describe the behavior representation used in our method.

Different applications have different cache behavior. In fact, even a single application may show multiple different behaviors over the period of its execution. A single algebraic function, like the 2-to-$\sqrt{2}$ rule [44], while good for approximation, is inadequate for capturing these different behaviors. Another problem with using such a rule is that it does not capture the notion of associativity which has a significant effect on the performance of a cache.

To handle both these issues, we collect cache behaviors of different applications using gem5 [19]. We can also treat different phases of an application as separate applications in their own right. For each application, the behavior of the application is collected for different cache sizes and associativities that can possibly be part of the cache hierarchy. Note that our method can also use algebraic functions like the 2-to-$\sqrt{2}$ rule.

3.2.1 Shared Cache Behavior

The application behavior obtained above suffices for the private levels of the hierarchy. Since we are designing the hierarchy for a manycore processor, we also need to obtain behavior of ensemble of applications. We generate a finite number of different scenarios, where each scenario consists of a subset of applications and the number of copies of the application that are executing on the processor. For example, scenarios for a 4-core processor might look like $\{A, A, B, C\}$, $\{A, B, B, C\}$, $\{A, B, C, C\}$ and so on. In section 3.4.4, we show how we use prior work and estimate the behavior of an ensemble of applications executing on a manycore processor with a shared cache hierarchy. Alternately, we can also collect statistics related to shared cache behavior of each scenario, as we did for individual applications.

3.3 Pruning Individual Cache Designs

In this section, we explain the method for generating C, the set of all single level cache designs from which designs for different levels of the cache hierarchy are sourced. While we source designs at each level from the same set, a designer may generate different sets for different levels of the cache hierarchy.

We use Cacti [75] as the cache design tool in our method. For a given set of parameters (size, associativity, number of banks, ports), Cacti first generates all possible tag and data array designs. Different tag and data array designs are obtained by varying the number of bitlines,

wordlines, sets per wordline, wire type, and few other parameters. The cross product of the tag and data array designs forms the set of cache designs for the given parameters. We observed that the number of possible cache designs runs into billions. Since we are interested in designing a hierarchy and each level in the hierarchy could be any of these cache designs, a huge number of possible hierarchies would make the problem intractable. Therefore, we first choose designs that are good and discard others.

3.3.1 Algorithmic Pruning of the Design Space

Consider the following four metrics for cache designs: access time, leakage power, area and dynamic energy. We looked at the values of these metrics for different designs generated by Cacti. It made us realize that most of the designs that Cacti generates for a given set of parameters, are actually worse than some design in the set. Thus there are few designs that are *maximal* with respect to the metrics mentioned. We eliminate the non-maximal designs since for each non-maximal design, there is some other design which is strictly better. While computing maximal designs, we only compare designs that have the same size and associativity. This because an application's cache behavior is largely dependent on these two parameters.

We compute the set of maximal designs using Bentley's [16] multi-dimensional divide and conquer algorithm for computing the maximal points in a given set. Given n points in a k-dimensional space, the algorithm computes all the maximal points in $O(n \log^{k-1} n)$ time. We first compute all the maximal data array designs, then we compute all the maximal tag array designs. Finally, we compute all the maximal cache designs. This approach is very effective in reducing the number of cache designs. It reduces the number of cache designs from 1.86×10^{12} to 1.11×10^{6}, which is about 0.000006% of the design space.

3.3.2 Further Pruning After Discretization

The number of maximal cache designs can be still large enough to make the rest of the method computationally prohibitive. To further prune the set of designs, we discretize the access latency in units of clock period of the processor, and the dynamic energy is rounded up to the nearest picoJoule.

Consider two individual cache designs A and B. We observed that even though both A and B are maximal, it is still possible that they are close to each other when considered as points in the space defined by the performance metrics. In such a case, we choose not to consider one of the two designs. Assume design A has a lower access latency compared to design B. Then, we eliminate design B if:

- B has a access latency which is same as A's when compared in terms of cycles of the underlying processor core.
- there is less than a picoJoule (pJ) of difference in dynamic energy consumed per access for the two designs. Typically several pJs of energy is consumed per access [75].

After applying these rules, we are left with only 1601 designs, which is small enough for our method. Note that we could not have heuristically pruned designs before the algorithmic pruning as the heuristic pruning approach has $O(N^2)$ complexity.

Alternately, we can also reduce the number of cache designs by using a clustering algorithm like k-means [43] or by applying rules that make use of relative distance in place of absolute distance. While different pruning methods may choose different set of cache designs, the pruning step itself does not affect the rest of the algorithm.

3.4 Computing Maximal Hierarchies

The designs obtained (or the set $\mathbb{C}$) after the procedure described in sections 3.3.1 and 3.3.2 are used to design the hierarchies. Any of these designs in principle can appear at any level of the hierarchy. For computing the optimal three-level cache hierarchies, we need to evaluate N^3 hierarchies, where $N = |C|$. In our case, $N = 1601$. We have $N^3 = 4 \times 10^9$ hierarchies to evaluate. If we were to assume that the size of caches increase with each level, we can reduce the number of possible designs that need to be evaluated. But it would not reduce the design space by a big factor. We would still have 1.2×10^9 designs to consider. Even if we had 1000 machines available and each hierarchy required only a second to be simulated, we would need an entire year to carry out the required simulations. Thus the design space for cache hierarchies is large enough to rule out simulation-based search. We need an efficient algorithm to reduce the size of the design space.

Our approach computes the set of optimal hierarchies by adding one level at a time. For example, for a two-level hierarchy, we first compute the cache designs that are optimal for the first level. This leaves us with N_1 designs for the first level. Intuitively, any two-level hierarchy with a sub-optimal first level cache design can be replaced with another hierarchy that has an optimal first level cache design and the same second level cache design. Thus, we only consider $N_1 \times N$ possible designs for the two-level hierarchy, the cross product of N_1 optimal cache designs for the first level and N cache designs for the second level. We again compute the optimal two-level hierarchies. The procedure ends when a specified number of levels have been computed, or when the performance of each feasible design is inferior to some design that was optimal at the previous level. We use Bentley's algorithm [16] to compute the optimal hierarchies. In section 3.5, we justify our approach with a formal proof of its optimality.

3.4.1 Comparing Two Hierarchies

To compute the set of optimal hierarchies, we need to compare one hierarchy with another. We consider hierarchy B to be better than hierarchy A if B is better, *on average*, on a chosen set of performance metrics. The average is computed over all the applications the hierarchy is being designed for. While computing the average, we assume that each application is executed on the processor with equal probability. We mentioned four performance metrics previously: access time, leakage power, dynamic energy and chip area. We also compare the global miss rate [47] for the two hierarchies. This miss rate needs to be considered since the rest of the hierarchy also impacts the overall performance. Hence, B is better if it has lower values for expected access time, expected dynamic energy, leakage power, chip area and the global miss rate.

3.4.2 Computing Expected Values for Private Levels

Let $E_j[*]$ denotes the expected value of the quantity $*$ for the first j levels for the hierarchy. For private levels of the hierarchy, we compute $E_j[*]$ for different quantities in the following fashion:

- *Expected Access Time,* $E_j[t]$: This is the expected time for accessing the first j levels of the hierarchy. $E_j[t]$ does not include time spent on the levels after level j for accesses that miss at j. It is computed as: $E_j[t] = |\mathbb{A}|^{-1} \sum_{\alpha \in \mathbb{A}} E_{j,\alpha}[t]$ where $E_{j,\alpha}[t]$ is the expected time spent on accessing the first j levels of the hierarchy by application α. We compute $E_{j,\alpha}[t]$ as: $E_{j,\alpha}[t] = \sum_{i=1}^{j} p_{i,\alpha} t_i$, where $p_{i,\alpha}$ is the global miss rate for the memory accesses from application α at the $(i-1)^{th}$ level of the hierarchy and t_i is the time required for accessing the i^{th} level of the cache (obtained from Cacti) and traversing the interconnect between the $(i-1)^{th}$ and the i^{th} levels. The values $p_{i,\alpha}$ were collected as described in section 3.2.

- *Expected Dynamic Energy,* $E_j[d]$: This is the expected dynamic energy consumed for accessing the first j levels of the hierarchy. It is computed in the same fashion as $E_j[t]$ with the access time for a cache replaced with its per access dynamic energy.

- *Leakage Power,* $E_j[l]$: This is the total leakage from the first j levels of the cache hierarchy and is independent of the application behavior. It is computed as: $E_j[l] = \sum_{i=1}^{j} l_i$, where l_i is the leakage power of the i^{th} level.

- *On-chip Area,* $E_j[a]$: This is the total area required for the first j levels of the hierarchy. It is computed as: $E_j[l] = \sum_{i=1}^{j} a_i$, where a_i is the area of the i^{th} level.

- *Expected Global Miss Rate,* $E_j[p]$: This is the probability with which the accesses to the hierarchy miss in the first j levels. It is computed as: $E_j[p] = |\mathbb{A}|^{-1} \sum_{\alpha \in \mathbb{A}} p_{j+1,\alpha}$ where $p_{j+1,\alpha}$ is the global miss rate for the application α after accessing the first j levels of the hierarchy.

3.4.3 Computing Expected Values for Shared Levels

In this section, we explain our process for computing the maximal hierarchy designs when there are shared levels in the hierarchy. The set of all possible designs for the hierarchy is obtained as the cross product of the set of maximal designs for the private levels and the set of designs for the shared level of the cache. We assume that the shared cache has a size which is at least double the total size of the cache at the second level of the hierarchy. This assumption is required since we assume that the hierarchy we are designing is inclusive.

For each hierarchy in the cross product, we estimate its expected performance on a set of scenarios. Each scenario consists of a number of applications (same as the number of cores on the chip). These applications are chosen uniformly at random from the given set of applications. We assume that each scenario is equally likely to occur in practice. The performance of a given design is then computed as an expectation over these different scenarios. Again, the performance of a hierarchy refers to the following quantities: expected access time, expected dynamic energy, total leakage power, total chip area and probability of a miss. These are the same quantities that we computed while designing the private levels of the hierarchy. After computing these

quantities for each hierarchy design, we again apply the multi-dimensional divide and conquer algorithm to compute the set of maximal hierarchies.

To evaluate the performance of a given hierarchy on a given scenario, we compute the quantities mentioned above in the following manner:

- The **expected access time** for an application consists of two parts: the latency for the accessing caches and the latency involved in traversing the on-chip network. For each application, we have an estimate of the rate at which it missed at the previous level of the hierarchy. This rate is the rate at which the current level is accessed by the application. Since there multiple requests queue up at each bank of the cache, we apply the $M/D/1$ queueing model described in section 3.4.5 for estimating the latency experienced by a request. For estimating the network latency, we use the method described in section 3.4.5. The expected access latency for the scenario is then computed as the expectation over all the applications in the scenario.

- The **expected dynamic energy** for the hierarchy is computed by taking the expectation over all the applications in the scenario. For each application, we have an estimate of the probability that a load /store request accesses the given shared cache. This probability multiplied with the dynamic energy of the shared cache gives the energy per access for that application. The network dynamic energy is accounted using the method described in section 3.4.5.

- The **leakage** or the **static power** for the design is computed as the sum of the leakage power for all the caches in hierarchy and the leakage power for the on-chip network (obtained as described in section 3.4.5).

- For each application in the scenario, we compute the behavior of the application with respect to the shared cache using the method described in section 3.4.4. Hence, we have

an estimate for the probability of a miss in the shared cache for each of the applications. Using the access rate for the shared cache, we compute the expected miss rate for the shared cache for each application which is averaged over all the applications in the scenario to get the miss rate for the hierarchy for the scenario under consideration.

3.4.4 Estimating Shared Cache Behavior

For choosing the shared levels of the hierarchy, we need estimates for the cache behavior of applications when they share cache space with other applications. But we collect data for cache behavior when applications are running alone. Since multiple different execution scenarios are possible, we avoid collecting data for each scenario. Instead we leverage prior work for estimating the shared cache behavior of applications from their private cache behaviors.

We combine the private cache behaviors of different applications by using the *Frequency of Access* model presented by Chandra *et al.* [25]. Sandberg *et al.* [86] proposed a similar model for caches with *random* replacement policy. In both these models, the system under analysis is assumed to be in a steady state. Then, the shared cache space that an execution thread gets is proportional to the rate with which it brings data into the cache. This yields the following system of equations:

$$\frac{c_\alpha}{C} = \frac{f_\alpha}{F} \quad \forall \alpha \in \mathbb{A} \tag{3.1}$$

$$F = \sum_{\alpha \in \mathbb{A}} f_\alpha \tag{3.2}$$

In equation (3.1), c_α represents the number of ways that application α gets. C is the total number of ways in the cache. f_α is the rate at which application α brings data into the cache. It is a function of c_α and the number of sets in each way of the cache. Note that each application can use any set in a given way. We know f_α from the per application data collected in section 3.2. F is the sum of f_α over all applications α, the total rate at which data is brought in the cache.

Once the applications reach an equilibrium state with respect to the shared cache, their respective shares (c_α) would not change that much. This means the rate at which application α brings data into the cache (f_α) would match the rate at which its data is evicted from the cache ($F\frac{c_\alpha}{C}$). Hence, we get equation (3.1).

Solving the System of Equations

There is a mutual dependence between f_α and c_α. The fraction of the shared cache space that an application gets is dependent on the rate at which data is fetched into the cache. This rate is dependent on the rate at which memory requests are issued by the application, which in turn depends on the cache space it can use. Therefore, starting from some initial solution, the system of equations above are solved repeatedly till an equilibrium point (fixed point) is reached.

We use Krasnoselskij Iteration [17] for computing a fixed point of the system of equations defined above. Let E be a real normed space, $T : E \to E$ be a selfmap and $x_0 \in E$. Then, the Krasnoselskij Iteration is defined by the following relation:

$$x_{n+1} = (1 - \lambda)x_n + \lambda T x_n \tag{3.3}$$

where $\lambda \in (0, 1)$ is the damping factor.

While solving for the fixed point, we need to evaluate the cache behavior of applications for associativities different from the ones we have data for. Therefore, we interpolate the data using piecewise cubic interpolation [33] as done by Sandberg *et al* [86].

Anomalous Situations

We have observed situations in which a fixed point is not reached within a certain number of iterations or the fixed point exists with the cache size of some core being less than the sizes of the private caches. Currently, we drop hierarchies that result in either of the two situations.

3.4.5 Estimating Network Performance

The amount of research effort that has gone into designing on-chip networks for multi and many core processors speaks volumes about the important role on-chip networks play in the performance of these processors. Hence, on-chip network design needs to be taken into account while designing the cache hierarchy. In this section, we explain how we estimate the expected values for the performance metrics we care about: network latency, dynamic energy and leakage power.

Estimating Average On-chip Network Latency

The last level of the caches private to cores generate packets that traverse the network links based on the routing protocol and end up at a bank of the shared cache. The banks then respond to these packets and send back appropriate reply packets. We use *Queueing Theory* for analyzing the average latency associated with these packets. We base our analysis on the discussion by Bertsekas and Gallager [18].

An on-chip network can be viewed as a set of queues that interact with each other. At each router, traffic from multiple different packet streams, where a stream is composed of packets from one cache controller to another, merge together. Assume that each packet stream individually behaves as a *Poisson process* [18]. Had there been no interaction between the packet streams arriving at different routers, it would have been relatively straight forward to analyze the latency suffered by a packet. But in presence of multiple interacting queues, there is no known analysis technique that can provide estimates for packet latencies. This is because the assumption that packet streams behave as Poisson processes does not necessarily hold once the stream has been through one of the routers.

Kleinrock [59] suggested that merging several packet streams restores the independence of inter-arrival times. Hence, one can use queueing models that are appropriate for a single-server system, even though the traffic at each router interacts with the traffic in the rest of the network. This is known as the **Kleinrock independence approximation**. We use this approximation in

estimating the time expected time for traversing the on-chip network. While Kleinrock experimentally verified that this approximation works well for networks he considered, it is not known if this holds for on-chip networks. In section 3.4.5, we further discuss this assumption and the experiments we performed to test its validity.

The routers associated with each node in the network, act as servers servicing packets. We model each router and its set of incoming links as an $M/D/1$ queueing system, assuming that a router requires a fixed deterministic amount of time to process a packet. Let λ_i be the rate at which packets arrive at router R_i, μ_i be the rate at which it processes packets and $\rho_i = \frac{\lambda_i}{\mu_i}$. Using the *Pollaczek-Khinchin formula* (P-K), we compute W_i, the time spent by a packet in router R_i's buffer as:

$$W_i = \frac{\rho_i}{2\mu_i(1-\rho_i)} \tag{3.4}$$

To this we add $\frac{1}{\mu_i}$, the time required by R_i to service a packet and d_i, the time required for traversing the next link. We estimate the total time required by a packet to traverse a path $\mathcal{P}$ as:

$$T_{\mathcal{P}} = \sum_{R_i \in \mathcal{P}} \left(\frac{\rho_i}{2\mu_i(1-\rho_i)} + \frac{1}{\mu_i} + d_i \right) \tag{3.5}$$

The estimate allows for using of multiple different router designs [70] in the same network. It is also independent of the topology of the on-chip network and routing protocol in use.

Estimating λ_i

The estimate described above depends on the request arrival rate, λ_i, at each router. We estimate λ_i in the following fashion. The rate at which a core issues load and store requests is given as the product of the rate of instruction execution and the expected number of loads and stores per instruction. We approximate the rate at which a core executes instructions is given as: $\frac{1}{t_{exec} + n_{avg}\, t_{mem}}$, where t_{exec} is the time required for executing an instruction when load and store instructions can be executed in a single cycle, n_{avg} is the expected number of loads and

stores per instruction and t_{mem} is the expected time that a load or a store access spends on traversing the hierarchy before it traverses the on-chip network. We compute the rate at which requests are injected into the on-chip network multiplying the rate computed above with the global miss rate of the cache which injects the requests into the on-chip network. The arrival rate estimate is conservative since the time spent in accessing the lower levels of the hierarchy and the on-chip network will lower the rate at which load and store requests are issued.

We assume that these requests are uniformly distributed over the set of possible recipients. For example, if some particular (shared) level of the cache hierarchy is distributed across the chip, we assume that the requests are uniformly distributed among the different parts of the level. This assumption allows us to compute the rate at which request traffic is received at each router. This also gives us the rate at which each shared cache controller receives the request traffic. Hence, we can also compute the rate at which response traffic is injected into the network by the cache controllers at the shared level. Combining these request and response traffic rates, we get the total rate at which each router receives traffic. This total rate is the required λ_i value. Symbolically, we have the following relation:

$$\lambda_i = \sum_{\substack{\text{all request stream } req \\ \text{that pass through } R_i}} x_{req} + \sum_{\substack{\text{all response stream } res \\ \text{that pass through } R_i}} x_{res} \tag{3.6}$$

Estimating Dynamic Energy, Leakage Power and Area

We use DSENT [94] to account for the area, the dynamic energy and the leakage power of the on-chip network.

Testing Poisson Property and Kleinrock's Independence Assumption

In the description above on the method for analyzing the latency experienced by a cache access due to the on-chip network, we made two assumptions: first, the request process follows Poisson

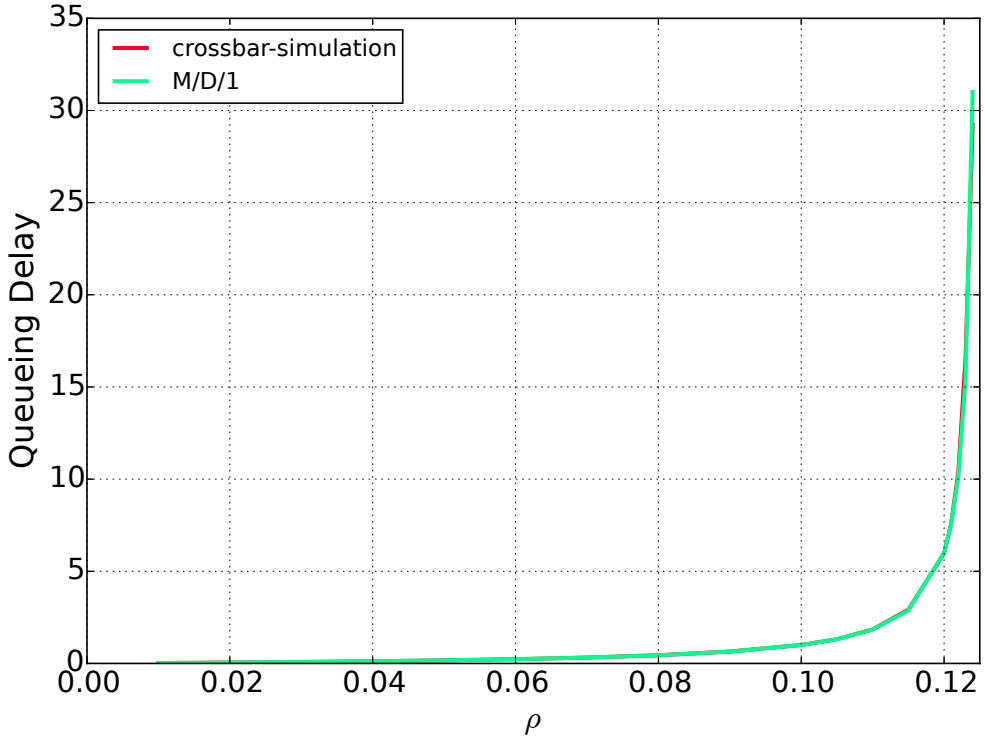


Figure 3.3: Results from synthetic simulation of the on-chip network. The graph shows the average queueing delay for different amounts of load (ρ) generated by the input sources. The graph also shows the theoretically predicted queueing delay for the same load using the *Pollaczek-Khinchin formula*.

distribution, and second, the streams of packets merging at each router restore the Poisson distribution which is affected due to the interaction amongst different queues. We need to verify these assumptions for the system we are working with.

To verify Kleinrock's independence assumption for an on-chip mesh network, we developed a simulator for an on-chip network. Each input source generates packets such that time interval between two packets is exponentially distributed. These patckets pass through routers and are ultimately delivered to their destination. For each router, we collected the number of packets that arrived at a router and the total waiting time across all the packets that arrived.

Using the simulator described above, we simulated a crossbar. In Figure 3.3, we show the

average queueing delay observed during the simulation. We also show the queueing delay for an M/D/1 queueing server. As can be seen in the Figure, the two curves match each other closely. Similar experiments were conducted for the research work presented in Chapter 5. The reader may refer to section 5.3.7 for details.

3.4.6 Estimating Physical Memory Performance

After adding the cache levels, we add the physical memory as the final level in the hierarchy. We assume that last level of the cache hierarchy and the memory controllers communicate via an on-chip network that is separate from the one which the cache controllers use. This simplifies the analysis as it avoids interference between cache-to-cache and cache-to-memory traffic. We also assume that the physical memory is sufficient enough that each access to the physical memory ends up in a hit. To evaluate each hierarchy along with the physical memory, we consider the following four performance metrics: *expected access time*, *expected dynamic energy*, *total leakage power* and *on-chip area*.

Timing Model for a Memory Controller

We base our timing model as per the description from Jacob, Ng and Wang [53]. We assume the traffic received by the memory controllers is uniformly distributed over all the memory controllers. Further, the traffic received by a particular controller is uniformly distributed over its dram channels. We model each channel as an $M/G/1$ queueing system under the assumption that the request process behaves as a $Poisson$ process. Though there are multiple types of commands that a dram device supports, we model only three of them:

- *read*: the service time is assumed to be t_{RC}.
- *write*: the service time is assumed to be t_{RC}.

- *refresh*: we assume that the refresh commands are issued by the controller as a Poisson process with the mean as required by the dram device. The service time is assumed to be t_{RFC}.

The average time required for processing a request is estimated to be:

$$T = \overline{X} + \frac{\lambda \overline{X^2}}{2(1-\rho)} \tag{3.7}$$

where λ is rate of request arrival, $\overline{X}$ is the average service time, $\overline{X^2}$ is the second moment of the service time and $\rho = \lambda \overline{X}$.

Power Model for Physical Memory

We assume a 1333 MHz DRAM and use the average dynamic energy values as published by David *et al*. [28].

3.5 Proof of Pareto-Optimality

In this section, we show why the proposed method may yield the set of maximal cache hierarchy designs. Assume we are designing a two-level cache hierarchy. We show that the set of maximal hierarchies for the two levels ($H_{1,2}$) can be computed by first computing the set of maximal hierarchies for the first level (H_1) and then computing the maximal hierarchies over the cross product of H_1 and $\mathbb{C}$, the set of all cache designs.

3.5.1 Statement of the Problem

More formally, assume two candidates designs for a single level cache hierarchy: $h_1 = (A_1)$ and $h_1' = (A_2)$, where A_1 and A_2 are individual cache designs. On all the performance metrics, the

expected values for h_1 are better than those for h'_1. Our method therefore discards the configuration h'_1 at the L_1 stage itself. Then, is it true that h_1 is better than h'_1 for all possible choices of the second level?

3.5.2 Proof

Since we are only working with expected values, it seems possible that some cache design for the second level cache may make h'_1 better than h_1. We prove that this is not possible.

Let B be a cache design to be added at the second level. Define $h_{1,2} = (A_1, B)$ and $h'_{1,2} = (A_2, B)$. Compute the **expected access time**, $E_2[t]$, for the two hierarchies as:

$$E_2[t] = E_1[t] + \frac{t_2}{|\mathcal{A}|} \sum_{\alpha \in \mathcal{A}} p_{\alpha,2} \tag{3.8}$$

where $E_1[t]$ is the expected access time for the first level, $p_{\alpha,2}$ is the global miss rate at the first level for application α and t_2 is the access time for the second level of the cache. As per our assumption h_1 is better than h'_1 for the first level. Therefore, $E_1[t]$ for h_1 lower compared to $E_1[t]$ for h'_1. Similarly, $\sum_{\alpha \in A} p_{\alpha,2}$ has a lower value for h_1 than for h'_1. Combining these two facts, we get $h_{1,2}$ has a lower $E_2[t]$ compared to $h'_{1,2}$. Similarly, one can show that other performance metrics are better for $h_{1,2}$ even after a new level of cache has been added. This completes the proof of optimality for our dynamic programming-based method.

The proof above is applicable to only the metrics we considered. For any other metrics, the proof should be revisited to make sure that the proposed method works correctly. Further, the discussion above assumes all levels of the hierarchy are private to the cores. For shared levels of caches, if we assume that lower access rates to the shared cache result in better performance, then it seems that any hierarchy that is dominated by any other hierarchy, in terms of the metrics in use, would still be dominated when the shared levels are added. But since the miss rates at the shared level are computed using a fixed point computation, it is not clear if a strong statement

can be made.

