RESOURCE MANAGEMENT ALGORITHMS FOR COMPUTING HARDWARE DESIGN AND OPERATIONS: FROM CIRCUITS TO SYSTEMS

A Dissertation

by

HAO HE

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

December 2016

Major Subject: Computer Engineering

ABSTRACT

The complexity of computation hardware has increased at an unprecedented rate for the last few decades. On the computer chip level, we have entered the era of multi/many-core processors made of billions of transistors. With transistor budget of this scale, many functions are integrated into a single chip. As such, chips today consist of many heterogeneous cores with intensive interaction among these cores. On the circuit level, with the end of Dennard scaling, continuously shrinking process technology has imposed a grand challenge on power density. The variation of circuit further exacerbated the problem by consuming a substantial time margin. On the system level, the rise of Warehouse Scale Computers and Data Centers have put resource management into new perspective. The ability of dynamically provision computation resource in these gigantic systems is crucial to their performance. In this thesis, three different resource management algorithms are discussed. The first algorithm assigns adaptivity resource to circuit blocks with a constraint on the overhead. The adaptivity improves resilience of the circuit to variation in a cost-effective way. The second algorithm manages the link bandwidth resource in application specific Networks-on-Chip. Quality-of-Service is guaranteed for time-critical traffic in the algorithm with an emphasis on power. The third algorithm manages the computation resource of the data center with precaution on the ill states of the system. Q-learning is employed to meet the dynamic nature of the system and Linear Temporal Logic is leveraged as a tool to describe temporal constraints. All three algorithms are evaluated by various experiments. The experimental results are compared to several previous work and show the advantage of our methods.

DEDICATION

To my family.

ACKNOWLEDGMENTS

It has been a long way in pursuit of a doctoral degree. In the years I spent in Texas A&M, there were times of exuberance, disappointment, frustration, and ecstasy. Along the way I received great help from many people, some I am close with, some I admire and look up to, and some I do not even know the name of.

First and foremost, I want to thank my advisor, Professor Jiang Hu, for being a great guidance in research and in life. His tolerant attitude, strong work ethic and high standards for himself are true examples of what a professional should be like.

I would also like to express my gratitude for Jiafan Wang, Yiren Shen and Gongming Yang. There were many bumps and obstacles during the projects we worked together. They showed great patience and worked hard in the process.

I met many friends in Mathworks, Natick where I spent half a year doing my internship. I want to thank my manager Andy Bartlett for giving me the opportunity. I had many inspiring discussions with my colleague Evangelous Denaxas. And many of my best memories there are with Weijia Zhang, who is always up for the bigger challenge.

Lastly, I need to thank my family and Dr. Yining Huang for their support. They always bring up the best of me.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis (or) dissertation committee consisting of Professor Jiang Hu [advisor], Professor Paul Gratz and Professor Shuguang Cui of the Department of Electrical & Computer Engineering and Professor Dilma Da Silva of the Department of Computer Science & Engineering.

The experiments presented in section 1 were conducted in collaboration with Jiafan Wang of the Department of Electrical & Computer Engineering and were published in 2015 in an article listed in the Biographical Sketch.

All other work conducted for the thesis (or) dissertation was completed by the student independently.

Funding Sources

Graduate study was supported by research funding from Semiconductor Research Corporation.

NOMENCLATURE

LR	Lagrangian Relaxation
ABB	Adaptive Body Bias
FBB	Forward Body Bias
DAG	Directed Acyclic Graph
SSTA	Statistical Static Timing Analysis
PCA	Principal Component Analysis
NoC	Networks-on-Chip
QoS	Quality-of-Service
GS	Guaranteed Service
BE	Best Effort
TDM	Time Division Multiplexing
SAT	Boolean Satisfiability
ILP	Integer Linear Programming
CNF	Conjunctive Normal Form
LTL	Linear Temporal Logic
MDP	Markov Decision Process
AP	Atomic Proposition
DRA	Deterministic Rabin Automaton
DoD	Depth of Discharge
DRF	Dominant Resource Fairness

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. RESOURCE MANAGEMENT IN COMPUTER HARDWARE

1.1 Resource Management in Different Scope

As the scale of computation system grows, proper resource management is becoming a crucial part to wield the power of computer hardware. Some examples of the resources to be allocated and provisioned are memory, cpu time, power budget and network bandwidth. These resources are usually competed for among instances in a large distributed system. For example, a data center may have more than $10K$ servers running collaborative tasks to perform a single big data analysis job. The servers are not homogenous because the hardwares are usually replaced batch by batch in a three-year rotation. So the performance and power profile of the servers may vary by a large extent inside the data center. In addition, the power supply network forms a hierarchical architecture. Power budget is split among different domains, racks and ultimately server machines. The complexity of resource management is well demonstrated considering the power capping problem in presence of server heterogeneity, location and power supply architecture.

In addition, resource management exists in various level of the system. Take power management as an example. In the data center level, the voltage and frequency of processors can be tuned down and servers can be put into sleep state by the power manager. In the chip level, part of the circuit that is not essential to the overall performance can be downsized to save power. The nature of the power management in different level is also very different. Power consumption of a data center is highly dynamic because of the workload fluctuation. A proper power manager should adapt its strategy, for example the power budget allocation, in the run time. The time scale for such adaptation is in hours, days or even months. In contrast, the computer chips cannot be altered after manufacture. Power management in this scale is mostly in the design time.

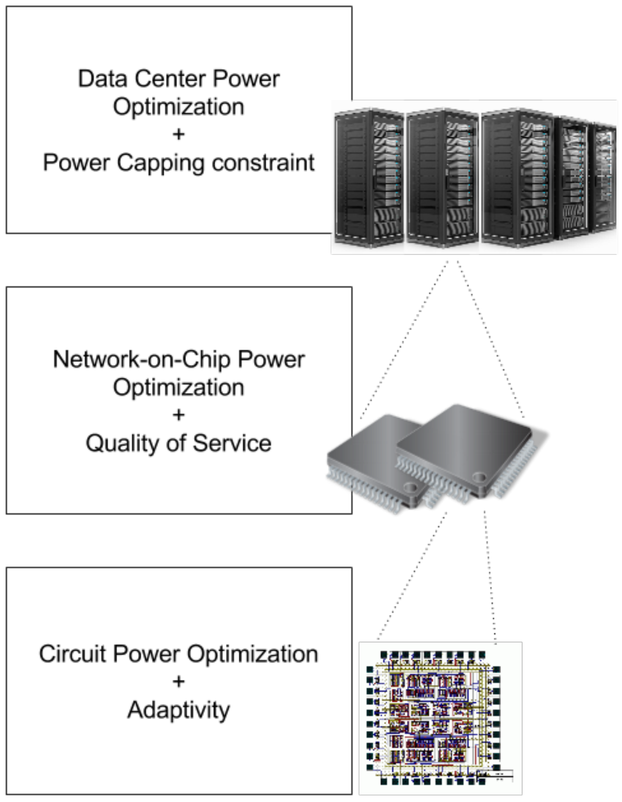

Data Center Power Optimization
+
Power Capping constraint
Network-on-Chip Power Optimization
+
Quality of Service
Circuit Power Optimization
+
Adaptivity

Figure 1.1: Resource management in multiple level.

What makes the problem more complex is the other design goals that are intertwined with the power management. For instance, to combat variation in the circuit, the ability to adapt to these variations needs to be added to the circuit. Adaptivity and power is intertwined because the power reduction techniques usually squeeze the safety work margin of the digital circuit, thus increases the need of adaptivity. In network resource management, power reduction can only be considered when the Quality-of-Services is not hurt. In the design of a power management policy, these other design goals should also be taken into account.

1.2 An Algorithmic Approach

In this dissertation, we study the resource management algorithms in different level of the computing hardware. In section 2, a Lagrangian-Relaxation-based algorithm is designed to optimize the gate implementation of the circuit. The algorithm minimizes the circuit power consumption and leaves a sufficient safety margin to combat variation for the circuit with adaptivity injected. The algorithm is run in design time with a probabilistic modeling of how much variation the circuit may suffer in the run time. In section 3, we look at the opportunity to tailor a Network-on-Chip specifically for a priori knowledge of the traffic pattern. We show the possibility to configure the packet routing and link bandwidth using a power-aware algorithm. With the help of proper network resource management, Quality-of-Service can be achieved in a power efficient manner. In the last section, we propose a way to enhance conventional resource managers with new design goals and constraints. The method is based on Q-learning and Linear Temporal Logic (LTL). LTL is used to formulate desired property of the system and then transformed into a state machine running alongside the original resource manager. The state machine warns the original resource manager about potential violations of the desired property. Our experiments show the method attains better tradeoff compared to existing methods.

2. OPTIMIZATION FOR ADAPTIVE CIRCUIT[*]

2.1 Introduction

Since the invention of silicon-based Integrated Circuit (IC), the density of transistors on a chip has grown exponentially over the years. In 1997, the Intel Pentium II processor is made of 7.5 million transistors with 350nm technology. Less than twenty years later, the 22-core Xeon Broadwell-E5 processor registered a transistor count of 7.2 billions with 14nm technology. As the process technology of digital circuit approaches the fundamental physical limit, this trend is unlikely to continue. Nevertheless, the integration density of modern IC has brought up great challenges in chip design. One of these challenges is the effect of variation.

Variation is the difference in the attributes of transistors such as length, width and oxide thickness. Variation is introduced when the chips are fabricated or in use. Sources of variation include manufacture process [1], device aging [2] and thermal fluctuations [3]. Process variation is the result of imperfections in the manufacturing process. These imperfections include the diffraction of light used in lithography, the variation in thickness when the gate oxide is grown, and the random fluctuation in the number of impurity atoms doped into the channel region of a transistor. Device aging causes the wear-out effect of Hot Carrier Injection (HCI) and Bias Temperature Instability (BTI), hence degrades the speed of transistors unevenly throughout the lifetime of the aging circuit. The temperature fluctuations of a chip may come from the ambient temperature change or the power dissipation of the circuit itself. An increase in temperature typically causes a decline of carrier mobility and an increase of interconnect resistance, hence slows down the circuit.

[*]Part of this section is reprinted with permission from H. He, J. Wang, and J. Hu, "Collaborative gate implementation selection and adaptivity assignment for robust combinational circuits", in Low Power Electronics and Design, 2015 IEEE/ACM International Symposium on, 2015.

Because of the dwindling process technology, the variation of digital circuits has a higher and higher impact on the overall circuit. This imposes a big threat to the performance as well as the efficiency of the circuit. To compensate for the potential loss in the speed of transistors, digital designs often have to target for the worst degree of variation expected. Consequently, the transistors will be over-sized for the typical case, and thus consume more power and die area.

Adaptive circuit has been recognized as a power-efficient technology to overcome various variation throughout the entire life of hardware chips. Adaptive circuit contains variation sensors and tuning units in addition to the original circuit. The power and performance tradeoff of the circuit is tuned to compensate the variation after manufacture. Because of the existence of this post-silicon tuning mechanism, the chip design no longer has to tackle the worst-case scenario. Various tuning mechanism are proposed, such as body biasing [1], voltage adaptation [4], circuit reconfiguration [5].

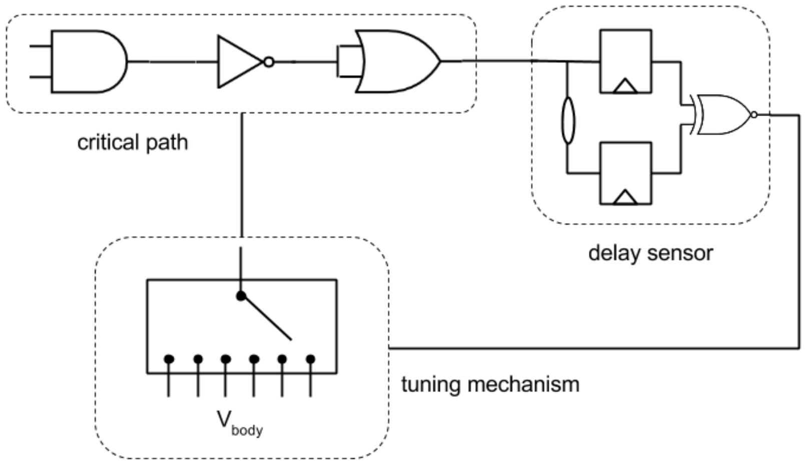


Figure 2.1: An example of adaptive circuit through body biasing.

The effectiveness and overhead of adaptive circuit relies largely on its granularity, or the amount of circuit that is tuned together. With fine-grained adaptivity [4, 6], the circuit is able to adjust itself for fine level of variation, but large area overhead is incurred (50% area overhead for voltage interpolation [7] and 20% in [5, 6]). In contrast, coarse-grained adaptivity is applied to the entire die uniformly and is mostly for inter-die variations only. So it has relatively small overhead (0.2% reported in [8]).

The granularity problem limits the application of adaptive circuits. Coarse-grained adaptive circuit must adapt to the worse-case variation across the die, therefore it may either boost up the not-so-bad parts, causing unnecessary power increase, or miss the opportunity to save power in those parts. The fine-grained adaptivity has the ability to pinpoint the right tuning for circuit blocks of hundreds or thousands of gates. However without a careful design, the area overhead of the additional adaptivity logic could easily out shadow its benefits.

In this work, we develop a general algorithm to optimize the adaptive circuit design with overhead control. Evidently, variation must be accounted and this makes the optimization problem rather difficult. We make use of Lagrangian relaxation (LR) that solves a multi-objective problem in two layers – subproblem and dual problem. The subproblem is focused on solution search for the weighted multiple objectives while the dual problem employs variability-aware models to guide the tradeoff among multiple objectives. As such, accurate models are used in a lightweight manner without causing too long runtime.

Our work provides a relatively complete adaptivity assignment solution for general adaptive circuit designs while the works of [9, 10] focus on clustering and ABB. Compared to [11], which is restricted to linear, continuous models, our work is a discrete approach and compatible with realistic models in industry. Moreover, area overhead is explicitly handled in our work but neglected in [11]. Experimental results show that our method usually reduces adaptivity overhead by more than a half compared to a naïve approach.

When gate area is counted together, our method often reduces the overall overhead by more than 70%.

2.2 Related Work

In adaptive VLSI circuits, power/performance is tuned by body bias [1], voltage adaptation [4], circuit reconfiguration [5] or a combination of them. The purpose of the adjustment is to compensate performance variability due to manufacturing process variations, device aging, thermal fluctuations, etc.

The effectiveness as well as overhead of adaptive design highly depend on its granularity. Coarse-grained adaptivity, such as uniform adaptivity for an entire processor core, has relatively small amortized overhead (0.2% reported in [8]). However, coarse-grained adaptivity is mostly for compensating inter-die variations. When intra-die variations are more pronounced [12], fine-grained adaptivity [4,6] (in blocks of hundreds or thousands of gates) brings significantly more power savings. Evidently, fine-grained adaptivity tends to entail large overhead of sensors, voltage regulators and control circuits (50% area overhead for voltage interpolation [7] and 20% in [5, 6]).

Obviously, one prefers the power savings from fine-grained adaptivity but not its large overhead. The adaptivity overhead has been mentioned in several previous works [5–7], however, it has rarely been a main emphasis. The objective of [9, 10] is to minimize the overhead of adaptive body bias (ABB), but it assumes ABB is applied to all clusters. Another work [13] restricts variation sensors only at timing critical paths so that the overhead is not excessively large. However, it does not consider control or voltage generation overhead.

Adaptive design is highly related with conventional circuit optimization. A joint design-time and post-silicon tuning optimization algorithm is proposed in [11]. It assumes that gate size can be continuously changed while most of modern designs are based on highly discrete cell libraries. Variability-aware discrete gate sizing is discussed in [14, 15]. These works are focused on how to propagate statistical timing information during sizing without much emphasis on the optimization aspect.

2.3 Background

2.3.1 Lagrange Relaxation

Lagrange relaxation is a mathematical optimization framework that converts a difficult constrained optimization problem to a simpler relaxed formulation. By solving this relaxed formulation, it is possible to obtain an approximation to the optimal solution of the original problem. When the problem has a certain structure, it can be proven that the solution of the Lagrange relaxation problem is actually optimal.

In the most general form, an optimization problem solves

$$\begin{aligned} \mathit{min}_x \quad & f(x) \\ \mathit{s.t.} \quad & g_i(x) \leq 0 \\ & h_j(x) = 0 \end{aligned} \tag{2.1}$$

The problem has an objective function $f(\cdot)$, several inequality constraints $g_i(\cdot)$ and several equality constraints $h_j(\cdot)$. Lagrange relaxation of the problem is

$$\begin{aligned} \mathit{max}_{u \geq 0, v} \quad \mathit{min}_x \quad & \mathcal{L}_{u,v}(x) \\ \mathcal{L}_{u,v}(x) = f(x) + \sum_i u_i g_i(x) + \sum_j v_j h_j(x) \end{aligned} \tag{2.2}$$

The Lagrange function $\mathcal{L}_{u,v}(x)$ can be viewed as a relaxation of the original problem. Instead of having hard equality and inequality constraints, the Lagrange relaxation allows constraint violations but penalizes the constraint violations in the objective. $\mathit{min}_x \quad \mathcal{L}_{u,v}(x)$ is called the primal problem, and $\mathit{max}_{u \geq 0, v} \mathcal{D}(u, v) = \mathit{max}_{u \geq 0, v} \quad \mathit{min}_x \quad \mathcal{L}_{u,v}(x)$ is called the dual problem. u, v are called dual variables or Lagrange multipliers. Note the range of the dual variables for inequality constraint is the half space $u \geq 0$.

$\mathcal{L}_{u,v}(x)$ has several interesting characteristics. Firstly, it is a linear function of dual variables u and v. This means $min_x \quad \mathcal{L}_{u,v}(x)$ is concave with respect to u, v. Secondly, the original problem can be viewed as a min-max problem

$$\begin{aligned} &min_x \quad max_{u \geq 0, v} \quad \mathcal{L}_{u,v}(x) \\ &\qquad\qquad \mathcal{L}_{u,v}(x) = f(x) + \sum_i u_i g_i(x) + \sum_j v_j h_j(x) \end{aligned} \tag{2.3}$$

If $g_i(x) \leq 0$ and $h_j(x) = 0$, then the maximizer of $\mathcal{L}_{u,v}(x)$ is $u_i = 0$. Otherwise, $max_{u \geq 0, v} \mathcal{L}_{u,v}(x) = \infty$ because either a coefficient of u_i is positive, or a coefficient of v_j is non-zero. So the minimization over x in the front will ensure the former condition is met, which means feasibility of the equality and inequality constraints. As such, the Lagrange relaxation swaps the minimization and the maximization operation. It can be shown the min-max problem (original problem) is equivalent to the max-min problem (Lagrange relaxation) when f, g_i, h_j are all convex functions.

2.3.2 Gate Sizing

Gate sizing is a well-studied topic in physical design of digital circuits. The input for the gate sizing process is a circuit with timing requirements. The algorithm adjusts the gate sizes to increase its power efficiency and to satisfy the timing constraint of the circuit.

Specifically, the circuit under design is represented as a Directed Acyclic Graph (DAG). The primary inputs of the circuit are the nodes in the graph without fanin. The primary outputs of the circuit are the nodes without fanout. The timing requirements are the signal arrival time a on the primary inputs and required arrival time q on the primary outputs. The timing requirement is that all signals that arrives at the primary inputs at a are transmitted to primary outputs no later than q.

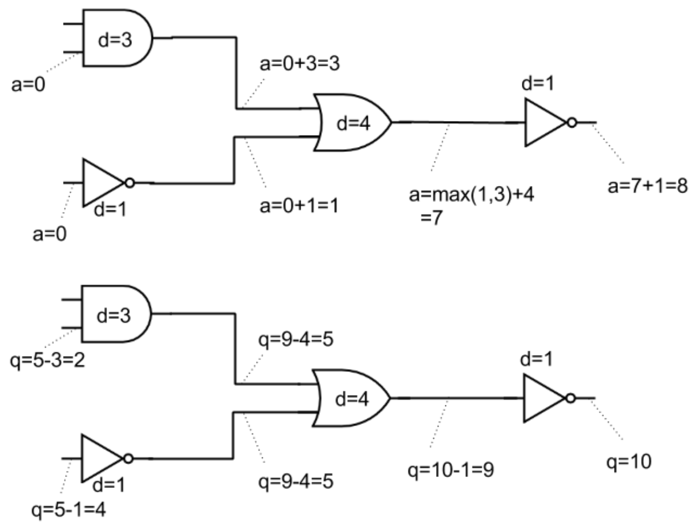
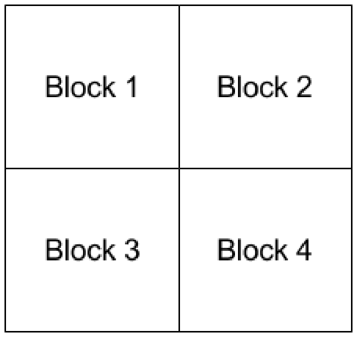


Figure 2.2: Timing constraint of circuit design.

Each gate has a discrete number of size options $1\times, 2\times, ...$. The size option has an effect on the power consumption as well as the gate delay. Gate sizing corresponds to the problem to find a size option for each gate such that the timing constraint is respected and the total circuit power is minimized.

2.3.3 Variation Modelling

There are two primary types of variations present in the circuit design. The first is inter-die variation, in which all devices in the chip shift towards the same variation but different chips bear different variations. The other is intra-die variation, in which the variation's impact on the same chip is not uniform. The intra-die variation can cause the components of the chip design to diverge from the design specifications. This poses a threat to the functionality and performance of the chips. For example, the timing profile of a computer chip may become different from the critical path analysis in the design time. In contrast, the inter-die variation is relatively easier to compensate because it is uniform across the entire chip.



$$\Sigma = \begin{pmatrix} \sigma_{11} & \sigma_{12} & \sigma_{13} & \sigma_{14} \\ \sigma_{21} & \sigma_{22} & \sigma_{23} & \sigma_{24} \\ \sigma_{31} & \sigma_{32} & \sigma_{33} & \sigma_{34} \\ \sigma_{41} & \sigma_{42} & \sigma_{43} & \sigma_{44} \end{pmatrix}$$

Figure 2.3: Correlation matrix is defined to model the variation.

There can be multiple sources for the intra-die variation such as the variations in channel length and gate oxide thickness. The variations from different sources are usually assumed statistically independent. For example, the delay of a single gate d is calculated as the Taylor expansion

$$d = d_0 + \sum_i \frac{\partial d}{\partial p_i} \Delta p_i \tag{2.4}$$

where p_i denotes a single source of variation, $\frac{\partial d}{\partial p_i}$ is the derivative of the delay model with respect to p_i, d_0 is the nominal delay.

Note variation Δp_i in quation 2.4 is actually a function of gate position (x, y). The intra-die variation $\Delta p_i(x, y)$ is often modeled as a multivariate Gaussian vector over the spatial location of the 2D circuit [16] [17]. Suppose a circuit is partitioned into 4 blocks, we can define the correlation of each pair of blocks in a 4×4 matrix Σ. The delay variation of the gates in each block then follows $\delta_{delay} \sim N(0, \Sigma)$.

2.3.4 Adaptive Body Biasing

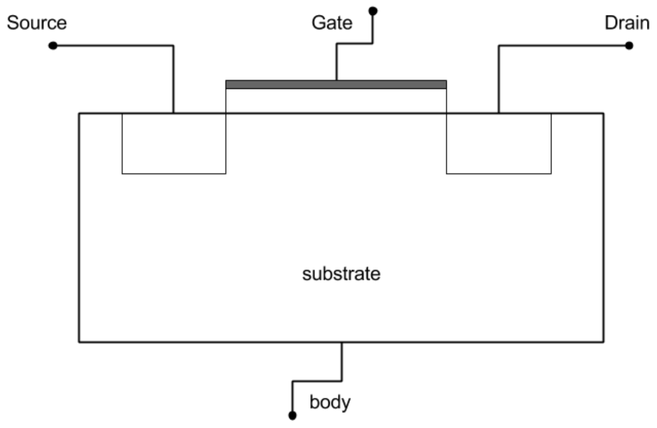


Figure 2.4: Body biasing a transistor.

Figure 2.4 shows the cross section through a MOSFET transistor. Body biasing is a technology to apply voltage to the substrate of the device. Reverse body biasing, i.e. applying a negative biasing voltage, is often used to reduce leakage power of the transistor. The bias voltage V_{bs} affects both the delay and the leakage power of the transistor.

$$\begin{aligned}
V_{th} &= V_{th0} + \gamma(\sqrt{2\phi_F + V_{bs}} - \sqrt{2\phi_F}) \\
P_{leakage} &= V_{dd}I_{subn} + |V_{bs}|(I_{jn} + I_{bn}) \\
I_{subn} &= \frac{W}{L}I_S[1 - e^{-\frac{V_{dd}}{V_T}}]e^{-\frac{V_{th}+V_{off}}{nV_T}} \\
d &= \frac{d_0}{(V_{dd} - V_{th})^\alpha}
\end{aligned} \tag{2.5}$$

$P_{leakage}$ is the leakage power. d is the gate delay. V_{dd} is the power supply voltage. I_{subn} is the sub threshold leakage current. $V_{th0}, \gamma, \phi_F, I_{jn}, I_{bn}.I_S, V_T, W, L, V_{off}, d_0, \alpha$ are device parameters or empirical constants. It is shown in [9] that $P_{leakage}$ can be approximated by a quadratic function of V_{bs} and d by a linear function.

2.3.5 Static Timing Analysis

Static Timing Analysis (STA) is a tool to analyze the timing of the combinational circuit in design time. The input of STA is the circuit topology, delays of gates in the circuit, arrival time a and required arrival time q in the primary inputs and outputs of the circuit. Figure 2.5 shows the a simple example of STA. Each node in the figure represents a gate and the delay is given by d. The results of STA are the arrival time a and required arrival time q on each gate (gates that are not primary inputs or primary outputs).

The arrival time a is calculated by a forward traversal of the circuit graph. The arrival time at the output of a gate is given by $a_g = max_{i \in fanin(g)}(a_i + d_g)$. That is, the arrival time is the worst case arrival time of its fanins plus the gate delay. The arrival time shows when a signal will propagate to the gate in the worst case. Similarly, the required arrival time is calculated by a backward traversal. The required arrival time at the input of a gate is given by $q_g = min_{i \in fanout(g)}(q_i) - d_g$. The required arrival time shows the time before which a signal must arrive at the point. Once the arrival time and required arrival time are obtained for every point in the graph, the slack is calculated as $q - a$. It shows the how tight the timing requirement is at that point.

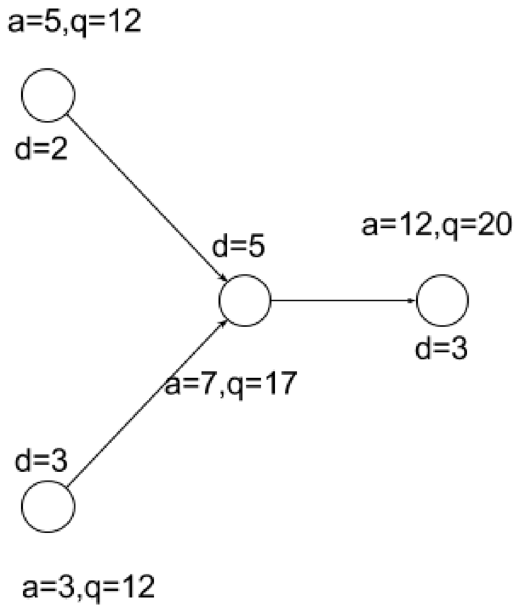


Figure 2.5: An example for Static Timing Analysis.

2.4 Problem Formulation

The input to our algorithm is a placed combinational circuit, timing constraints and adaptivity clusters. The combinational circuit can be viewed as a DAG. The nodes in this graph correspond to digital gates in the combinational circuit. The edges represent wire connections among gates. A combinational circuit has a set of primary inputs and outputs, through which the signals flow in and out of the circuit of interest. For all the primary inputs, the arrival time a_i of signals are specified. For all the primary outputs, the required arrival time q_j are specified. Timing constraint is the requirement that the longest time needed for a signal to propagate from input i to output j is less than $q_j - a_i$. In addition, we assume the circuit has been clustered into a set of blocks $\mathcal{B} = \{B_1, B_2, ...\}$. These blocks are the candidates to which the algorithm will assign adaptivity. If a block is assigned with adaptivity, the full set of variation sensor, tuning unit and control policy implementation will be added to the design for the selected block. Once a tuning level (the voltage of body-biasing, the power supply of a dual Vdd gate, etc.) is set by the tuning unit, all gates in the same block will be affected. Note our algorithm decides whether to assign adaptivity to each block in the design time, but the tuning level is determined by the variations of the circuit and set in run time. The optimization objective and constraints include power, timing, robustness to variations and area overhead.

2.4.1 Placement and Clustering

To find the appropriate clustering for a combinational circuit is a non-trivial task. Several aspects need to be taken into consideration. First of all, the variation of the clustered gates should be statistically correlated, since these gates are tuned together. The gates on the same signal path, for example, have correlated aging effect because they are usually turned on and off together. Secondly, gates in a circuit are not equally important for the timing constraint of the whole circuit. When variation is not present, timing of a digital circuit is determined by a set of critical paths. In the event of variation, path delay is a random variable, so the non-critical paths in the nominal sense may become critical. However the gates still impose an uneven impact to the final timing. As such, we want to make the cluster contain the most impactful gates. In addition, to properly implement the tuning mechanism by circuit, it is often required for the gates in a cluster to be placed in a contiguous region.

In our experiment, we only use the spatial location of the placed gates to form the clusters $\mathcal{B} = \{B_1, B_2, ...\}$. However, several work [9] [18] have been published on the clustering and placement of adaptive circuit. These more sophisticated algorithms can be easily incorporated into our adaptivity assignment method.

2.4.2 Gate Implementation Selection

Gate implementation selection is closely coupled with adaptivity assignment problem, therefore our method performs a joint optimization of the two tasks. Gate implementation selection is a well-studied area in physical design of digital circuit. After the logic design of a digital circuit is finalized, the physical implementation of the gates in the circuit plays an important role on the correctness metric (e.g. timing) as well as the efficiency metric (e.g. power). Since the major concern of circuit variation is on the timing of the circuit, it is possible to rely solely on the gate implementation selection to combat the variation. However, this approach would be inefficient in the sense that all the manufactured chips must be designed to handle the worst-case variation, while only a small fraction will actually experience the worst-case variation. With the option to make part of the circuit adaptive, this worst-case oriented design can be avoid. Since adaptivity itself also incur an implementation cost, we rely on the algorithm to determine the right trade-off between adaptivity and gate implementation selection.

In our work, we consider two kinds of gate implementation options: gate size and threshold voltage. The larger size a gate is implemented in circuit, the faster the gate is and the more die area it consumes. Similarly, a decrease in the threshold voltage of a gate will cause an increase in speed as well as leakage power consumption.

Apart from gate size and threshold voltage, we consider the case of Adaptive Body Biasing (ABB). The adaptive circuit tunes the biasing voltage of the circuit blocks to compensate for variation. The power and delay model for a single gate v is

$$\begin{aligned} power_v(V_{th}, X_{size}, V_{bias}) &\approx leakage^v_{V_{th},X_{size}} * quadratic(V_{bias}) \\ &\quad + \alpha V_{dd}^2 f \sum_{u \in fanout(v)} C^u_{V_{th},X_{size}} \\ delay_v(V_{th}, X_{size}, V_{bias}) &\approx linear(V_{bias}) * R^v_{V_{th},X_{size}} * \sum_{u \in fanout(v)} C^u_{V_{th},X_{size}} \end{aligned} \tag{2.6}$$

In equation 2.6, X_{size} denotes the size of the gate, V_{th} denotes the threshold voltage, V_{bias} denotes the body biasing voltage level. $leakage_{V_{th},X_{size}}$,$R_{V_{th},X_{size}}$,$C_{V_{th},X_{size}}$ are the leakage power, output resistance and input capacitance of a gate, respectively. These parameters for available V_{th}, X_{size} can be found in a standard cell library. Previous work [9] shows the effect of body biasing on power can be accurately approximated by a quadratic function $quadratic\ (V_{bias})$ and the effect on delay by $linear\ (V_{bias})$.

2.4.3 Adaptivity Assignment

For each block $\mathcal{B} = \{B_1, B_2, ...\}$, the 0-1 decision variable $\Phi(B_i)$ chooses whether the block should become adaptive. If the block is adaptive, the tuning mechanism can offset the body biasing level of the block to one of several discrete options $V^1_{bias}, V^2_{bias}, ..., V^n_{bias}$. The body biasing voltage affects the power and delay of the circuit as formulated in 2.6. An adaptive block enables the optimization to choose smaller size and higher threshold voltage implementation for the gates, and thus reduce the average-case power consumption. However, it also pays an area overhead for implementing the tuning mechanism in circuit. We approximates these overhead using a linear function.

$$A(B_i) = k \sum_{v \in B_i} Area_v + b \tag{2.7}$$

2.4.4 Collaborative Gate Implementation Selection and Adaptivity Assignment

We formulate the gate implementation selection and adaptivity assignment problem as a joint optimization.

$$\begin{aligned} min \quad & E_{V_{bias}}(power(\vec{\boldsymbol{\xi}}, V_{bias})) \\ \textit{subject to} \quad & max_{V_{bias}} \mathcal{P}(a_v + delay_v(\vec{\boldsymbol{\xi}}, V_{bias}) + \delta_{delay_v} < a_u) > \textit{probability limit} \\ & A(G) = \sum_i A(B_i)\Phi_{B_i} < \textit{overhead limit} \end{aligned} \tag{2.8}$$

We let $\vec{\boldsymbol{\xi}} = (V_{th}, X_{size})$ for simple notation. G is the circuit. a_v is the signal arrival time of gate v. a_v for the primary inputs and outputs are constants obtained from the design. a_v for the intermediate nodes are variables set by the optimization.

In this formulation, the objective is set to minimize the expected power consumption of the circuit. The expectation is over the body biasing level V_{bias} tuned by the adaptive circuit. Because the variation is random, the tuning level V_{bias} is also random. To calculate $E_{V_{bias}}(power(\vec{\boldsymbol{\xi}}, V_{bias}))$, the probability of each tuning level is required. We assume a monotonic relation between the minimum slack of the circuit and the tuning level. The probability is calculated by quantizing the distribution of the minimum slack in the circuit.

$$\begin{aligned} E_{V_{bias}}(power(\vec{\boldsymbol{\xi}}, V_{bias})) &= \sum_i \mathcal{P}(V^i_{bias} * power(\vec{\boldsymbol{\xi}}, V^i_{bias}) \\ \mathcal{P}(V^i_{bias}) &= \mathcal{P}(l_i < \textit{min slack} < h_i) \end{aligned} \tag{2.9}$$

The timing constraint asserts the gate needs to be fast enough to propagate signal from v to u, when the tuning level V_{bias} is set to the the highest compensating level. Because the variation δ_{delay} is Gaussian, we can easily translate the probability limit to a variance limit. For example, 99.7% probability means the gate delay must have at least 3 standard deviation slack.

2.5 Overview of Adaptivity Assignment Algorithm

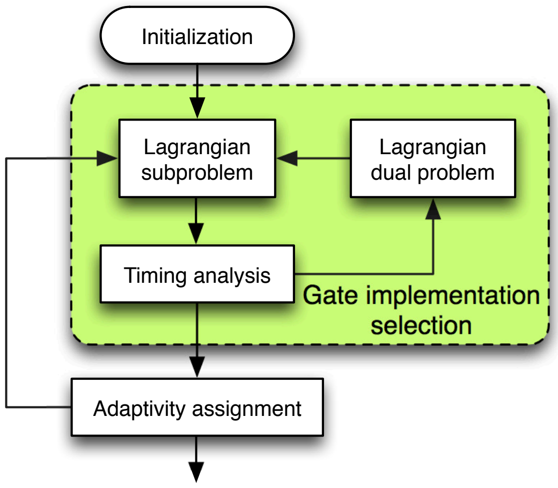


Figure 2.6: Overview of algorithm flow.

The overall algorithm is shown in figure 2.6. The algorithm iterates between gate implementation selection and adaptivity assignment. The gate implementation selection part is handled by Lagrangian relaxation (LR). Its formulation is to minimize power dissipation subject to timing constraints with consideration of variations. Area is not explicitly in the formulation as power and area are correlated in gate sizing. By solving the problem in two layers of Lagrangian subproblem and dual problem, the calls to SSTA can be restricted to the dual problem part. Then, the subproblem can be solved using simple models while the overall solution quality is not compromised due to the SSTA guidance in solving the dual problem. The problem size of adaptivity assignment is significantly smaller and allows SSTA to be called more frequently. Therefore, the adaptivity assignment is solved by a sensitivity-based heuristic.

2.5.1 Lagrange Primal Problem: Gate Implementation Selection

The primal problem follows the standard Lagrangian Relaxation framework. The constrained optimization problem described in 2.8 is transformed to an unconstrained primal problem. In this step, the adaptivity assignment is fixed. This means whether a block has adaptivity is known for the primal problem.

$$min \quad \mathcal{L}_{\vec{\mu}}(\vec{\boldsymbol{\xi}}) = E(power(\vec{\boldsymbol{\xi}})) + \sum_{v} \mu_v(delay_v(\vec{\boldsymbol{\xi}}) + \delta_{delay_v}) \tag{2.10}$$

where $\vec{\mu}$ is the Lagrangian dual variables, $\vec{\xi}$ is the decision variables for gate size and threshold voltage [19]. $power_v(\cdot)$ is the power model of an individual gate. $delay_v(\cdot)$ is the delay model of the gate v.

The objective is a weighted sum of power consumption and timing constraint, the Lagrangian dual problem is responsible for finding the optimal weights μ_v to balance power and speed. Because the problem is combinatorial, we employ a dynamic-programming-based previous work [20] to solve it. Here we use a simple example to show the algorithm.

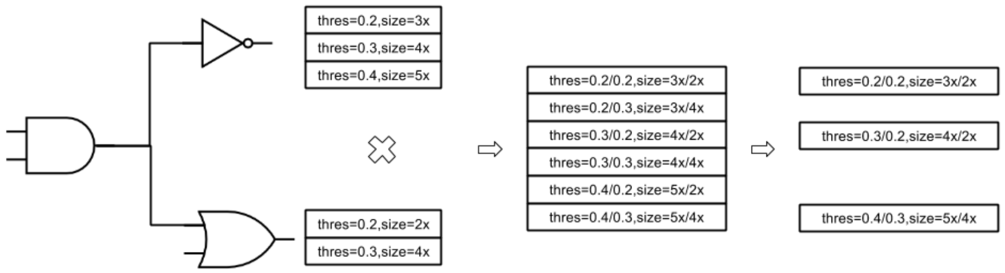


Figure 2.7: An example of dynamic programming algorithm.