3.5.3 Why Work with Expected Values

In our method, we do not work with values for individual applications or scenarios. Instead we combine them and work with expected values. This is because the computational complexity of the multi-dimensional divide and conquer algorithm for computing maximal designs is $O(n \log^{k-1} n)$. Here n is the number of points in the set and k is the number of dimensions of a point. The algorithm is exponential in the number of dimensions and the base of the exponent is $\log n$ which is large (> 2). If we treat each individual application / scenario as a dimension, the exponent will be raised to a very high value. For W applications, the number of dimensions would be cW where c is number of performance metrics involved. In our case, c is five. The algorithm would be $O(n \log^{5W-1} n)$ in time complexity. Even for 10 applications, the algorithm would require prohibitive amount of time to compute the maximal designs with so many dimensions. Hence, we only require optimality for the expected values of the performance metrics.

3.6 Extending the Method

In this section, we discuss how the method presented in section 3.4 should be extended to handle some of the intricacies of modern processors.

3.6.1 Pipelined Out-of-order Cores

Designing cache hierarchy for out-of-order cores required two changes. First, we computed the rate at which the processor issued memory instructions under the assumption that one instruction is committed each cycle. This assumption does not hold true for out-of-order processors as they can commit multiple instructions per cycle. For each application, we need an estimate of the rate at which it can commit instructions on the out-of-order core we are designing the hierarchy

for. We collect data for computing the clocks cycles required per instruction (cpi) for an out-of-order core as a function of the latency involved in accessing a perfect memory system (infinite bandwidth, fixed latency).

Second, we assumed that the processor blocks after issuing a load / store request and waits till the request is fulfilled. This meant a request to the caches private to core never had to wait for another request to get done. With out-of-order cores, a request may need to wait before it is serviced. Hence, we apply queueing theory to model the delay involved in servicing a request. Moreover, caches may have multiple read/write ports to allow for the increased rate of request. These multiple ports are modeled using $M/D/k$ queueing system where k is the number of ports.

3.6.2 Exclusive caches

For designing an inclusive cache hierarchy, we assumed that the probability of missing in the cache at level i is approximately the probability of a miss if this cache were the first level of the hierarchy. For an exclusive cache in the hierarchy, we modify this assumption. We assume that the probability of a missing in the cache at level i is approximately the product of the probability that this level is accessed and the probability of missing in this cache if it were the first level in the hierarchy.

For example, assume we have two cache designs C_1 and C_2 with miss probabilities p_1 and p_2 respectively. When estimate these probabilities from the workload behavior, we assume that the respective caches were the first level of the hierarchy. For the exclusive cache hierarchy in which C_1 is at the first level and C_2 is at the second level, the probability of accessing C_2 is given by p_1. The probability of missing in the second level of the hierarchy is then assumed to be $p_1 p_2$.

3.6.3 Cache replacement policies

Steps 2 (capturing workload behavior) and 5 (analyzing behavior of shared caches) need to be

designed specifically for the replacement policy in use.

3.7 Comparison With a Continuous Model

In this section we first formulate a model that assumes cache designs can be represented as continuous functions of the design parameters. Then, we compare this continuous model with the discrete model we described earlier. This model is inspired by the models presented by Jacob *et al.* [54] and by Sun *et al.* [95]. We assume that the hierarchy consists of n levels. The first level is closest to the processor and the last level is the main memory. The complete model appears in Figure 3.4.

3.7.1 Variables

The model consists of following variables:

- s_i: size of the cache at level i.
- t_i: access latency of the cache at level i.
- p_i: the global miss ratio for the cache at level $i-1$. This is also the probability that a memory request needs to access level i of the cache hierarchy. It is modelled as:

$$p_i = \beta s_{i-1}^{\alpha}$$

  Note that the access probability of the cache at level i depends on the cache at the previous level. This is because we assume that a cache at level i includes all the data stored in the levels 1 to $i-1$. The constants α and β vary with the application being studied.

- $static_i$: power dissipated by the cache at level i even when the cache is not in operation.

- $dynamic_i$: the energy dissipated by the cache at level i when it is in operation.
- $area_i$: the on-chip area required for the cache at level i.

Other than p_i, we model t_i, $static_i$, $dynamic_i$ and $area_i$ as linear functions of the size of the cache at level i. This necessitates use of constants $a_0, a_1, b_0, b_1, c_0, c_1, d_0$ and d_1. The value of these constants were estimated by fitting linear regression models to the data obtained for different cache designs from Cacti.

3.7.2 Constraints

Apart from the definitional equations presented above, we have three constraints (3.13), (3.14) and (3.15) in the model. These constraints, respectively, limit the total static power, the total dynamic energy and the total on-chip area that can be consumed by the cache hierarchy.

3.7.3 Objective Function

The objective of the model is to minimize the total expected delay involved in completing an access to the hierarchy. We use the following definition for total expected delay:

$$total_delay = \sum_{i=1}^{n} (t_i + \text{network delay}) p_i$$

The equation above has an extra term for **network delay**. The t_i term represents the time involved in accessing just the cache. The network delay is time spent on moving from one cache level to another.

3.7.4 Comparison With the Dynamic Programming-based Algorithm

We implemented the model from Figure 3.4 using GAMS modelling language and solved it using Baron [96] as the solver. The values used for various parameters involved in the model

$$\text{minimize} \quad \sum_i (t_i + \text{network delay}) p_i \tag{3.9}$$

$$\text{subject to} \quad t_i = a_0 + a_1 s_i \tag{3.10}$$

$$t_n = \text{memory delay} \tag{3.11}$$

$$\text{For all cache levels} \begin{cases} static_i & = b_0 + b_1 s_i \\ dynamic_i & = p_i(c_0 + c_1 s_i) \\ area_i & = d_0 + d_1 s_i \\ p_i & = \beta s_{i-1}^{\alpha} \end{cases} \tag{3.12}$$

$$\sum_{i=1}^{n-1} static_i \leq \text{power budget} \tag{3.13}$$

$$\sum_{i=1}^{n-1} dynamic_i \leq \text{energy budget} \tag{3.14}$$

$$\sum_{i=1}^{n-1} area_i \leq \text{area budget} \tag{3.15}$$

$$s_i \geq 0$$

Figure 3.4: Continuous Optimization Model for Hierarchy Design

appear in Table 3.4. The solver was constrained to output the best solution found in $60s$. The solution sizes for different levels in the hierarchy had to be rounded to match some available cache design. In our rounding procedure, we search for the cache design that is closest to the computed optimal design along the four dimensions: access time, dynamic energy, static power and area. We also impose the specified budget constraints during this search procedure. Since the continuous model does not have any notion of associativity, we compute optimal hierarchy for possible associativities separately.

To compare the efficacy of the continuous model and the discrete model, we use our method to compute the Pareto-optimal hierarchies for the same parameter values. Note that our method does not require any rounding procedure. We simulate the hierarchies obtained from the two

Param Name	Value
static power budget	0.6 mW
energy budget	4 nJ
area budget	$1.5 \times 10^6 \mu m^2$
memory delay	50 ns
network delay	0.25 ns

Table 3.4: Parameter Values used for solving the Continuous Model

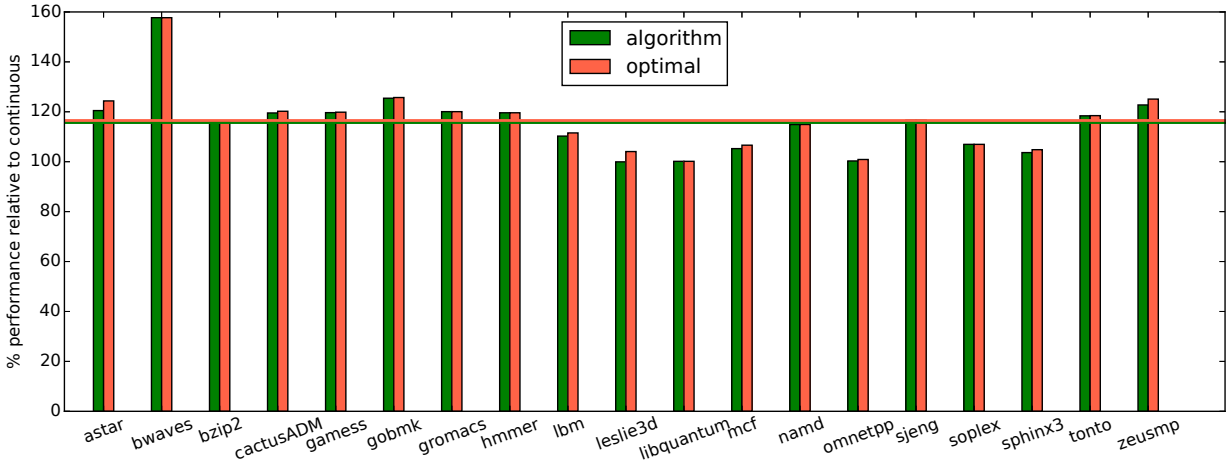
models using gem5 [19]. We carried out simulation-based experiments for applications from the SPEC CPU2006 [48] benchmarks. Each application simulated was first fast forwarded 10 billion instructions and 100 million instructions were simulated. In Figure 3.5, we show the improvement in CPI provided by the best hierarchy computed using the discrete model over the best hierarchy computed using the continuous model.

3.8 How Effective is the Method

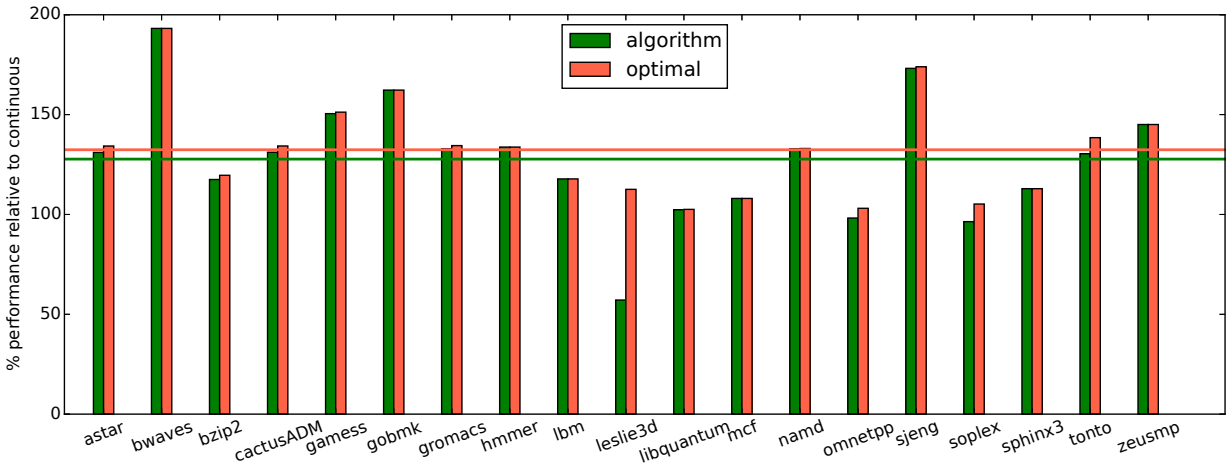
To evaluate the efficacy of our method, we perform multiple experiments and compare the performance of the hierarchies obtained using our method other possible hierarchies. We describe these experiments and results in this section. Our experimental data was obtained for $32nm$ process technology.

We collected the cache behavior of different SPEC CPU2006 applications for different cache sizes and associativities using gem5 [19] and ref input set. While collecting information, we skip the first ten billion instructions, and then recorded observations over the next 100 million instructions.

Each of our experiments consists of a set of applications for which we compute the set of maximal hierarchies for different number of cores. The design of the processor assumed is shown in Figure 3.6. While computing these hierarchies, we imposed an area budget of 25 mm^2 and static power budget of 1 mW. With this budget, there are about 1.2×10^9 three-level



(a) Inorder pipelined CPU core used for simulation.



(b) Out-of-order pipelined CPU core used for simulation.

Figure 3.5: Improvement in IPC for the best hierarchy computed using discrete model as percentage of the IPC for the best hierarchy computed using continuous model. The horizontal lines show the average improvement in IPC

hierarchies that are feasible. We would need more than two years to simulate these hierarchies assuming we need a minute of simulation for each and we have 1000 machines available. Most of these hierarchies are very well dominated by other hierarchies and should be neglected. Testing our method with multiple different workloads, we found that less than 1000 hierarchies have expected access time and expected dynamic energy that is at most 25% worse compared to the hierarchies predicted to be Pareto-optimal. We carried out detailed simulation of such hierarchies

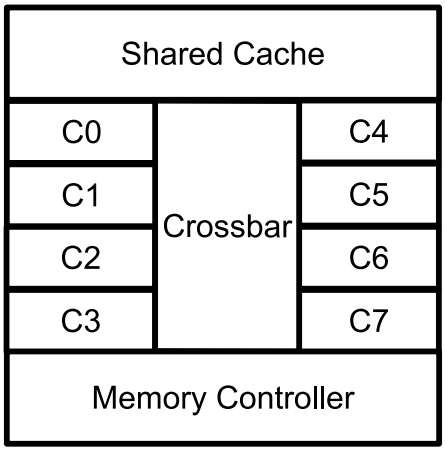


Figure 3.6: Multicore Processor Layout

Parameter	Value
Number of Cores	8
Processor Core	4GHz, pipelined, 8-wide out-of-order
Coherence Protocol	MESI
Main Memory	16 GB, single DDR-style controller
Warmup Time	50,000,000 cycles
Instructions per core	at least 100 million

Table 3.5: Simulation Parameters

only.

The detailed simulation was carried out using gem5 [19]. The simulation parameters appear in Table 3.5. These simulations provide us the actual values for the expected access time and the expected dynamic energy per access to the hierarchy. Using the simulation results, we recompute the set of maximal hierarchies.

3.8.1 Experimental Results

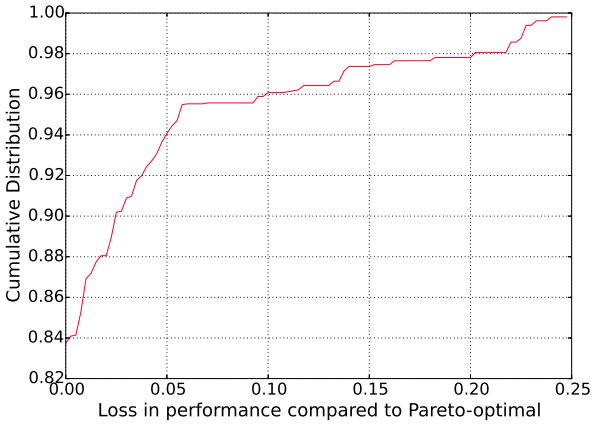
In Figure 3.7, we show the results obtained for 8-core cache hierarchies for homogeneous scenarios. Each of these scenarios consists of eight copies of a single benchmark application from the SPEC CPU2006 benchmark suite [48].

- In Figure 3.7a, we show the cumulative distribution function for actual Pareto-optimal cache hierarchies as a function of their relative distance from the Pareto-optimal frontier predicted by the method. This function is similar to the *precision* metric used for measuring effectiveness of machine learning algorithms [31]. About 84% of the actual Pareto-optimal cache hierarchies were correctly predicted by our method. If we also simulate hierarchies that are predicted to perform at most 5% worse than the predicted Pareto-optimal hierarchies, then we can find about 94% of the actual Pareto-optimal hierarchies.

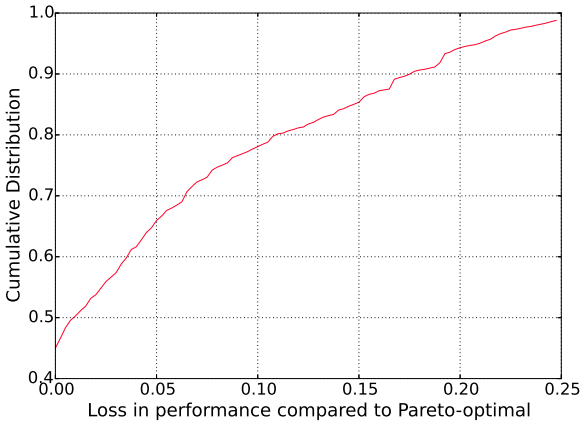
- In Figure 3.7b, we show the cumulative distribution function for simulated cache hierarchies as a function of their relative distance from the Pareto-optimal frontier predicted by the method. This function is similar to the *recall* metric used for measuring effectiveness of machine learning algorithms [31]. All the cache hierarchies that were found to perform at most 25% worse than the predicted Pareto-optimal frontier were simulated. About 45% of these were predicted to be on the frontier by our method and 65% of these were predicted to perform at most 5% worse than those predicted to be on the frontier. From Figures 3.7a and 3.7b, we can infer that we need to simulate about half of these hierarchies to find 90% of the hierarchies on the Pareto-frontier.

- In Figures 3.7c and 3.7d, we compare the actual values for cycles per instruction (cpi) and dynamic energy consumed with those predicted by our method. The cpi values seem to be closely spread around the line with slope $45°$. It seems we under predict the energy consumed. This likely due to the variation in memory access patterns. While the latency incurred due to a larger number of memory accesses can be hidden by the out-of-order processing core, it cannot hide the energy consumed for these accesses.

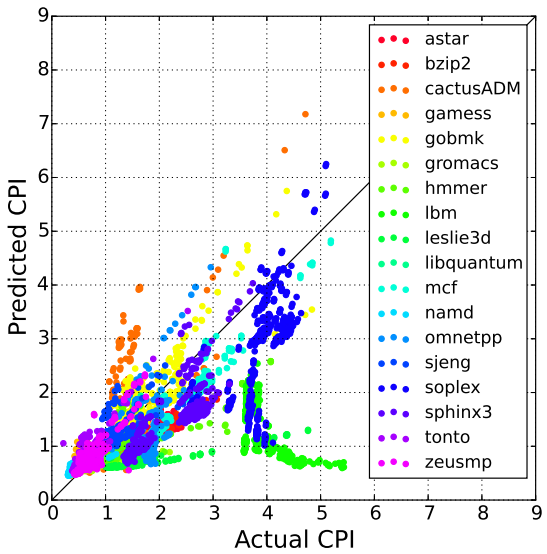
In Figure 3.8, we show the results obtained for heterogeneous scenarios consisting of applications from SPEC CPU 2006. The graphs for heterogeneous scenarios are similar to the ones described above for homogeneous scenarios.



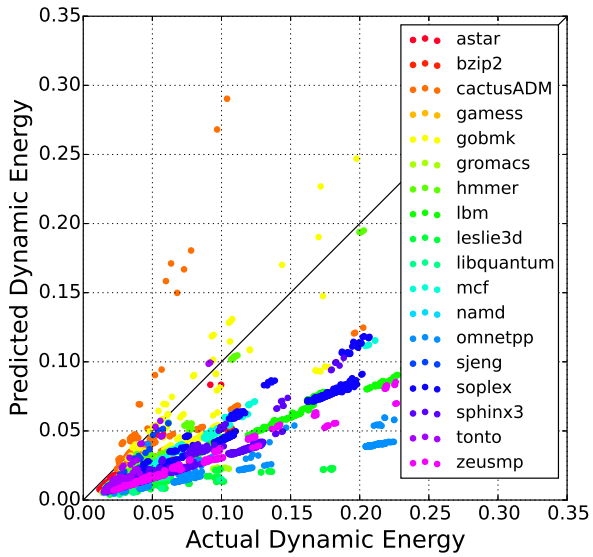
(a) Cumulative Distribution of true Pareto-optimal hierarchies as a function of their predicted performance. 0.0 on x-axis represents predicted Pareto-optimal hierarchies, 0.25 represents hierarchies predicted to perform about 25% worse.



(b) Cumulative Distribution of all simulated hierarchies as a function of their predicted performance. 0.0 on x-axis represents predicted Pareto-optimal hierarchies, 0.25 represents hierarchies predicted to perform about 25% worse.

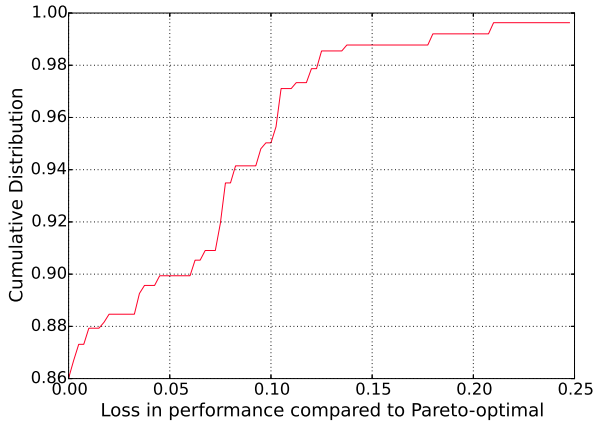


(c) CPI Scatter (units are cycles)

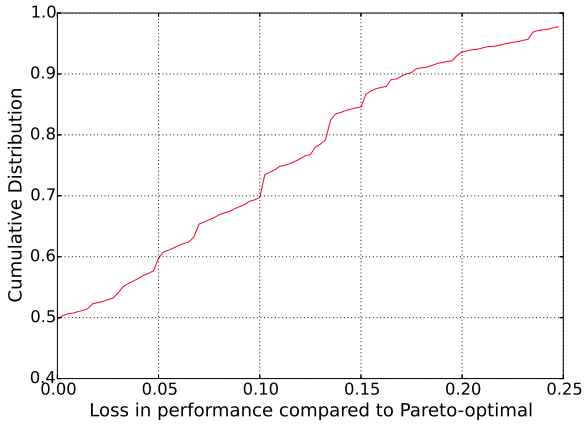


(d) Dynamic Energy Scatter (units are nanoJoules)

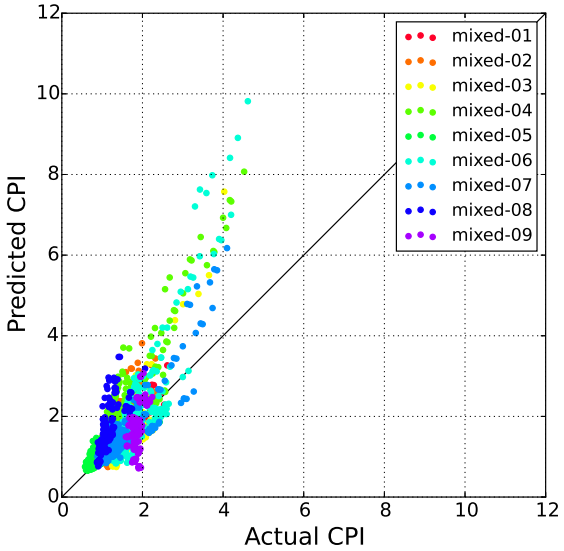
Figure 3.7: Graphs for homogeneous scenarios created from SPEC CPU2006 applications.



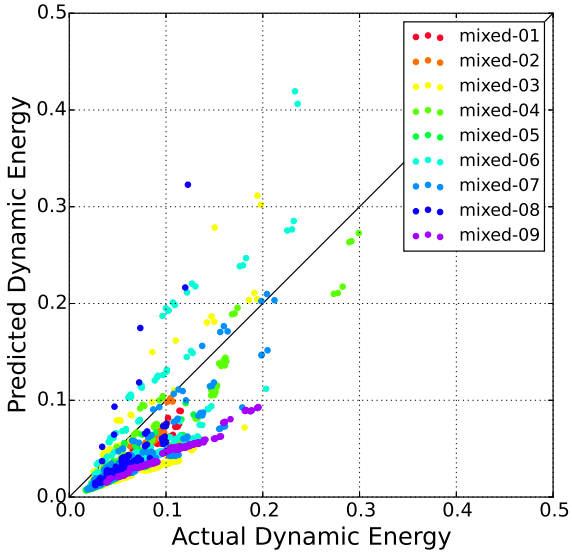
(a) Cumulative Distribution of true Pareto-optimal hierarchies as a function of their predicted performance. 0.0 on x-axis represents predicted Pareto-optimal hierarchies, 0.25 represents hierarchies predicted to perform about 25% worse.



(b) Cumulative Distribution of all simulated hierarchies as a function of their predicted performance. 0.0 on x-axis represents predicted Pareto-optimal hierarchies, 0.25 represents hierarchies predicted to perform about 25% worse.



(c) CPI Scatter (units are cycles)



(d) Dynamic Energy Scatter (units are nanoJoules)

Figure 3.8: Graphs for heterogeneous scenarios created from SPEC CPU2006 applications.

3.8.2 Comparison With Prior Work

We used our method to compute the Pareto-optimal cache hierarchies using the two approximations suggested by Sun *et al.* [95]. These approximations were also mentioned in the section 3.1.

- The cache hit rate was modeled using the $2 - to - \sqrt{2}$ rule. As per the rule, a cache of size $2x$ has a miss rate which is less by a factor of $\sqrt{2}$ from the miss rate for a cache of size x.

- The access power of a cache is given by: $\rho\sqrt{size} \times bandwidth$. Here ρ is constant determined using Cacti [75].

Sun *et al.* required these approximations since their method uses continuous functions. We changed our method to make use of these assumptions in place of the data that was collected from applications (discussed in section 3.2) and from Cacti(discussed in section 3.3). We then computed Pareto-optimal cache hierarchies for the same parameter as described earlier in this section. After that, we performed detailed simulations of the computed hierarchies. We make two observations about the results from these experiments:

- Only about 25% of actual Pareto-optimal hierarchies were found using the suggested approximations. As mentioned earlier, we had computed 84% of the Pareto-optimal hierarchies when we used the data obtained directly instead of using the data to compute the parameters for the approximations mentioned above.

- The hierarchies obtained with the approximations were dominated in performance by the hierarchies obtained without the approximations.

3.9 Lessons Learnt

In this section we highlight few things that we learned while designing and evaluating our method.

Detailed processor model is imperative for architectural evaluation. We initially evaluated the method using a very simple model for the processing core (no pipelining, no speculation, one cycle per non-memory instruction, detailed memory model). We obtained very accurate results with such a core model. Then, we switched to using a detailed out-of-order core model

(pipelined, speculative and super-scalar execution) and things broke down completely. This is because the memory access pattern and the access rate for an out-of-order core are significantly different from that for a simple core described above.

Detailed memory model is imperative as well. Evaluating models using very simple cache controller designs is also not good. These simple designs tend to ignore the affect of queueing delays in cache controllers. We observed significant performance degradation due to lack of bandwidth at cache controllers. This is especially true when caches are small in size and multiple pending requests may map to the same cache line.

3.10 Related Work

In this section, we briefly describe some of the prior research that is of a similar nature as our work. Over the years, multiple different approaches have been taken towards design of processors through the use of mathematical models. While initially these approaches were directed towards designing performance optimal designs, the recent literature tends to focus both on the performance and the power consumption of the design. These works include Lee and Brooks [61, 62] on regression modeling for exploring microarchitectural design space, Azizi *et al.* [11] on jointly optimizing over the space of architectural and circuit design choices, and Eyerman, Eeckhout and De Bosschere [30] on performance of various search algorithms for exploring design space for out-of-order processors. These works, as we understand, represent the design space using continuous variables. Our work tries to prune the design space of cache hierarchies with a discrete model.

Van Craeynest and Eeckhout [99] and Eklov *et al.* [29] present mathematical models for estimating the performance of multi-core multi-programmed workloads. Since we also carry out such modelling, it is possible to combine our method with these models. It is likely that we will be able to more accurately predict the Pareto-optimal hierarchies.

Karkhanis and Smith [56] use mathematical modeling for arriving at Pareto-optimal design points for out-of-order superscalar processors. The goal of this work is similar to ours, but our use of dynamic programming and Bentley's multi-dimensional divide and conquer algorithm for computing maximal points, allows us to explore a much larger space. Lastly, Przybylski [81] in his thesis work discusses a dynamic programming algorithm for designing access time optimal multi-level cache hierarchies. Our algorithm is structurally similar. But we allow for multiple objectives, multiple execution scenarios, designing for shared caches and the on-chip network.

3.11 Conclusion

In this work, we presented an algorithm for pruning the design space for cache hierarchies. We justifies our method with a formal proof that hierarchies computed using the method are optimal under the assumptions of our model. We presented arguments on why continuous modeling of the design space prevents modeling of important design requirements and makes it hard to optimize for multiple different objectives. We also presented a continuous model for designing cache hierarchy for single core processors and showed that our method computes better designs than the designs computed as solutions to the continuous model. Lastly, we provided simulation-based experimental results for the effectiveness of the proposed method and showed that our proposed method performs better than the continuous modelling-based method proposed in an earlier work. We hope that our method would appeal to processor designers.

Chapter 4

Resource Placement and Distribution

In this chapter, we use mathematical optimization to manage the growing design complexity. We demonstrate this approach using three problems related to on-chip networks. The first two are: memory controller placement [5] (section 4.1), and resource allocation in heterogeneous on-chip networks [70] (section 4.2). The combination of these two problems is a much more challenging optimization problem (NP-Hard) but explores the tradeoffs between shared resources in the design (section 4.3). While mathematical optimization has been used before in different facets of computer architecture including on-chip network design [93, 58, 67], compilers [34, 4] and design space exploration [12], we believe we are the first to use it for these three problems. Futhermore, we demonstrate that use of mathematical optimization for computer architecture-related problems can result in nonlinear models and how one might overcome the challenges in solving them. Simulation-based experiments provide evidence that our mathematically optimal designs provide better performance than previous solutions. We believe these results provide compelling evidence that architects should adopt mathematical optimization as another important design tool.

The reader may note that this work has been published previously in ACM Transactions on Architecture and Code Optimization [98].

4.1 Placement of Memory Controllers

Chip Multiprocessors (CMPs) need abundant DRAM bandwidth to feed the increasing number of cores. Due to limited pin bandwidth [5], there will be more cores compared to the number of memory controllers on a chip. For example, the Oracle Sparc T5 [32] has 4 memory controllers for 16 cores. This raises the question of how to place the memory controllers within the on-chip network. Careful placement of the controllers can lead to lower latency and better bandwidth utilization for on-chip communication. This problem was introduced by Abts *et al.* [5].

Consider a design problem with n cores and m memory ports. Assuming the memory ports are co-located with the cores, there are $\binom{n}{m}$ possible ways of placing the memory controllers. For a 64-core, 16-port design, this number is about 4.9×10^{14}. It is not possible to explore each and every placement of the memory controllers. Abts *et al*. search this design space using a combination of intuition (experience), exhaustive simulation of smaller designs, and a *Genetic Algorithm* (GA) based approach to arrive at a reasonably good placement. Instead, we use mathematical optimization for searching the design space by creating an optimization-based model for the problem. We next discuss the assumptions used in the model.

4.1.1 Assumptions

Similar to Abts *et al*., we assume that the cores are laid out on a 2D-plane and are connected using an on-chip network. The memory ports are co-located with the cores. The on-chip network uses a deterministic routing protocol, i.e., all messages from node A to node B always traverse the same path [55]. Lastly, since CMPs typically distribute memory addresses across controllers using lower-order address bits [32], we assume the traffic is distributed uniformly across all memory controllers.

Since the memory controllers are distributed (possibly) over the entire chip area, the chip would need to be manufactured such that the I/O pins cover the entire area of the chip and not

just the periphery, as is the case with flip chip technology [97]. The design constraints on these I/O pins are not considered in our model. We also ignore the difference in areas of cores that have memory ports and those without.

4.1.2 Notation Used in the Model

We model the problem as a Mixed Integer Linear Program (MILP). Figure 4.1 shows the formulated model. In the model, each memory controller is denoted as a point (x, y) on the 2-D plane. Similarly, a core is referred to with coordinates (x', y'). For each (x, y), $I_{x,y}$ is a binary variable denoting whether a memory controller is placed at (x, y). $LoadOnLink(l)$ denotes the load on the link l due to the communication between the cores and the memory controllers. This load depends on the placement of the controllers. The set $Path(x, y, x', y')$ contains all the links l that are used for going from (x, y) to (x', y'); since the routing protocol is deterministic, $Path()$ is an input to the problem. z bounds the maximum load a link can be assigned. We further assume that the read to write requests have a ratio of $R : 1$ and that a packet with data is K-times the size of a packet with no data.

4.1.3 Description of the Model

There are three constraints involved in the model.

- Equation (4.1) enforces that a specified number of memory controllers (m) are placed on the chip. $I_{x,y}$ can be either 0 or 1. So when the equation is satisfied, exactly m of the n $I_{x,y}$-variables are set to 1, the rest are 0.

- Equation (4.2) defines the load on each link in the on-chip network. The first term on the right hand side of the constraint is a sum over all pairs (x, y) and (x', y') such that a memory controller is placed at (x, y) and the path from (x', y') to (x, y) uses link l. This term represents the request traffic going from the cores to the memory controllers. We assume

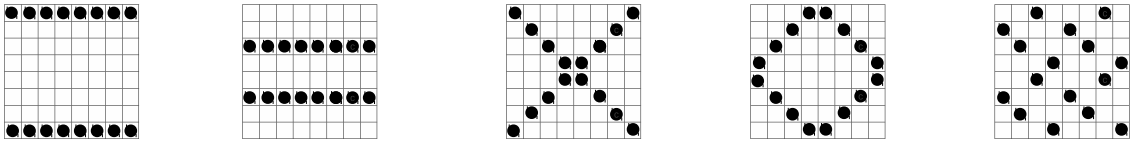
$$
\begin{aligned}
&\text{minimize} && z && \\
&\text{subject to} && && \\
&\sum_{(x,y)} I_{x,y} = && \text{number of memory controllers} && (4.1)\\
&LoadOnLink(l) = && \sum_{(x,y,x',y'):l\in Path(x',y',x,y)} I_{x,y}(R+K) && \\
&+ && \sum_{(x,y,x',y'):l\in Path(x,y,x',y')} I_{x,y}(RK+1) && (4.2)\\
&\text{where } l \text{ varies} && \text{over all the links in the network} && \\
&LoadOnLink(l) \leq && z && (4.3)\\
&\text{where } l \text{ varies} && \text{over all the links in the network} && \\
&I_{x,y} \in \{0,1\}, && z \in \mathbb{R}_+ &&
\end{aligned}
$$

Figure 4.1: MILP for Placing Memory Controllers

the traffic has $R:1$ ratio for reads and writes. Each read request requires a single flit, while a write request needs K flits. Hence, the total request traffic is proportional to $R+K$. The second term, which represents the response traffic from the memory controllers to the cores, can be interpreted in a similar fashion. The total response traffic is proportional to $RK+1$, where RK is for flits with response data for read requests and 1 is for flits with acknowledgment for write requests.

- Equation (4.3) bounds the load on any link to be at most as large as z.

The objective of the problem is to minimize z, the maximum load on any of the links i.e. the *maximum channel load*. Thus an optimal solution for the model would place the memory controllers such that the maximum load placed on any of the links is as low as possible. A lower value for z means less congestion in the network and lower queueing delays. Alternately, we would like to use the available bandwidth efficiently by spreading the load across as many links as possible in a uniform manner. We chose this objective so as to keep the model linear.



(a) row 0-7 (b) row 2-5 (c) diagonal (d) diamond (e) mc-opt

Figure 4.2: Different Controller Placements for an 8×8 Mesh/Torus Network. Tiles with black-colored sphere represent a memory port co-located with a core.

4.1.4 Solving the Model

For solving the model, we assume the cores are connected using a $k \times k$ 2D- mesh or torus on-chip network. The number of memory controllers that need to be placed is $2k$. We also assume that the network uses dimension-ordered routing. Abts *et al*. made similar assumptions.

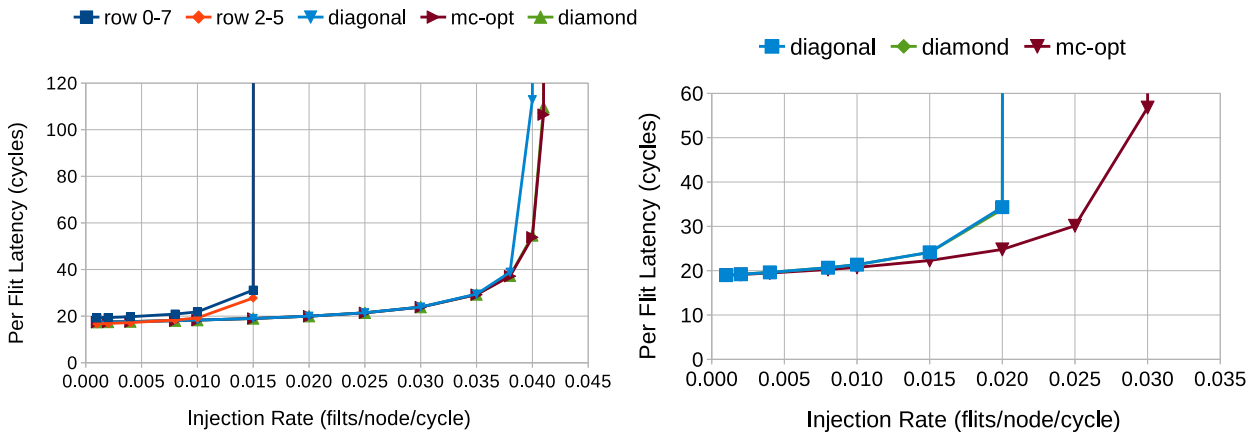
We express the model using GAMS [1] and solve it using Gurobi [42]. Figure 4.2 shows some of the possible placements for $k = 8$. The `diamond` placement, shown in Figure 4.2d, was reported as the best placement by Abts *et al*. Figure 4.2e, here on termed as `mc-opt`, is the placement obtained as a solution to the model when R and K are both set to 1. The same placement is optimal for both mesh and torus networks. Note that many other placements are also optimal.

4.1.5 Evaluation of the Model

Our model takes a very simplified view of an on-chip network. To validate that the simplified view is sufficient for the purpose of pruning the design space, we carry out different simulation-based experiments. All experiments use 8×8 mesh and torus networks.

Performance with Equi-probable Read and Write Synthetic Traffic

Our first experiment uses a detailed on-chip network simulator, which is part of gem5 [19] and is based on GARNET [6]. During the simulation, all the cores inject request packets into the network at a fixed rate. The request addresses are generated in a manner so that the requests are



(a) Mesh Network (b) Torus Network

Figure 4.3: Average Latency versus Injection Rate (Uniform Random Traffic)

uniformly distributed amongst the memory controllers. The memory controllers then respond to these requests, thus completing the request-response loop. Statistical results are collected after running the simulation for $2,000,000$ cycles.

Figure 4.3 shows the plots of average flit latency versus the rate of request injection. Figure 4.3a is for a mesh network, while Figure 4.3b is for a torus network. From the graphs, we conclude that for the mesh network, `mc-opt` has performance similar to that of `diamond` and `diagonal`. For the torus network, `mc-opt` supports about 50% higher request injection rate before saturation, compared to `diagonal` and `diamond`. It also provides lower average latency. Thus, the design obtained from our optimization-based model works well when the traffic is uniformly distributed.

Performance with SPEC CPU2006 Applications.

In our second experiment, we evaluate the designs by simulating applications from the SPEC CPU2006 benchmark suite [48] on the gem5 simulator. Such an experiment evaluates the efficacy of the design suggested by optimization-based model in a close to realistic situation.

We briefly describe our experimental setup. We chose ten applications from the suite, namely,

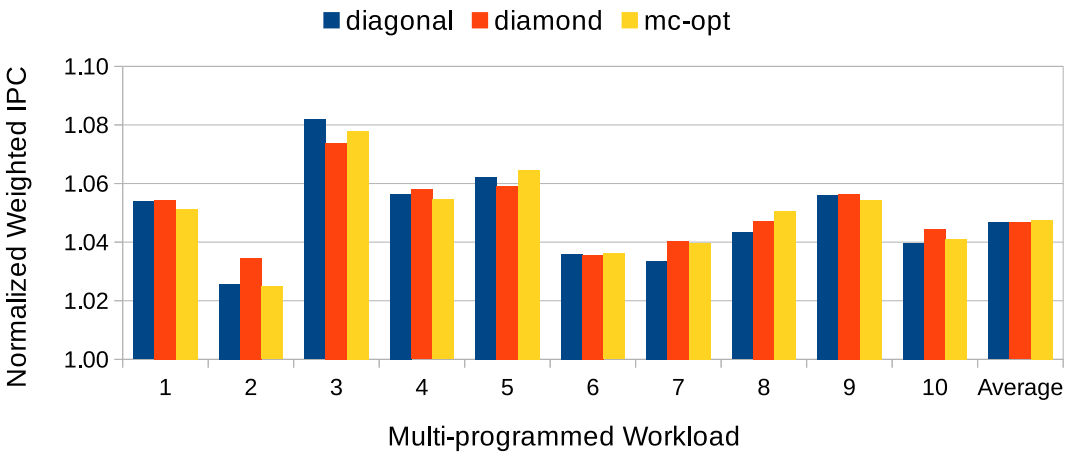
Parameter	Value
Processors	64 out-of-order cores operating at 2GHz
L1 I Cache	32 KB, 2 way set associative, 2 cycle latency
L1 D Cache	64 KB, 2 way set associative, 2 cycle latency
L2 Cache (Private)	2 MB, 8 way set associative, 10 cycle latency
Main Memory	48 GB, 16 DDR-style controllers, addressed using bits 15:12.

Table 4.1: Processor and Cache Simulation Parameters

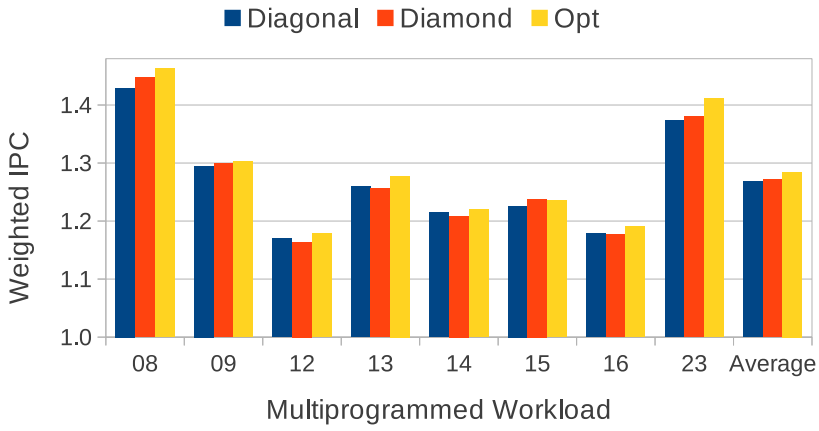
astar, lbm, mcf, milc, omnetpp, libquantum, leslie3d, soplex, sphinx3, and *GemsFDTD*. We created checkpoints for these applications after skipping first $100\ billion$ instructions. For each simulation, we dropped two applications and simulate eight copies of each of the eight remaining applications. These applications were mapped to the cores randomly. Each simulation was allowed to run till every core had executed at least $25\ million$ instructions. On average about $4\ billion$ instructions were simulated in each simulation. We begin statistics collection after the first $2\ million$ cycles complete. Relevant simulation parameters appear in Table 4.1.

While there are a total of $\binom{10}{2}$ different combinations of the applications, we present results for only ten of these due to space constraints. We show the average over all the combinations. In Figure 4.4, we present the weighted speedup [89] obtained for the different application combinations by the different designs. The speedup is normalized to that for the `row 0-7` design. For the mesh network, `mc-opt` performs as well as `diagonal` and `diamond` designs. All these designs have, on average, about 4.5% better performance than the `row 0-7` design. For the torus network, `mc-opt` performs better than `diagonal` and `diamond` on almost all the combinations. On average, `mc-opt` improves performance by slightly more than 1% over `diagonal` and `diamond`.

Thus, the solution obtained from the model works well even in real situations. We expect `mc-opt` to perform even better on workloads that have higher cache miss rates and hence access



(a) Mesh Network



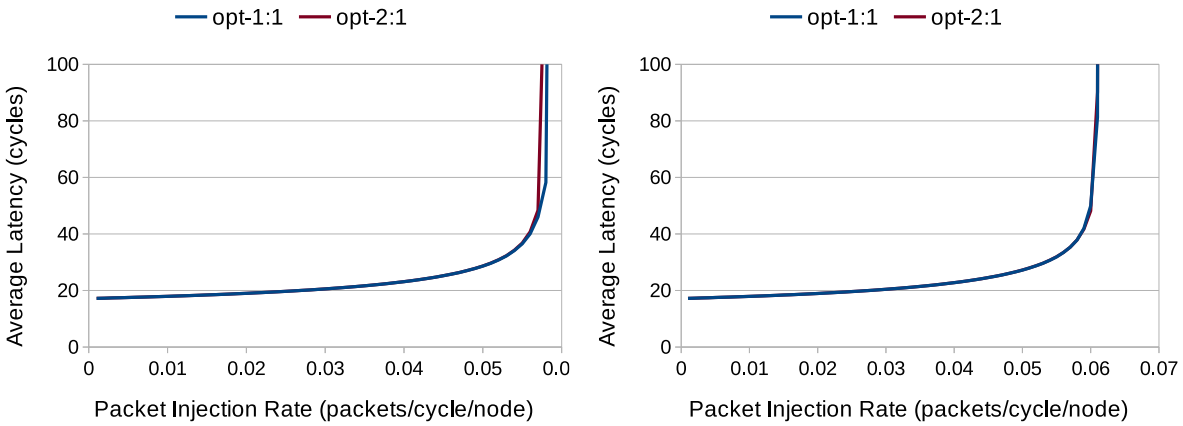
(b) Torus Network

Figure 4.4: Weighted speedup for different application combinations and controller placements normalized to `row 0-7`.

the main memory more frequently.

4.1.6 Why `mc-opt` Performs Better

For a torus network, `mc-opt` performs better than `diamond` and `diagonal` (which are *isomorphic* for this network). Our synthetic simulations show that `mc-opt` better spreads out the traffic. `diagonal` exhibits traffic hotspots around the clusters of four adjacent memory controllers, resulting in only 6.25% of the links observing at least 90% of the maximum channel



(a) Simulated traffic with read to write ratio 2:1 (b) Simulated traffic read to write ratio 1:1

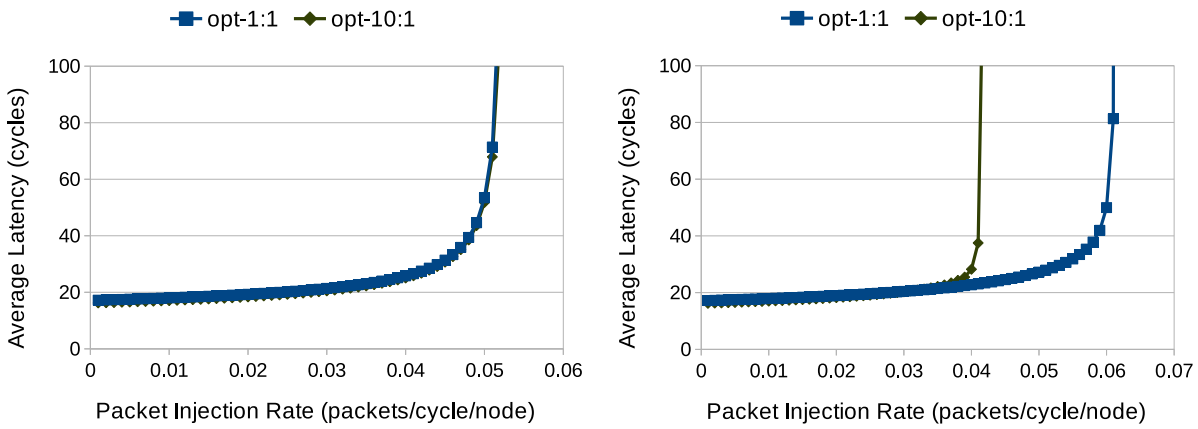
Figure 4.5: Average Latency versus Injection Rate (Uniform Random Traffic). The figure on the right is for traffic with read and write requests being equi-probable. The figure on the left is for traffic with read requests twice as probable as write requests. Both figures show optimal designs obtained by solving the model. $opt-1:1$ is for the setting $R=1$ and $opt-2:1$ is for the setting $R=2$.

load. In contrast, about 25% of links in `mc-opt` observe at least 90% of the maximum channel load. By more uniformly distributing memory controllers, `mc-opt` achieves more uniformly distributed traffic.

4.1.7 Effect of Read and Write Traffic Ratio on Controller Placement

To gauge the effect of ratio of read versus write traffic on placement of controllers, we also solved our optimization model for two more settings of parameters: $(R=2, K=5)$ and $(R=10, K=5)$. The first of these settings represents the typical ratio of $2:1$ for reads to writes, while the second one is a rather extreme setting of $10:1$ for reads to writes ratio. A design obtained for read to write ratio $R:1$ would be referred to as $opt-R:1$. For example, the design with read to write ratio $2:1$ is referred to as $opt-2:1$.

We evaluated the designs obtained using the on-chip network simulator described earlier. We show the results in Figure 4.5 and 4.6. As can be seen in the figures, we observed almost no



(a) Simulated traffic with read to write ratio 10:1

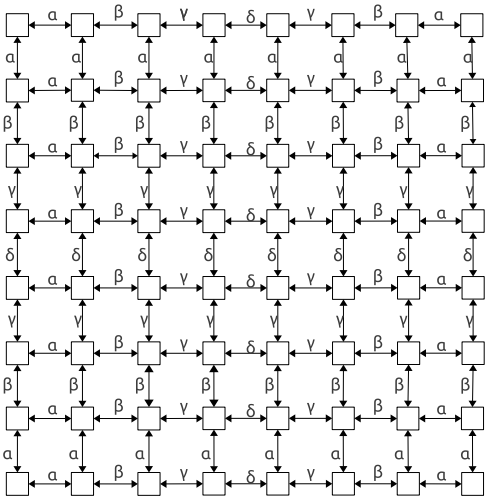
(b) Simulated traffic read to write ratio 1:1

Figure 4.6: Average Latency versus Injection Rate (Uniform Random Traffic). The figure on the right is for traffic with read and write requests being equi-probable. The figure on the left is for traffic with read requests ten times as probable as write requests. Both figures show optimal designs obtained by solving the model. $opt-1:1$ is for the setting $R=1$ and $opt-10:1$ is for the setting $R=10$.

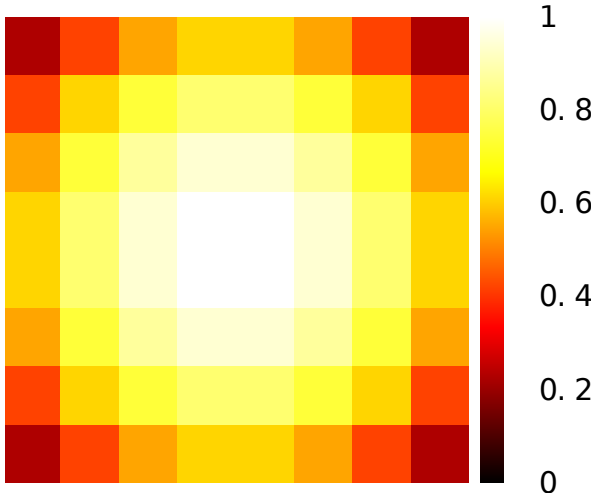
difference in performance of the designs $opt-1:1$ and $opt-2:1$. We think this is because the difference in the volume of traffic from cores to memory controllers and from memory controllers to cores is not that significant for the designs considered. We also observed that $opt-1:1$ performs slightly worse than $opt-10:1$ on traffic with reads ten times as probable as writes. But $opt-10:1$ performs significantly worse on traffic with read to write ratio 1:1. This is because $opt-10:1$ is optimized for the case when most of the traffic flows from memory controllers to cores.

4.2 Resource Allocation in On-chip Network

Mishra *et al.* observed that in an on-chip network based on a mesh topology, the routers and links closer to the center of the mesh handle more traffic than those near the edge of the mesh [70]. This is represented graphically in Figure 4.7. The graph on the left shows the distribution of traffic across the network links, while the graph on th right shows traffic distribution across



(a) Distribution of traffic on different links. Each link is marked with the amount of traffic it observes relative to the links that observe the maximum amount of traffic. $\alpha = 0.4375$, $\beta = 0.75$, $\gamma = 0.9375$, $\delta = 1$.



(b) Heatmap for traffic distribution in a Mesh Network under uniform traffic assumption. Each cell represents a router. Its color represents the amount traffic passing through the router. Lighter colors represent more traffic.

Figure 4.7: Traffic Distribution in a Mesh Network.

routers. But the routers, whether at the center or at the edge, are typically provided with the same amount of buffers and virtual channels. Mishra *et al.* therefore raised the question on how the resources should be distributed across the routers as it appears that allocating each router the same resources may not be optimal. These observations were made assuming that the routing protocol is deterministic and dimension-ordered, and that the traffic is uniformly distributed over all the paths permissible under the routing protocol.