Figure 2.7 shows a simple circuit of 3 gates. The algorithm starts from the primary outputs and work its way back in reverse topological order. The NOT gate has 3 options for gate implementation and the OR gate has 2 options. Each option achieves a different trade-off between the power consumption and the speed of the gate. Because the NOT gate and the OR gate are only examined individually at this stage, it is not possible to tell which option is better. So all options are kept in the gates. In the next stage, the upstream AND gate is examined. A cross product of the options of its fan-out gates is performed. This includes all possible combination of gate implementation of two branches of the AND gate. Then, the algorithm carries out a pruning operation on the 6 cross product options. An option is said to be "inferior" if both the weighted objective and the input capacitance are larger than the other option. The weighted objective indicates the power and speed cost of downstream gates. The input capacitance determines the option's effect on upstream gates. After the pruning operation, only the options that are in the Pareto frontier of upstream-downstream tradeoff are kept.

2.5.2 Statistical Static Timing Analysis

Static Timing Analysis (STA) is widely adopted to analyze the timing aspect of digital circuit. In essence, STA finds the worst-case signal arrival time of a gate's inputs, and add the delay of the gate itself. When large variation is present, the gate delay becomes a random variable. More importantly, the delay of different gates are correlated because the variation is correlated. A proper timing analysis is supposed to represent and calculate the correlation among gates in the circuit. To this end, several Statistical Static Timing Analysis (SSTA) methods are proposed. In this work, we use a previous work [17] to perform SSTA for the circuit under design.

The work in [17] models the spatial correlation of variation as a Gaussian multivariate function. The circuit is first partitioned into blocks $L_1, L_2, ..., L_n$ and a $n \times n$ correlation matrix Σ is specified. Σ captures the correlation of any two blocks. Then a Principal

Component Analysis (PCA) is performed. The result of PCA enables the representation of gate delay variation in the form

$$delay_v = d + \sum_i k_i \epsilon_i \tag{2.11}$$

where d is the nominal delay that is calculated through the same procedure as STA, k_i are the PCA coefficients and ϵ_i are standard Gaussian variables $\epsilon_i \sim N(0,1)$ that are independent with each other.

The power of this representation lies in its ability to compute correlation quickly. For example, if gate A has delay $d_A + 3\epsilon_1 - 2\epsilon_2$ and gate B has delay $d_B + 2\epsilon_1 + 2\epsilon_2$, then $corr(A,B) = E[(3\epsilon_1 - 2\epsilon_2) * (2\epsilon_1 + 2\epsilon_2)] = E[6\epsilon_1^2 - 4\epsilon_2^2] = 2$.

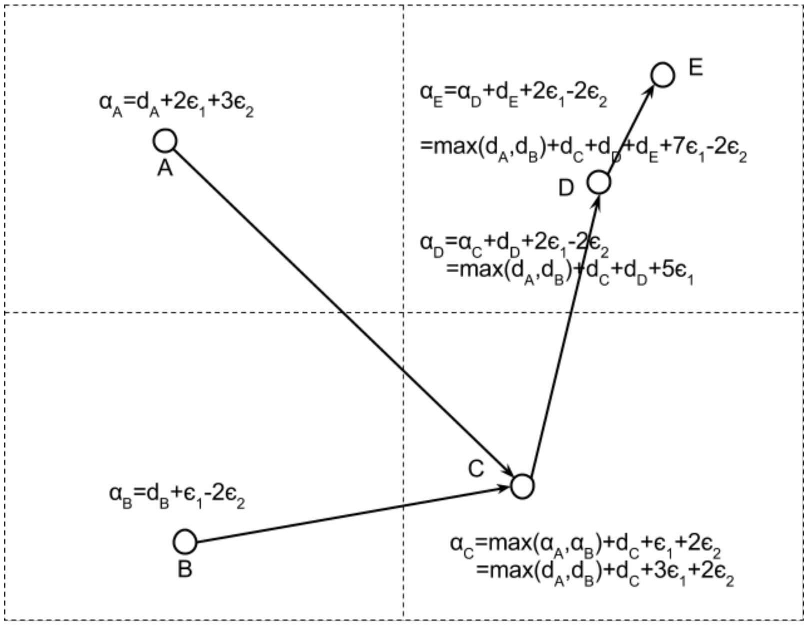

$\alpha_A = d_A + 2\epsilon_1 + 3\epsilon_2$
A
$\alpha_E = \alpha_D + d_E + 2\epsilon_1 - 2\epsilon_2$
$= \max(d_A, d_B) + d_C + d_D + d_E + 7\epsilon_1 - 2\epsilon_2$
E
D
$\alpha_D = \alpha_C + d_D + 2\epsilon_1 - 2\epsilon_2$
$= \max(d_A, d_B) + d_C + d_D + 5\epsilon_1$
C
$\alpha_B = d_B + \epsilon_1 - 2\epsilon_2$
B
$\alpha_C = \max(\alpha_A, \alpha_B) + d_C + \epsilon_1 + 2\epsilon_2$
$= \max(d_A, d_B) + d_C + 3\epsilon_1 + 2\epsilon_2$

Figure 2.8: An example of SSTA.

Again we use an example to show how SSTA works. Figure 2.8 is an example of circuit broken into 4 blocks. After performing PCA on the correlation matrix of the 4 blocks, we obtain the variational delay of gate A,B,C,D,E in terms of two independent Gaussian variables ϵ_1 and ϵ_2. Note gate D and E have the same variation because they are in the same block. We assume gate A and B are the primary inputs that input signals arrive at time 0. Then the signal arrival time at the output of each gate $\alpha_A, \alpha_B, \alpha_C, \alpha_D, \alpha_E$ can be calculated by performing a topological order traversal of the graph. For single input gates, the variation on its output is the sum of variation on its input and the variation of the gate delay. Hence the output variation is still Gaussian. When a gate has multiple inputs, a $MAX(\cdot)$ operation is applied, and the output variation is no long Gaussian. But we can calculate a linear approximation [17] of the resulting variation such that the computation is tractable.

From SSTA we can also obtain the distribution of timing slack (the slack of the gate delay beyond which the circuit will fail). Timing yield of a circuit is the probability that any gate in the circuit has a negative slack.

2.5.3 Lagrange Dual Problem

The dual problem solves $\mu_{u,v}$ in equation 2.10. It is well known the dual problem

$$max_{\vec{\mu}} \quad \mathcal{D}_{\vec{\mu}} = max_{\vec{\mu}} \quad min_{\vec{\boldsymbol{\xi}}} \quad \mathcal{L}_{\vec{\mu}}(\vec{\boldsymbol{\xi}}) \tag{2.12}$$

is a concave problem, and a subgradient for μ_v is

$$\frac{\partial \mathcal{D}_{\vec{\mu}}}{\partial \mu_v} = delay_v(\vec{\boldsymbol{\xi}}) \tag{2.13}$$

We use subgradient ascent to update μ_v at each inner iteration of figure 2.6.

Algorithm 1 Sensitivity-based adaptivity assignment.

```
Require: Circuit $G$ composed by blocks $\mathcal{B} = \{B_1, B_2, ...\}$
 1: Timing yield constraint $\Upsilon$
 2: Adaptivity area constraint $\Omega$
 3: Perform statistical static timing analysis
 4: while (true) do
 5:     if $Y(G) < \Upsilon$ then
 6:         $mode \leftarrow timing$  $\mathcal{B}^* \leftarrow \{B_i | \Phi(B_i) = 0\}$
 7:     else
 8:         $mode \leftarrow overhead$  $\mathcal{B}^* \leftarrow \{B_i | \Phi(B_i) = 1\}$
 9:     end if
10:     $\alpha \leftarrow A(G) > \Omega$ ? 1 : 0 ; // $A(G)$: area overhead
11:     $B^* \leftarrow \arg\max_{B \in \mathcal{B}^*} \theta(B)$
12:     $\Phi(B^*) \leftarrow \Phi(B^*)$ ? 0 : 1 ; // Trial adaptivity change
13:     perform SSTA, obtain $Y^*(G)$ and $A^*(G)$
14:     if $mode == timing$ and $Y^*(G) > Y(G)$ then
15:         continue
16:     end if
17:     if $mode == overhead$ then
18:         if $\alpha$ and $A^*(G) < A(G)$ then
19:             continue
20:         end if
21:         if (power reduced) and $A^*(G) \leq \Omega$ then
22:             continue
23:         end if
24:     end if
25:     $\Phi(B^*) \leftarrow \Phi(B^*)$ ? 0 : 1 ; // Undo adaptivity change
26: end while
```

2.6 Adaptivity Assignment Algorithm

Our algorithm iteratively assigns or deassigns adaptivity for a block. Depending on if timing yield $Y(G)$ evaluated by an SSTA, each iteration may be in either timing mode or overhead mode. If $Y(G) < \Upsilon$, the timing mode tries to add adaptivity to improve circuit robustness. If $Y(G) \geq \Upsilon$, the overhead mode attempts to remove adaptivity from a block to reduce power and area overhead.

Timing mode sensitivity $\theta_t(B)$ for a block B is defined by

$$\begin{aligned}\theta_t(B) &= \sum_{g\in B} \Delta_{delay}(g) \cdot \sqrt{\psi_t(g)} \\ \Delta_{delay}(g) &= (d_g(\phi_0) - \sum_{i=1}^{max} P_B(\phi_i) \cdot d_g(\phi_i))\end{aligned} \tag{2.14}$$

where $d_g(\phi_i)$ is the gate g delay with adaptivity level ϕ_i and $P_B(\phi_i)$ is the probability that block B operates with adaptivity level ϕ_i. The number of timing critical paths $\psi_t(g)$ passing through g is defined by

$$\psi_t(g) = \sum_{u\in fanin(g),fanout(g)} \begin{cases} 1 & \text{if } \tilde{s}(u) \leq 0.5\tilde{s}_{min} \\ 0 & \text{otherwise} \end{cases} \tag{2.15}$$

where $\tilde{s}(u)$ is the slack at node u and $\tilde{s}_{min}$ is the minimum slack over the entire circuit. The tilde here indicates that they are mean plus certain σ (standard deviation) value.

The overhead mode sensitivity $\theta_o(B)$ is defined by

$$\begin{aligned}\theta_o(B) &= \frac{\sum_{g\in B} \Delta_W(g) + W_B}{\sum_{g\in B} \Delta_{delay}(g) \cdot \sqrt{\psi_o(g)}} \\ \Delta_W(g) &= (\sum_{i=1}^{max} P_B(\phi_i) \cdot w_g(\phi_i) - w_g(\phi_0))\end{aligned} \tag{2.16}$$

where $w_g(\phi_i)$ is the power dissipation of gate g when it is at adaptivity level ϕ_i and W_B is the adaptivity power overhead for block B. The number of timing critical paths $\psi_o(g)$ passing through g in the overhead mode is defined by

$$\psi_o(g) = \sum_{u\in fanin(g),fanout(g)} \begin{cases} 1 & \text{if } \tilde{s}(u) \leq 1.5\tilde{s}_{min} \\ 0 & \text{otherwise} \end{cases} \tag{2.17}$$

In timing (overhead) mode, the block B^* without (with) adaptivity and the maximum θ_t (θ_o) is selected. A trial adaptivity change is made for B^* based on the sensitivity. Then, we consider if to commit this trial change according to SSTA, area and power analysis. In timing mode, the commitment is based on timing yield improvement. In overhead mode, we first check adaptivity area overhead, which has a hard constraint. If the constraint is not satisfied and the area is reduced, then the change is committed. The third case for commitment is when power dissipation is reduced. The iteration continues as long as we see improvements on either of these three cases, and terminates when no such improvement is obtained. The pseudo code for this heuristic is given in Algorithm 0.

2.7 Experiment Result

Table 2.1: Naïve method with only forward body bias (FBB). Power overhead, total area overhead, number of adaptive blocks% are denoted by $\Delta W\ (\mu W)$, $\Delta A\ (unit)$, $\#B$, respectively.

			Baseline	Naïve			
Circuit	$\#$gates	$\lvert\mathcal{B}\rvert$	Yield	Yield	ΔW	$\Delta A/\#B$	CPU (s)
c432	171	4	94.9%	99.3%	6564	707/4	1
c499	218	5	91.6%	97.7%	10975	1433/5	1
c880	383	5	96.3%	98.9%	5123	809/4	1
c1355	562	4	88.8%	99.9%	26442	1587/4	2
c1908	972	6	75.9%	99.9%	19049	1380/4	4
c2670	1287	5	94.6%	98.2%	6156	947/2	5
c3540	1705	5	73.6%	99.9%	21952	1759/4	8
c5315	2351	6	90.9%	99.8%	29364	2602/4	10
c6288	2416	6	93.9%	99.9%	50323	1931/2	11
c7552	3625	5	41.8%	99.9%	42878	3291/4	18
usb_phy	609	6	88.0%	99.3%	1729	518/2	2
edit_dist	130661	29	81.3%	99.9%	15460	24640/5	804
fft	32281	20	81.2%	99.1%	194576	15742/5	310
cordic	41601	20	73.9%	99.5%	443511	22618/10	493
des_perf	112644	22	83.5%	99.2%	204159	43608/6	750
matrix_mult	155325	20	44.0%	99.1%	1382050	78028/14	1378
Average			80.8%	99.3%	153769	12600/4.9	237

Table 2.2: Our method with only forward body bias (FBB). Gate area overhead% is denoted by ΔA_g $(unit)$.

			Baseline	Ours				
Circuit	#gates	$\|\mathcal{B}\|$	Yield	Yield	ΔW	$\Delta A/\#B$	ΔA_g	CPU (s)
c432	171	4	94.9%	99.9%	2524	323/1	7%	1
c499	218	5	91.6%	99.9%	3688	355/2	-26%	3
c880	383	5	96.3%	99.3%	1790	504/1	14%	3
c1355	562	4	88.8%	99.4%	12922	388/2	-17%	5
c1908	972	6	75.9%	99.8%	9162	762/2	-5%	9
c2670	1287	5	94.6%	99.1%	544	176/1	-7%	12
c3540	1705	5	73.6%	99.6%	13924	603/2	-13%	16
c5315	2351	6	90.9%	99.2%	3350	293/1	0%	24
c6288	2416	6	93.9%	98.9%	10549	1248/1	4%	24
c7552	3625	5	41.8%	99.9%	20053	404/3	-16%	40
usb_phy	609	6	88.0%	99.3%	779	167/1	-3.9%	6
edit_dist	130661	29	81.3%	99.3%	4430	7781/2	0%	1937
fft	32281	20	81.2%	99.2%	32167	10376/3	0%	759
cordic	41601	20	73.9%	99.1%	146446	9590/4	0%	1141
des_perf	112644	22	83.5%	99.4%	5726	15060/2	0%	1795
matrix_mult	155325	20	44.0%	99.0%	-200650	-47184/6	-15%	3193
Average			80.8%	99.4%	4213	15/2.1	-4.9%	561
% difference vs. naïve = $\frac{(ours-naïve)}{abs(naïve)}$					-97.3%	-99.8%/-57.1%		

To the best of our knowledge, there is no previous work on joint gate implementation selection and adaptivity assignment with consideration of overhead control. Therefore, we compare with the following approaches.

- Baseline. Variability-aware gate implementation selection without adaptivity. This is to emulate conventional non-adaptive designs.

- Naïve adaptivity assignment. If only forward body bias (FBB) is considered, adaptivity is assigned to any block that has negative slack in terms of mean plus certain σ value. This is to emulate what designers may do for adaptive circuit design without adaptivity optimization tools. In ABB where both FBB and reverse body bias are allowed, the naïve method simply assigns adaptivity for all blocks. Actually this is

Table 2.3: Naïve method with forward body bias and reverse body bias (ABB). Power overhead, total area overhead, number of adaptive blocks% are denoted by ΔW (μW), ΔA $(unit)$, $\#B$, respectively.

			Baseline	Naïve			
Circuit	#gates	$\lvert\mathcal{B}\rvert$	Yield	Yield	ΔW	$\Delta A/\#B$	CPU (s)
c432	171	4	99.7%	99.5%	1857	689/4	1
c499	218	5	99.9%	99.9%	664	1358/5	1
c880	383	6	99.9%	99.6%	1452	921/6	1
c1355	562	4	99.8%	99.1%	271	1354/4	2
c1908	972	6	99.5%	99.2%	2623	1550/6	4
c2670	1287	5	99.9%	99.9%	-435	1543/5	5
c3540	1705	5	99.9%	99.7%	-1481	1821/5	7
c5315	2351	6	99.9%	99.9%	-3949	2668/6	10
c6288	2416	6	99.9%	99.9%	-8302	2175/6	11
c7552	3625	5	99.9%	99.9%	-3307	3103/5	17
usb_phy	609	6	99.2%	99.2%	1529	1174/6	2
edit_dist	130661	29	99.3%	99.3%	-217934	72890/29	788
fft	32281	20	99.8%	99.7%	-137438	41061/20	297
cordic	41601	20	99.5%	99.3%	-160335	43106/20	488
des_perf	112644	22	99.4%	99.4%	-204797	67013/22	734
matrix_mult	155325	20	99.0%	99.3%	-287264	90859/20	1336
Average			99.6%	99.6%	-63550	20830/10.5	232

the approach of [9].

In the experiments, gates are modeled by RC switches and the Elmore delay model is employed. We extend a previous SSTA work [17] to perform timing analysis and estimate timing yield. We consider gate length variations with standard deviation σ being 5% of nominal value, and gate width variations with σ of 2.7% of nominal width. We use adaptive body bias (ABB) [1, 9] as adaptivity. The power model, including dynamic and leakage power, and impact of ABB on delay and power are based on [9]. The adaptivity area overhead includes two parts. Per-adaptive-gate area increase due to manufacturing process requirement is derived from [9]. Per-block overhead due to sensor, tuning circuits and routing of several control signal lines is estimated according to [1]. The experiments are performed on ISCAS85 and ISPD13 [21] benchmark circuits. The circuits are placed by

Table 2.4: Our method with forward body bias and reverse body bias (ABB). Gate area overhead% is denoted by ΔA_g $(unit)$.

			Baseline	Ours				
Circuit	#gates	$\lvert\mathcal{B}\rvert$	Yield	Yield	ΔW	$\Delta A/\#B$	ΔA_g	CPU (s)
c432	171	4	99.7%	99.3%	0	0/0	0%	2
c499	218	5	99.9%	99.1%	0	0/0	0%	3
c880	383	6	99.9%	99.0%	-71	-43/0	-3%	3
c1355	562	4	99.8%	99.1%	0	0/0	0%	5
c1908	972	6	99.5%	99.1%	0	0/0	0%	9
c2670	1287	5	99.9%	99.7%	-562	458/1	0%	12
c3540	1705	5	99.9%	99.5%	-1378	522/1	-2%	16
c5315	2351	6	99.9%	99.8%	0	0/0	0%	24
c6288	2416	6	99.9%	99.9%	-7791	-1595/1	-29%	25
c7552	3625	5	99.9%	99.9%	-6316	925/2	-9%	40
usb_phy	609	6	99.2%	99.4%	-628	246/1	0%	5
edit_dist	130661	29	99.3%	99.4%	-175907	56604/19	0%	2093
fft	32281	20	99.8%	99.5%	-91955	22368/9	0%	824
cordic	41601	20	99.5%	99.3%	-63189	-1050/6	-7%	1121
des_perf	112644	22	99.4%	99.3%	-79359	19979/3	0%	1805
matrix_mult	155325	20	99.0%	99.3%	-213180	-2734/7	-9%	3203
Average			99.6%	99.4%	-40021	5980/3.1	-3.7%	574
% difference vs. naïve = $\frac{(ours-naïve)}{abs(naïve)}$					37.0%	-71.3%/-70.5%		

FengShui [22] and clustered as elaborated in section 3.1.