As an answer, Mishra *et al.* designed a heterogeneous mesh network composed of two types of links—*wide* and *narrow*—and two types of routers – *big* and *small*. To arrive at a distribution of these links and routers for an 8×8 mesh network, they evaluated several thousand design configurations for a 4×4 mesh network. The best configurations for the 4×4 mesh network were extrapolated to the 8×8 network. Figure 4.11 shows the designs proposed by Mishra *et al.*

An 8×8 mesh network requires 64 routers. Assuming $16-$big and $48-$small routers, there are $\binom{64}{48} \approx 4.89 \times 10^{14}$ possible ways in which these routers can be placed in the network. If the

assumption that routers can only be *big* and *small* is dropped, the solution space explodes further. Given the size of the solution space, we use mathematical optimization for solving this resource allocation problem. In a nutshell, the designer has a fixed budget of resources and he/she needs to figure out the optimal resource division amongst the routers. The following sections describe a Mixed Integer Linear Program (MILP) for the problem.

4.2.1 Assumptions Made in the Model

To facilitate comparison, our optimization model relies on similar assumptions as Mishra *et al*. The resources available for designing an on-chip network are—links, virtual channels and buffers. We can vary the bandwidth of the physical links connecting different routers, the number of virtual channels and the number of buffers associated with each physical link. A physical link can be either *wide* or *narrow*. A wide link has twice the bandwidth of a narrow link. The sum total of the bandwidths of all the links in the network is bounded by the *link budget*. This budget is only for the links between the routers. The links between the cache controllers and the routers are assumed to be thin. The total number of virtual channels and buffers across all routers are bounded by the *vc budget* and the *buffer budget* respectively. There is an upper bound on the number of virtual channels and buffers that can be associated with a physical link. Apart from these assumptions on the resources, the traffic distribution in the network is assumed to be known a priori.

4.2.2 Notation Used in the Model

Our model appears in Figure 4.8. For each uni-directional link l in the mesh network, we introduce the following variables. W_l is an integer variable denoting whether link l is wide or narrow. A narrow link has width of 1, while a wide link has a width of 2. VC_l is an integer variable denoting the number of virtual channels associated with the physical link l. B_l is an

$$
\begin{aligned}
& maximize && t + s + w && \\
& subject\ to && && \\
& \sum_l W_l && \leq \text{ link budget} && (4.4)\\
& \sum_l VC_l && \leq \text{ vc budget} && (4.5)\\
& \sum_l B_l && \leq \text{ buffer budget} && (4.6)\\
& W_l && \geq LoadOnLink(l) * t && (4.7)\\
& VC_l && \geq LoadOnLink(l) * s && (4.8)\\
& B_l && \geq LoadOnLink(l) * w && (4.9)\\
& \text{where } l \text{ varies over all} && \text{the links in the network} && \\
& W_l \in \{1, 2\}, && VC_l, B_l \in \mathbb{Z}_+, s, t, w \in \mathbb{R}_+ &&
\end{aligned}
$$

Figure 4.8: MILP for Distributing Network Resources

integer variable for the number of buffers associated with link l. Variables t, s, and w denote the bandwidth, virtual channels, and buffers allocated to a link that has a load of one unit. Last of all, the function $LoadOnLink(l)$ gives the load on link l. This function is an input to the problem as the traffic distribution is known beforehand.

4.2.3 Description of the Model

Now we describe our model in detail. Equations (4.4), (4.5) and (4.6) impose the budgetary constraints on the design. Intuitively, each link should have resources in proportion to the traffic that goes over that link. Such a distribution would allocate more resources to the routers at the center of the mesh network compared to the routers on the edge. Equations (4.7), (4.8) and (4.9) impose this proportionality constraint. We use $\geq$ (greater than or equals) relation in these constraints because W_l, VC_l and B_l are integer-valued and the terms appearing on the right-hand side are non-negative reals. The objective of the model is to maximize the sum: $t + s + w$. We chose this objective for two reasons:

- As mentioned before, we would like to assign resources to each link in proportion to the traffic going over that link. In our model, variables t, s and w represent the constants of proportionality for the distribution of different resources. Equations (4.7), (4.8) and (4.9) ensure the proportionality constraints. But the resources, left after the proportional distribution, can be distributed arbitrarily. For example, if t, s and w were set to 0, all the resources can be distributed arbitrarily amongst the links. On the other hand, if these variables took the maximum value they can possibly have, then almost all the resources will be distributed proportionately.

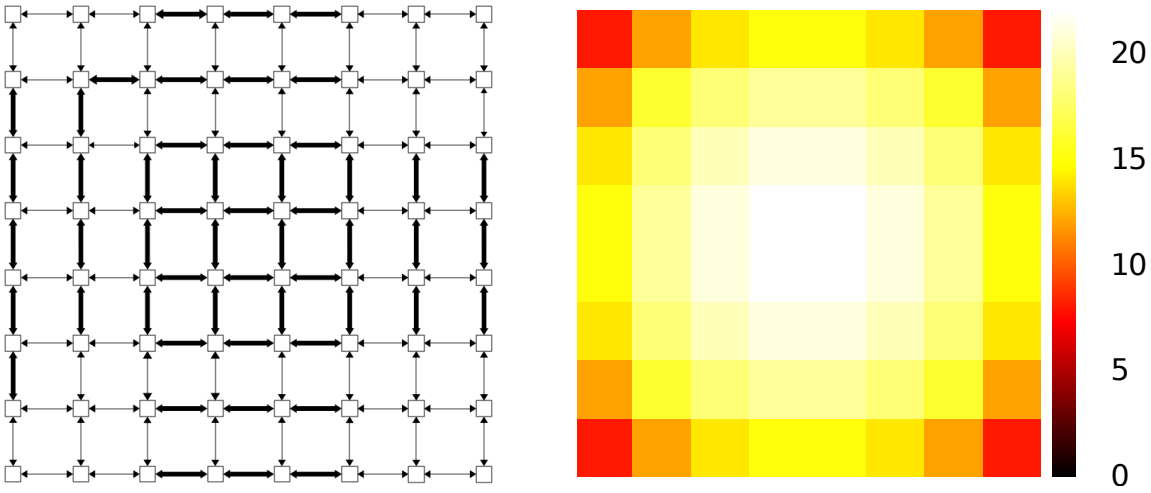
- The model also minimizes the maximum amount of traffic handled by a unit amount of resource. Consider equation (4.8). We can rewrite it as: $\frac{1}{s} \geq \frac{LoadOnLink(l)}{VC_l}$. Since s is maximized, $\frac{1}{s}$ is minimized. Thus the maximum load that a single virtual channel needs to bear is minimized.

From the model, it can be seen that t, s, and w are independent of each other. Therefore, each of them will be independently maximized. In fact, the overall problem could be split into three independent optimizations.

4.2.4 Design Obtained from the Model

We solve the model under certain assumptions. The network is assumed to be an 8×8 mesh network, with dimension-order routing. Routers have up to six input and output ports. Two of these ports are for the private cache and the shared memory controllers respectively. The other four ports are for the four different directions. Lastly we assume that the traffic is distributed uniformly i.e. each cache controller generates requests for any of the memory controllers with equal probability. Thus the traffic that a link needs to service is directly proportional to the number of paths using that link.

Figure 4.9 shows the design obtained on solving the model. Figure 4.9a shows the distribution



(a) Link Width Distribution. Wide links have been shown wider compared to narrow links.

(b) Virtual Channel Distribution. Each cell represents a router. Its color represents the number of virtual channels assigned to the router.

Figure 4.9: `net-opt` Network Design

of wide (256-bit) and narrow (128-bit) links across the 8×8 mesh network. All the links which observe heavy load (marked with γ and δ in Figure 4.7a) are wide in our design. Figure 4.9b shows a heat map for the distribution of the virtual channels amongst the routers. It can be observed that the number of virtual channels go down on moving from the center of the mesh towards the periphery. Since the traffic is assumed to be uniformly distributed, the routers at the center of the mesh service more traffic than those at the periphery, as shown in Figure 4.7b. Therefore, the design has most resources assigned to the routers at the center, and least to the ones farthest from the center. In the sequel, this design will be referred to as the `net-opt` design.

Note that there is slight asymmetry in the links in Figure 4.9a. Wider links, other than those in the middle three rows and columns, were assigned more bandwidth because of the available budget. If symmetry is important (e.g., to simplify layout), it is straight forward to add symmetry constraints.

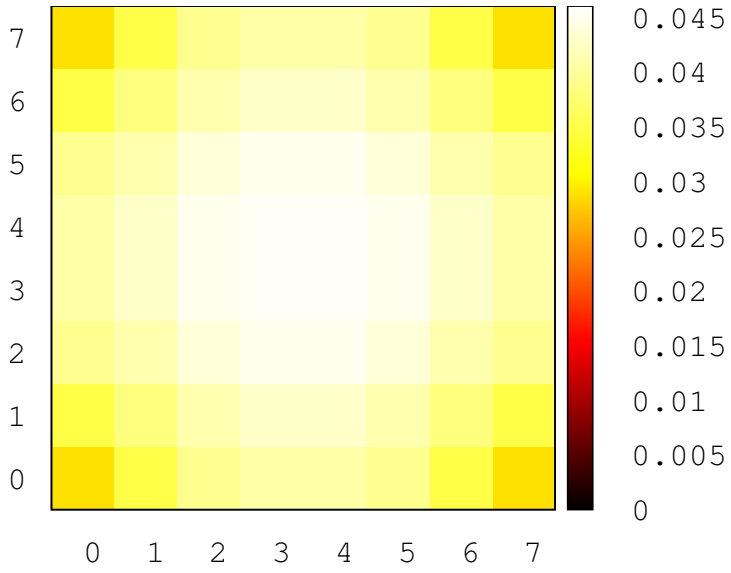


Figure 4.10: Virtual Channel to Traffic Distribution. Each cell represents a router. Its color represents the ratio of virtual channels assigned to the router and the traffic that goes through that router.

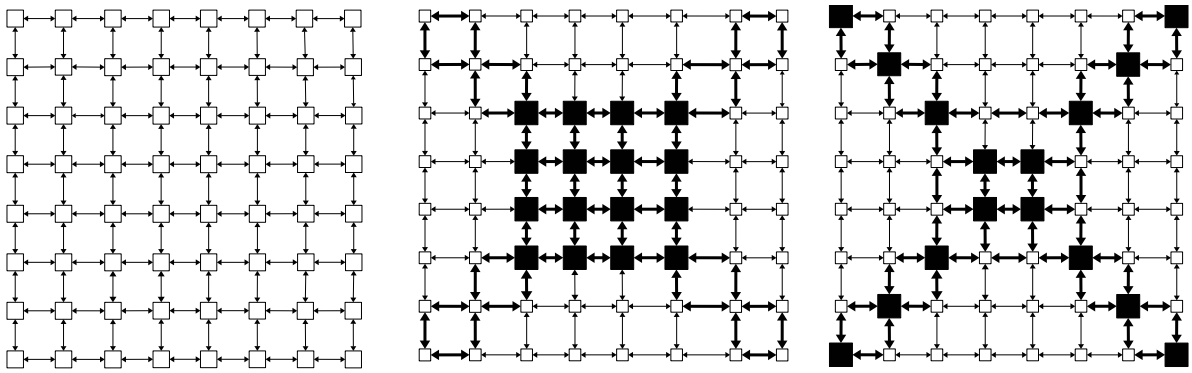
4.2.5 Evaluation of the Design

We compare our design against the three designs analyzed by Mishra *et al*. These designs have been shown in Figure 4.11.

- `base`: All routers are provided equal resources. Each port has 3 virtual channels per virtual network. Each link is 192-bit wide.

- `center`: 16 routers in the center are big (6 virtual channels/ virtual network / port). The rest are small (2 virtual channels / virtual network / port). Half the links are wide (256-bit), while others are narrow (128-bit) as shown in Figure 4.11b.

- `diagonal`: Routers along the diagonals are big, rest are small. Again, half the links are wide and others are narrow. The design appears in Figure 4.11c. Mishra *et al* found this design to be the best.



(a) base (b) center (c) diagonal

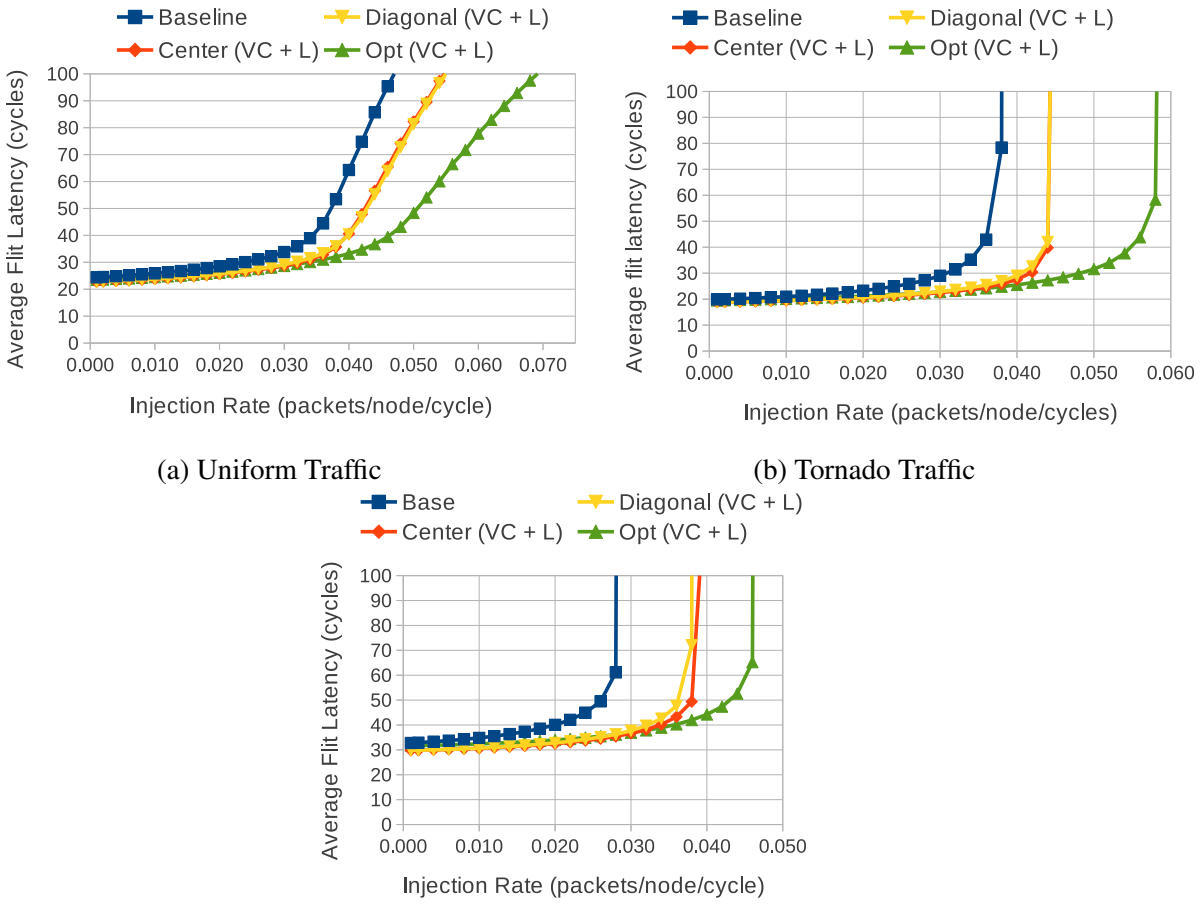
Figure 4.11: Different Network Designs. Each box represents a router. Shaded routers are big. Wider links have been shown wider.

All the designs in our experiments adhere to the same resource limits. Further, we assume that `base` operates at a clock frequency of 2.20 GHz and all the other designs operate at 2.07 GHz. This assumption is required since the routers in `center`, `diagonal` and `net-opt` are possibly bigger than the routers in `base` design. The frequencies assumed are same as those assumed by Mishra *et al*. Since `net-opt` makes use of routers that are smaller than the big routers in `center` and `diagonal` designs, our assumption on the frequency of these routers should be sufficient.

Performance With Synthetic Traffic.

We evaluate the candidate designs using the network simulator available in gem5 [19]. The standard simulator supports a homogeneous network in which each physical link has the same width and the same number of virtual channels. We modified the simulator so that different links can be assigned different number of virtual channels and link widths.

In each simulation, the L2 controllers inject request packets into the network at a fixed rate. A memory controller, on receiving a request, injects the response packet for that request, thus completing the request-response loop. The simulation is allowed to run 2,000,000 cycles. At the end of the simulation, we note the average latency involved in transporting a flit from its source



(a) Uniform Traffic

(b) Tornado Traffic

(c) Bit-Complement Traffic

Figure 4.12: Average Latency vs Request Injection Rate for different request patterns

to the destination. The experiment is repeated with different rates of request injection, and with three different request patterns—uniform random, tornado, and bit-complement [27].

Figure 4.12 shows the graphs for the average latency of a flit versus the rate of request injection into the network for these request patterns. As can be seen in the graphs, for all three request patterns, the `net-opt` design has a higher saturation bandwidth and provides lower average latency compared all other designs. This experiment shows the design obtained from the optimization-based model works well when a detailed network simulation is carried out, even when the uniformity assumption for the traffic is not observed. `net-opt` performs better than `diagonal` and `center` because the latter two designs distribute resources better than `base`

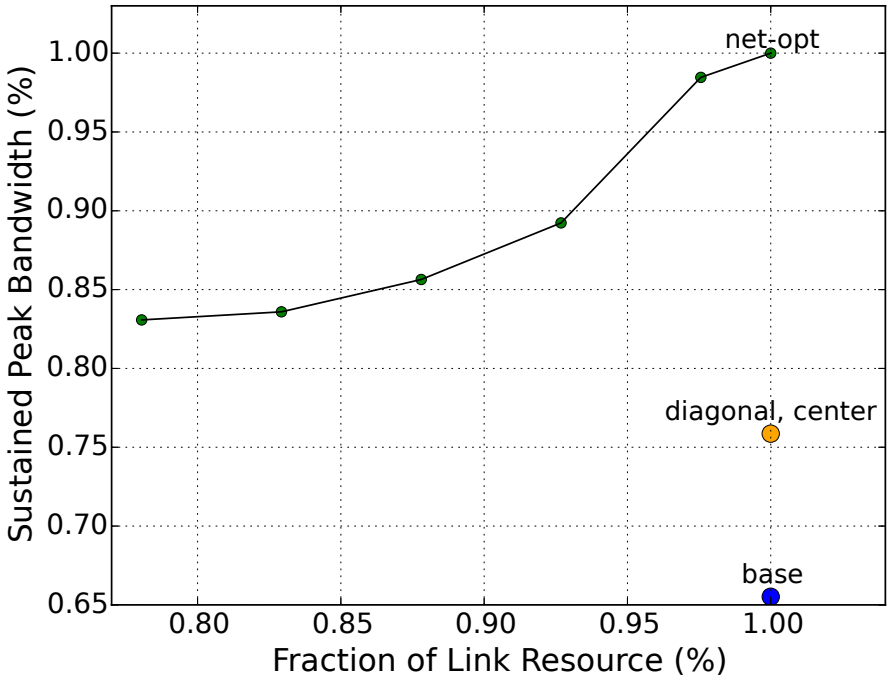


Figure 4.13: Sustained peak bandwidth as a function of the link resources used.

but do not do so completely. In particular all the routers and links in rows and columns: 2, 3, 4 and 5 of the mesh network observe more traffic than rest as shown in Figure 4.7. `net-opt` provides most resources to these routers and links. In contrast, `diagonal` and `center` provide more resources to routers and links lying along the diagonals.

With the same setup as described above, we also simulated network designs that use lesser amount of link resources by reducing the number of wider links. In Figure 4.13, we show how the peak sustained bandwidth varies as a function of the amount of link resources. As we can see from the figure, optimal distribution of the link resources provides for similar or better performance even when the number of wider links used in the design is 30% less compared to `diagonal` and `center`.

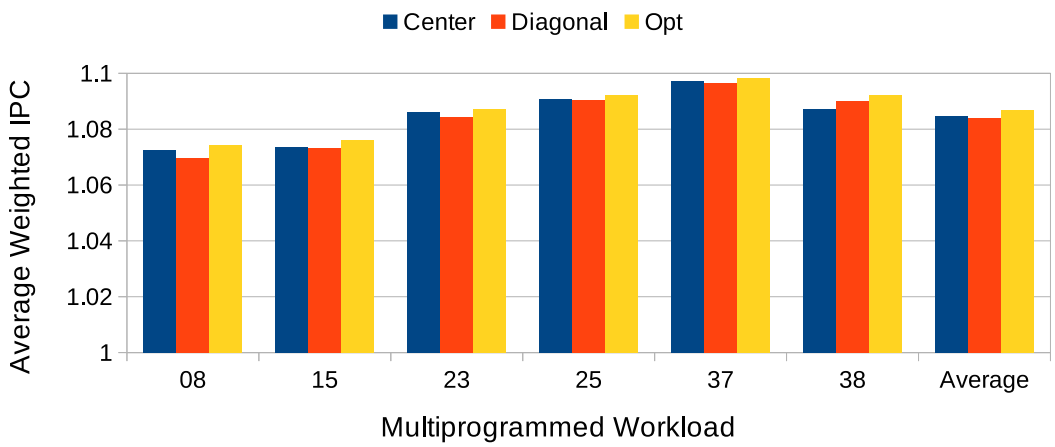


Figure 4.14: Weighted Speedup for different application combinations and network designs normalized to `base`.

Performance with SPEC CPU2006 Applications

We further evaluated the designs by simulating applications from the SPEC CPU2006 benchmark suite. The experimental setup was described in Section 4.1.5. In Figure 4.14, we present the weighted speedup [89] obtained for the different application combinations by the different designs. The speedup is normalized to that for the `base` design. `net-opt` performs as well as `diagonal` and `center` designs. All these designs have, on average, about 8% better performance than the `base` design. `net-opt` fails to perform better than `diagonal` and `center` as the network utilization is very low and all the designs observe nearly zero load latency for all the workloads.

4.3 On-chip Network Design Combined With Placement of Controllers

In section 4.1, we formulated an optimization model for placing memory controllers in an on-chip network. In section 4.2, we formulated an optimization model for allocating resources to links and routers in an on-chip network. That model assumes the traffic distribution as an input.

$$
\begin{aligned}
&\text{minimize} && W + S + T && \\
&\text{subject to} && && \\
&\sum_l W_l && \leq \text{ link budget} && (4.10)\\
&\sum_l VC_l && \leq \text{ vc budget} && (4.11)\\
&\sum_l B_l && \leq \text{ buffer budget} && (4.12)\\
&W_l * T && \geq LoadOnLink(l) && (4.13)\\
&VC_l * S && \geq LoadOnLink(l) && (4.14)\\
&B_l * W && \geq LoadOnLink(l) && (4.15)\\
&\text{where } l \text{ varies over} && \text{all the links in the network} && \\
&LoadOnLink(l) && = \sum_{(x,y,x',y'):l \in Path(x',y',x,y)} I_{xy}(R+K) && \\
&&& + \sum_{(x,y,x',y'):l \in Path(x,y,x',y')} I_{xy}(RK+1) && \\
&\text{where } l \text{ varies over} && \text{all the links in the network} && (4.16)\\
&\sum_{(x,y)} I_{x,y} && = \text{ total memory controllers} && (4.17)\\
&I_{x,y} + I_{x,y+1} && \leq 1 && (4.18)\\
&I_{x,y} + I_{x+1,y} && \leq 1 && (4.19)\\
&I_{x,y} \in \{0,1\}, \;\; S,T,W && \in \mathbb{R}_+, W_l \in \{1,2\}, VC_l, B_l \in \mathbb{Z}_+ &&
\end{aligned}
$$

Figure 4.15: Non-linear Program for the Combined Problem

But this distribution depends on the placement of the memory controllers, which in turn may depend upon the network design. Ideally we would like to place memory controllers and allocate network resources in a single combined problem. This may result in a better design than obtained by solving the two problems sequentially.

With this intuition, we formulated the optimization model presented in Figure 4.15. The model we formulated has some constraints that are non-linear. As we discussed in section 2.3.4, non-linear models are hard to solve. Thus the combination of the two design problems is potentially harder to solve.

4.3.1 Analysis of the Model

Most of the variables and the constraints used in the model have been described in sections 4.1 and 4.2. We describe the additional variables and constraints:

- Variables T, S and W represent the load per unit bandwidth, virtual channel, and buffer respectively. They are the inverses of the variables t, s and w introduced in section 4.2.2. Note that while t, s and w can be independently optimized, T, S and W cannot be since they are linked by variables I_{xy}.

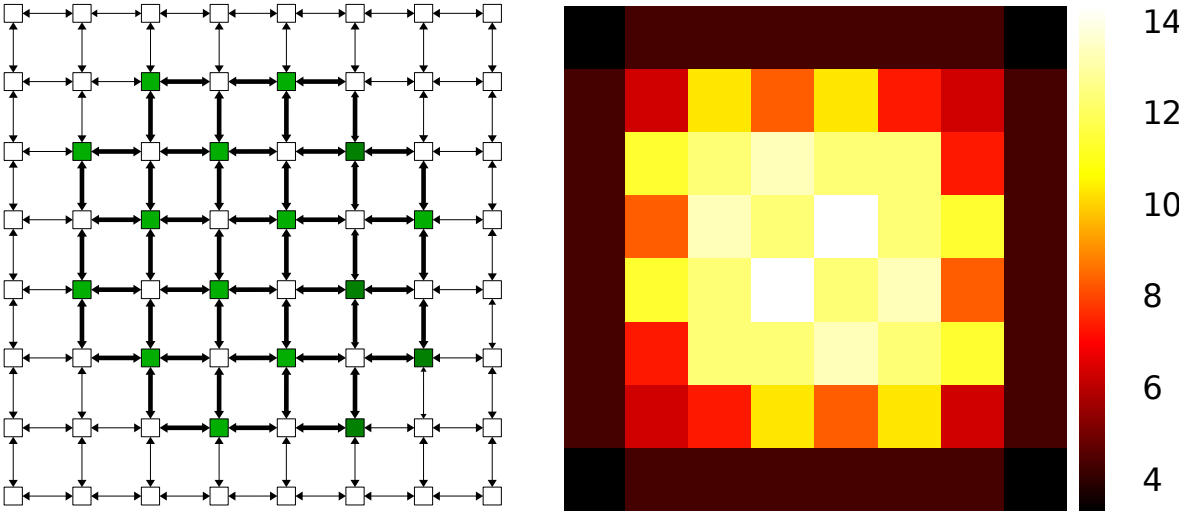
- Constraints (4.18) and (4.19) together avoid designs in which adjacent cells in the mesh network have memory controllers. We observed that without these (or similar) constraints, all the memory controllers are placed in the center portion of the chip. Such a design is likely to cause congested wire routing and thermal hot spots, hence the additional constraints in the model.

- Since memory controller placement determines traffic patterns, the function $LoadOnLink(l)$ is no longer an input. Also, we assume that caches are private to the cores and the on-chip network is only used for communication between the last level (private) caches and the memory controllers. Hence constraint (4.16) only accounts for traffic to and from memory controllers.

- Constraints (4.13), (4.14), and (4.15) have terms where two variables are being multiplied. These product terms make these constraints non-linear. Hence, the model is a *mixed integer non-linear program* (MINLP).

- The reasons behind the choice of objective function are similar to the ones discussed in section 4.2.3.

Our model is **non-convex** because of two reasons. In our model, variables: W_l, VC_l, B_l and $I_{x,y}$ are constrained to be integer-valued and models with integrality constraints are non-convex.



(a) Distribution of Memory Controllers and Link Widths

(b) Virtual Channel Distribution

Figure 4.16: `com-opt` design for the Combined Problem

Even if we were to drop the integrality requirements, the model remains non-convex because constraints (4.13), (4.14), and (4.15) are not convex. These three constraints are of the form: $f(x, y, z) = z - xy \leq 0$. If we choose, for example, $a = (1, 1, 1)$ and $b = (2, 2, 4)$, then $f(a) = f(b) = 0$ but with $\theta = 0.5$, $f(\theta a + (1 - \theta)b) = 0.25$ so that

$$f(\theta a + (1 - \theta)b) \leq \theta f(a) + (1 - \theta) f(b)$$

is not satisfied. Thus f is not convex, implying that constraints (4.13), (4.14), and (4.15) and the overall model is not convex.

4.3.2 Solving the Model

We solved the model under the same assumptions as made in sections 4.1 and 4.2. The network topology is assumed to be an 8×8 mesh, with dimension-ordered routing, and uniformly distributed traffic.

No polynomial-time methods are known for solving non-convex MINLPs [22]. We explored the solution space for our problem using Baron [96], an NLP solver. The solver uses a branching scheme to optimize over the (bounded) feasible set. A good initial solution is very useful for pruning the search tree in order to find the globally optimal solution. To get closer to the optimal design, we seeded the solver with multiple initial designs, and experimented with the bounds on different variables. We used the placements described in section 4.1 as initial designs for the solver. The solver ultimately computed designs with improved objective function values. Figure 4.16 shows the best design we found.

As we show in Section 4.3.3, this is in fact the globally optimal solution. However, in the given time limit, Baron could not prove the global optimality of this solution. Rather, by relaxing the non-convex constraints and iteratively performing a branching procedure, it provides a lower bound on the optimal value of the objective function. The design shown in Figure 4.16 has an objective value which is 14% higher than the algorithmically computed lower bound. In comparison, the design of Mishra *et al.* (shown in Figure 4.17) has an objective value 55% higher than the lower bound.

4.3.3 Linearized Model

Our initial model has a linear objective function and nonlinear constraints with the form: $f(x, y, z) = z - xy \leq 0$. These are called bilinear terms since each is the product of two variables and the whole model is called a Bilinear Program (BLP). Moreover, each bilinear term is the product of a continuous variable and an integer variable, which can be converted to linear terms using binary expansion [41]. After this conversion, we can solve the model using MILP techniques.

In our model, constraints (4.13), (4.14), and (4.15) are of the form: $z \leq xy$, where x is a continuous variable and y is an integer variable. Further, both x and y are non-negative and bounded from above i.e. $0 \leq x \leq a$ and $0 \leq y \leq b$. The set of points satisfying such a constraint

can be represented as:

$$\mathcal{P} = \left\{(x, y, z) \in \mathbb{R}_+ \times \mathbb{Z}_+ \times \mathbb{R} : \; z \leq xy, \; x \leq a, \; y \leq b\right\} \tag{4.20}$$

Using y's binary expansion, we get: $y = \sum_{i=1}^{k} 2^{i-1} w_i$. Here w_i are 0-1 integer variables and $k = \lfloor \log_2 b \rfloor + 1$. Define $\mathcal{B}$ as:

$$\begin{aligned} \mathcal{B} = \Big\{(x, y, z, w, v) \in \mathbb{R} \times \mathbb{Z} \times \mathbb{R} \times \{0,1\}^k \times \mathbb{R}^k : \\ y = \sum_{i=1}^{k} 2^{i-1} w_i, \; y \leq b, \; z \leq \sum_{i=1}^{k} 2^{i-1} v_i, \; 0 \leq v_i \leq a w_i, \\ v_i \leq x, \; v_i \geq x + a w_i - a, \text{for all } i \in \{1, \cdots, k\} \Big\} \end{aligned} \tag{4.21}$$

It can be shown that $\mathcal{P} = \text{Proj}_{x,y,z}(\mathcal{B})$. Here $\text{Proj}_{x,y,z}$ represents the projection operator that maps (x, y, z, w, v) to (x, y, z). Since $\mathcal{B}$ does not have any nonlinear term in its representation, it is an exact linearization of $\mathcal{P}$. We linearize our model by replacing constraints (4.13), (4.14), and (4.15) with the constraints used in defining $\mathcal{B}$. With this new model, it took CPLEX [3], another solver for MILP, less than 5 minutes to prove that the design presented in Figure 4.16 is in fact optimal.

4.3.4 Optimal Design

Figure 4.16 illustrates the design—referred to as `com-opt`—obtained from solving the model. Figure 4.16a shows the distribution of the memory controllers (solid boxes) and the wide (bold) and narrow (thin) links in the network. Figure 4.16b shows a heat map indicating the distribution of virtual channels. Note the significant differences from the optimal solutions to the individual problems in Figures 4.2e and 4.9.

We also obtained the design `center-opt` as the solution to our model from Figure 4.15 without constraints (4.18) and (4.19). In this design, the memory controllers are placed in the center of the chip. The objective value for `center-opt` is better than that of `com-opt` by 1.25%. We therefore expect `center-opt` to perform marginally better than `com-opt`.

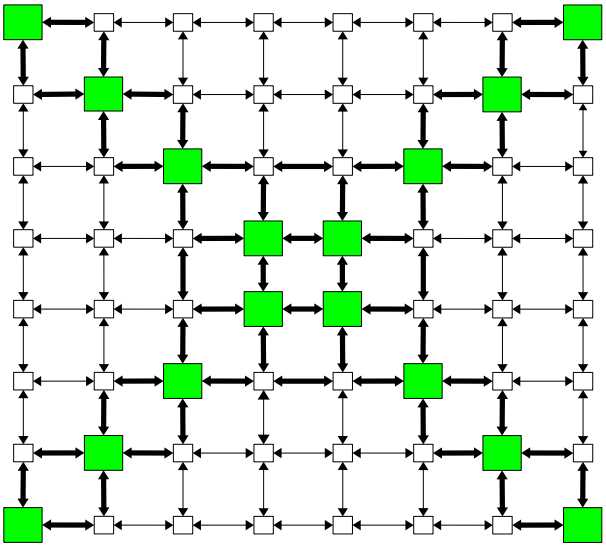


Figure 4.17: Distribution of Memory Controllers, Buffers and Link Widths for `diagonal`.

4.3.5 Sensitivity of the Optimal Design vis-à-vis the Objective Function

The objective function in the combined model essentially gives equal consideration to links, virtual channels, and buffers. To test the sensitivity to this assumption, we generalized the objective function to provide weights to the variables S, T and W. Specifically, the objective from Figure 4.15 was changed to $\omega W + \psi S + \tau T$. We considered the following cases:

1. $\omega = \psi = \tau = 1$: this case has been evaluated in section 4.3.4 and `com-opt` was obtained as the optimal design for the model.

2. $\omega = 2, \psi = \tau = 1$: the design `com-opt` is optimal for this setting as well.

3. $\omega = 1, \psi = 2, \tau = 1$: `com-opt` found to be optimal.

4. $\omega = 10, \psi = 1, \tau = 1$: `com-opt` found to be optimal.

5. $\omega = 1, \psi = 10, \tau = 1$: `com-opt` was the best design found, but the solver CPLEX was not able to prove that it is the optimal design. The optimality gap i.e. the gap between the objective of the best design found and the best under-estimate, was 18%.

4.3.6 Evaluation of the Designs

The objective function value for `com-opt` and `center-opt` is better compared to that for the design proposed by Mishra *et al*. Their design, referred to as `diagonal` and shown in Figure 4.17, places memory controllers on the diagonal nodes, along with big routers (6 virtual channels / port) and wide links. Routers on non-diagonal nodes are small (2 virtual channels / port) and use narrow links. To validate this result, we compare these designs using simulation. We also evaluate two other designs: `row0-7+opt` and `diamond+opt`. The placement of memory controllers for these is shown in Figure 4.2. The distribution of link widths and virtual channels was obtained by solving our optimization model with variables for memory controllers ($I_{x,y}$) set to fixed values. All designs use the same number of virtual channels, wide and narrow links, and buffers.

Synthetic Traffic.

We evaluated the designs using the NoC simulator described in section 4.2.5. In Figure 4.18, we present the average latency experienced by a flit as a function of the rate of request injection into the network. From the graph, we can observe that `com-opt`, `center-opt` and `diamond+opt` support about 30% higher saturation bandwidth and provide lower latency compared to `diagonal`.

SPEC CPU2006 Benchmark.

We also evaluated the designs by simulating combinations of SPEC CPU2006 benchmarks [48] on the gem5 simulator as described in section 4.1.5. In Figure 4.19a, we present the weighted speedup obtained for the different combinations. The speedup is normalized to the `diagonal` design. It can be observed that, on average, `com-opt` improves weighted speedup by 5.4% and `center-opt` improves weighted speedup by 6.7% over `diagonal`.

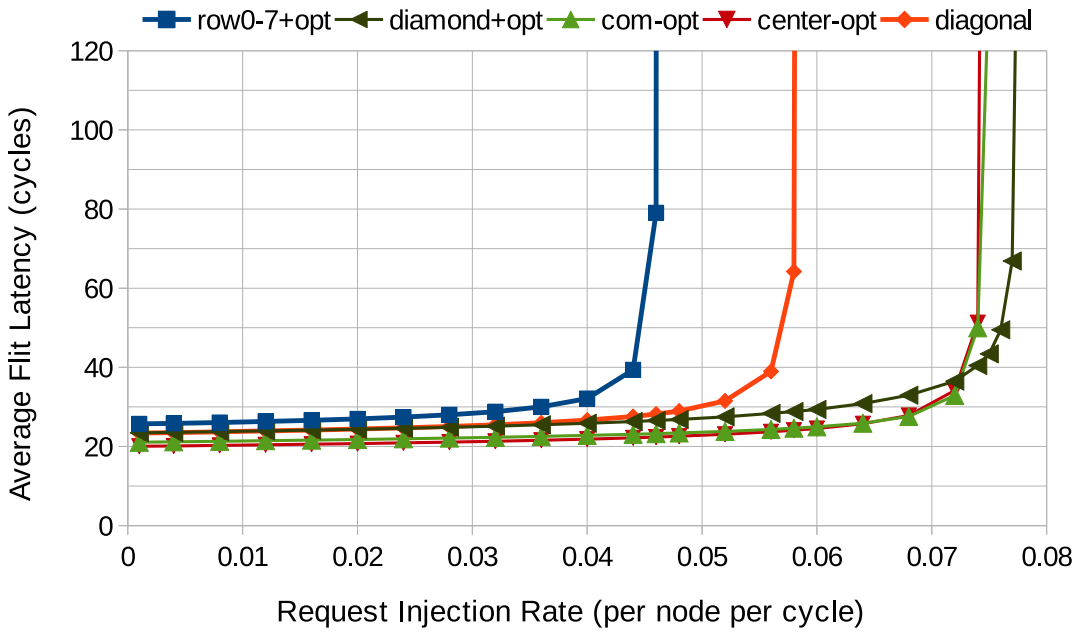
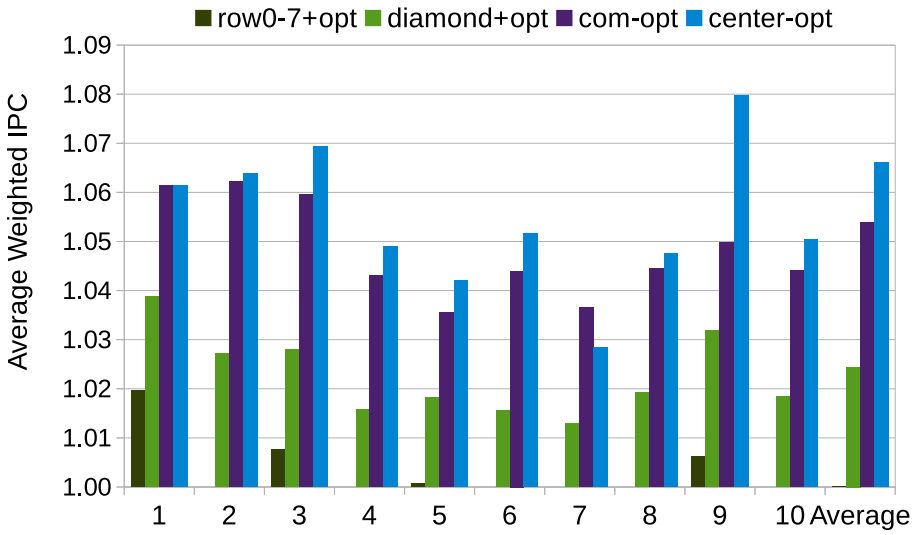


Figure 4.18: Graph for Average Flit Latency vs Request Injection Rate for synthetically generated uniform random traffic.

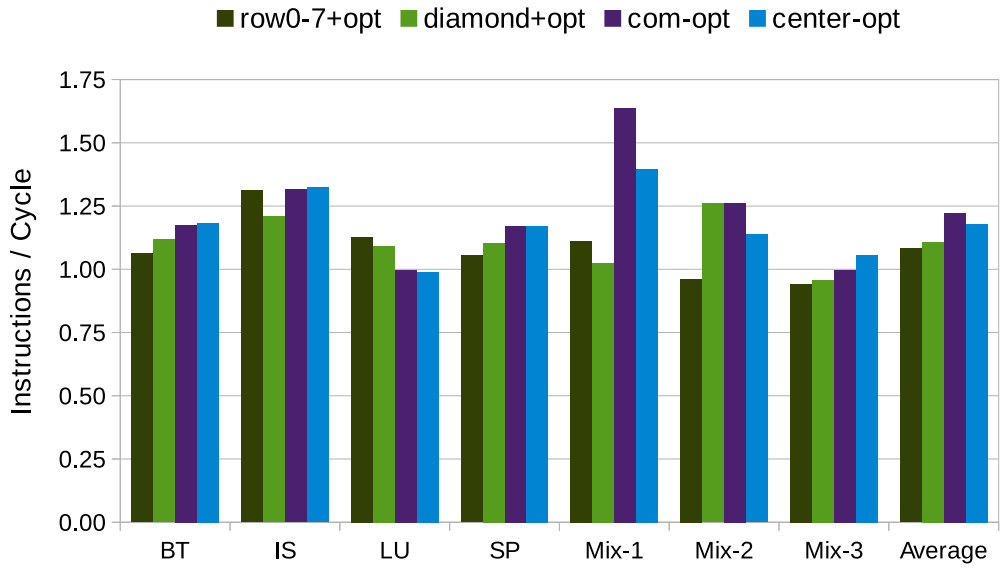
NAS Parallel Benchmark.

We further evaluated the designs by executing applications from NAS Parallel Benchmarks (NPB) [13]. There are eight applications that comprise NPB. We provide results only for the ones that gem5 can execute properly. Others fail due to lack of support for x87 instructions in gem5. Each application is executed with 4 threads running on adjacent cores. Thus, we run 16 applications for a particular simulation to cover all the 64 cores. We also ran some workloads with a mix of these applications. The applications were mapped randomly to the cores for such workloads. Note that in these simulations, the on-chip network traffic is not uniformly distributed since the cache-to-cache transfers take place amongst caches private to cores executing threads from the same application.

We executed each workload five times with different random seeds to harmonize any effects arising due to scheduling of threads. In Figure 4.19b, we present the improvement in IPC obtained for different designs. The speedup is normalized to the `diagonal` design. It can be

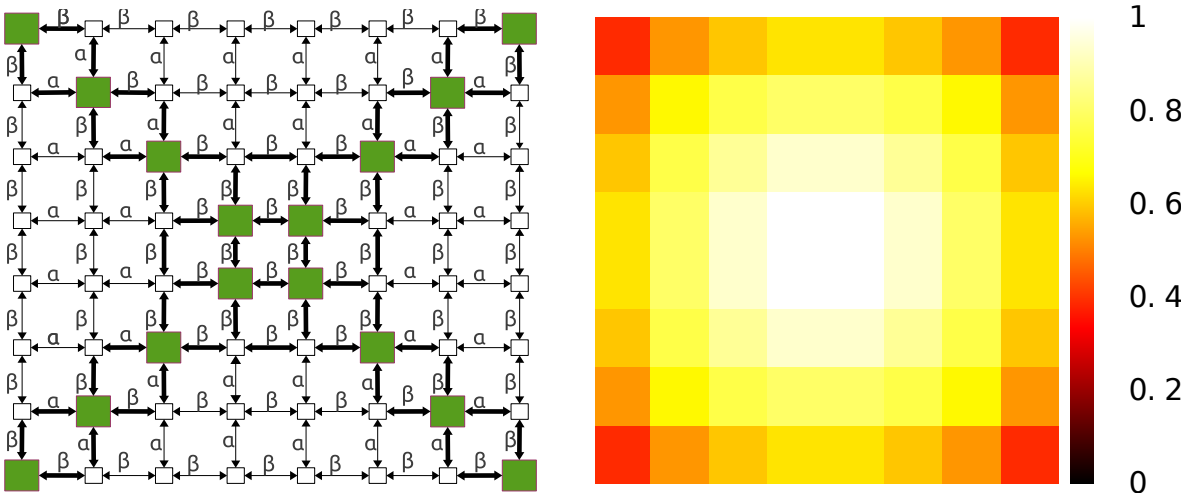


(a) Weighted Speedup, normalized to `diagonal` for multiprogrammmed workloads composed from SPEC CPU2006 Applications.



(b) Instructions per cycle, normalized to `diagonal` for multithreaded workloads composed from NAS Parallel Benchmarks.

Figure 4.19: Graphs from experimental evaluation of designs for the combined problem.



(a) Traffic load on different links. Links marked α are lightly loaded, while links marked β are heavily loaded.

(b) Per Router Traffic Load

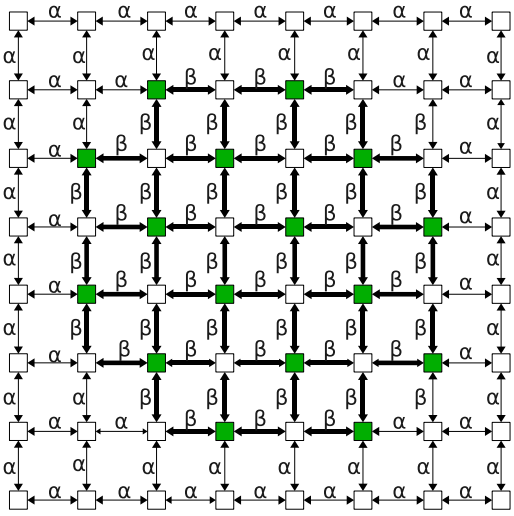
Figure 4.20: Traffic Distribution for the `diagonal` Design

observed that `com-opt` performs about 22% better than the `diagonal`.

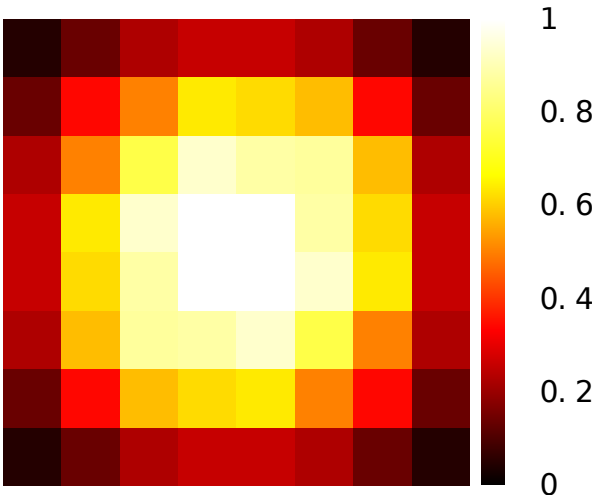
4.3.7 Analysis of the Designs

`com-opt` performs better than `diagonal` for two reasons:

- The zero load latency of `com-opt` is slightly lower than that of `diagonal`, as illustrated in Figure 4.18. Hence, under low traffic intensity `com-opt` results in lower latency.
- For higher traffic intensity `com-opt` performs better since it does a better job of matching network resources to network traffic. Figures 4.20 and 4.21 illustrate the traffic load observed by the links and the routers for the two designs. Links marked α observe less traffic compared to links marked β. Ideally, the α links should be narrow and the β links should be wide. But `diagonal` has 16 wide α and 32 narrow β links, indicating a mismatched resource allocation. Similarly, `diagonal` assigns more virtual channels to routers near the corners even though they observe less traffic. By simultaneously placing memory



(a) Traffic load on different links. Links marked α are lightly loaded, while links marked β are heavily loaded.



(b) Per Router Traffic Load

Figure 4.21: Traffic Distribution for the `com-opt` Design

controllers and allocating network resources, `com-opt` eliminates these resource mismatches.

4.4 Why Use Mathematical Optimization

In this section, we discuss reasons for preferring mathematical optimization over some of the other approaches that can be employed for designing manycore processors.

4.4.1 Scalability

For the memory controller placement problem, the genetic-algorithm approach of Abts *et al.* [5] required running heuristic-based algorithms on multiple machines over more than a day [36]. Our approach found an optimal solution to the 64-processor, 16-port problem in less than a minute on a four core, eight thread machine. The same machine took less than 15 minutes to solve the 100-processor, 20-port problem.

Similarly, Mishra *et al*. [70] evaluated only several thousands of the possible 4×4 mesh network designs to arrive at what they thought was the best design for an 8×8 mesh network. Evaluating each design took 5-10 minutes, requiring multiple machines in parallel to reduce the total time [69]. Since the design space is huge, exploration via exhaustive / randomized simulation is impractical. In comparison, our optimization-based model took less than a minute for computing the optimal solution (under the given constraints).

Thus, mathematical optimization can potentially reduce the time required for design space exploration and is better suited for exploring larger and more complex design problems.

4.4.2 Theoretical Bounds on Performance

While mathematical optimization can find an optimal solution in many cases, there are problems where it cannot. For many such problems, it is possible to compute a theoretical bound on the performance objective using problem relaxations. Problem relaxations are enlargements of the feasible set that are computationally more tractable. For integer linear programs, this is done by removing the integrality constraints. For non-convex programs, convex under and over estimators can bound the optimal objective function value. These bounds can be used to compare the *unknown* optimal design and the computed (feasible) candidate designs.

4.4.3 Flexibility

Extrapolation and divide-and-conquer based approaches require the design problem to be symmetric. These approaches use the symmetry to reduce the complexity of the solution space. For example, the methods used by Abts *et al.* [5] and Mishra *et al.* [70] required the on-chip network to have the same dimension along the X and the Y axes. This was because the authors were trying to extrapolate results from a smaller 4×4 on-chip network to a larger 8×8 on-chip network. While optimization-based models also benefit from using symmetry, given the speed of

computation, it may not be necessary to restrict the design space to symmetric designs only. For example, in optimization-based model, the X and the Y dimensions can differ. Similarly, it is not required that the number of memory ports be an integral multiple of the dimensions of the mesh / torus network. This was required in the original approach, again because of the need for extrapolation.

4.5 Related Work

We have made a case for using mathematical optimization for solving design problems in the field of Computer Architecture. Our approach yields designs that perform better than those obtained via other techniques used before. Our work is highly influenced by the works of Abts *et al*. [5] and Mishra *et al*. [70]. In this section, we highlight other recent works in the area and discuss why our work is different.

On-chip Networks: Designing on-chip networks is a fertile area of research. Significant effort has been devoted towards improving the performance and reducing the cost of on-chip networks. Prior works have proposed adaptive routing [64, 65], bufferless NoC [45, 72], and QOS support for NoCs [40]. These works alter the dynamic behavior of the network. Our approach focuses on carrying out the best possible allocation of the resources at the design time. Ben-Itzhak *et al.* [15] proposed using simulated annealing for designing heterogeneous NoCs. We discussed earlier that mathematical optimization works better when the problem can be expressed as a linear / integer linear / convex program. Our work shows that this is true for certain problems related to NoC-design. Heuristic-based approaches may work better for non-convex design problems.

On-chip Placement: Prior works [10, 100] have focused on figuring out the best mapping for applications and data on to cores and memory controllers at execution time, while we have presented a design time approach. A lot work for on-chip placement has been done in the SoC

domain. These works [104, 51] propose using genetic algorithms for generating solutions.

Xu *et al*. [103] also tackled the problem of placing memory controllers for chip multiprocessors. They solved the problem for a 4×4 CMP through exhaustive search. To find the best placement for the 8×8 problem, they exhaustively search through solutions obtained by stitching solutions obtained for the 4×4 problem. This reduces the solution space that needs to be searched, but the idea is not generic. It assumes that the chip can be divided in to smaller regions and solutions for the smaller regions can composed to get optimal solutions for larger regions. This may not hold true in general. Our approach of using mathematical optimization does not rely on any such assumption.

Theory: Network design problem has been widely studied in theoretical computer science [66, 37], particularly with respect to designing distribution, transportation, telecommunication and other types networks. These works mainly focus on designing approximation algorithms for the different variants of the network design problem and on analyzing their theoretical complexity. We focus on on-chip network design and on-chip placement.

4.6 Future Work and Extensions

The design problems discussed in the chapter have several possible variations that can be handled with out significant changes to the approach presented. The formulations provided can handle different types of resources, routing protocols, and topologies. If the network traffic is not distributed uniformly, a weighting function can be introduced to associate a weight with each path. For example, in the network design problem, if the memory controllers were located at the corners of the mesh, it is likely that links near the corners would observe more traffic. This fact can be captured by providing higher weights for the paths that use those links compared to the paths that don't. Similarly, in asymmetric designs, it is likely that the traffic distribution would not be uniform. Formulations would have to be adapted to take care of the asymmetry.