The first experiment is to evaluate the effectiveness of our approach for forward body bias (FBB)-only and ABB, which allows both forward and reverse body bias (RBB). Relatively tight timing constraints are applied to FBB cases as FBB is mostly for timing improvement. The ABB cases have relatively loose timing constraints to see the effect of RBB on leakage power reduction. The results are shown in Table 2.1$\sim$2.4. In all tables, the number of gates and blocks of each circuit are displayed in the second and third column. The fourth column is for timing yield of the baseline, where no adaptivity is applied. Columns 5-8 provide results from the naïve method or our method. For each method, we examine the power overhead ΔW, number of adaptive block $\#B$ and total area overhead ΔA w.r.t. baseline in addition to timing yield and CPU runtime. For our method, we also show the gate area overhead ΔA_g. All area numbers are presented in the unit of ISPD13 [21] cell library. All overheads are with respect to the baseline results. We also show the average results and the percentage difference of our method versus the naïve approach.

For the cases of FBB, our methods can reduce power and area overhead by around 100% compared with the naïve approach. Due to the collaboration between gate implementation selection and adaptivity assignment, our method often reduces gate area from the baseline. Of course, both methods can largely fix the timing problem from the baseline. In the ABB cases, our method causes 71% less area overhead than the naïve method. It has 37% less power savings than the naïve method, but the power savings compared to the baseline is still significant.

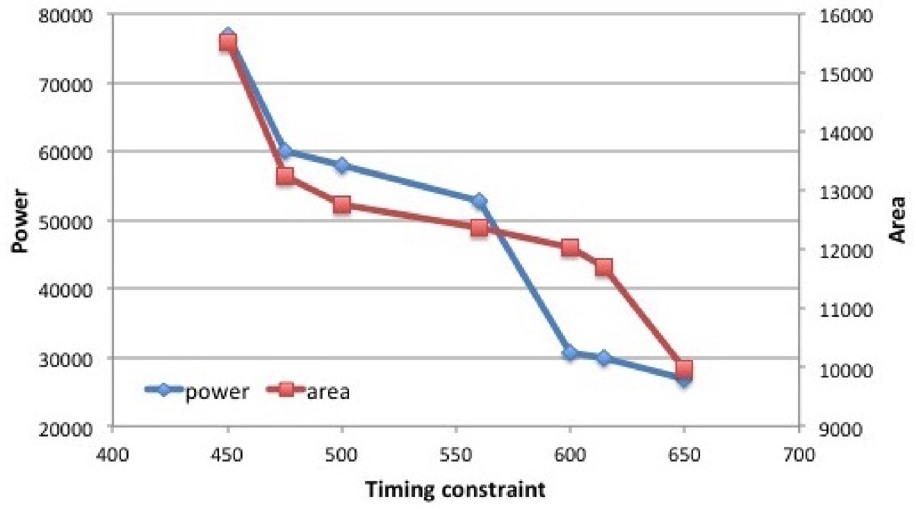


Figure 2.9: Power/area-timing tradeoff for circuit c7552.

The second experiment is to investigate the power/area versus timing tradeoff of our approach. We alter the required arrival time on the primary outputs of C7552, and observe the area and power resulted from our algorithm. The result in Figure 3.10 shows that area and power increase with increasingly tight timing constraint as expected.

In the last experiment, we vary the number of adaptivity blocks $|\mathcal{B}|$ of ISPD13 circuit *fft* and examine the effect on power and area overhead. The results from our method are plotted in Figure 2.10. When the granularity is too coarse, each block is relatively large and must involve nodes of different timing behaviors. The adaptive tuning in this case must be targeted toward the worst case gates and unnecessary power and area overhead are paid on non-critical gates. When the adaptivity is too fine-grained, the per-block overhead due to sensors, control and tuning circuits becomes very large. Therefore, there is sweet spot for adaptivity granularity.

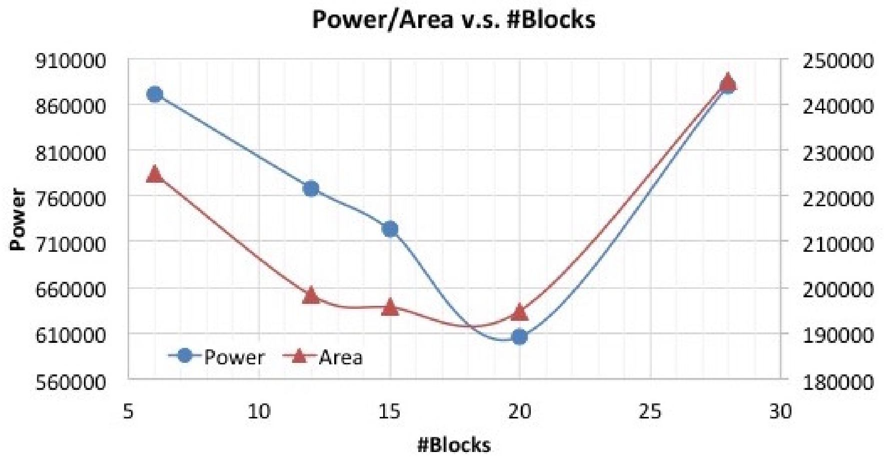


Figure 2.10: Power/area vs. granularity for circuit fft.

3. POWER EFFICIENT QUALITY-OF-SERVICE FOR APPLICATION SPECIFIC NETWORK-ON-CHIPS*

3.1 Introduction

As the degree of chip integration grows, the demand for on-chip communication bandwidth also increases. Networks-on-Chip (NoC) has received a great deal of research attention [23] as a scalable substitute to the conventional bus architecture. A critical part of NoC design is Quality of Service (QoS), which is usually categorized into guaranteed service (GS) and best-efforts service (BE). Though guaranteed service is typically more difficult to achieve, many embedded System-on-Chips are application-specific and therefore their communication patterns are by and large traceable. In such scenarios, one can characterize the communication requests and reserve NoC bandwidth for them at design-time, so as to provide guaranteed service.

NoC communication bandwidth is embodied by packet/flit routing, which identifies links that constitutes a connection, and time slot assignment at these links [23] [24]. In early works like [25] [26], packet routing and time slot assignment are performed separately. The work of [27] couples the two parts together, but applies exhaustive search on the routing part, which is too expensive, and a greedy heuristic on the time slot assignment, which provides very little assurance on solution quality.

The operations of Multi-Processor System-on-Chip (MPSoC) can often be characterized by multiple user-cases. For example, a smart phone SoC performs voice calls, messaging, and video streaming for different user-cases. Each user-case has a different traffic pattern on NoC. This application-specific nature, as opposed to the largely random traffic

*Part of this section is reprinted with permission from G. Yang, H. He, and J. Hu, "Resource allocation algorithms for guaranteed service in application-specific NoCs", in Proceedings of the IEEE International Conference on Computer Design, 2013 and H. He, G. Yang, and J. Hu, "Algorithms for power-efficient qos in application specific Network-on-Chips," in Low Power Electronics and Design, IEEE/ACM International Symposium on, 2014.

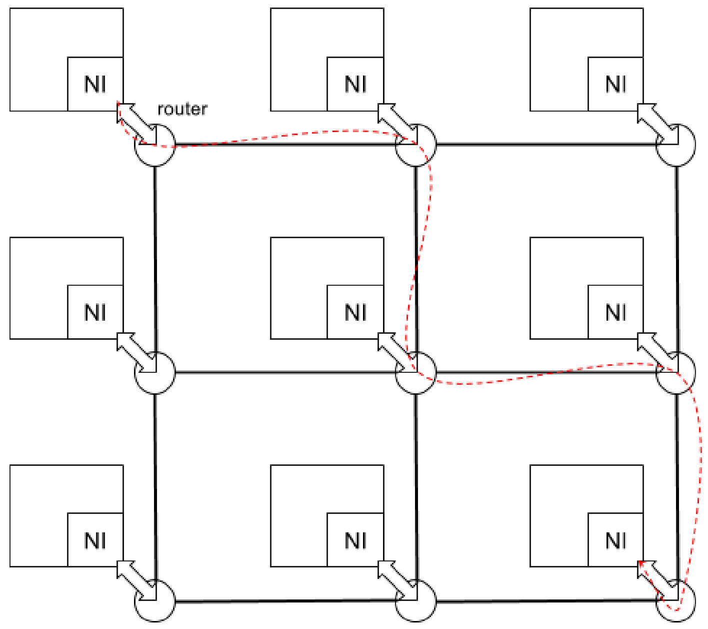


Figure 3.1: Networks on Chip.

in chip multiprocessors, allows guaranteed service to be obtained by reserving resources at design time [25–30]. The resources include both physical ones - links along a packet routing path, and temporal ones - time slots at each link, i.e., the resource allocation is based on Time Division Multiplexing (TDM). Many works [25–30] take this approach, but pay almost no attention on the power issue. The work in [31] solves task mapping and scheduling problem under fixed routing and fixed link/buffer capacity. As such, the role of NoC in [31] is more of online optimization than design space exploration.

We first show a path-based boolean Satisfiability (SAT) formulation for the time slot assignment problem alone without link/buffer capacity optimization. The path-based SAT formulation is very straightforward and light-weighted. The formulation is then fed to a highly scalable SAT solver to generate the time slots. Then, we propose a design-time algorithm of simultaneous packet routing, time slot assignment and link/buffer capacity optimization for guaranteed service. Distinguished from the previous works [25–30], we minimize the total of dynamic energy, which depends on packet routing, and static energy, which is decided by link capacity and buffer size. This method follows an Integer Linear Programming (ILP) formulation, which has substantially lower computation cost than conventional edge-based ILP formulation [29]. Our ILP technique permits multi-route and in-order delivery [28]. Since flit-level static scheduling and routing has been shown feasible in Æthereal network [28] and FPGA [29], we focus on algorithmic techniques instead of hardware implementation. For comparison, we extend two previous works with related but different goals. One is the NoC capacity optimization algorithm [32], which is an iterative greedy heuristic, and the other is bandwidth allocation algorithm for only QoS [29], which is a conventional edge-based ILP approach. Experimental results show that our technique significantly outperforms iterative greedy heuristic and is dramatically faster than edge-based ILP.

3.2 Background

3.2.1 NoC Topology

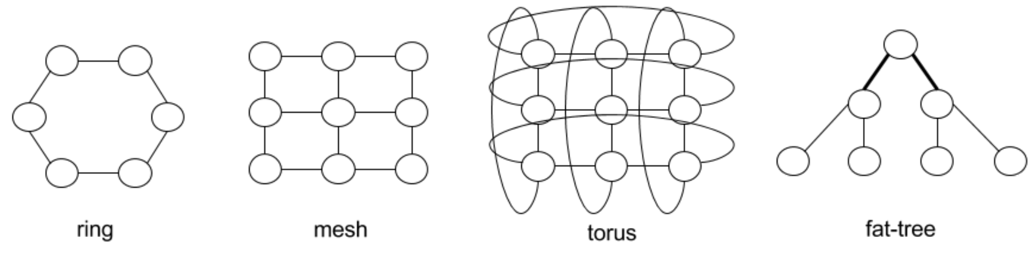
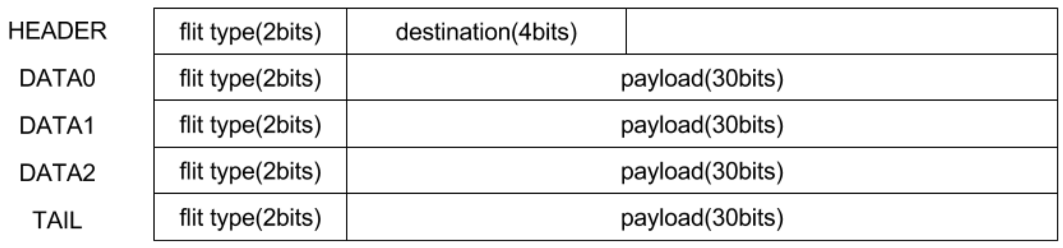


Figure 3.2: Common NoC topologies.

Figure 3.2 shows several common topologies of NoC. These topologies differ in node degree, diameter (the maximum shortest path between any two nodes), link complexity and bisection width (the bandwidth available if the network is broken into two partitions of the same size). It is also possible to design an application-specific topology for better hardware efficiency and performance.

3.2.2 Packet Format

As in the computer networks, the message in NoC is transmitted by packets. Each packet consists of one or multiple flits. A flit is the minimum unit transmitted from a router to another router. During the transmission, the flits in a packet might spread in different routers. Figure 3.3 shows an example of packet layout. The packet contains a header flit, 3 payload flits and a tail flit. Which type of flit is indicated by the first two bits of a 32-bit flit. The header flit may contain information such as the destination router.



HEADER	flit type(2bits)	destination(4bits)	
DATA0	flit type(2bits)	payload(30bits)	
DATA1	flit type(2bits)	payload(30bits)	
DATA2	flit type(2bits)	payload(30bits)	
TAIL	flit type(2bits)	payload(30bits)	

Figure 3.3: The layout of an NoC packet.

3.2.3 Router Design

Three architecture of NoC routers are shown in figure 3.4. In the event of a network congestion, flits are saved in the buffer inside the NoC routers. The buffers can either be at the input port, the output port, or shared among all the ports. In our formulation, the buffer capacity is also deemed as a decision variable for power efficiency optimization.

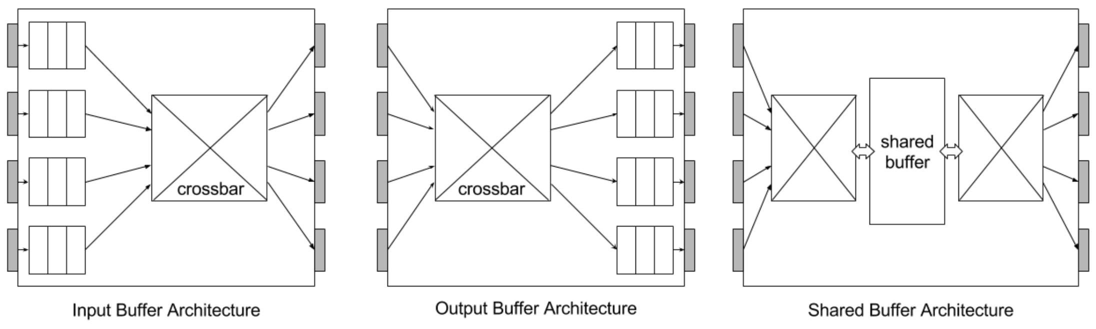

crossbar
crossbar
shared buffer
Input Buffer Architecture
Output Buffer Architecture
Shared Buffer Architecture

Figure 3.4: Router Microarchitecture.

3.3 Problem Formulation

3.3.1 Power Efficient QoS

The problem inputs include a set of user-cases on a fixed NoC topology. In each user-case, every guaranteed service (GS) packet has a specified injection time and a latency constraint. Additionally, there is the minimum bandwidth required for overall best effort (BE) packets. In a router, BE buffers and GS buffers are separated. This is because otherwise BE flits may block GS flits for an arbitrarily long time. Time on each link/buffer is divided into slots that can be assigned to different flits, i.e., it is a time division multiplexing (TDM) system.

For each GS packet/flit, the decisions are to find routing path in terms of links/buffers, and time slots along the path[†]. Additionally, a lumped sum of BE bandwidth is reserved,

[†]The flit level static scheduling and routing is demonstrated to be feasible in Æthereal network [28] and FPGA [29].

and link/buffer capacities are also decided like in [32]. The objective is to minimize the average power consumption among all user-cases.

When there is no resource contention in the network, a flit is routed along the shortest path without buffering. In the presence of congestion, in contrast, the algorithm can choose among the following three options: (1) increasing link capacity, (2) waiting in a buffer and (3) routing detour. Options (1) and (2) increase static energy while option (3) causes more dynamic energy. The trade-off among these options depends on the traffic pattern and the hardware parameters of the NoC. Hence we rely on the algorithm to determine the best configuration. The problem formulation is given as follows.

PEQoS (Power-Efficient QoS): *Given an NoC topology, a set of traffic user-cases, find routing paths and time slot assignment for each GS flit, and decide link and GS buffer capacity such that the weighted average of power consumption among all user-cases is minimized, every GS packet satisfies its latency constraint and sufficient link bandwidth remains for BE packets.*

The weighing factors for user-cases are design parameters. For example, they can be the estimated probabilities of individual user-cases. Customers can also decide how much link bandwidth should be kept for BE packets, which are routed in a distributed manner. In contrast, GS packets employ source routing. Since the TDM switchings are determined at design time, deadlock can be easily avoided for GS packets like in [28, 29]. As GS buffers are separated from BE buffers, there will be no deadlock between GS and BE packets. Our techniques can be applied to various NoC topologies, from regular mesh to customized topology. We also allow flits of the same packet to take different routes as long as the in-order delivery constraint is satisfied.

3.3.2 Graph Model

The physical and temporal resources of an NoC can be described in a unified graph model, where solving PEQoS is equivalent to finding a path for every flit. We first define physical graph $G(V, E_L, E_B)$, which does not contain temporal information. It has a set of nodes V, each of which represents a router, and two sets of edges, E_L for links and E_B indicating buffers. An example is shown in Figure 3.5(a), where dashed edges are for E_B.

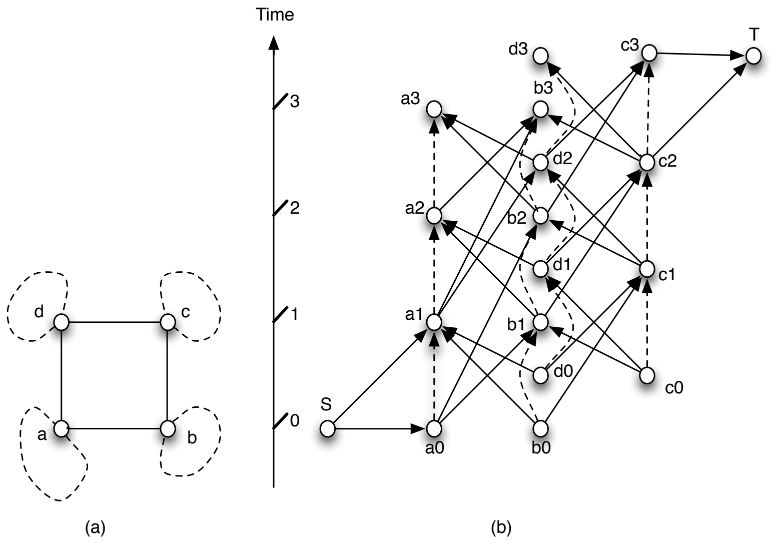


Figure 3.5: (a) Physical graph; (b) Resource graph.

In a resource graph, temporal resource is embraced by duplicating physical nodes along the time axis and connecting nodes at different time planes with edges. This is illustrated in Figure 3.5(b), where two adjacent time planes are separated by one clock cycle. If the latency of a link is m clock cycles, the corresponding edge spans $m + 1$ time planes. To limit the graph size, we assume that the traffic pattern repeat itself in the periodic windows. This assumption is reasonable if the window size is sufficiently large [28, 30].

In a resource graph $\mathcal{G}(\mathcal{V}, \mathcal{E}_L, \mathcal{E}_B)$, a node $v \in V$ is duplicated into $v^0, v^1, ..., v^\Psi \in \mathcal{V}$ at time plane $0, 1, ..., \Psi$, where Ψ is the window size. Any time τ corresponds to $k\Psi + \tau$, where k is an integer. An edge elapsing through two adjacent windows wraps around in the graph [28, 30]. For example, an edge starting from $v_i^\Psi \in \mathcal{V}$ may end at $v_j^1 \in \mathcal{V}$. The time range for a packet ready to inject can be modeled by a super source node like S in Figure 3.5(b). The deadline for a flit can be enforced by super target node. In Figure 3.5(b), the super target node T requires that the corresponding flit must arrive physical node c by time 3. Edges $\mathcal{E}_L$ and $\mathcal{E}_B$ correspond to links and buffers, respectively.