Designing for multiple different traffic distributions using a stochastic optimization-based approach is also conceivable. The formulations can be extended by adding constraints on the power/energy consumed by the designs. Designers may also impose thermal constraints to avoid hot spots on the chip.

4.7 Conclusion

In this chapter, we solved three problems related to on-chip networks. The solutions were obtained by using mathematical optimization models for searching and pruning the design space. We believe that there are many other computer architecture problems that can benefit from optimization. It might be worthwhile to explore whether there are optimization techniques that are better suited for problems in computer architecture.

Chapter 5

Latency-optimized Models for On-chip Resource Distribution

5.1 Introduction

Prior work [5, 70] and the work presented in chapter 4 presented models for design problems related to on-chip networks for tiled processors like one shown in Figure 5.1. Those models distribute memory controllers and design heterogeneous on-chip networks so as to minimize the maximum bandwidth that a link in the on-chip network may need to service.

For many applications performance is not bound by the communication bandwidth provided by the on-chip network, but by the latency of communication it provides. Hence, the models presented in prior work may not yield good designs for processors for such latency-sensitive applications. In this work, we present models that optimize for the quantity (latency or bandwidth) which affects the performance more. By setting a single parameter—traffic input rate—a designer can explore the design space and choose models that balance bandwidth utilization and communication latency. We present designs obtained as solutions to these for a range of traffic input rate and evaluate the designs using synthetic simulations. Our show that the proposed

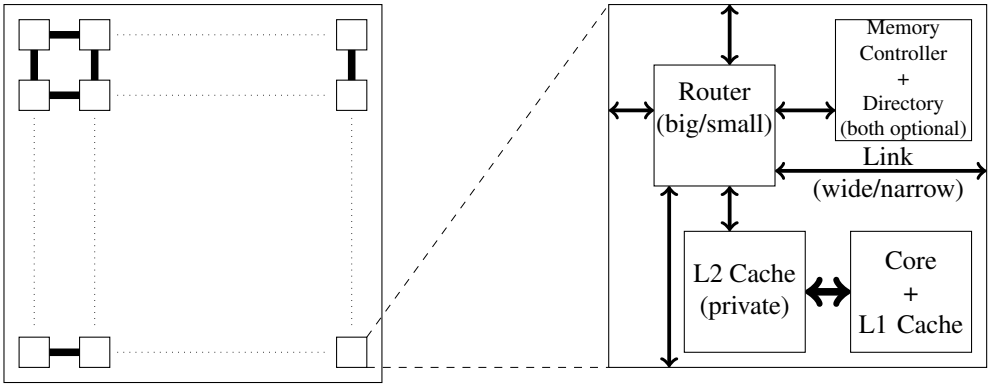


Figure 5.1: Processor Layout (not to scale). On the left is an $n \times n$-tiled processor. The tiles are connected using a mesh/torus network. The figure on the right shows a single tile. The caches are kept coherent using a directory-based coherence protocol. The memory controller is optional. Routers for different tiles may have different amounts of resources. The links in between tiles can be wide or narrow.

models work well for the entire range of the possible traffic input rate.

5.1.1 Our Contributions

We summarize our contributions in this chapter below:

- We developed two optimization models for placement of memory controllers and distribution of network link bandwidth. The first is for minimizing the maximum latency of communication. The second is for minimizing the average latency of communication. These models take as input a single parameter: traffic input rate. Designers interested in these design problems (or similar ones) can set this parameter and explore the design space. We believe designers would find these crisp models useful.

- We initially framed the models using non-linear functions. This resulted in models that were hard to solve computationally. We therefore employ two different techniques to formulate linear models that approximate the non-linear ones. These linear models are far more tractable. We believe that these linearization techniques have not been demonstrated in optimization models for problems related to computer architecture. Our models act as

examples for designers interested in developing models involving similar non-linear functions.

- Using these linear models, we present optimal designs for a range of traffic input rate and evaluate the designs using synthetic simulations. Our results show that the models perform well over the entire range of the traffic input rate. Thus, designers can use the same model, irrespective of the application space, to compute designs that are likely to perform well in practice.

5.1.2 Organization of the chapter

In the next section, we review the work of Abts *et al.*, Mishra *et al.* and that presented in chapter 4. We also discuss the limitations of the work presented in the chapter 4. In section 5.3, we present our model that minimizes the maximum latency a flit may have on any path. In section 5.4, we present our model for minimizing the average latency a flit has to incur when traversing the on-chip network. We discuss research related to our work in section 5.5. We conclude the chapter in section 5.6.

5.2 Overview of Previous Work

Abts *et al.* [5] introduced the problem of placing memory controllers within the on-chip network for multi-core processors. They observed that different placements provide different bandwidth and therefore a prudent placement can lead to better performance. Abts *et al.* used Genetic Algorithms and exhaustive simulations to arrive at their proposed designs.

Mishra *et al.* [70] observed that in an on-chip network based on a mesh topology, the distribution of traffic is non-uniform. The routers and links closer to the center of the mesh handle more traffic than those near the edge of the mesh. But typically all the routers are provided with

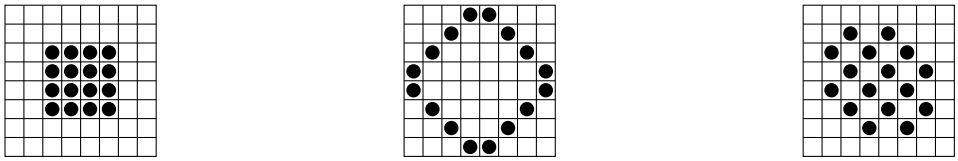
$$
\begin{aligned}
&\text{minimize} && T && \\
&\text{subject to} && && \\
&\sum_{l} W_l && \leq \quad \text{link budget} && (5.1)\\
&T && \geq \quad \frac{LoadOnLink(l)}{W_l} && (5.2)\\
&\text{where } l \text{ varies over} && \quad \text{all the links in the network} && \\
&LoadOnLink(l) && = \sum_{(x,y,x',y'):l \in Path(x',y',x,y)} I_{xy}(R+K) && \\
& && + \sum_{(x,y,x',y'):l \in Path(x,y,x',y')} I_{xy}(RK+1) && \\
&\text{where } l \text{ varies over} && \quad \text{all the links in the network} && (5.3)\\
&\sum_{(x,y)} I_{x,y} && = \quad \text{total memory controllers} && (5.4)\\
&I_{x,y} + I_{x,y+1} && \leq \quad 1 && (5.5)\\
&I_{x,y} + I_{x+1,y} && \leq \quad 1 && (5.6)\\
&I_{x,y} \in \{0,1\}, \quad T && \in \quad \mathbb{R}_+, W_l \in \{1,2\} &&
\end{aligned}
$$

Figure 5.2: Non-linear Program for the combined problem in Chapter 4

the same amount of buffers and virtual channels. Mishra *et al.* used exhaustive simulations to explore the design space. They presented designs with non-uniform distribution of resources to the routers and showed that those designs perform better than the uniform design.

Optimization Model from Chapter 4. The distribution of traffic in an on-chip network depends on the placement of the memory controllers, which in turn may depend upon the network design. Ideally we would like to place memory controllers and allocate network resources in a single combined problem. This may result in a better design than obtained by solving the two problems sequentially. We solved this problem using mathematical optimization in section 4.3 of the previous chapter.

In Figure 5.2, we reproduce a portion of the model presented in Figure 4.15 in the previous chapter. The model is for distributing a fixed number of memory controllers and bandwidths (W_l) for links in an on-chip network for a tiled processor like one shown in Figure 5.1. Constraints



(a) `center` (b) `diamond` (c) `com-opt`

Figure 5.3: Designs evaluated for the combined problem in Chapter 4.

(4.10) and (4.17) limit the total amount of resources that can be devoted to links and memory controllers. Constraint (4.16) defines the load on a link as sum total of flits that pass over that link due to communication between every pair of memory controller and core that use that particular link for communication. The model aims at minimizing the variable T. This variable is defined by constraint(4.13) as: $T \geq \frac{LoadOnLink(l)}{W_l}$, where l varies over all the links in the network. Describing in words, we set T to be at least as much as the maximum over links, the ratio of the load on a link and its bandwidth. The model then tries to minimize T. Thus, the model minimizes the maximum traffic-volume that a link may need to service. We show the designs we evaluated in Figure 5.3. The design `com-opt`, shown in Figure 5.3c, was obtained from solving the model in Figure 5.2.

Shortcomings in the previous work. We think there are two shortcomings in the previous work from Abts *et al.*, Mishra *et al.* and us:

- it assumes that increase in the load serviced by a link linearly affects the performance of the link. This is not true in general. Even for the simple Markovian queueing models, the latency of service increases non-linearly with increase in load [39].

- it assumes that the applications only care about the bandwidth offered by the on-chip network. There are many applications that do not communicate at high enough rate. Such applications benefit more from lower latency communication network.

In the following sections, we present work that do away with these shortcomings. To address

the first shortcoming, we introduce functions that model nonlinear behavior observed in a link's performance as the traffic input rate increases. To address the second one, we introduce two new objective functions that seek to minimize latency. The two objective functions capture the maximum and the average latency respectively. These two changes together help strike a balance between the latency and the bandwidth of communication offered by the on-chip network.

5.3 Min-Max Latency Optimization Model

In this section, we present the *MinMax Latency* model. The model, shown in Figure 5.4, minimizes the maximum latency that a flit on any path has to incur. The latency that a flit needs to incur is modelled as sum of the length of the path (proxy for time spent in traversing a link) and the time that a flit has to spend in buffers in routers while waiting for its turn to traverse links. The waiting time for a flit at a router has been modelled by assuming each router to be a $M/D/1$ queueing system. This means that inter-arrival time between successive flits is assumed to exponentially distributed and the router is assumed to require a fixed amount of time once it starts serving a particular flit. This assumption on distribution of inter-arrival times is just for the purpose of illustration. We later on discuss why this assumption can be dropped.

5.3.1 Notation Used in the Model

In the model shown in Figure 5.4, each memory controller is denoted as a point (x, y) on the 2-D plane. Similarly, a core is referred to with coordinates (x', y'). For each (x, y), $I_{x,y}$ is a binary variable denoting whether a memory controller is placed at (x, y). The set $Path(x, y, x', y')$ contains all the links l that are used for going from (x, y) to (x', y'); since the routing protocol is deterministic, $Path()$ is an input to the problem. We further assume that the read to write requests have a ratio of $R : 1$ and that a packet with data is K-times the size of a packet with no data. We use λ to denote the average rate at which a core makes requests to and receives

$$\text{minimize } Z \text{ subject to}$$

$$\sum_{l} BW_l \leq \text{link budget} \tag{5.7}$$

$$\sum_{(x,y)} I_{x,y} = \text{total memory controllers} \tag{5.8}$$

For all links l in the network

$$LoadOnLink(l) = \sum_{(x,y,x',y'):l \in Path(x',y',x,y)} \lambda(R+K) I_{xy} + \sum_{(x,y,x',y'):l \in Path(x,y,x',y')} \lambda(RK+1) I_{xy} \tag{5.9}$$

$$\lambda_l \geq \frac{LoadOnLink(l)}{(1+BW_l)} \tag{5.10}$$

$$W_l \geq \frac{\lambda_l}{2\mu(\mu-\lambda_l)} \tag{5.11}$$

For all possible values of (x,y) and(x',y')

$$Z_{x,y,x',y'} \geq I_{xy}\Big(\sum_{l \in Path(x',y',x,y)} \Big(\frac{1}{\mu} + W_l\Big)\Big) \tag{5.12}$$

$$Z_{x',y',x,y} \geq I_{xy}\Big(\sum_{l \in Path(x,y,x',y')} \Big(\frac{1}{\mu} + W_l\Big)\Big) \tag{5.13}$$

$$Z \geq Z_{x,y,x',y'} \tag{5.14}$$

$$Z \geq Z_{x',y',x,y} \tag{5.15}$$

$$I_{x,y} \in \{0,1\}, \quad BW_l \in \{0,1\}$$

Figure 5.4: *MinMax* Optimization Model for Latency

responses from a particular memory controller per unit time.

For each uni-directional link l in the mesh network, we use the following variables. BW_l is a binary variable denoting whether link l is wide or narrow. A narrow link has width of 1, while a wide link has a width of 2. $LoadOnLink(l)$ denotes the load on the link l due to the communication between the cores and the memory controllers. This load depends on the placement of the controllers. λ_l denotes the amount of load that a unit amount of bandwidth has to bear on link l. W_l denotes the average time a flit traversing link l has to wait for. μ denotes the

rate which a router processes flits it receives.

$Z_{x,y,x',y'}$ upper bounds the latency incurred by a flit traversing the path from a memory controller at (x, y) to a core at (x', y'). Similarly, $Z_{x',y',x,y}$ upper bounds the latency incurred by a flit traversing the path from a core at (x', y') to a memory controller at (x, y). Z upper bounds all $Z_{x,y,x',y'}$ and $Z_{x',y',x,y}$.

5.3.2 Description of the Model

There are nine constraints involved in the model.

- Constraint (5.7) impose the budgetary constraint on link bandwidths.
- Constraint (5.8) enforces that a specified number of memory controllers (m) are placed on the chip. $I_{x,y}$ can be either 0 or 1. So when the equation is satisfied, exactly m of the n $I_{x,y}$-variables are set to 1, the rest are 0.
- Constraint (5.9) defines the load on each link in the on-chip network. The first term on the right hand side of the constraint is a sum over all pairs (x, y) and (x', y') such that a memory controller is placed at (x, y) and the path from (x', y') to (x, y) uses link l. This term represents the request traffic going from the cores to the memory controllers. We assume that a core, on average, generates λ requests per unit time per memory controller and that these requests have $R : 1$ ratio for reads and writes. Each read request requires a single flit, while a write request needs K flits. Hence, the total request traffic is proportional to $\lambda(R + K)$. The second term, which represents the response traffic from the memory controllers to the cores, can be interpreted in a similar fashion. The total response traffic is proportional to $\lambda(RK + 1)$, where RK is for flits with response data for read requests and 1 is for flits with acknowledgment for write requests.
- Constraint (5.10) defines λ_l as the total load on l divided by its bandwidth.

- Constraint (5.11) uses the $M/D/1$ queueing model [39] to estimate the average wait time for a flit at a given link l. This constraint assumes that the inter-arrival time between successive flits is exponentially distributed and the router requires a fixed amount of time for serving a flit.

- Constraints (5.12) and (5.13) estimate the total latency along each path from a memory controller to a core and vice versa.

- Constraints (5.14) and (5.15) set the objective variable Z to at least as much as $Z_{x,y,x',y'}$ and $Z_{x',y',x,y}$ i.e. Z should be at least as much as the latency incurred on any path in the on-chip network.

Note that the model presented above is a nonlinear model since the constraints (5.9), (5.10), (5.11), (5.12) and (5.13) involve nonlinear functions. It is generally harder to solve optimization models involving nonlinear functions.

5.3.3 Computational Results with the Non-linear Model

We used GAMS [1] for representing the model and tried solving it using BARON [96]. We solved the model for different values of the parameter $\rho = \frac{\lambda}{\mu}$. Here λ is the average rate which a core sends requests and receives responses and μ is rate at which routers service flits. μ was fixed to 1 in our experiments. Also the link budget was set to zero, meaning all links are narrow. The number of memory controllers was set to 16.

For $\rho = 0.004$ (we discuss the numerical range for ρ in section 5.3.7), without any initial solution being provided, the non-linear model failed to find any solution even after five hours of computation. We then successively initialized the solver with the designs shown in Figure 5.3: `diamond`, `center` and `com-opt`. In all three cases, the solver only improved on the lower bound to about 10.00, while the the upper bound was remained at objective value of the design used for initialization. The objective value for `diamond`, `center` and `com-opt` are 12.41,

11.37 and 12.45 respectively. Thus, with the nonlinear model, we were unable to compute the optimal solution or explore the design space. Therefore, we linearized the model which we present in the next section.

5.3.4 Linearizing the Model

Constraints (5.10), (5.11), (5.12) and (5.13) in the model are non-linear. These constraints make solving the model hard as was observed when we used a non-linear solver for solving the model. We next discuss our approach for linearizing these non-linear constraints.

- Constraint (5.10): This constraint originally looked like:

$$\lambda_l \geq \frac{LoadOnLink(l)}{(1 + BW_l)}$$

Here BW_l is a binary variable: it takes values 0 and 1 only. We replace the original constraint by two new constraints, one for each value. If BW_l takes the value 1, the constraint can be written as:

$$\lambda_l \geq \frac{LoadOnLink(l)}{2} \tag{5.16}$$

Now note that this constraint will also be satisfied even if BW_l equals zero since then: $\lambda_l \geq LoadOnLink(l)$. Now consider the constraint:

$$\lambda_l + BW_l \times M^+ \geq LoadOnLink(l) \tag{5.17}$$

Here M^+ is much bigger than $LoadOnLink(l)$. So when BW_l is zero, the constraint just defined would impose the original constraint (5.10), but would be trivially satisfied if BW_l equals one. Hence, we replace the constraint (5.10) with constraints (5.16) and (5.17).

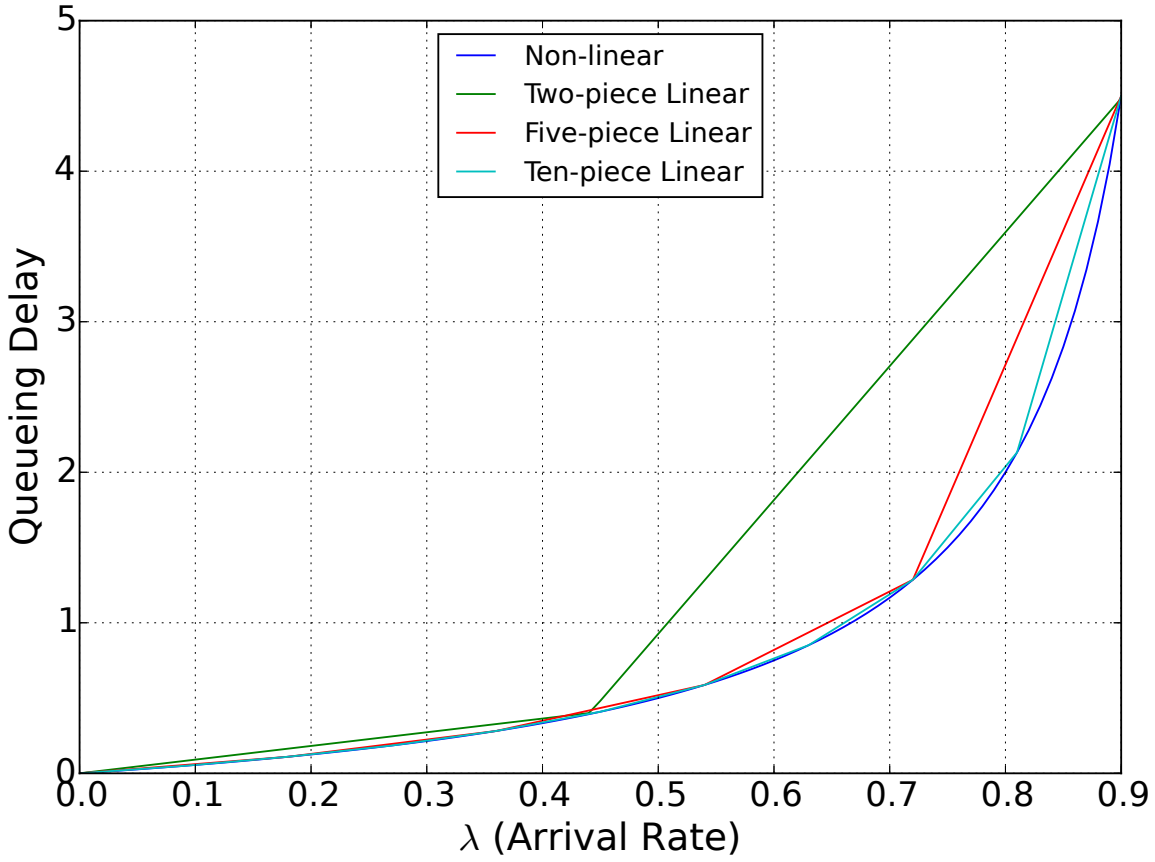


Figure 5.5: Graph showing the queueing delay of an $M/D/1$ model as function of the arrival rate. The service rate μ was assumed to be 1. Also shown are piece-wise linear approximations.

- Constraint (5.11): This constraint originally looks like following:

$$W_l \geq \frac{\lambda_l}{2\mu(\mu - \lambda_l)}$$

In Figure 5.5, we plot the non-linear function represented by the right hand side of this constraint. We also plot $2, 5$ and 10-piece linear functions that over estimate the non-linear function. As is visible from the plot, the 10-piece estimate nearly matches the non-linear function. We use these piece-wise linear functions for linearizing this constraint.

The modelling language GAMS provides a special datatype **SOS2** for representing piece-wise linear functions. SOS2 stands for *Specially Ordered Sets of Type 2*. A variable of

this type is part of a linearly ordered set. At most two elements from set are allowed to be non-zero and those two non-zero variables have to be adjacent to each other in the ordering. Note that these variables can only be used in mixed integer linear (MIP) models. Non-linear functions are not allowed. This forced us to linearize other constraints.

Assume that we would like to have a p-piece linear function to represent a given non-linear function. Then, we introduce a $p+1$ point **SOS2 set** S. We define λ_i to be the value of the i^{th} breakpoint in S and $W_i = \dfrac{\lambda_i}{2\mu(\mu - \lambda_i)}$. For each link l, we also introduce new variables: $b_{l,i}$, where $i \in S$. We introduce new constraints:

$$\sum_{i \in S} b_{l,i} = 1 \tag{5.18}$$

This forces the $b_{l,i}$'s variables to sum up to 1. We re-write constraints (5.16) and (5.17) as:

$$\sum_{i \in S} b_{l,i} \times \lambda_i \geq \frac{LoadOnLink(l)}{2} \tag{5.19}$$

$$\sum_{l,i} b_{l,i} \times \lambda_i + BW_l \times M^+ \geq LoadOnLink(l) \tag{5.20}$$

Here λ_i is the value of λ at the i^{th} breakpoint. We re-write the non-linear constraint (5.11) as:

$$W_l \geq \sum_{i \in S} b_{l,i} \times W_i \tag{5.21}$$

- Constraints (5.12) and (5.13): These constraint come into play only when I_{xy} equals one. So, we can employ the same tactic that we used for linearizing (5.10). Consider the following constraint:

$$Z_{x,y,x',y'} \geq (1 - I_{xy})M^- + \sum_{l \in Path(x',y',x,y)} (\frac{1}{\mu} + W_l) \tag{5.22}$$

$$\begin{aligned}
\text{minimize} \quad & Z \quad \text{subject to} \\
\sum_{l} BW_l \quad & \leq \quad \text{link budget} && (5.23)\\
\sum_{(x,y)} I_{x,y} \quad & = \quad \text{total memory controllers} && (5.24)\\
\text{For all links } l \text{ in the network} & \\
LoadOnLink(l) \quad & = \quad \sum_{(x,y,x',y'):l\in Path(x',y',x,y)} \lambda(R+K)I_{xy} \\
& + \quad \sum_{(x,y,x',y'):l\in Path(x,y,x',y')} \lambda(RK+1)I_{xy} && (5.25)\\
\sum_{i\in S} b_{l,i}\times\lambda_i \quad & \geq \quad \frac{LoadOnLink(l)}{2} && (5.26)\\
\sum_{l,i} b_{l,i}\times\lambda_i + BW_l \times M^+ \quad & \geq \quad LoadOnLink(l) && (5.27)\\
\sum_{i\in S} b_{l,i} \quad & = \quad 1 && (5.28)\\
W_l \quad & \geq \quad \sum_{i\in S} b_{l,i}\times W_i && (5.29)\\
\text{For all possible values of } (x,y) \quad & \text{and} \quad (x',y') \\
Z_{x,y,x',y'} \quad & \geq \quad (1-I_{xy})M^- + \sum_{l\in Path(x',y',x,y)} (\frac{1}{\mu}+W_l) && (5.30)\\
Z_{x',y',x,y} \quad & \geq \quad (1-I_{xy})M^- + \sum_{l\in Path(x,y,x',y')} (\frac{1}{\mu}+W_l) && (5.31)\\
Z \quad & \geq \quad Z_{x,y,x',y'} && (5.32)\\
Z \quad & \geq \quad Z_{x',y',x,y} && (5.33)\\
I_{x,y}\in\{0,1\}, & \qquad BW_l\in\{0,1\}, \;\; b_l \text{ are SOS2}
\end{aligned}$$

Figure 5.6: Linearized *MinMax* Optimization Model for Latency

Here M^- is a highly negative constant value. If I_{xy} equals zero, then the new constraint is trivially satisfied. Otherwise, the original constraint is forced.

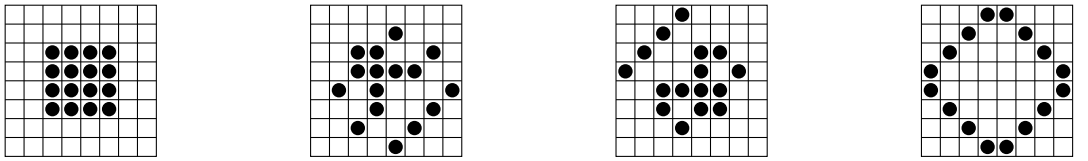
5.3.5 Computational Results with Linearized Model

We provide the complete linearized model in Figure 5.6. We solved the model for different values of the parameter $\rho = \frac{\lambda}{\mu}$. Here λ is the average rate which a core sends requests and receives responses and μ is rate at which routers service flits. μ was fixed to 1 in our experiments. The link budget was set to 0, thus each link was set to be narrow. The number of memory controllers was set to 16. The non-linear function in constraint (5.11) was broken down into 10 linear pieces. For solving the linearized model, we used the tool CPLEX [3]. In Table 5.1 and Figure 5.7, we present the results obtained. In the results, `min-max-opt` represents the best design found by the solver for a given value of the parameter ρ. From Vaish *et al.*, we know `center` performs well when the available bandwidth is high compared to the traffic rate, while `diamond` performs well when available bandwidth is low. The fact that our *Linearized MinMaxLatency* model recovers the `center` design for lower values of ρ and the `diamond` for the higher values provides evidence that the model is working well.

5.3.6 Synthetic Simulation Results

We implemented a C++-based software simulator for simulation of network traffic in an $N \times N$ mesh network. Each processor-core in the network generates traffic at a specified rate. The memory controllers, distributed as per the design being simulated, also generate response traffic to the cores at the same rate. We simulated the designs presented in Chapter 4 (shown in Figure 5.3) and those obtained by solving our model (shown in Figure 5.7) for different rates of traffic generation. The simulation was allowed to run for $1,000,000$ cycles, at the end of which we accumulated the statistical quantities.

In Figure 5.8, we show the maximum latency observed for any path in the network as a function of the rate at which the traffic is generated. The graph on the left shows the maximum latency for the entire range of ρ, the per node flit injection rate. The graph on the right shows



(a) $\rho = 0.0075$ (b) $\rho = 0.0077$ (c) $\rho = 0.008$ (d) $\rho = 0.0085$

Figure 5.7: Designs obtained for *Min-Max* Latency Optimization Model for different values of ρ.

Design	ρ		
	0.0075	0.008	0.0085
`center`	14.48	16.08	18.47
`diamond`	14.81	15.35	15.97
`com-opt`	15.52	16.89	18.85
`min-max-opt`	14.48	15.25	15.97

Table 5.1: Value of the objective function for the optimal designs for the Linearized Min-Max Latency Model.

the same data for a limited range of ρ, when the network is not congested. From the graphs obtained via synthetic simulations, one can observe that the model succeeded in computing a design(`min-max-0.008`) that has lower latency for a major portion of the traffic injection rates. We believe this points to the usefulness of model in working successfully for different injection rates.

5.3.7 Discussion on the model and the designs

Numerical range for ρ. The reader might wonder as to why even for low values of ρ (≤ 0.012), the network saturates. Note that ρ represents the per node injection rate. So the nodes are injecting less than one flit per cycle. But the network will saturate as soon as even a single link reaches its capacity. This is possible at even low values for ρ since a link may receive flits from multiple different nodes. For the four different designs we evaluated above, the link with maximum load serves traffic from a large number of processor-core and memory controller pairs. This number for designs: `center`, `diamond`, `com-opt` and `min-max-0.008` is 196, 78,

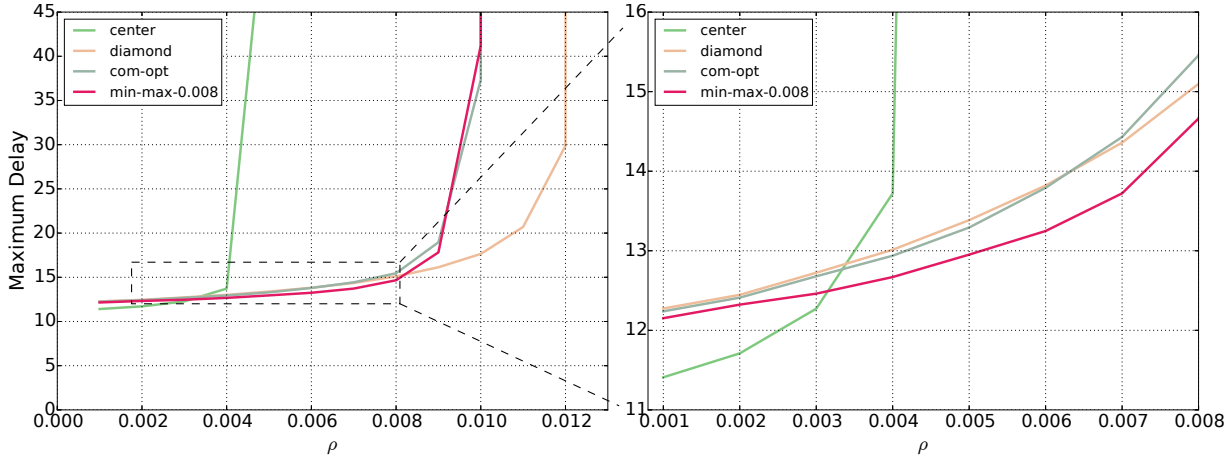


Figure 5.8: Results from synthetic simulation of the designs obtained from the *Min-Max Latency Model*. The graph on the left shows maximum latency for different designs and the entire range of ρ, the node injection rate. The graph on the right shows the same data for a limited range of ρ.

96 and 97 respectively. The inverse of this number gives the upper bound on ρ for each design. That inverse closely matches the maximum value we observed for ρ in our experiments.

To further elaborate on this fact, we present some data in Figure 5.9. In this Figure, we have graphed the increase in the load serviced by the most loaded link as a function of ρ. As can be seen in the graph, the link is saturated with traffic at same value of ρ at which the particular design saturates.

M/D/1 approximation. In Figure 5.10, we show the queueing delay observed at the link servicing the maximum load for each design. We also show the queueing delay for an M/D/1 queueing server. As can be seen from the figure, curves for the queueing delay for different designs match nicely with the curve for M/D/1 system. Thus the $M/D/1$ approximation for modeling queueing delays works reasonable well for major portion of the input.

We ultimately work with a piecewise linear approximation to the nonlinear function that represents queueing latency for an $M/D/1$ system. This means that if we have a piecewise linear approximation for queueing delay in any other system, our model should still work fine.

Asymmetry in the designs obtained. As can be seen in Figures 5.7b and 5.7c, we obtain

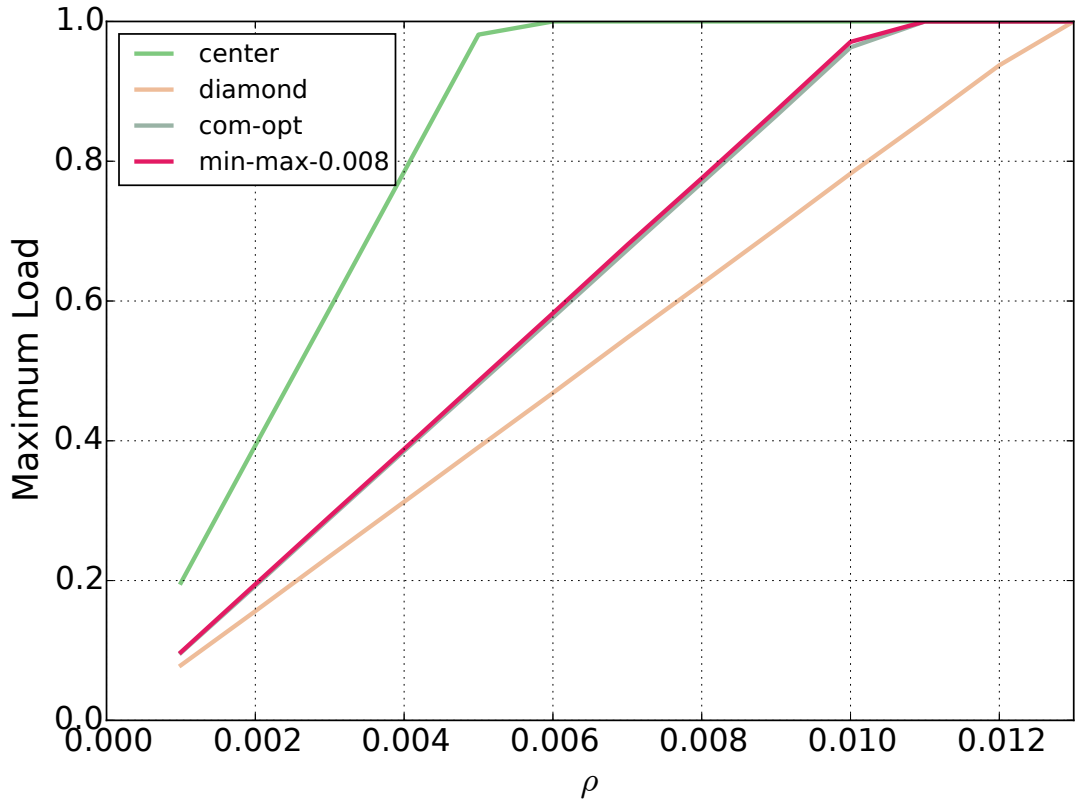


Figure 5.9: Results from synthetic simulation of the designs obtained from the *Min-Max Latency Model*. The graph shows maximum load that a link has to service for different designs and different values of ρ, the node injection rate.

designs that are not symmetric. Since layout and verification may be simpler with symmetric designs, we further tried solving the optimization model in by adding two different symmetry constraints:

- symmetric with respect to the X and the Y axes.
- symmetric with respect to the left and the right diagonals.

For both these cases and $\rho = 0.008$, we obtained the `diamond` design as the solution. Note that `diamond` performs worse than `min-max-0.008` for $\rho = 0.008$. We also tried computing other optimal designs using the model. But we only succeeded in obtaining designs that are rotated version of `min-max-0.008`.

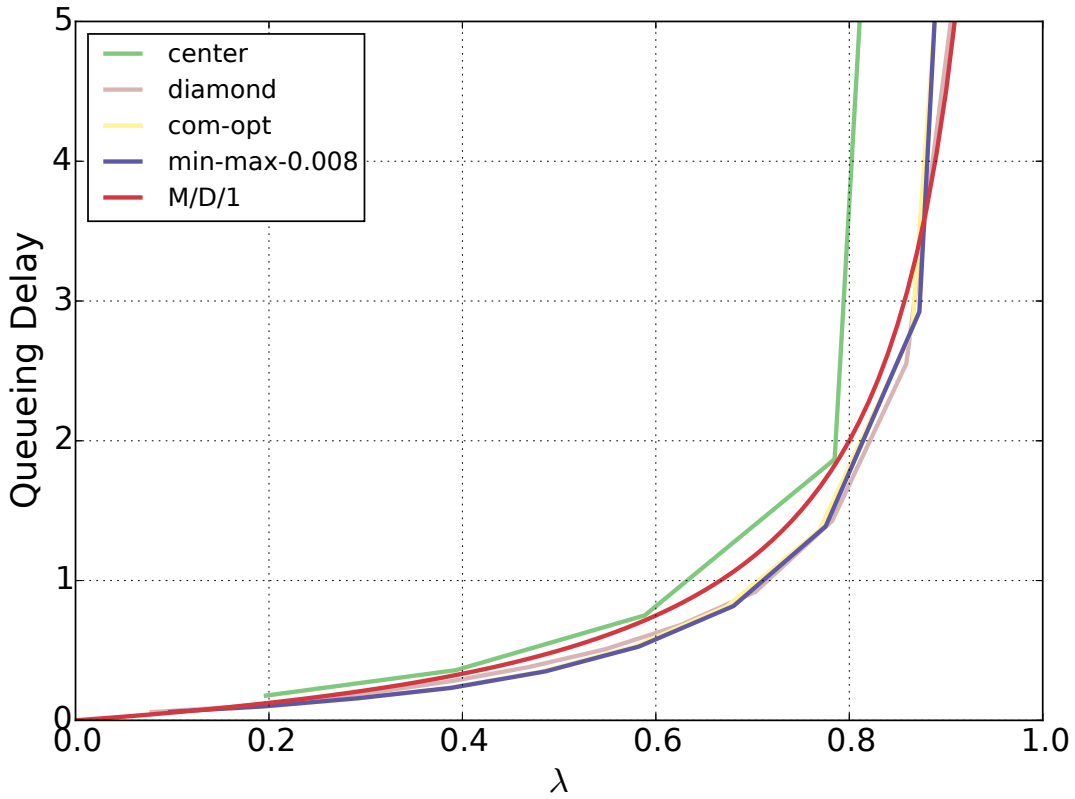


Figure 5.10: Results from synthetic simulation of the designs obtained from the *Min-Max Latency Model*. The graph shows the queueing delay observed for the link with maximum load for different designs and different values of λ, the link's traffic arrival rate.

5.4 Average Latency Optimization Model

In this section we present the *Average Latency* model for distributing resources in an on-chip network. This model aims at minimizing the average latency a flit has to experience in going from a processor core to a memory controller or vice versa. The model has been shown in Figure 5.11. The model is very similar to the *MinMax Latency* model. Therefore, we only describe the changes made to the *MinMax Latency* model to obtain the *Average Latency* model.

5.4.1 Nonlinear Model

To obtain the *Average Latency* model from the *MinMax Latency model*, we dropped constraints: (5.12), (5.13), (5.14) and (5.15). These constraints were used for estimating the total latency along each path and then taking their maximum to arrive at the objective value. In place of these, we added constraints: (5.39), (5.40) and (5.41). Constraint (5.39) assigns to the variable $Y_{x,y,x',y'}$ the latency involved in moving a flit from the core at (x', y') to the memory controller at (x, y). Constraint (5.40) similarly assigns variables $Y_{x',y',x,y}$. Constraint (5.41) assigns to Y the sum total of latencies observed by flits along all possible paths of communication. Finally, we changed the objective function variable from Z to Y. This means the models aims at minimizing the average latency required for communication.

Computational Results With Non-linear Model. We tried solving the model from Figure 5.11 with the solver Baron [96] and $\rho = 0.007$. The solver failed to make any progress in solving the problem and we were not able to compute any meaningful results. We know a lower-bound of 4.25 on the objective function. This value can be computed by computing the zero-load average latency when a memory controller is placed at all possible values of (x, y) and choosing the least M (number of memory controllers) positions to place the memory controllers. This process places all the memory controllers in the center of the mesh network. The solver is not able to improve upon this lower bound. Further, it is not able to find a solution on its own. We provided the solver with the following initial designs: `center`, `com-opt` and `diamond`. The solver is not able to improve on these designs either. These designs have the objective values 7.47, 7.07 and 7.71 respectively.

5.4.2 Linearized Model

Constraints (5.37), (5.38), (5.39) and (5.40) in the model are non-linear. All of these constraints also appeared in the *MinMax* model above. We discussed in section 5.3.4 how these constraints

$$\begin{aligned}
\text{minimize} \quad & Y \quad \text{subject to} \\
\sum_{l} BW_l \quad & \leq \quad \text{link budget} && (5.34)\\
\sum_{(x,y)} I_{x,y} \quad & = \quad \text{M} && (5.35)\\
\text{For all links } l \text{ in the network} & \\
LoadOnLink(l) \quad & = \quad \sum_{(x,y,x',y'):l \in Path(x',y',x,y)} \lambda(R+K) I_{xy} \\
& + \quad \sum_{(x,y,x',y'):l \in Path(x,y,x',y')} \lambda(RK+1) I_{xy} && (5.36)\\
\lambda_l \quad & = \quad \frac{LoadOnLink(l)}{(1+BW_l)} && (5.37)\\
W_l \quad & \geq \quad \frac{\lambda_l}{2\mu(\mu-\lambda_l)} && (5.38)\\
\text{For all possible values of } (x,y) \quad & \text{and} \quad (x',y') \\
Y_{x,y,x',y'} \quad & \geq \quad I_{xy}\Big(\sum_{l \in Path(x',y',x,y)} \big(\frac{1}{\mu} + W_l\big)\Big) && (5.39)\\
Y_{x',y',x,y} \quad & \geq \quad I_{xy}\Big(\sum_{l \in Path(x',y',x,y)} \big(\frac{1}{\mu} + W_l\big)\Big) && (5.40)\\
Y \quad & \geq \quad \frac{1}{2\text{NM}} \sum_{x,y,x',y'} (Y_{x,y,x',y'} + Y_{x',y',x,y}) && (5.41)\\
I_{x,y} \in \{0,1\}, \quad & BW_l \in \{0,1\}
\end{aligned}$$

Figure 5.11: *Average Latency* Optimization Model

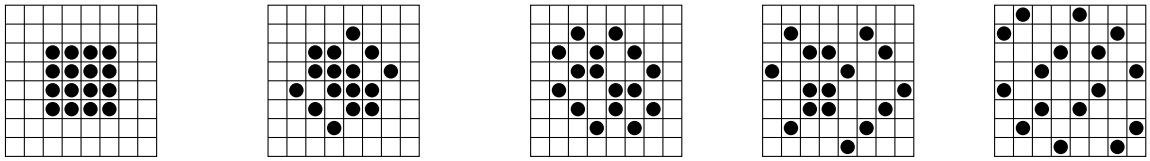
can be linearized. We use the same method again for the *Average Latency* model here. The resulting linearized *Average Latency* model appears in Figure 5.12.

Computational Results with Linearized Model. We solved the linearized model for different values of the parameter $\rho = \frac{\lambda}{\mu}$. The non-linear function was broken down into 10 linear pieces. For solving, we used the tool CPLEX [3]. In Figure 5.13, we show the designs obtained by solving the optimization model. In Table 5.2, we show the average latency for different designs as obtained from the model. From the obtained data, we conclude two things. The presented model is able to correctly compute the `center` design for low values of ρ and a design

$$\begin{aligned}
\text{minimize} \quad & Y \quad \text{subject to} \\
\sum_{l} BW_l \quad & \leq \quad \text{link budget} && (5.42)\\
\sum_{(x,y)} I_{x,y} \quad & = \quad \text{M} && (5.43)\\
\text{For all links } l \text{ in the network} & \\
LoadOnLink(l) \quad & = \quad \sum_{(x,y,x',y'):l \in Path(x',y',x,y)} I_{xy}(R+K) & \\
& + \quad \sum_{(x,y,x',y'):l \in Path(x,y,x',y')} I_{xy}(RK+1) && (5.44)\\
\sum_{i \in S} b_{l,i} \times \lambda_i \quad & \geq \quad \frac{LoadOnLink(l)}{2} && (5.45)\\
\sum_{l,i} b_{l,i} \times \lambda_i + BW_l \times M^+ \quad & \geq \quad LoadOnLink(l) && (5.46)\\
\sum_{i \in S} b_{l,i} \quad & = \quad 1 && (5.47)\\
W_l \quad & \geq \quad \sum_{i \in S} b_{l,i} \times W_i && (5.48)\\
\text{For all possible values of } (x,y) \text{ and} & (x',y') \\
Y_{x,y,x',y'} \quad & \geq \quad (1-I_{xy})M^- + \sum_{l \in Path(x',y',x,y)} (\frac{1}{\mu} + W_l) && (5.49)\\
Y_{x',y',x,y} \quad & \geq \quad (1-I_{xy})M^- + \sum_{l \in Path(x,y,x',y')} (\frac{1}{\mu} + W_l) && (5.50)\\
Y \quad & \geq \quad \frac{1}{2\text{NM}} \sum_{x,y,x',y'} (Y_{x,y,x',y'} + Y_{x',y',x,y}) && (5.51)\\
I_{x,y} \in \{0,1\}, \quad BW_l \in \{0,1\}, & \qquad b_l \text{ are SOS2}, \quad Y_{x,y,x',y'} \geq 0, Y_{x',y',x,y} \geq 0
\end{aligned}$$

Figure 5.12: Linearized *Average Latency* Optimization Model

that performs similar to the `diamond` design for high values of ρ. Secondly, there are designs that perform better than `center` and `diamond` for intermediate values of ρ. Designers might be more interested in these designs. The value of ρ seems crucial. Designers should estimate it with reasonable certainty to zero-in on optimal designs.



(a) $\rho = 0.005$ (b) $\rho = 0.006 - 0.007$ (c) $\rho = 0.008$ (d) $\rho = 0.009$ (e) $\rho = 0.010$

Figure 5.13: Designs obtained for the Linearized *Average* Latency Optimization Model for different values of ρ.

Design	ρ					
	0.005	0.006	0.007	0.008	0.009	0.010
`center`	5.35	5.80	6.45	7.55	10.01	∞
`diamond`	7.96	6.71	7.16	7.73	8.52	9.70
`com-opt`	7.45	5.92	6.40	7.11	8.44	∞
`average-opt`	5.35 (9.03%)	5.78 (14.33%)	6.33 (22.24%)	7.11 (28.30%)	8.20 (33.28%)	9.70 (42.39%)

Table 5.2: Average Latency for Different Designs on the Linearized Average Latency Model. Integrality gap for the model solution is shown in parentheses. ∞ means that a design cannot sustain that value of ρ.

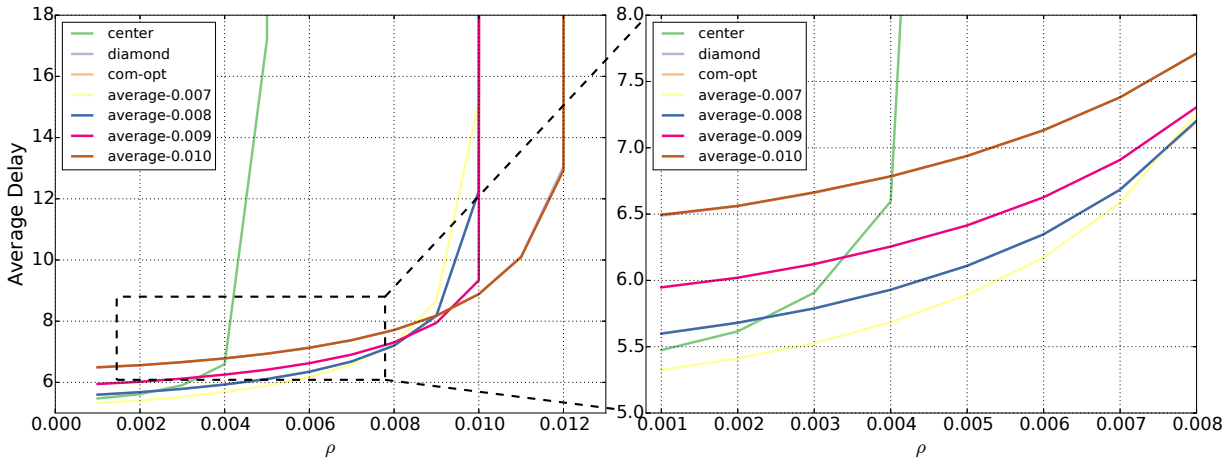


Figure 5.14: Results from synthetic simulation of the designs obtained from the *Average Latency Model*. The graph on the left shows the average latency for different designs and for the entire range of ρ, the node injection rate. The graph on the right shows the same data for a smaller range of ρ. Note that in the graphs the following pairs overlap each other: (`diamond`, `average-0.010`) and (`com-opt`, `average-0.008`).

Synthetic Simulation Results. We simulated the designs from Figure 5.13 using the simulator described in section 5.3.6. In Figure 5.14, we show the average latency observed for any

path in the network as a function of the rate at which the traffic is generated. The graph on the left shows the average latency for different designs and for the entire range of ρ. The graph on the right shows the same data for a smaller range of ρ.

From the graphs obtained via synthetic simulations, we believe that designs: `average-0.007` and `average-0.008` perform better than others for intermediate rates of traffic injection. While they do not achieve as high bandwidth as `diamond` or `average-0.010`, for most injection rates, they provide the best average latency. Thus, our solutions from the mathematical model agree with the results obtained from synthetic simulations. We think these two designs were not known before and that they might be of interest to designers.

Moreover, these results from simulations further provide evidence that there are designs that perform better than `center` and `diamond` for intermediate values of ρ. These designs can be found using optimization models. Designers should evaluate them alongwith `center` and `diamond` which represent the two extremes.

5.4.3 Discussion about the designs obtained

Asymmetry of the designs. We initially obtained asymmetric designs for most of values of ρ. We then tried solving the optimization model along with symmetry constraints with respect to the left and the right diagonals. For all cases other than $\rho = 0.009$, we obtained symmetric designs that matched the objective value of the asymmetric design. For $\rho = 0.009$, we obtained a symmetric design whose objective value is 8.31 compared to 8.20, the objective value of design `average-0.009` shown in Figure 5.13d.

Optimality of the designs obtained. In Table 5.2, we have noted that there was significant gap between the objective value of the solution obtained and the best possible estimate for lower bound. This points to the computational hardness of the problem that we are trying to solve. The source of this hardness is not the number of pieces used in linearization. To support this statement, we fixed $\rho = 0.010$ and varied the number of pieces used for linearization. For 4-,6-

,8- and 10-piece linearizations, the gaps are 40.81%, 41.68%, 41.62% and 42.39% respectively. These results show that the computational hardness of the linearized model is, to a very large extent, independent of the number of pieces used.

5.5 Related Work

Our work is based on the work of Abts *et al.* [5], Mishra *et al.* [70] and Vaish *et al.* [98]. In this section, we highlight other work that solve problems similar to ours.

Models for On-chip Networks. Analytical models for on-chip networks are needed to quickly evaluate designs while carrying out design-space exploration. Significant effort has been devoted towards improving the predictions accuracy of these models and reducing the time required to make those predictions [102, 21, 63, 101, 60, 85, 91]. These models can be used towards improving the models we have presented. In particular the assumption that inter-arrival times for flits are exponentially distributed does not work well when the traffic injection rate is high. One may want to incorporate these analytical models into ours.

We discuss some of the more recent work on analytical models for on-chip networks. Qian *et al.* [82] present a $GE/G/1/K$ queueing model method for latency prediction for on-chip networks. The proposed model can handle bursty traffic and dependent traffic patterns. Qian *et al.* [83] present a latency model based on support vector regression (SVR). In their approach, first a queueing theory based analytical model is formulated. To improve the accuracy of the model, they use SVR to learn features of delay distribution. The combined model is shown to have higher accuracy for predictions.

Kiasari *et al.* [57] survey four different formal techniques—queueing theory, network calculus, schedulability analysis, and dataflow analysis—and their application in analyzing on-chip networks. They present examples how these techniques are used in practice. In another survey paper, Qian *et al.* [84] review the workloads used for evaluating on-chip networks, the techniques

for analyzing and modeling short and long-range behaviors and the queueing theory based models for predicting end-to-end delays. They also discuss some of the open problems in the area on-chip networks.

On-chip Placement. Significant amount of work on on-chip placement has been done in the SoC domain. Different methods including finite element analysis [38], simulated annealing [46], genetic algorithms [104, 51], exhaustive search [103], linear programming [92] and via problem-specific algorithms [50, 87, 7, 74]. The above work mainly focuses on how to prepare a floorplan for the chip. This planning happens late in the design process. Our models are applicable inital design space exploration when initial resource allocation is carrier out for the processor being designed.

5.6 Conclusion

In this chapter, we presented two new optimization models for placing memory controllers in an on-chip network. The first model minimized the maximum latency of communication. The second one minimized the average latency of communication. We provide the designer with a single parameter—traffic input rate—that they can set and explore the design space using these models. We presented optimal designs for a range of traffic input rate. These designs have not appeared in the literature so far. Synthetic simulations show that the computed designs perform as expected by the model. We believe that these single parameter models are effective and can help a designer in pruning the design space efficiently.

Chapter 6

Conclusion

Manycore processors are going to be ubiquitous in coming years. There are already several different processors available that might be classified as manycore processors. The state of the art in designing such processors is mainly based on simulation. But this reduces the design possibilities we can explore. Hence, in this dissertation, we suggested modeling and algorithmic approaches that should help in exploring a larger area of the design space.