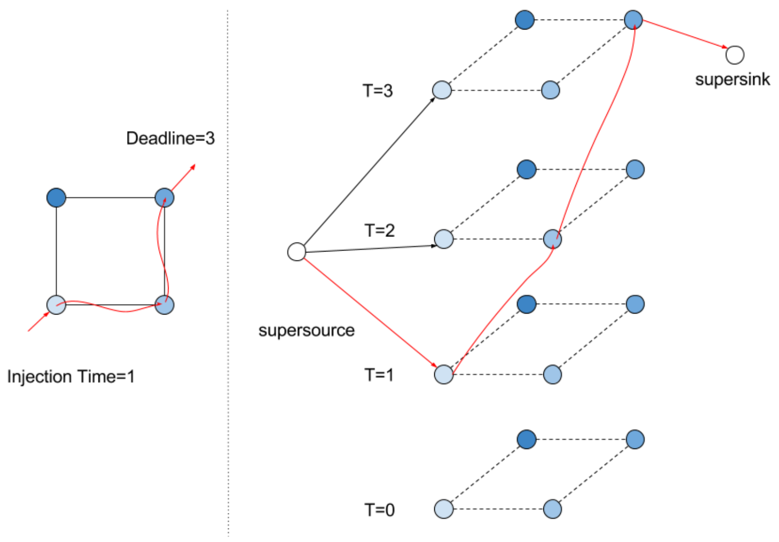


Figure 3.6: A valid flit route that respects the injection time and latency constraint

Figure 3.6 shows the correspondence between injection time and latency constraint. A flit is available at time $T = 1$ and must be delivered before time $T = 3$. The supersource that represents this flit is connected to all the time instances of the injection node after $T = 1$. Similarly, the supersink admits routes from all the time instances of the destination node after $T = 3$. As such, any path from the supersource to the supersink is a valid time slot allocation for the flit of interest.

3.4 Boolean-Satisfiability-based Method

3.4.1 Boolean Satisfiability Problem

Given a Boolean function $f(x_0, x_1, \cdots, x_n)$, Boolean Satisfiability (SAT) is the problem of finding a binary value for every variable $x_0, x_1, \cdots, x_n$ such that f is evaluated to be $true$. In spite that SAT is a well-known NP-complete problem, there are solvers that can efficiently solve problems of moderate size.

For a SAT solver, it is usually required that the Boolean function is in the Conjunctive Normal Form (CNF). CNF represents a boolean function as a conjunction (AND) of clauses, and each clause is a disjunction (OR) of variables. Mathematically, if we denote AND as multiplication and OR as addition,

$$f(x_0, x_1, \cdots, x_n) = \prod_k \sum_{i \in S_k} x_i \tag{3.1}$$

where S_k are some subsets of $0, 1, \cdots, n$

Because SAT is a satisfiability problem instead of an optimization problem, in this section we explore a restricted version of PEQoS problem described in section 3.3. Here we assume link/buffer capacity is fixed and only one user-case exists. For each user-case, our algorithm finds a valid time slot allocation that satisfies the latency constraint.

3.4.2 Candidate Path Generation

There are two significantly different approaches for the SAT formulation of the problem: *Edge-Based* where a decision variable $x_{e,i} \in \{0, 1\}$ indicates if an edge $e \in \mathcal{E}$ is utilized by a data flit ϕ_i; and *Path-Based* where a decision variable $x_{p,i} \in \{0, 1\}$ tells if a path p in the resource graph is selected to be used by flit ϕ_i.

The edge-based formulation is computationally expensive in that it multiplies the number of decision variables by the number of flits, while the path-based formulation requires a premise that a set of candidate paths P_i are pre-selected. We propose a simple yet effective algorithm that generates candidate paths and its pseudo code is shown in Figure 3.7. For each flit ϕ_i the algorithm iteratively generates a shortest path on $\mathcal{G}$ using the Dijkstra's algorithm (step 5), and adds this path into the candidate set (step 6). The key part is that an edge weight is increased by a small amount if the path passes through this edge (step 7-8). Consequently, the path search in later iterations attempts to avoid using previously used edges, and therefore the generated paths are diversified.

```
Procedure: $CandidatePathGeneration(\mathcal{G}, \Phi, k, \delta)$
Input: Resource graph $\mathcal{G}$
       A set of flits $\Phi = \{\phi_1, \phi_2, \cdots\}$
Output: A set of paths $P_i$ for each $\phi_i \in \Phi$
1. For each flit $\phi_i \in \Phi$
2.   $P_i \leftarrow \emptyset$
3.   Initialize weight for each edge $e_j \in \mathcal{E}$
4.   Repeat for up to $k$ iterations
5.     Find a shortest path $p$ on $\mathcal{G}$ for $\phi_i$
6.     $P_i \leftarrow P_i \cup \{p\}$
7.     For each edge $e \in p$
8.       Increase weight of $e$ by $\delta$
```

Figure 3.7: Algorithm of candidate paths generation.

Note the candidate path selection is performed on the resource graph rather than the physical graph. Each path corresponds a time slot allocation for transmitting a flit from source to sink.

3.4.3 SAT Formulation

Once a diversified set of candidate paths are generated, each path is associated with a decision boolean variable x_p. $x_p = 1$ means the candidate path is selected for flit transportation. To ensure a valid time slot allocation, we enforce two kinds of constraints in our formulation:

(a) the edge capacity constraints

For each edge $e \in \mathcal{E}$, we have the following CNF to enforce its capacity constraint:

$$\prod_{p_i \neq p_j, e \in p_i, e \in p_j} (\bar{x}_{p_i} + \bar{x}_{p_j}) \tag{3.2}$$

where p_i and p_j are two distinctive paths passing through e. Each disjunction clause $(\bar{x}_{p_i} + \bar{x}_{p_j})$ implies that at most one path is selected between p_i and p_j. The conjunction of all such clauses ensures that this condition is true for all pairs of paths that going through edge e.

(b) the demand constraints

The idea of formulating the demand constraint is based on similar idea. For each flit ϕ_i, we enforce

$$\prod_{p_j, p_k \in P_i,\ p_j \neq p_k} (\bar{x}_{p_j} + \bar{x}_{p_k}) \cdot \sum_{p_j \in P_i} x_{p_j} \tag{3.3}$$

where P_i is the set of candidate paths for flit ϕ_i. The disjunction clause $\sum_{p_j \in P_i} x_{p_j}$ requires that at least 1 candidate path is selected. The remaining part of expression (3.3) enforces that at most 1 path is selected.

3.4.4 In-order Flits Delivery

When multiple flits come from the same packet, it is sometimes required in NoC that the order in which the flits are received is the same as the order in which they are sent. In our formulation discussed above, the path length is not considered when the SAT solver searches for a solution. Therefore, we need additional constraint to guarantee in-order delivery.

Consider a set of flits $\{\phi_1, \phi_2, \cdots\}$ that belong to the same packet, and their indices indicate the injection order. Each flit ϕ_i is associated with a super target T_i. Each super target T_i has multiple incident edges $e_{T_i}^{\tau}$, $e_{T_i}^{\tau+1}$, ..., where τ represents a time instant. Now we introduce an indicator variable $y_{T_i}^{\tau} \in \{0, 1\}$ to tell if edge $e_{T_i}^{\tau}$ is utilized. It is defined by

$$y_{T_i}^{\tau} = \sum_{p \in P_i,\ e_{T_i}^{\tau} \in p} x_p \tag{3.4}$$

If flit ϕ_i arrives at time τ, which means $y_{T_i}^{\tau} = 1$, we require that flit ϕ_{i+1} must arrive at $\tau + 1$, i.e., $y_{T_{i+1}}^{\tau+1} = 1$. For each pair of consecutive flits ϕ_i and ϕ_{i+1}, we add the following clause

$$\overline{y_{T_i}^{\tau}} + y_{T_{i+1}}^{\tau+1} \tag{3.5}$$

3.4.5 Experiment Result

Table 3.1: Experimental results on 144 cases with timeout limit as 4 hours. The runtime T is for only the successful runs.

		Previous work [27]			SAT-based (Section 3.4)		
Cases	# packets	Success	T	Timeout	Success	T	Timeout
Mesh 6×6	25 - 90	20.0%	11s	64.0%	76.0%	2s	24.0%
Mesh 8×8	30 - 160	23.1%	1s	73.1%	69.2%	6s	30.8%
Mesh 10×10	35 - 250	18.2%	4s	72.7%	59.1%	28s	40.9%
Random 36	25 - 110	20.8%	1s	79.2%	75.0%	3s	25.0%
Random 64	30 - 270	17.4%	1s	82.6%	82.6%	25s	17.4%
Random 100	35 - 450	16.7%	1s	83.3%	83.3%	101s	16.7%
Average		19.3%	3s	75.8%	74.2%	28s	25.8%

The SAT problems are solved by *zChaff* (http://www.princeton.edu/˜chaff/zchaff.html). We also implemented the method of [27] for comparison. The leftmost column of Table 3.1 tells the types of testcases. For example, *Mesh* 6×6 is a 36 node mesh and *Random 64* is a 64 node randomly generated topolgy. For each type of topology, we tested over 20 cases with different amount of requests, and from uniform to bursty injection patterns. The ranges of the number of packets for each time period are listed in column 2 of Table 3.1. In each case, about 85% packets are single-flit and 15% packets are multi-flits.

The main results for the 144 cases are shown in Table 3.1. Column 3 and 6 display the success rate for [27] and our SAT-based methods, respectively. The success rate is the percentage of testcases whose all flits are successfully routed. Our methods can averagely improve the success rate to 67% and 74% respectively, compared to the 19% from [27]. The average runtime of successful runs are displayed in column 4 and 7. We set a timeout limit of 4 hours, beyond which a run is counted as a failure. Column 5 and 8 indicate the percentage of cases where the timeout occurs. Although the method of [27] is fast on the successful runs, it runs longer than 4 hours in about 76% of the cases.

Stress tests are further performed for our SAT-based method and the previous work [27]. In a stress test, the number of packets $|\mathcal{R}|$ is increased to the point a method fails to find any feasible solution. The stress test results are displayed in Figure 3.8. The first 5 cases are 6×6 meshes and the other 5 cases are in random topologies of 30 physical nodes. The bar graph shows that the SAT-based methods can find solutions for twice as many cases as [27].

3.5 Integer Linear-Programming-based Method

In this section, we discuss the details of solving PEQoS problem with Integer Linear Programming. The proposed method optimizes the power consumption of NoC communication and guarantees the QoS of GS traffic at the same time. The design time algorithm

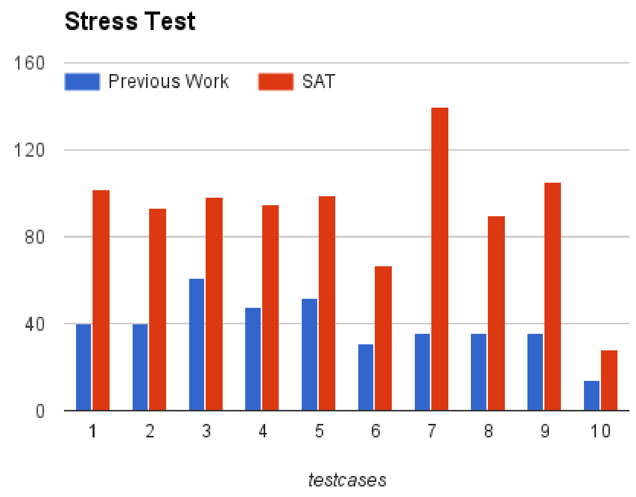


Figure 3.8: The maximal number of packets can be routed. Cases 1-5: 6×6 mesh; cases 6-10: random topology.

not only decides the exact time slot reservation for GS traffic with known pattern, but also provides guidance to the right amount of link capacity and buffer size in the network. It also take into consideration multiple user-cases of the NoC.

3.5.1 Problem Formulation

Integer Linear Programming has been applied to solve NoC QoS before [29]. In their approaches, the binary decision variables are defined to indicate if *edges* in the resource graph are selected to transport flits. Such formulation guarantees that optimal solution is in the search space. However, such approach does not scale well with problem size and can be applied at only very small cases in practice.

$$\begin{aligned} \sum_{e_f \in incident(v)} x_{e_f} &= 0, \forall f \in F, v \in V \textit{(flow conservation)} \\ \sum_{f \in F} x_{e_f} &\leq C_e, \forall e \in E \textit{(capacity)} \end{aligned} \quad (3.6)$$

Equation 3.6 shows the constraints of edge-based formulation. x_{e_f} denotes the amount of flow (0-1 variable in our application) on edge $e \in E$ and flit $f \in F$. Flow conservation constraint asserts that the number of flits entering an internal node v is equal to the number of flits exiting v. Capacity constraints enforces the bandwidth limit of a physical link. As it can be seen from equation 3.6, the number of decision variables is $|F| \times |E|$. Note $|E|$ is the number of edges in the resource graph, which is multiple times of the number of physical links because of time domain duplication. Even though edge-based formulation captures all possible routes of transmitting a set of flits F, the problem size is very large for big graph.

We propose a path-based ILP formulation, where a decision variable tells if to select a *path* in the resource graph for a flit. This approach requires that a set of candidate paths are generated for each flit in advance. Since it is not practical to include all possible paths in this set, there is no guarantee for optimality. However, by carefully generating the candidates, one can attain near optimal solutions with much better scalability than the edge-based ILP method.

We describe the path-based ILP formulation followed by an introduction to the candidate paths generation. An MPSoC design has a set of $\mathcal{M}$ user-cases. In each user-case $\mu \in \mathcal{M}$, there is a set of packet requests per time window and each packet is composed by one or multiple flits. To simplify the description without loss of generality, we specify the requests in flits, i.e., a set of flit requests Φ_μ for each user-case μ per time window. A set of candidate paths $P_i = \{p_{i,1}, p_{i,2}, ...\}$ on $\mathcal{G}$ are found for flit ϕ_i, and each of the paths has a variable $x_{i,j} \in \{0, 1\}$ indicating if path $p_{i,j}$ is selected. Each link $l \in E_L$ has a variable capacity y_l telling how many flits it can simultaneously transport. Similarly, each buffer $b \in E_B$ has capacity z_b, which is the number of flits it can accommodate. Dynamic energy for a time window in user-case μ is represented by Δ_μ. The static energies per time window for unit link and buffer capacity are denoted by ϵ_l and ϵ_b, respectively. We use

$\phi_i \prec \phi_j$ to indicate that ϕ_i is injected into the network earlier than ϕ_j and they belong to the same packet. The ILP formulation is as follows.

$$\text{Min} \quad \textstyle\sum_{\mu \in \mathcal{M}} \omega_\mu \Delta_\mu + \sum_{l \in E_L} \epsilon_l y_l + \sum_{b \in E_B} \epsilon_b z_b \tag{3.7}$$

$$\text{s.t.} \quad \Delta_\mu = \textstyle\sum_{\phi_i \in \Phi_\mu} \sum_{p_{i,j} \in P_i} \delta_{i,j} x_{i,j} \tag{3.8}$$

$$\textstyle\sum_{p_{i,j} \in P_i} x_{i,j} = 1, \;\; \forall \phi_i \in \Phi_\mu, \;\; \forall \mu \in \mathcal{M} \tag{3.9}$$

$$\textstyle\sum_{e_l \in p_{i,j}} x_{i,j} \leq y_l \leq U_l, \;\; \forall e_l \in \mathcal{E}_L, \;\; \forall \mu \in \mathcal{M} \tag{3.10}$$

$$\textstyle\sum_{e_b \in p_{i,j}} x_{i,j} \leq z_b \leq U_b, \;\; \forall e_b \in \mathcal{E}_B, \;\; \forall \mu \in \mathcal{M} \tag{3.11}$$

$$\textstyle\sum_{p_{i,j} \in P_i} |p_{i,j}| x_{i,j} \leq \sum_{p_{k,q} \in P_k} |p_{k,q}| x_{k,q}, \forall \phi_i \prec \phi_k \tag{3.12}$$

$$\Psi y_l - \textstyle\sum_{e_l \in \mathcal{E}_l, \in p_{i,j}} x_{i,j} \geq \beta_l, \; \forall l \in E_L \tag{3.13}$$

$$x_{i,j} \in \{0, 1\}, \;\; \forall x_{i,j} \tag{3.14}$$

$$y_l, z_b \in Z^*, \;\; \forall y_l, z_b \tag{3.15}$$

The objective (3.7) is to minimize total energy consumption per time window. The first term is in (3.7) is a weighted average of dynamic energy among all user-cases. The weighting factors ω_μ are user specified parameters, and can be obtained by system level characterization. The dynamic energy of each user-case is defined by constraint (3.8). The dynamic energy of propagating flit ϕ_i through path $p_{i,j}$ is represented by $\delta_{i,j}$. The second (third) term in (3.7) is for the static energy of all links (buffers). Constraint (3.9) enforces that one and only one path is selected for each flit. Constraints (3.10) and (3.11) ensure that each link/buffer capacity does not exceed certain bound U_l/U_b. In (3.12), $|p_{i,j}|$ means the path length in term of clock cycles for path $p_{i,j}$. Then, this is the constraint for in-order delivery. The last significant constraint (3.13) is to make sure at least β_l bandwidth is left for BE flits at link l per time window.

Now we discuss the candidate paths generation. The candidate paths should all satisfy latency constraints and consist of short paths so as to increase the chance of low dynamic energy solutions. To this end, we first generate all the paths that has minimum number of hops, which can be found by Breadth First Search. In addition, the candidate paths for a flit need to be diversified so that contention with other flits can be easily avoided. Thus, we also generate candidate paths by the same method as [30]. For each flit, our method iteratively performs the shortest path algorithm on the resource graph and the result is added to the candidate path set. At the end of each iteration, the cost of each edge along this path is increased by a fixed amount so that later iterations attempt to circumvent these edges and thereby improve path diversity.

After the candidate path generation, the ILP formulation is fed to a ILP solver which tries to optimize it and may use different algorithms such as Branch and Bound and Cutting Plane Method.

3.5.2 Experimental Results

In the experiment, we attempt to evaluate the effectiveness of our techniques by comparing with extensions of two related but different works. One is the *iterative greedy* method for NoC capacity optimization [32] and the other is the conventional *edge-based ILP*, which is recently used [29] in NoC QoS without considering power dissipation.

The comparisons are conducted on two types of testcases. One is random benchmarks generated by TGFF [33], which has been employed in many other NoC works such as [31, 34]. The other is a more realistic benchmark developed for multimedia SoC in the NaNoC project (http://www.nanoc-project.eu). The energy dissipation, including both dynamic and static energy, is estimated by ORION3.0 [35] based on 65nm technology. All methods are implemented and simulated with C/C++ and ILP is solved by LPSolve (http://lpsolve.sourceforge.net/). The experiment is performed on AMD Opteron processor with 2.2GHz frequency and Linux operating system.

Table 3.2: Main results for TGFF cases.