Throughout this dissertation, our focus has been on designing the memory system for these manycore processors. It seems that years of performance improvements that we had for processor cores is now behind us. Major improvements in processing power would come from increased thread-level parallel, heterogeneity and special purpose functional units [20]. This dissertation focused on modeling techniques for designing the memory system for increased parallelism.

We first described a model and an algorithm for designing the caches for these processors (Chapter 3). Caches are critical in reducing the time and energy required to access data. Therefore, our suggested technique aimed at computing designs that are close to the Pareto-frontier formed by multiple objectives that are required from a manycore processor design.

We then moved onto designing the on-chip network and distributing the memory controllers across the processor. This is an important part of the design process since inter-thread commu-

nication, whether through caches or through the memory controllers, has a strong bearing on the performance of the processor. We first modeled the problem assuming that application performance was only dependent on the bandwidth available for communication (Chapter 4). We then improved on our model so as to take into account that as the utilization of the available communication bandwidth goes up, the latency of communication also goes up, and that too in a non-linear fashion(Chapter 5).

6.1 Limitations of Our Work

- In our work on designing cache hierarchy for manycore processor, our method performs poorly when resources shared amongst cores are close to saturation. The method should be improved with a better model in place for shared cache performance.

- As we experienced, optimization-based models involving integer-valued variables or non-linear constraints, are hard to solve and do not always yield provably optimal solutions. Similarly, our models do not scale well beyond tiled processors of size more than 10×10. One may have to invest significant computational time and effort for solving these harder and larger problems to optimality. Otherwise, one would need to reformulate the problem such that the new problems are less complex.

- We used ideal queueing models that ignore the complexity involved in actual implementations like finite buffers and irregular request patterns. There might be other queueing models available in the literature that are more suitable for architectural design.

Further research on the problems tackled in this dissertation should focus on removing the limitations described above.

6.2 Future Work

Multi-scale Design Process

Going even further, it might be useful to construct a formal multi-scale design process for many-core processors. Such a design process should have at least three levels. The first level will be based on rudimentary design principles like Amdahl's law [9, 49]. This level would help in making some of the high level design decisions like resource budget allocated to some of the major on-chip components. For example, one may decide the number of cores sufficient to achieve the desired level of performance. At the next scale, we would have models similar to the ones described in this dissertation. Such models, based on queueing theory, machine learning, mathematical optimization and other quantitative techniques, would help explore the design space quickly and zero in on portions of the design space that demand a deeper exploration. The final level (possibly) would use a software-based simulator that is used to explore the designs obtained at the previous level in much more detail. The design steps outlined above should be carried out in an iterative fashion till a design which achieves the design goals is achieved.

Computer architects already have an informal understanding of such a multi-scale design process. A more formal study should aim at providing both theoretical and empirical error bounds for each of levels in the process. We think such a multi-scale process is imperative for managing the complexity involved in designing future manycore processors.

Simulation Optimization

Another approach that seems worth exploring in depth is a design process that interweaves software-based simulators with a mathematical optimization based approach. Borrowing the notation from Fu [35], we define the process of designing a processor as finding a design that minimizes the objective function $J(\theta) = E[L(\theta, \omega)]$ over all $\theta \in \Theta$. Here θ represents a design, Θ is the set of all possible designs, ω represents a computation which the design needs to process

and L is a performance measure. Typically it is assumed that the performance of a given solution can be computed quickly. So greater emphasis is placed on finding improved solution. But in many problems, including that of processor design, estimating the performance of a given solution is costly. Therefore, one may need to trade-off between the computational effort required for the two parts of the design process. The field of simulation optimization focuses on how we can integrate techniques from optimization into software-based simulation while making the required trade-off. As computer architects, we might want to consider the approaches suggested in the research literature of simulation optimization.

Bibliography

[1] “Gams,” "http://www.gams.com". 17, 64, 103

[2] “Spec,” "http://www.spec.org". ix, 2, 3

[3] “Ilog cplex 10.1 user’s manual,” 2006. 84, 108, 114

[4] A. H. Abdel-Gawad and M. Thottethodi, “Transcom: transforming stream communication for load balance and efficiency in networks-on-chip,” ser. MICRO-44 ’11. New York, NY, USA: ACM, pp. 237–247. [Online]. Available: http://doi.acm.org/10.1145/2155620.2155648 60

[5] D. Abts, N. D. Enright Jerger, J. Kim, D. Gibson, and M. H. Lipasti, “Achieving Predictable Performance through Better Memory Controller Placement in Many-Core CMPs,” in *ISCA ’09*. 60, 61, 90, 91, 92, 95, 97, 118

[6] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, “Garnet: A detailed on-chip network model inside a full-system simulator.” in *ISPASS*. IEEE, 2009, pp. 33–42. [Online]. Available: http://dblp.uni-trier.de/db/conf/ispass/ispass2009.html#AgarwalKPJ09 64

[7] T. Ahonen, D. A. Sigüenza-Tortosa, H. Bin, and J. Nurmi, “Topology optimization for application-specific networks-on-chip,” in *Proceedings of the 2004 International Workshop on System Level Interconnect Prediction*, ser. SLIP ’04, 2004, pp. 53–60. [Online]. Available: http://doi.acm.org/10.1145/966747.966758 119

[8] K. Aingaran, S. Jairath, G. Konstadinidis, S. Leung, P. Loewenstein, C. McAllister, S. Phillips, Z. Radovic, R. Sivaramakrishnan, D. Smentek, and T. Wicki, "M7: Oracle's next-generation sparc processor," *IEEE Micro*, vol. 35, no. 2, pp. 36–45, Mar 2015. 20

[9] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring). New York, NY, USA: ACM, 1967, pp. 483–485. [Online]. Available: http://doi.acm.org/10.1145/1465482.1465560 122

[10] M. Awasthi, D. W. Nellans, K. Sudan, R. Balasubramonian, and A. Davis, "Handling the Problems and Opportunities Posed by Multiple On-Chip Memory Controllers," in *PACT*, 2010, pp. 319–330. [Online]. Available: http://doi.acm.org/10.1145/1854273.1854314 92

[11] O. Azizi, A. Mahesri, B. C. Lee, S. J. Patel, and M. Horowitz, "Energy-performance tradeoffs in processor architecture and circuit design: A marginal cost analysis," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 26–36. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815967 58

[12] ——, "Energy-performance tradeoffs in processor architecture and circuit design: a marginal cost analysis," in *Proceedings of the 37th annual international symposium on Computer architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 26–36. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815967 60

[13] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks—Summary and Preliminary Results," in *Proceedings of the 1991 ACM/IEEE Conference on*

Supercomputing, ser. Supercomputing '91. New York, NY, USA: ACM, 1991, pp. 158–165. [Online]. Available: http://doi.acm.org/10.1145/125826.125925 87

[14] L. A. Barroso, "The Price of Performance," *Queue*, vol. 3, no. 7, pp. 48–53, Sep. 2005. [Online]. Available: http://doi.acm.org/10.1145/1095408.1095420 3

[15] Y. Ben-Itzhak, I. Cidon, and A. Kolodny, "Optimizing heterogeneous noc design," ser. SLIP '12, pp. 32–39. [Online]. Available: http://doi.acm.org/10.1145/2347655.2347670 92

[16] J. L. Bentley, "Multidimensional Divide-and-Conquer," *Commun. ACM*, vol. 23, no. 4, pp. 214–229, Apr. 1980. [Online]. Available: http://doi.acm.org/10.1145/358841.358850 13, 31, 33

[17] V. Berinde, *Iterative Approximation of Fixed Points*, ser. Lecture Notes in Mathematics. Springer, 2007. 38

[18] D. Bertsekas and R. Gallager, *Data Networks*. Prentice Hall, 1992. 39

[19] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718 30, 51, 53, 64, 76

[20] S. Borkar and A. A. Chien, "The Future of Microprocessors," *Commun. ACM*, vol. 54, pp. 67–77, May 2011. [Online]. Available: http://doi.acm.org/10.1145/1941487.1941507 3, 120

[21] S. C. Borst, O. J. Boxma, and M. B. Combé, "Collection of customers: A correlated m/g/1 queue," *SIGMETRICS Perform. Eval. Rev.*, vol. 20, no. 1, pp. 47–59, Jun. 1992. [Online]. Available: http://doi.acm.org/10.1145/149439.133085 118

[22] S. Boyd and L. Vandenberghe, *Convex Optimization*. New York, NY, USA: Cambridge University Press, 2004. 14, 17, 83

[23] Cavium, *ThunderX ARM Processors*. [Online]. Available: http://www.cavium.com/pdfFiles/ThunderX_PB_Rev2.pdf?x=2 20

[24] ——, *ThunderX2 ARM Processors*. [Online]. Available: http://cavium.com/pdfFiles/ThunderX2_PB_Rev1.pdf?x=2 20

[25] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture," ser. HPCA. IEEE Computer Society, 2005, pp. 340–351. [Online]. Available: http://dx.doi.org/10.1109/HPCA.2005.27 37

[26] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001. 10, 11, 23

[27] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003. 77

[28] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu, "Memory power management via dynamic voltage/frequency scaling," in *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ser. ICAC '11. New York, NY, USA: ACM, 2011, pp. 31–40. [Online]. Available: http://doi.acm.org/10.1145/1998582.1998590 44

[29] D. Eklov, D. Black-Schaffer, and E. Hagersten, "Fast modeling of shared caches in multicore systems," in *Proceedings of the 6th International Conference on*

High Performance and Embedded Architectures and Compilers, ser. HiPEAC '11. New York, NY, USA: ACM, 2011, pp. 147–157. [Online]. Available: http://doi.acm.org/10.1145/1944862.1944885 20, 58

[30] S. Eyerman, L. Eeckhout, and K. De Bosschere, "Efficient design space exploration of high performance embedded out-of-order processors," in *Proceedings of the Conference on Design, Automation and Test in Europe: Proceedings*, ser. DATE '06. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2006, pp. 351–356. [Online]. Available: http://dl.acm.org/citation.cfm?id=1131481.1131578 58

[31] T. Fawcett, "An introduction to {ROC} analysis," *Pattern Recognition Letters*, vol. 27, no. 8, pp. 861 – 874, 2006, {ROC} Analysis in Pattern Recognition. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016786550500303X 54

[32] J. Feehrer, S. Jairath, P. Loewenstein, R. Sivaramakrishnan, D. Smentek, S. Turullols, and A. Vahidsafa, "The oracle sparc t5 16-core processor scales to eight sockets," *Micro, IEEE*, vol. 33, no. 2, pp. 48–57, March 2013. 61

[33] F. Fritsch and R. Carlson, "Monotone Piecewise Cubic Interpolation," *SIAM Journal on Numerical Analysis*, vol. 17, no. 2, pp. 238–246, 1980. [Online]. Available: http://epubs.siam.org/doi/abs/10.1137/0717021 38

[34] C. Fu and K. Wilken, "A faster optimal register allocator," in *MICRO 35*. IEEE Computer Society Press, 2002, pp. 245–256. [Online]. Available: http://dl.acm.org/citation.cfm?id=774861.774888 60

[35] M. C. Fu, "Feature article: Optimization for simulation: Theory vs. practice," *INFORMS J. on Computing*, vol. 14, no. 3, pp. 192–215, Jul. 2002. [Online]. Available: http://dx.doi.org/10.1287/ijoc.14.3.192.113 122

[36] D. Gibson, "Private communication." 90

[37] M. X. Goemans, A. V. Goldberg, S. Plotkin, D. B. Shmoys, E. Tardos, and D. P. Williamson, "Improved approximation algorithms for network design problems," in *SODA*, 1994, pp. 223–232. [Online]. Available: http://dl.acm.org/citation.cfm?id=314464.314497 93

[38] B. Goplen and S. Sapatnekar, "Thermal via placement in 3d ics," in *Proceedings of the 2005 International Symposium on Physical Design*, ser. ISPD '05. New York, NY, USA: ACM, 2005, pp. 167–174. [Online]. Available: http://doi.acm.org/10.1145/1055137.1055171 119

[39] D. Gross, J. F. Shortle, J. M. Thompson, and C. M. Harris, *Fundamentals of Queueing Theory*, 4th ed. New York, NY, USA: Wiley-Interscience, 2008. 99, 103

[40] B. Grot, J. Hestness, S. W. Keckler, and O. Mutlu, "Kilo-noc: a heterogeneous network-on-chip architecture for scalability and service guarantees," in *ISCA*, 2011, pp. 401–412. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000112 92

[41] A. Gupte, S. Ahmed, M. S. Cheon, and S. S. Dey, "Solving mixed integer bilinear problems using mip formulations," 2012. 83

[42] I. Gurobi Optimization, "Gurobi optimizer reference manual," 2012. [Online]. Available: http://www.gurobi.com 17, 64

[43] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 28, no. 1, pp. 100–108, 1979. [Online]. Available: http://www.jstor.org/stable/2346830 32

[44] A. Hartstein, V. Srinivasan, T. R. Puzak, and P. G. Emma, "Cache Miss Behavior: Is It $\sqrt{2}$?" ser. CF '06, pp. 313–320. [Online]. Available: http://doi.acm.org/10.1145/1128022.1128064 24, 27, 29

[45] M. Hayenga, N. E. Jerger, and M. Lipasti, "Scarab: a single cycle adaptive routing and bufferless network," in *MICRO 42*, 2009, pp. 244–254. [Online]. Available: http://doi.acm.org/10.1145/1669112.1669144 92

[46] M. Healy, M. Vittes, M. Ekpanyapong, C. S. Ballapuram, S. K. Lim, H. H. S. Lee, and G. H. Loh, "Multiobjective microarchitectural floorplanning for 2-d and 3-d ics," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 1, pp. 38–52, Jan 2007. 119

[47] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003. 34

[48] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006. [Online]. Available: http://doi.acm.org/10.1145/1186736.1186737 51, 53, 65, 86

[49] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, Jul. 2008. [Online]. Available: http://dx.doi.org/10.1109/MC.2008.209 122

[50] W. Hung, C. Addo-Quaye, T. Theocharides, Y. Xie, N. Vijakrishnan, and M. J. Irwin, "Thermal-aware ip virtualization and placement for networks-on-chip architecture," in *IEEE International Conference on Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings.*, Oct 2004, pp. 430–437. 119

[51] W.-L. Hung, Y. Xie, N. Vijaykrishnan, C. Addo-Quaye, T. Theocharides, and M. Irwin, "Thermal-aware floorplanning using genetic algorithms," in *International Symposium on Quality of Electronic Design, 2005.*, pp. 634 – 639. 93, 119

[52] H. Iwai, "Roadmap for 22nm and beyond (invited paper)," *Microelectron. Eng.*, vol. 86, no. 7-9, pp. 1520–1528, Jul. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.mee.2009.03.129 1

[53] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk*. Elsevier Science, 2010. [Online]. Available: http://books.google.com/books?id=SrP3aWed-esC 27, 43

[54] B. L. Jacob, P. M. Chen, S. R. Silverman, and T. N. Mudge, "An Analytical Model for Designing Memory Hierarchies," *IEEE Trans. Comput.*, vol. 45, no. 10, pp. 1180–1194, Oct. 1996. [Online]. Available: http://dx.doi.org/10.1109/12.543711 20, 27, 48

[55] N. D. E. Jerger and L.-S. Peh, *On-Chip Networks*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2009. 61

[56] T. S. Karkhanis and J. E. Smith, "Automated design of application specific superscalar processors: An analytical approach," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. New York, NY, USA: ACM, 2007, pp. 402–411. [Online]. Available: http://doi.acm.org/10.1145/1250662.1250712 59

[57] A. E. Kiasari, A. Jantsch, and Z. Lu, "Mathematical formalisms for performance evaluation of networks-on-chip," *ACM Comput. Surv.*, vol. 45, no. 3, pp. 38:1–38:41, Jul. 2013. [Online]. Available: http://doi.acm.org/10.1145/2480741.2480755 118

[58] M. A. Kinsy, M. H. Cho, T. Wen, E. Suh, M. van Dijk, and S. Devadas, "Application-aware deadlock-free oblivious routing," in *ISCA '09*. ACM, pp. 208–219. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555782 60

[59] L. Kleinrock, *Communication Nets: Stochastic Message Flow and Delay*. McGraw-Hill, New York, 1964. 39

[60] D. D. Kouvatsos, "A maximum entropy queue length distribution for the g/g/1 finite capacity queue," in *Proceedings of the 1986 ACM SIGMETRICS Joint International Conference on Computer Performance Modelling, Measurement and Evaluation*, ser. SIGMETRICS '86/PERFORMANCE '86. New York, NY, USA: ACM, 1986, pp. 224–236. [Online]. Available: http://doi.acm.org/10.1145/317499.317555 118

[61] B. C. Lee and D. M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII. New York, NY, USA: ACM, 2006, pp. 185–194. [Online]. Available: http://doi.acm.org/10.1145/1168857.1168881 58

[62] ——, "Illustrative design space studies with microarchitectural regression models," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, ser. HPCA '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 340–351. [Online]. Available: http://dx.doi.org/10.1109/HPCA.2007.346211 58

[63] T. Lin and L. Kleinrock, "Performance analysis of finite-buffered multistage interconnection networks with a general traffic pattern," *SIGMETRICS Perform. Eval. Rev.*, vol. 19, no. 1, pp. 68–78, Apr. 1991. [Online]. Available: http://doi.acm.org/10.1145/107972.107980 118

[64] S. Ma, N. Enright Jerger, and Z. Wang, "Dbar: an efficient routing algorithm to support multiple concurrent applications in networks-on-chip," in *ISCA-38*, 2011, pp. 413–424. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000113 92

[65] S. Ma, N. D. E. Jerger, and Z. Wang, "Whole packet forwarding: Efficient design of fully adaptive routing algorithms for networks-on-chip." in *HPCA*, 2012, pp. 467–478. [Online]. Available: http://dblp.uni-trier.de/db/conf/hpca/hpca2012.html#MaJW12a 92

[66] T. L. Magnanti and R. T. Wong, "Network Design and Transportation Planning: Models and Algorithms," *Transportation Science*, vol. 18, pp. 1–56, 1984. 93

[67] R. Marculescu and P. Bogdan, *The Chip Is the Network: Toward a Science of Network-on-Chip Design*, 2009. [Online]. Available: http://dx.doi.org/10.1561/1000000011 60

[68] Mellanox, *Tile-Gx72*. [Online]. Available: http://www.mellanox.com/related-docs/prod_multi_core/PB_TILE-Gx72.pdf 20

[69] A. K. Mishra, "Private communication." 91

[70] A. K. Mishra, N. Vijaykrishnan, and C. R. Das, "A Case for Heterogeneous On-Chip Interconnects for CMPs," in *ISCA '11*. [Online]. Available: http://doi.acm.org/10.1145/2024723.2000111 40, 60, 69, 91, 92, 95, 97, 118

[71] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, 1965. 1

[72] T. Moscibroda and O. Mutlu, "A case for bufferless routing in on-chip networks," in *ISCA*, 2009, pp. 196–207. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555781 92

[73] T. Mudge, "Power: A First-Class Architectural Design Constraint," *Computer*, vol. 34, no. 4, pp. 52–58, 2001. 3

[74] S. Murali, P. Meloni, F. Angiolini, D. Atienza, S. Carta, L. Benini, G. De Micheli, and L. Raffo, "Designing application-specific networks on chips with floorplan information,"

in *Proceedings of the 2006 IEEE/ACM International Conference on Computer-aided Design*, ser. ICCAD '06. New York, NY, USA: ACM, 2006, pp. 355–362. [Online]. Available: http://doi.acm.org/10.1145/1233501.1233573 119

[75] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP Laboratories*, 2009. [Online]. Available: http://www.hpl.hp.com/techreports/2009/HPL-2009-85.html 21, 24, 27, 30, 32, 57

[76] T. Oh, H. Lee, K. Lee, and S. Cho, "An Analytical Model to Study Optimal Area Breakdown between Cores and Caches in a Chip Multiprocessor," ser. ISVLSI '09. IEEE Computer Society, pp. 181–186. [Online]. Available: http://dx.doi.org/10.1109/ISVLSI.2009.27 20

[77] C. H. Papadimitriou, "On the complexity of integer programming," *J. ACM*, vol. 28, no. 4, pp. 765–768, Oct. 1981. [Online]. Available: http://doi.acm.org/10.1145/322276.322287 17

[78] J. Pershing. [Online]. Available: http://preshing.com/20120208/a-look-back-at-single-threaded-cpu-performance ix, 2

[79] W. B. Powell, *Approximate Dynamic Programming: Solving the Curses of Dimensionality (Wiley Series in Probability and Statistics)*. Wiley-Interscience, 2007. 11, 13

[80] P. Prieto, V. Puente, and J.-A. Gregorio, "Multilevel Cache Modeling for Chip-Multiprocessor Systems," *IEEE Comput. Archit. Lett.*, vol. 10, no. 2, pp. 49–52, Jul. 2011. [Online]. Available: http://dx.doi.org/10.1109/L-CA.2011.20 20

[81] S. A. Przybylski, *Performance-directed memory hierarchy design*. Stanford University, 1988. 59

[82] Z. Qian, D. C. Juan, P. Bogdan, C. Y. Tsui, D. Marculescu, and R. Marculescu, "A comprehensive and accurate latency model for network-on-chip performance analysis," in *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2014, pp. 323–328. 118

[83] Z. L. Qian, D. C. Juan, P. Bogdan, C. Y. Tsui, D. Marculescu, and R. Marculescu, "A support vector regression (svr)-based latency model for network-on-chip (noc) architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 3, pp. 471–484, March 2016. 118

[84] Z. Qian, P. Bogdan, C.-Y. Tsui, and R. Marculescu, "Performance evaluation of noc-based multicore systems: From traffic analysis to noc latency modeling," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 21, no. 3, pp. 52:1–52:38, May 2016. [Online]. Available: http://doi.acm.org/10.1145/2870633 118

[85] D. A. Reed, "Queueing network models of multimicrocomputer networks," in *Proceedings of the 1983 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '83. New York, NY, USA: ACM, 1983, pp. 190–197. [Online]. Available: http://doi.acm.org/10.1145/800040.801406 118

[86] A. Sandberg, D. Black-Schaffer, and E. Hagersten, "Efficient Techniques for Predicting Cache Sharing and Throughput," ser. PACT, 2012, pp. 305–314. [Online]. Available: http://doi.acm.org/10.1145/2370816.2370861 37, 38

[87] M. K. F. Schafer, T. Hollstein, H. Zimmer, and M. Glesner, "Deadlock-free routing and component placement for irregular mesh-based networks-on-chip," in *ICCAD-2005. IEEE/ACM International Conference on Computer-Aided Design, 2005.*, Nov 2005, pp. 238–245. 119

[88] R. Sivaramakrishnan and S. Jairath, "Next generaron sparc processor cache hierarchy," 2014. 20

[89] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreaded processor," ser. ASPLOS IX. ACM, 2000, pp. 234–244. [Online]. Available: http://doi.acm.org/10.1145/378993.379244 66, 79

[90] A. Sodani, R. Gramunt, J. Corbal, H. S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. C. Liu, "Knights landing: Second-generation intel xeon phi product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, Mar 2016. 20

[91] J. R. Spirn, "Network modeling with bursty traffic and finite buffer space," *SIGMETRICS Perform. Eval. Rev.*, vol. 11, no. 1, pp. 21–28, Apr. 1982. [Online]. Available: http://doi.acm.org/10.1145/1010631.801687 118

[92] K. Srinivasan, K. S. Chatha, and G. Konjevod, "Linear-programming-based techniques for synthesis of network-on-chip architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 4, pp. 407–420, April 2006. 119

[93] K. Srinivasan, K. Chatha, and G. Konjevod, "Linear programming based techniques for synthesis of network-on-chip architectures," in *ICCD*, 2004, pp. 422–429. [Online]. Available: http://dx.doi.org/10.1109/ICCD.2004.1347957 60

[94] C. Sun, C.-H. O. Chen, G. Kurian, L. Wei, J. Miller, A. Agarwal, L.-S. Peh, and V. Stojanovic, "Dsent - a Tool Connecting Emerging Photonics with Electronics for Opto-Electronic Networks-on-Chip Modeling," ser. NOCS '12, pp. 201–210. [Online]. Available: http://dx.doi.org/10.1109/NOCS.2012.31 41

[95] G. Sun, C. J. Hughes, C. Kim, J. Zhao, C. Xu, Y. Xie, and Y.-K. Chen, "Moguls: A Model to Explore the Memory Hierarchy for Bandwidth Improvements," ser. ISCA '11,

pp. 377–388. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000109 20, 21, 27, 48, 56

[96] M. Tawarmalani and N. V. Sahinidis, "A polyhedral branch-and-cut approach to global optimization," *Math. Program.*, vol. 103, no. 2, pp. 225–249, Jun. 2005. [Online]. Available: http://dx.doi.org/10.1007/s10107-005-0581-8 17, 49, 83, 103, 113

[97] H.-M. Tong, Y.-S. Lai, and C. Wong, *Advanced Flip Chip Packaging*. Springer, 2013. 62

[98] N. Vaish, M. C. Ferris, and D. A. Wood, "Optimization models for three on-chip network problems," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 3, pp. 26:1–26:27, Sep. 2016. [Online]. Available: http://doi.acm.org/10.1145/2943781 60, 118

[99] K. Van Craeynest and L. Eeckhout, "The multi-program performance model: Debunking current practice in multi-core simulation," in *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, Nov 2011, pp. 26–37. 20, 58

[100] Z. Wang and M. F. O'Boyle, "Mapping parallelism to multi-cores: a machine learning based approach," in *PPoPP '09*, pp. 75–84. [Online]. Available: http://doi.acm.org/10.1145/1504176.1504189 92

[101] D. L. Willick and D. L. Eager, "An analytic model of multistage interconnection networks," *SIGMETRICS Perform. Eval. Rev.*, vol. 18, no. 1, pp. 192–202, Apr. 1990. [Online]. Available: http://doi.acm.org/10.1145/98460.98758 118

[102] C. H. Xia and Z. Liu, "Queueing systems with long-range dependent input process and subexponential service times," in *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '03. New York, NY, USA: ACM, 2003, pp. 25–36. [Online]. Available: http://doi.acm.org/10.1145/781027.781032 118

[103] T. C. Xu, P. Liljeberg, and H. Tenhunen, "Optimal memory controller placement for chip multiprocessor," in *CODES+ISSS*, 2011, pp. 217–226. [Online]. Available: http://doi.acm.org/10.1145/2039370.2039405 93, 119

[104] W. Zhou, Y. Zhang, and Z. Mao, "Pareto based multi-objective mapping ip cores onto noc architectures," in *IEEE Asia Pacific Conference on Circuits and Systems, 2006.*, pp. 331 –334. 93, 119