				Iterative Greedy		Path-Based ILP	
Testcases	cases	Total $\|\mathcal{V}\|$	$\|\mathcal{M}\|$	Energy	Runtime	Enery	Runtime
Group 1	9	1250-1500	1,5	459	161s	324	288s
Group 2	14	2000-2250	3,10	299	224s	222	1022s
Group 3	12	2400-2500	3,5	375	320s	273	2047s
Group 4	14	3000-3200	5,8,15	235	233s	177	1642s
Group 5	10	4800	10, 15	146	188s	116	2961s
Group 6	10	5000	8, 10	220	192s	151	5239s
Group 7	5	6400	8	266	529s	208	3817s
Group 8	9	7200-7500	12, 15	178	329s	141	5439s
Normalized sum	83			1	1	0.75	10.3

The first experiment is on 83 TGFF cases and the results are summarized in Table 3.2. To save space, the results are presented in 8 groups and the second column tells the number of cases in each group. The third column lists the total number of nodes $|\mathcal{V}|$ for the resource graphs of all user-cases in one testcase. Please note $|\mathcal{V}| = |V| \cdot \Psi \cdot |\mathcal{M}|$, where $|V|$ is the number of physical nodes and takes value of 10, 16 or 25 for each case. The number of user-cases $|\mathcal{M}|$ is in the fourth column. Please note that the runtime also depends on the number of flits in addition to $|\mathcal{V}|$. Each packet contains 1-3 flits and the latency across each link can be 1 or 2 clock cycles. The energy and computation runtime results are shown in the right six columns. As the energy is a weighted average among all user-cases, it does not necessarily grow with the total $|\mathcal{V}|$. The last line indicates that our path-based ILP can reduce energy dissipation by 25% compared to extension of [32]. Among these cases, $10\% - 60\%$ bandwidth is allocated to GS packets and the others are to be used by BE packets. Please note that the greedy heuristic terminates when it gets stuck at a local optima, hence further iterations would not improve its results.

In the second experiment, we compare our path-based ILP with optimal edge-based ILP solutions. Since the edge-based ILP is very slow, this part of experiment is carried out

Table 3.3: Optimality test on small TGFF cases.

	Total	Edge-ILP (Optimal)		Path-Based ILP	
	$\|\mathcal{V}\|$	Energy	Runtime	Energy	Runtime
Case 1	216	3.18	466	3.23	<1
Case 2	220	1.81	249	1.81	<1
Case 3	240	1.59	18	1.59	<1
Case 4	250	2.67	3009	2.71	<1
Case 5	288	2.80	279	2.84	<1
Case 6	288	3.13	1179	3.17	<1
Case 7	300	2.29	5769	2.34	<1
Case 8	312	2.93	6934	2.93	<1
Case 9	405	2.14	2807	2.14	<1
Normalized Ave.		1	2301	1.01	1

only on small cases. The results are displayed in Table 3.3. One can see that our technique is only 1% worse than the optimal but over 2000X faster than the edge-based ILP. In a small example, our path-based ILP entails about 1K variables while the edge-based ILP requires 54K variables. This explains why the path-based ILP is much faster.

The multimedia SoC case (http://www.nanoc-project.eu) has 25 cores and 8 user-cases. The NoC topology of 10 router nodes and traffic patterns are given. We vary the latency constraints and window size to obtain 9 variants of this case. The energy comparison among different techniques for these cases are depicted in Figure 3.9. Compared to the iterative greedy heuristic, our path-based ILP algorithm achieves energy reduction of 15%. The runtime of our path-based ILP is actually less than the greedy heuristic in these cases.

In the last part of the experiment, we examine the energy-latency tradeoff that can be obtained by our technique. For three TGFF cases, we vary the latency constraints and observe the impact on energy consumption. Figure 3.10 exhibits the results from our path-based ILP. Its horizontal axis indicates normalized packet latency and the vertical axis is normalized energy dissipation. The curves show a few small kinks because the problem is non-convex and our technique cannot guarantee the optimality. The overall trend of these

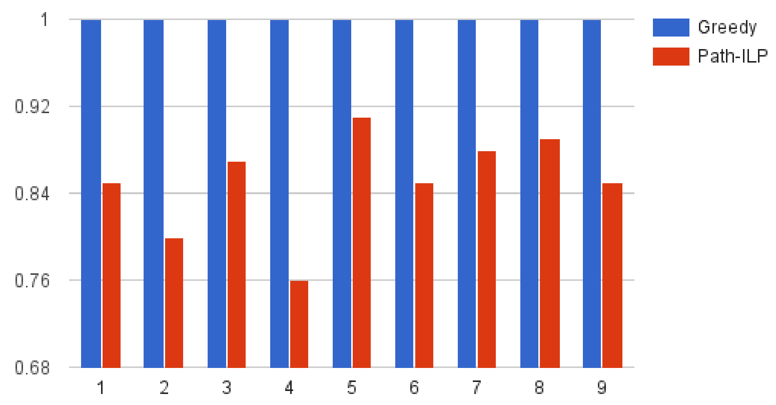


Figure 3.9: Normalized energy comparison for multimedia SoC cases.

curves does provide tradeoff between energy and latency.

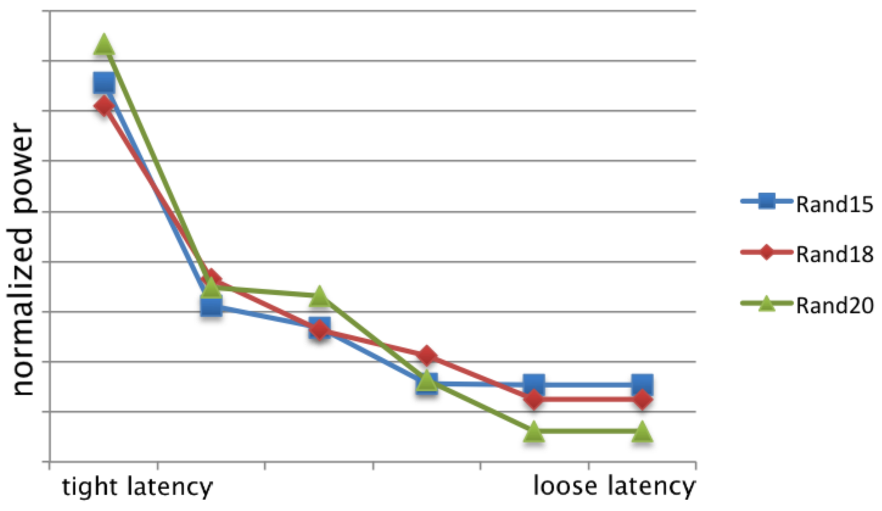


Figure 3.10: Energy-latency tradeoff of 3 different cases.

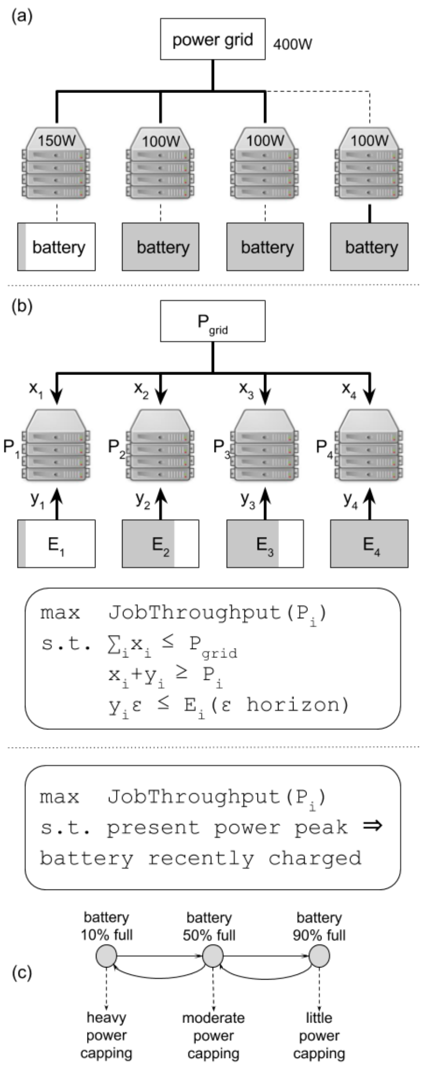
4. MODEL CHECKING BASED RESOURCE MANAGEMENT IN DATA CENTER

4.1 Introduction

As the size of datacenters reaches $30K$ servers/$10MW$ scale and beyond, resource management becomes an increasingly complex task. A proper resource manager, for example, provisions and de-provisions the computing hardware to match the workload changes [36], adjusts the effort of cooling equipments according to the dynamic thermal environment [37], and migrates computation in the event of a hardware failure. Moreover, the intricate interplay among these design goals often entails a joint optimization instead of a divide-and-conquer approach [38]. In such complex systems, to design and validate a resource manager that complies with all the goals, which often conflict with each other, is tricky yet important.

Take power management as an example. In contrast to a large, centralized Uninterruptible Power Supply (UPS) system, modern datacenters adopt a number of small UPS in the rack or server level [39] to push Power Usage Effectiveness (PUE) close to 1. This distributed UPS system is more scalable and efficient in terms of the power delivery loss, but increases the complication for power capping to prevent overdrawing batteries, which escalates the risk of server failure and shortens battery life. In a distributed UPS system, a single depleted battery does not entail the power demand to be capped even if the datacenter is experiencing a power peak (figure 4.1.a). The restriction to battery use conflicts with the goal of maintaining high system performance. It is possible to formulate the datacenter power distribution problem with static linear constraints (figure 4.1.b), but the formulation is only valid for a small horizon as the server power consumption and battery energy level are highly dynamic in nature.

To address scenarios where static constraints are not sufficient, we propose to enhance



```
max  JobThroughput(P_i)
s.t. ∑_i x_i ≤ P_grid
     x_i+y_i ≥ P_i
     y_i ε ≤ E_i (ε horizon)
```

```
max  JobThroughput(P_i)
s.t. present power peak ⇒
battery recently charged
```

Figure 4.1: (a) Single battery is depleted yet no need to cap power demand. (b) A static formulation: x_i and y_i are power provided by the power grid and battery, respectively. P_i are the server power consumption. E_i are the energy left in batteries. ϵ is a short time horizon starting from the current moment. (c) A Linear Temporal Logic formulation: the LTL constraint is transformed to a state machine that runs dynamically alongside the system.

datacenter resource management with Linear Temporal Logic (LTL) constraint. LTL is a form of property description used in model checking, which is a formal verification technique [40]. For the example in figure 4.1.c, LTL can describe the constraint that a battery can be discharged to supply a rack only when it has been recently charged for sufficiently long time. The LTL constraint is then transformed to a state machine that runs dynamically alongside the datacenter's resource management policy to monitor the battery conditions. Our approach is based on the recent theory of Markov Decision Process (MDP) control synthesis with LTL constrains [41]. It bears several advantages:

1) Our method provides a framework for considering subtle or complex constraints in datacenter resource management. It helps to resolve conflicting objectives under complicated system interactions.

2) Our method allows the consideration of property checking constraints to be directly augmented with almost any conventional management scheme, rather than radically altering conventional methods.

3) Although our method is based on the control synthesis theory [41], we make a significant improvement to avoid the state explosion problem in [41]. In [41], the learning and decision-making operate on the same state space, which is the cross product of the underlying MDP states and the automaton states arising from LTL constraints. In contrast, the learning in our method is solely upon the automaton state space. Our decision-making operates on the MDP states and the automaton states in tandem. As such, the state explosion problem is largely avoided.

We apply this methodology in two resource management scenarios: power capping in a distributed UPS datacenter and fair scheduling in a multi-tenant cloud service. In the first scenario, our method caps the power demand to improve the availability of UPS batteries should a power outage occur. The scale of the system under test is so large that the previous model checking method [42] is no longer applicable. The experimental results show our approach reduces the battery unavailability time by half and maintains the power efficiency level. In the second scenario, we implemented a MapReduce job scheduler that balances throughput and fairness among users. The emulation on Amazon AWS shows the degree of fairness can be tuned by the LTL formulation in contrast to several Hadoop schedulers. And a better tuning range is achieved with LTL over a previous work.

4.2 Related Work

The application of model checking in resource management problems has been explored in a number of previous literature. Johnson *et al.* [43] proposed an offline verification framework. In this framework, design changes can be made incrementally if the desired properties are not satisfied. Bersani *et al.* [44] showed the feasibility of formalizing elasticity, Quality-of-Service (QoS) and several correctness properties for resource management in Timed Constraint Linear Temporal Logic, an extension of LTL. However, their method is also offline and works on the system trace.

Techniques that adopt model checking online have also been proposed. Calinescu *et al.* [45] designed an autonomic manager to ensure QoS in a service-based system. The autonomic manager uses PRISM [46] model checker tool to experiment on a Markov Chain system model and a set of QoS requirements encoded in Probabilistic Computation Tree Logic. An optimization is then conducted to choose the best control parameters from those tested during the experiments. The procedure described in [45] is more of a trial-and-error approach than one with theoretical guarantees like our method. Gounaris *et al.* [42] applied the model checking policy synthesis technique [40] to the resource provisioning problem in the cloud. The synthesis is performed also by PRISM, which only supports "reachability" constraints rather than the full power of LTL. In their work, the system is explicitly modeled as a MDP and the resource manager is the policy synthesized. However, many realistic design goals are hard to optimize by a pure MDP controller because of the state space explosion problem, especially when the states are continuous. In contrast, our method allows the engineers to develop and curtail their resource manager for the specific goals of their application. Our method focuses only on the correctness properties that can be added on top of the existing resource manager.

In addition, practically all the previous work require precise modeling of the system. This is difficult for highly dynamic systems such as datacenters. Reinforcement learning (section 4.3.4) is well-known for the ability to work without models. Theoretical study on adopting reinforcement learning in online model checking problem has been documented [41]. However, we did not find such combination of reinforcement learning and model checking in the resource management literature.

4.3 Background

This section discusses the Linear Temporal Logic (section 4.3.1) and the LTL constrained Markov Decision Process control policy synthesis (section 4.3.3), which is the foundation of our approach. This theory supports the integration of any LTL constraints into a system MDP model. It is much more powerful than the "reachability" constraints that the previous work [42] [45] is based on. The synthesis method is performed offline and requires precise modeling of the system. More specifically, it requires explicit values for a transition probability matrix P, which is hard to estimate in large systems. This requirement can be avoided by reinforcement learning (section 4.3.4).

4.3.1 Linear Temporal Logic

An LTL expression describes the temporal properties of a discrete time system. LTL expressions consist of a set of boolean variables APs (termed as atomic propositions in model checking) and operators that assemble the APs to form complex semantics. APs are usually observable conditions of interest drawn from the system, for instance a = "power demand$<$power budget". A list of operators is given in table 4.1. A system is said to satisfy LTL expression ϕ if ϕ is true since time 0. For example, $GF_m a$ indicates that power demand$<$power budget cannot be false for more than m ticks since the system starts.

operator	semantic	example
$\&, \|\|, !$	and,or,not	!a is true at time t iff a is false at t
X	next	Xa is true at time t iff a is true at t+1
G	always	Ga is true iff a is true for $[0, \infty)$
F	eventually	Fa is true at time t iff a is true for some $h > t$
U	until	aUb is true at time t iff a remains true for $[t, h)$ and b is true at time h
U_m	m-until	$aU_mb = a\&(Xa)\&(XXa)...(\underbrace{XXX}_{m}b)$ *
F_m	m-eventually	$F_ma = a\|\|(Xa)\|\|(XXa)...(\underbrace{XXX}_{m}a)$ *

* X is the "next" defined in the third row.

Table 4.1: LTL syntax.

4.3.2 Deterministic Rabin Automaton

To formally verify an LTL expression ϕ, ϕ is transformed to a Deterministic Rabin Automaton (DRA) [47].

A DRA is a tuple $\Gamma_\phi = (T, AP, \delta, Rej)$ where T is the state space, $\delta : T \times AP \to T$ is the deterministic state transition function, $Rej = \{(Y_i \subseteq T) | 0 < i \leq k\}$ are sets of rejection states, which will be elaborated later.

The state space T of DRA is automatically generated. DRA performs state transitions based on the observed APs and δ. It can be shown that the LTL expression ϕ is true if and only if $\exists (Y_i) \in Rej$, Y_i is visited finitely often [40] (actually there are also accepting sets, but omitting them do not affect our results). Without loss of generality, we assume $k = 1$ hereafter. Two features of DRA make it ideal for verification purpose. Firstly, the state transition is deterministic. More importantly, with a DRA running alongside the original system it is possible to verify the LTL properties on the fly.

4.3.3 LTL Constrained MDP

Previous work [41] [48] shows that if we can model a system by Markov Decision Process, it is possible to embed a Deterministic Rabin Automaton (equivalent to an LTL constraint) into the MDP. Then a control policy that observes the LTL constraint can be synthesized offline. Here we briefly introduce the basic concepts and main results in [41] [48].

A labeled MDP is a tuple $\boldsymbol{M} = (S, A, P_M, \mathcal{L})$ where S is the state space, A is a set of actions, $P_M : S \times A \times S \to [0,1]$ is the probability transition matrix, and $\mathcal{L} : S \times A \to 2^{AP}$ labels the states by the APs that are true. A control policy $\pi : S \to A$ is a look-up table that holds the right action to take in each state.

Here S encodes the status of the system such as "server utilization" and should not be confused with the state space T of DRA. A encodes the decisions made in resource management like turning off a server or scheduling a task. P gives a probability to every state transition $s \in S \xrightarrow{a \in A} s' \in S$. $\mathcal{L}$ indicates whether a condition (AP) is true given the system status S.

To synthesize a control policy that conforms to an LTL expression ϕ, the system MDP model $\boldsymbol{M}$ and DRA Γ_ϕ are first combined together as a product MDP. The product MDP between a labeled MDP $\boldsymbol{M} = (S, A, P_M, \mathcal{L})$ and a DRA $\Gamma_\phi = (T, AP, \delta, Rej)$ is defined as a tuple $\boldsymbol{M}_\phi = (S \times T, A, P, W)$ where

$$P(s \times t, a, s' \times t') = \begin{cases} P_M(s, a, s') & \mathit{if}\, t' = \delta(t, \mathcal{L}(s)) \\ 0 & otherwise \end{cases}$$

$W : S \times T \to \mathbb{R}$ is a penalty function defined by

$$W(s,t) = \begin{cases} Constant > 0 & \mathit{if}\, t \in Y(Rej) \\ 0 & otherwise \end{cases}$$

In essence, the product MDP M_ϕ runs the DRA Γ_ϕ in parallel with the system model $\boldsymbol{M}$. M_ϕ extracts interesting conditions $AP = \mathcal{L}(s)$ from the system state s and feeds AP

to DRA Γ_ϕ. Γ_ϕ is given a penalty for the "bad" states $Y(Rej)$. The following theorem holds for product MDP:

Theorem 1. Given MDP *M* and a LTL induced DRA Γ_ϕ, if there exists a control policy $\pi : S \rightarrow A$ that satisfies ϕ with probability 1, then any policy synthesis algorithm that minimizes the *expected total penalty* of the product MDP M_ϕ will find such a policy.

A complete proof can be found in [41]. Intuitively, if the penalty given to the "bad" states is large enough, the algorithm that minimize expected total penalty would avoid these states as much as possible. Consequently, the target LTL ϕ is respected.

4.3.4 Q-learning

Q-learning [49] is a reinforcement learning-based algorithm to find ϕ-compatible policy π in Theorem 1. The core idea of Q-learning is to keep track of the historical total penalty $Q : S \times T \times A \rightarrow \mathbb{R}$. This Q table records the total penalty seen after making decision $a \in A$ at state $s \times t$, or "the penalty of increasing the power demand when the power budget is violated" translating to a datacenter example.

Q-learning updates the Q table when it observes the system made a state transition $s \times t \xrightarrow{a \in A} s' \times t'$ and was penalized c.

$$Q(s,t,a) = (1-\eta)Q(s,t,a) + \eta(c + min_{b \in A} Q(s',t',b)) \tag{4.1}$$

where $0 < \eta < 1$ controls the learning rate.

Once the Q table is obtained, the control policy can be derived:

$$\pi(s,t) = argmin_{a \in A} Q(s,t,a) \tag{4.2}$$

A major advantage of Q-learning is that it does not require the probability transition matrix P_M. In our method, the system designer determines the penalty of immediately "bad" states $Y(Rej)$ and Q-learning serves to "spread" the penalty to other states. By equation (4.1), the more frequently a state can transit into an immediately "bad" state, the higher its penalty is. However, the Q table has $|S| \times |T| \times |A|$ entries, which can be a huge number in a complex system. Learning for a huge state space is usually impractical. We show in section 4.4 this can be reduced to $|T| \times |A|$.

4.4 LTL-based Resource Management

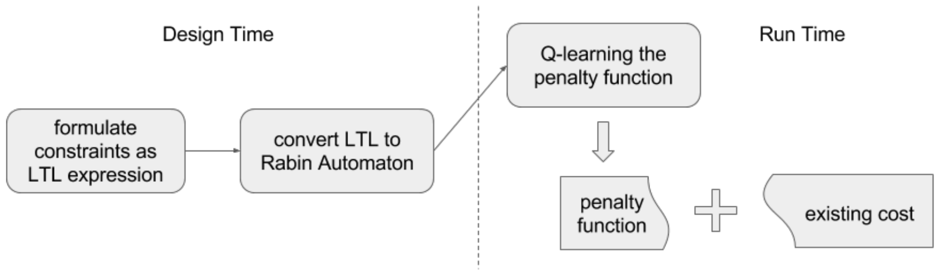

Design Time
Run Time
formulate constraints as LTL expression
convert LTL to Rabin Automaton
Q-learning the penalty function
penalty function
existing cost

Figure 4.2: An overview of LTL-based resources management.

The framework of our method is depicted in figure 4.2. The input of our method is an LTL expression that formulates the desired constraints of the system. The output of our method is the DRA and a learned penalty function that can be combined with a conventional resource manager. The decisions of the enhanced resource manager are made by optimizing the summation of the penalty and the cost function $F(s, a)$ of the conventional resource manager.

Our method is split into the design time phase and the run time phase. In design time, the desired correctness property is formulated in an LTL expression ϕ and then transformed to a DRA $\Gamma_\phi = (T, AP, \delta, Rej)$. This transformation is well studied in the model checking theory [47].

The run time phase starts with a learning period where a penalty function $Q : T \times A \to [0, \infty)$ is trained. This is a major distinction over Theorem 1 of previous work [41]: the penalty function Q is created on the space $T \times A$. In contrast, Theorem 1 dictates a table of $S \times T \times A \to \mathbb{R}$. The system state space S for datacenter resource management problems can easily explode to hundreds of thousands, but the DRA state space T is usually limited.

The reason our approach removes S is intuitive. In the previous work [42], the synthesized policy not only needs to observe the correctness constraints but also has to optimize the other design goals (efficiency, fault-tolerance, etc.). Therefore system states S is indispensable for the decision making. In our method, the conventional resource manager $F(s, a)$ is responsible for the other design goals. The learning phase merely seeks to measure the level of emergency of the LTL violations. Therefore DRA state space T suffices.

The constant c in algorithm 2 is the immediate penalty of violating ϕ, a parameter set by system designer. A history of system trace (t_i, a_i) is used as the input. t_i is the DRA state and a_i is the decision made by the conventional resource manager without LTL.

Algorithm 2 learning algorithm

```
Require: DRA $\Gamma_\phi = (T, AP, \delta, Rej)$, learning rate $\eta$, constant $c > 0$, trace
    $(t_0, a_0), (t_1, a_1), ..., (t_n, a_n)$.
1: $Q(t, a) \leftarrow 0, i \leftarrow 0$
2: for all $i < n$ do
3:     $p \leftarrow t_{i+1} \in Y(Rej)?c : 0$
4:     $Q(t_i, a_i) \leftarrow (1 - \eta) * Q(t_i, a_i) + \eta * (max_{b \in A}(Q(t_{i+1}, b)) + p)$
5: end for
6: return Penalty function $Q$
```

After the learning phase is completed, the penalty function Q is added to the cost function of the conventional resource management algorithm $F(s, a)$ (equation (4.3)). Many resource management algorithms are in the form of or can be translated to minimizing a carefully designed cost function [50] [51] [52]. During normal operation, the penalty magnitude is close to zero and thus has no impact on the resource manager. When the DRA is heading towards a "bad" state, the penalty begins to add bias to the resource manager. Parameter λ is a weighing factor between F and Q.

$$min_{a \in A} F(s, a) + \lambda Q(t, a) \tag{4.3}$$

4.5 Power Capping in Distributed UPS Data Center

In this section we apply our method to the power capping problem in distributed UPS data centers.

4.5.1 The Problem

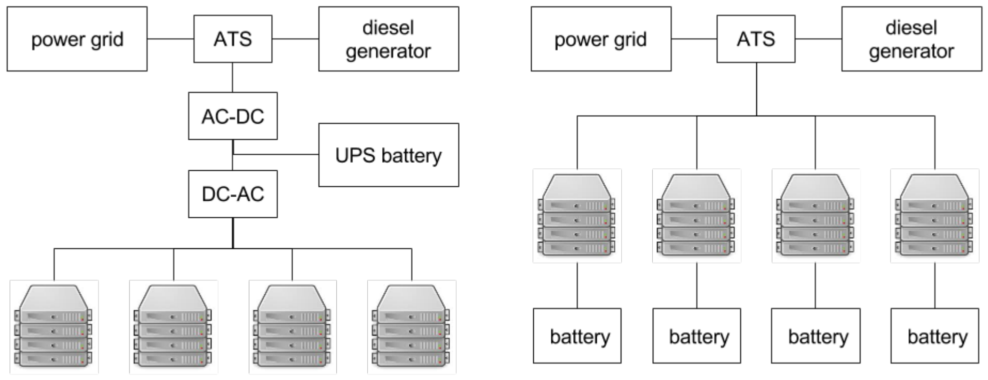


Figure 4.3: Two power architecture of data centers.

Traditionally the UPS system is placed on the top level of the warehouse scale data center (figure 4.3). Its sole purpose is to provide power during transition from external power grid to internal diesel generator should a power outage occur. The centralized UPS system has a poor scalability and requires an AC-DC-AC conversion, which can have a low efficiency (88% reported in [53]). This double conversion stage can be removed if the UPS system is distributed in server or rack level. The distributed UPS system is adopted in the leading data center designs including Google [54] and Facebook [55]. Furthermore, the distributed UPS system can be leveraged as an energy buffer for effective power capping [39]. The UPS batteries provide additional power during the power peaks and are recharged in the power valley. The power management infrastructure can thus cap the total power drawn from the grid without slowing down or shutting off the server machines.

Despite its advantages, distributed UPS system brings up one problem for power capping and management: it is harder to guarantee the high availability of UPS system. The problem actually is two-fold.

- Firstly, it is not straightforward to determine when and where to cap the power demand so that the batteries are not overdrawn.
- Secondly, the frequent discharge of UPS battery deteriorates the battery's lifetime and this rate of deterioration is non-uniform among distributed batteries.

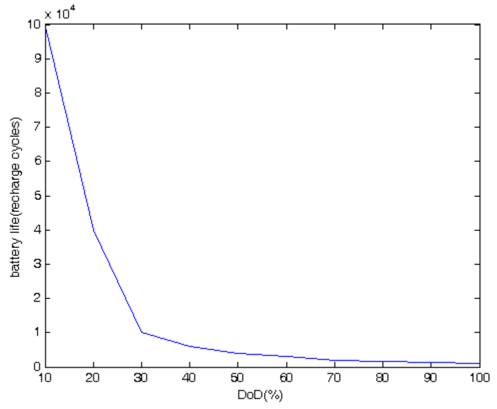


Figure 4.4: Battery life deteriorates substantially for deeper discharge cycles.

Depth of Discharge (DoD) is a term that refers to the designed maximum percentage of battery capacity drained at a discharge cycle. The battery life, usually measured in the number of recharge cycles, deteriorates 100 times faster in large DoDs(figure 4.4). Even if only a single battery fails when it should provide power for an outage or a power peak, the consequence can be disastrous.

4.5.2 LTL Constrained Power Capping

Table 4.2 summarizes a power manager designed with our methodology.

field	value	description
LTL ϕ	$\mathbf{G}[(true\mathbf{U}_{k+n}d)$ $\|\|\mathbf{F}_k(r\mathbf{U}_n true)]$	power peak $\Rightarrow$ battery recently charged
$d \in AP$	boolean	cluster power demand $<$ cluster-level budget
$r \in AP$	boolean	p% of racks: rack power demand $<$ rack-level budget
s	$cpuReq$, $memReq$, $cpuUtil$, $memUtil$	cpu & mem request of queued jobs cpu & mem utilization of servers
L	$(d, r) = L(s)$	sum up server power to get d, r
a	booleans	turn off servers

Table 4.2: LTL constrained power capping summary.

We assume an existing power efficiency server manager F that determines which servers are left ON to accommodate the current workload. The cost function of F :

$S \times A \rightarrow [0, \infty)$ is designed as:

$$\begin{aligned}
cpuD(s) &= \sum_{i \in DC} cpuUtil_i * mips_i + \sum_{j \in ShedQ} cpuReq_j \\
memD(s) &= \sum_{i \in DC} memUtil_i * mem_i + \sum_{j \in ShedQ} memReq_j \\
F(s,a) &= w_{cpu} * max(0, cpuD(s) - \sum_{i \in a} mips_i) \\
&\quad + w_{mem} * max(0, memD(s) - \sum_{i \in a} mem_i)
\end{aligned} \tag{4.4}$$

where DC is the datacenter, SchedQ is the scheduling queue, w_{cpu} and w_{mem} are weights for cpu and memory resources, $cpuReq_j$ and $memReq_j$ are the resource request from the jobs in the scheduling queue, $cpuUtil_i$ and $memUtil_i$ are cpu and memory utilization of individual servers. $cpuReq_j$, $memReq_j$, $cpuUtil_i$, $memUtil_i$ are the observed system states that the conventional resource manager bases its decision on. $mips_i$ and mem_i are cpu and memory capacities of server i, they capture the hardware heterogeneity of the datacenter. The power consumption of a server is calculated by the power model in table 4.3. $cpuD$ and $memD$ sum up the cpu and memory demand of executing workload and resources requests in the scheduling queue. $\sum_{i \in a} mips_i$ and $\sum_{i \in a} mem_i$ are the aggregated computation resources supply after turning off servers indicated by a. By minimizing F, server manager merely strives to promote power efficiency: it turns off as much servers as possible before computation resources are insufficient for the workload. In a real datacenter, server manager needs to take into account additional factors such as thermal variation. However, it requires little change for our algorithm to work with more sophisticated resource manager.

We further assume a per-rack UPS system as in [55]. The LTL formula ϕ is designed to impose the following property: power peak $\Rightarrow$ $p\%$ rack-level batteries are recently charged. In ϕ, only $p\%$ instead of 100% of the rack-level batteries are required to be charged. This is because the cluster-level power budget is shared by all racks in the datacenter, one insufficiently charged battery does not entail power capping (figure 4.1).

More specifically, at any time t, if cluster-level power demand>cluster-level budget (power peak), then rack-level power demand<rack-level budget (enable recharge) in time span $[h-n, h]$ for at least $p\%$ of all racks where $t-k<h<t$.

The equivalent LTL formula ϕ for this property is $\mathbf{G}[(true\mathbf{U}_{k+n}d)||\mathbf{F}_k(r\mathbf{U}_n true)]$. ϕ is built on two system conditions (AP): d = "cluster-level power demand<budget", and r = "rack-level power demand<budget". $(true\mathbf{U}_{k+n}d)$ states that it is not a power peak in $k+n$ time. $\mathbf{F}_k(r\mathbf{U}_n true)$ dictates a continuous charging period of length n within k time. d and r are extracted from system states s by label function $\mathcal{L}$, and drive DRA $\Gamma_\phi = (T, AP, \delta)$ through state transition table $t = \delta(t, d, r)$. Note the batteries are required to be charged continuously.

A DRA that corresponds to ϕ is generated through an implementation of [47]. The DRA Γ_ϕ is shown in figure 4.5. The states are arranged in an $n \times k$ matrix. The grey scale of the circle indicates the emergency of constraint violation and will later be learned by the penalty Q. Thin arrows in the graph indicates the state transitions that deteriorate the constraint and the thick arrows mean the contrary. A thin arrow transition is executed if $r = false$ and a thick arrow transition if $r = true$. The blue arrow is driven by $d = true$.

Now we inspect Γ_ϕ in greater detail and explain its logic. The first row of Γ_ϕ corresponds to the trace of a continuous peak $\underbrace{!r \quad !r \quad ... \quad !r}_{k}$. At any time along this trace, Γ_ϕ can break into charging mode if $r = true$. In charging mode (the columns), the state records the charging length. If a power peak interrupts the charge, the state goes back to the first row (the diagonal arrows). Otherwise it reaches the n-th row and stays there until $r = false$ again. d only comes into effect when Γ_ϕ is right before the constraint violation ($s1$). It gives Γ_ϕ a second chance if the datacenter does not see a present power

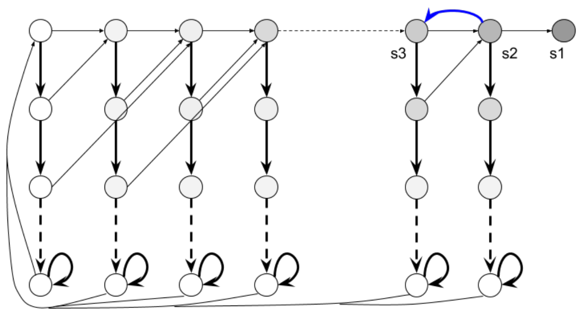


Figure 4.5: DRA Γ_ϕ for power capping constraint.

peak.

Even though Γ_ϕ has a clear semantic interpretation, it is generated by algorithms [47] that are proven not to overlook any corner cases of the LTL constraint. This is the key idea of LTL-based resource management.

4.5.3 Experimental Results

For our experiments, we developed a discrete event simulator for the datacenter trace published by Google in 2011 [56]. The trace contains $12.5K$ heterogenous servers with cpu and memory utilization data updated for every 5 minutes. We partition the servers in 306 racks assuming 42U racks and one UPS battery per rack. The UPS battery capacity is set to $500ah$ [39], which can support the rack for 37.5 minutes if fully charged. The rack can be powered either by the power grid or by the rack-level UPS battery. We measure the performance loss by average scheduling delay of jobs during the power peak. Scheduling delay has been a primary metric to measure performance in datacenters [57]. One reason for choosing scheduling delay is that our traces include jobs that execute for months, so other metrics such as job throughput and execution time are largely biased by these jobs. Important simulation parameters are summarized in table 4.3.

Inability of Previous Model Checking Method

We first show that the previous model-checking-based approach can not handle prob-

peak power	$2.85MW$
cluster-level power budget	$2.43MW$
capped power magnitude	14.7%
#servers	12551
#racks	306
rack-level power budget	$8KW$
power model (server is on)	$P_i = P_{cpu}\frac{mips_i}{max_{i\in DC} mips_i} cpuUtil_i$ $+P_{mem}\frac{mem_i}{max_{i\in DC} mem_i} memUtil_i$ $+P_{idle}(\frac{w_{cpu}}{(w_{cpu}+w_{mem})}\frac{mips_i}{max_{i\in DC} mips_i}$ $+\frac{w_{mem}}{(w_{cpu}+w_{mem})}\frac{mem_i}{max_{i\in DC} mem_i})$
$P_{idle}/P_{cpu}/P_{mem}$	$175W/105W/70W$ [39]
P_{sleep} (server is off)	$30W$ [36]
w_{cpu} / w_{mem}	$\frac{P_{cpu}}{P_{cpu}+P_{mem}} / \frac{P_{mem}}{P_{cpu}+P_{mem}}$
simulated time	$\sim$10 days
rack-level UPS battery	500 $amp\ hour/12V$ [39]
UPS sustain time for outage	37.5 minutes when fully charged
server manager invocation	every 5 minutes
k (history length)	60 minutes
n (recharge time)	30 minutes
$p\%$ (recharge percentage)	75%

Table 4.3: Simulation and design parameters.

lems of this scale. In the previous work [42] [45], PRISM [46] is extensively used as a model checking tool and performs the core functionality in their methods. We formulated an MDP model for the datacenter of varying size, and tested the runtime of a simple "reachability" property (the only kind of properties supported for policy synthesis in PRISM). As shown in table 4.4, even in the Monte-Carlo approximation mode, the model-checking-based approach is too slow for online problems.

#servers	10	100	1000	5000
run time (s)	1.7	11.3	268.5	timeout (3h)

Table 4.4: Run time of PRISM model checker.

Power Efficiency and Power Capping Trade-off

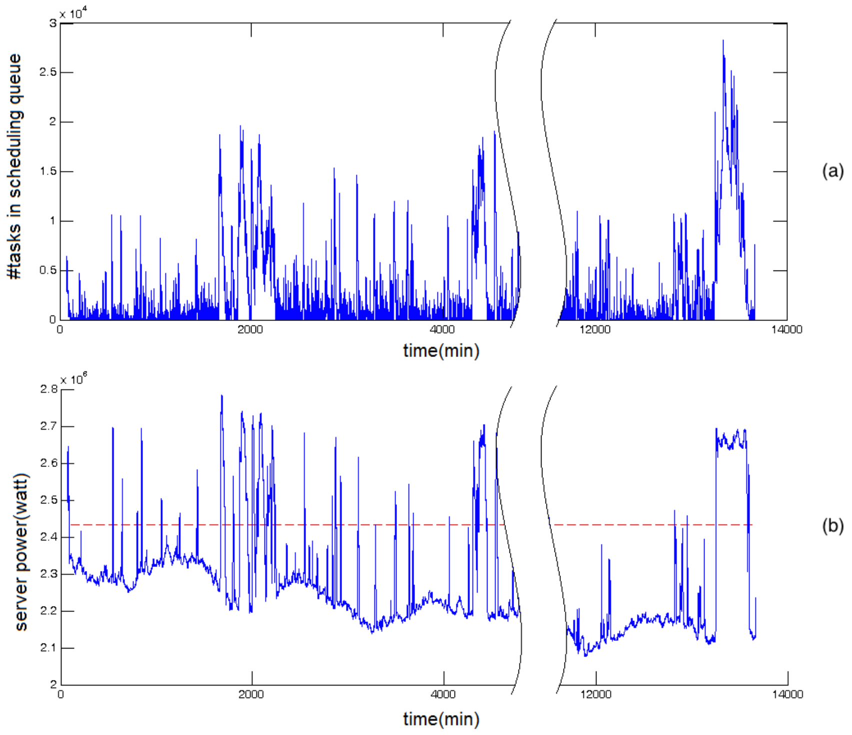


Figure 4.6: Simulation of 10-day trace.

An overview of the trace is shown in figure 4.6. We compare three strategies:

- *F: a conventional server manager that optimizes power efficiency alone* (equation (4.4)).
- *ClustCtrl: power capping server manager adapted from a previous work* [39].
- *F+LTL: power efficiency server manager F with power capping temporal LTL constraint.*

In [39], the design of a $10MW$ datacenter of Google distributed UPS architecture is explored. Several policies are described to utilize UPS battery during power peaks and shave the power profile of the datacenter. We adapt the best policy *ClustCtrl* in [39] for power capping purpose. The strategy is described in algorithm 3. Note *ClustCtrl* ignores the heterogeneity of server power consumption.

Algorithm 3 ClustCtrl Policy

```
    $Diff \leftarrow ClusterPower - PowerBudget$
 2: $\Delta Bats \leftarrow |Diff|/AvgRackPower$
    **if** $Diff \geq 0$ & $\geq \Delta bats$ batteries available **then**
 4:     Enable $\Delta bats$ batteries
    **else if** $Diff \geq 0$ & $< \Delta bats$ batteries available **then**
 6:     Enable available batteries
        Cap the most power demanding rack until $Diff \leq 0$
 8: **else if** $\Delta Bats \geq 5$ **then**
        Disable and Recharge $\Delta bats$ batteries
10: **end if**
```

Table 4.5 shows the effect of the three methods throughout the 10-day trace. AvgPower is the average power consumption over the 10 day period. BatDepTime is the total amount of time the rack batteries are depleted. The number is aggregated over 306 racks and 10 days. It represents hazardous status of the datacenter. The SchedDelay column shows the average scheduling delay. AvgDoD is the worst Depth of Discharge value experienced throughout 10 days. It is the average value of 306 racks and reflects the battery usage of all racks.

method	AvgPower	BatDepTime	SchedDelay	AvgDoD
F	$2.23MW$	$7356min$	$315s$	93%
$ClustCtrl$	$2.43MW$	$3257min$	$342s$	66%
$F + LTL$	$2.22MW$	$3414min$	$351s$	69%

Table 4.5: Overall result of the 10-day trace. BatDepTime is the total amount of time rack batteries are depleted and records hazardous status of the datacenter.

The efficiency only server manager F strives to boost energy efficiency without considering power capping necessity. Power capping reduces the battery depletion time by half at the expense of larger power consumption. Because $ClustCtrl$ will recharge batteries when the power demand is below the power budget and discharge them when the power demand is above the power budget, the average power consumption converges to the power budget value. With LTL-enhanced server manager F, power capping decisions from LTL do not interfere with power efficiency decisions of F unless it is necessary. The performance loss of $ClustCtrl$ and $F + LTL$ due to power capping are comparable.

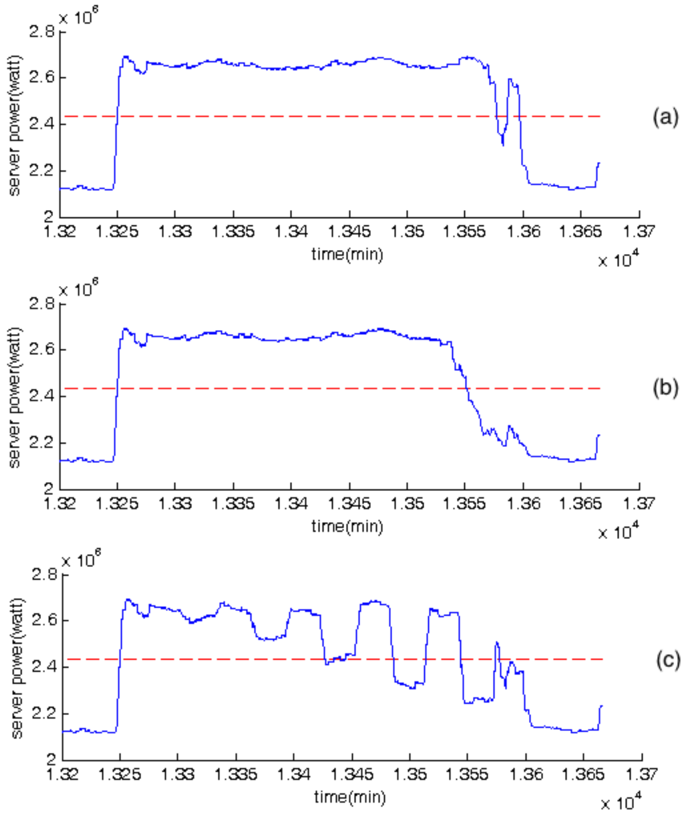


Figure 4.7: Power demand for peak at 1320 minute with (a) F, (b) $ClustCtrl$, (c) $F + LTL$.

We further show in detail the longest peak in the trace, the 6-hour period that starts at 1320 minute in figure 4.6. In case of $ClustCtrl$, power capping does not come into effect until the last portion of the peak. In comparison, the LTL constrained server manager identifies the emergency of power capping and creates deeper power valleys as the peak proceeds.

F+Hard Bounding Method

Imposing a hard bound on the energy left in the UPS batteries is a simple strategy to prevent overdrawn batteries without the ability to formulate temporal constraints, $\sum_{i \in batteries} Energy_i < threshold$. However, hard bounding does not deal with all possible use cases. In particular, servers in a datacenter have a very diverse power profile due to hardware heterogeneity, workload fluctuation and task placement constraints [58]. These heterogeneity is not captured by the hard bounding method as it treats all the batteries as an aggregated entity. Note it is possible to enforce hard bound on the individual rack level, however this approach suffers from the serious intra-rack coordination problem detailed in [39].

load balance	$F + LTL$	$F + HB$
Homogeneous	$872min$	$1113min$
Heterogeneous	$1180min$	$7499min$

Table 4.6: Deterioration of hard bounding method with heterogeneous job distribution.

Table 4.6 shows another power peak from 1600 minute to 2400 minute. $F + HB$ is the power efficiency server manager with a hard bound (30% of the total capacity). We disturb the load balancing in the original trace to generate a more heterogeneous job distribution in the datacenter. It can be seen from table 4.6 that $F + HB$ deteriorates by almost 7 fold with heterogeneous job distribution.

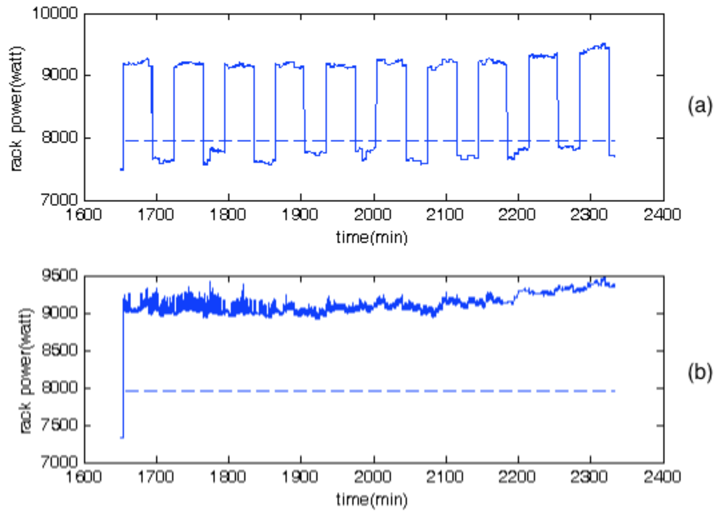


Figure 4.8: $F + LTL$ and hard bounding method on a particular rack with heterogenous job distribution. (a) Power capping by $F + LTL$. (b) Power capping is not exerted for hard bounding method and a high battery depletion time ensues.

To explain the results above, we further analyze the power consumption of one particular rack in the datacenter in figure 4.8. Because of the heterogeneous job distribution, this rack sees constant high power demand throughout the peak even though the cluster level demand fluctuates around the power budget. For hard bounding method (figure 4.8.b), power capping is not exerted because energy left in other rack batteries are still abundant. This results in a high BatDepTime for the rack in question.

Effect of Power Budget

Power budget is an important design parameter that affects datacenter capital investment and power capping strategy. A low power budget downsizes the power infrastructure and the supporting cooling equipment, at the cost of performance degradation and the difficulty of power capping.

By increasing the power budget, it is always possible to lower the battery usage and prevent low energy state. Table 4.7 shows the power budget increase needed to achieve the

Power Budget	2433kw⇒2515kw
Discharge Energy	5582wh⇒4142wh
Depth of Discharge	93%⇒69%
Capital Increase	\$1.6million *

* assume \$20 capital investment per watt [39].

Table 4.7: Increase power budget to achieve the same AvgDoD as $F + LTL$.

same AvgDoD (69%) as the LTL constrained approach. This increase is further translated into an additional capital investment of \$1.6million.

Adjust Battery Life with Recharge Time

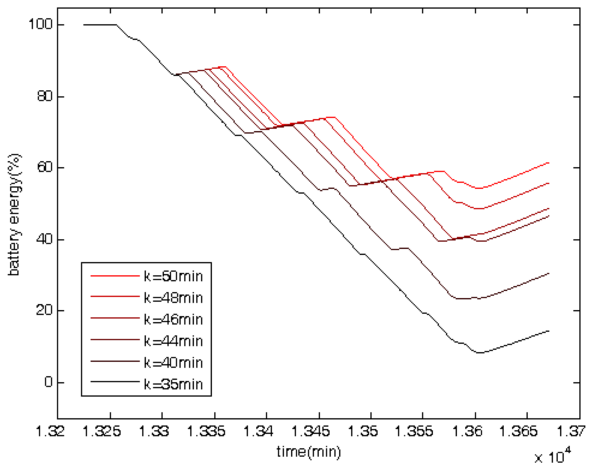


Figure 4.9: Modulate n to adjust battery life to a desired level.

We show that by changing the recharge time n, it is possible to tune the battery DoD. Figure 4.9 contains energy loss curves for 6 different n in peak starting $1320min$. The history length $k = 60$ minutes and $p = 100\%$ remains constant in the experiment. Larger values of n dictate longer recharge time and reduces the DoD. As can be seen in figure 4.9, DoD for the peak changes from 93% to 46% in response to n. This implies we can adjust the average battery life to a desired level.

4.6 Fair Scheduling

In this section we show another application of our methodology. We assume a multi-tenant cloud service that provides IT capacity to multiple users. The users submit their jobs to the datacenter and wait for the execution results. We assume the users have the same priority so the datacenter should serve them in a fair manner (For users with priority, our method can be adapted by adjusting the parameter in LTL for each user). From the perspective of the datacenter, however, resources management seeks to maximize the computation throughput. We design an experiment to demonstrate that:

- Conventional schedulers can be at two extreme ends of the performance-fairness tradeoff;
- LTL can achieve a continuous tradeoff between performance and fairness.

4.6.1 Emulation Platform

In this experiment, we created a small cluster of 20 virtual cpus and 50 GB memory running hadoop 2.4.1 [59] on Amazon AWS cloud. We execute the 2010 Facebook MapReduce trace [60] on this cluster to simulate a multi-tenant datacenter. To curb the experiment time, we only run a 2-hour fragment of the 24-hour trace. The fragment contains 200 MapReduce Jobs, the job submissions are shown in figure 4.10.a. We randomly assign the MapReduce jobs to 10 of the users. Because the number of tasks and the shuffle ratios of each job vary drastically, the workload of each user is very different. We found $\textit{amount of data written} \approx \textit{input splits} \cdot \textit{input shuffle ratio} \cdot (1 + \textit{output shuffle ratio})$ is a good estimate of the workload. The workload estimate per user is shown in figure 4.10.b. To make the problem more interesting, we also added a bursty user (user0) whose jobs are

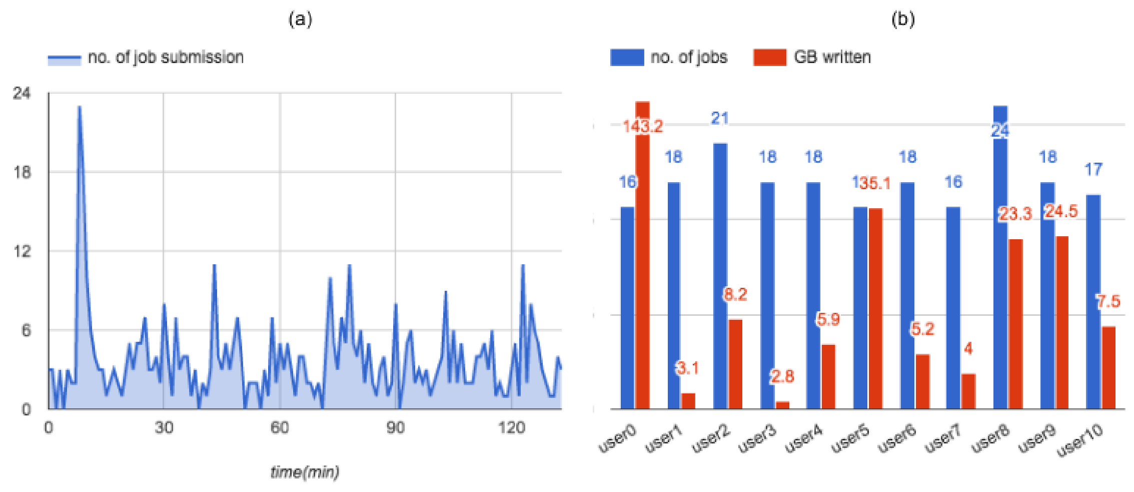


Figure 4.10: 200 MapReduce jobs are randomly assigned to 11 users. The job submission lasts for 2 hours.

all heavy weight ones.

4.6.2 Conventional Job Schedulers

When the aggregated resource demand is below the total resource supply of the cluster, all users' demands are granted and the resource allocation is uneven. When multiple users are competing for resource, a strictly fair scheduler should allocate equal amount of resource to each user despite the variation in demands. In more rigid terms, a scheduler is said to be *max-min fair* [61] if that the resource is allocated in infinitesimal pieces and the user with the minimum allocation who is still demanding gets the next piece. The *max-min fairness* metric only addresses the resource sharing problem, so it may result in a penalty for throughput.

Conversely, a max-throughput scheduler prioritizes light weight jobs. When cluster resource is released by a finishing job, the scheduler allocates the resource to a demanding job whose resource request is lowest:

$$\begin{aligned} & min_{j \in SchedQ} F(j) \\ & F(j) = ResourceRequest_j \text{ (either cpu or memory)} \end{aligned} \tag{4.5}$$

The max-throughput scheduler maximizes the performance of the datacenter in terms of throughput but ignores the fairness among users. As we will show, this could starve some users of the cluster resource.

Hadoop has three embedded schedulers: capacity, fair share and dominant resource fairness (DRF) [62] scheduler. Capacity scheduler manages multiple users by assigning a fixed proportion of total resource to each user job queue. But this does not suggest each user can only use resource below its share. An user queue is also allowed to seize the idle resource beyond their assigned capacity. In contrast, fair share scheduler assigns weights to users and determines the resource allocation according to the weight ratios. It approximates *max-min fairness* when the weights are equal for all users. In this experiment we set all the user queues to equal capacity share and equal weights. Fair share and capacity scheduler consider only a single resource: memory, whereas DRF takes into account both cpu and memory. The reasoning behind DRF is that jobs should be compared by their "dominant resource share", which is the bigger of the cpu percentage and the memory percentage that a job wants to claim from a cluster. It is obvious that DRF is more fair measure to compare a cpu-bound job with a memory-bound job.

4.6.3 F+LTL Scheduler

In addition, we designed an LTL expression to enforce "no starvation" constraint for all users. Let boolean h_i denote there is a queued job from user i. Let boolean p_i denote that a job from user i is dispatched. $\phi = (h_i \Rightarrow GF_m p_i) = (!h_i \| GF_m p_i)$ asserts if there are jobs in queue from user i, then there must be at least a job from user i dispatched within m

time. Following the procedure described in section 3, we transform ϕ into an DRA, train its penalty function and add this penalty to the cost function F in equation (4.8).

4.6.4 Experimental Results

Performance-Fairness Tradeoff of Hadoop Schedulers

scheduler	turn-around time(s)	Jain's index
Max Thrpt F	204(100%)	0.54
Capacity	234(115%)	0.40
F+LTL	271(133%)	0.75
Fair Share	316(155%)	0.81
DRF	333(163%)	0.82

Table 4.8: Schedulers of different turn-around time and fairness. Jain's index = 1 is the most fair case.

Table 4.8 shows the result of executing the 2-hour trace with LTL, three hadoop schedulers and our implementation of max-throughput scheduler. The turn-around time is the time from when a job is submitted to when it is finished, table 4.8 shows the average value of the 200 jobs. To quantify the level of fairness, we use Jain's fairness index [63] as a metric. For n numbers $x_1, ..., x_n$, Jain's index is calculated by $\frac{(\sum_j x_j)^2}{n \cdot \sum_j x_j^2}$. It attains the maximum value 1 when $x_1 = x_2 ... = x_n$ and the minimum $\frac{1}{n}$ when $x_j = 0$ for all but one x_j. We also show the scheduling delay and turn-around time of each user in figure 4.11.

Among the 5 schedulers, max-throughput scheduler and capacity scheduler have the fastest throughput but the lowest fairness score. The unfairness is better illustrated by the large variations in the heights of the bars in figure 4.11. To speed up the overall job execution, the scheduling delay of the max-throughput scheduler for user0 increases by 2.6 times compared to that of DRF. In contrast, DRF is much more fair with an average turn-around time of 332 seconds, 63% slower than the max-throughput scheduler. Fair

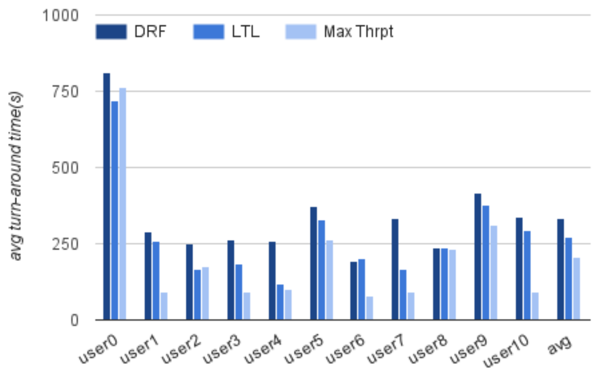


Figure 4.11: The height of the last bar shows the average performance. The evenness of the bars indicates the resource allocation fairness. Max-throughput and DRF schedulers are on the extremes of the performance-fairness tradeoff. LTL falls in-between.

share scheduler achieves similar results as DRF. This is primarily because most of the MapReduce jobs in the trace are cpu-bounded.

As is shown in table 4.8, the LTL scheduler achieves a balance between max-throughput scheduler and DRF. In fact, figure 4.12 shows we can achieve a continuous trade-off by changing the parameter m in the LTL expression.

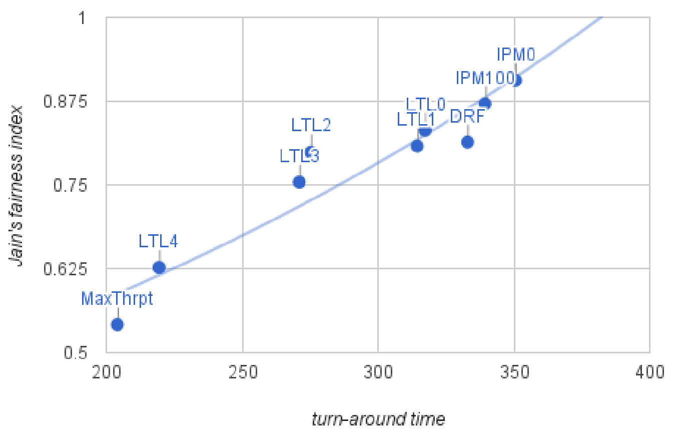


Figure 4.12: Performance-fairness tradeoff of DRF, 5 LTL and 2 IPM schedulers.

Comparison to Fairness Function Tuning

Despite *max-min fairness*, other less strict definitions of fairness such as α*-fairness* [64] and *proportional fairness* [65] have also been proposed. Recently it is shown that many notions of fairness including the above mentioned can be unified under a single family of fairness functions [66]:

$$f_{\beta,\lambda}(x) = sign(1-\beta)\left(\sum_{j=1}^{n}\left(\frac{x_j}{\sum_{k=1}^{n} x_k}\right)^{1-\beta}\right)^{\frac{1}{\beta}}(\sum_{j=1}^{n} x_j)^{\lambda} \tag{4.6}$$

In equation (4.6), x_j indicates the resource allocation of user j, λ is the efficiency weight and β determines what kind of fairness notion is used. The term $(\sum_{j=1}^{n} x_j)^{\lambda}$ captures the efficiency of the allocation: the scheduler should maximize the total amount of resource utilized $\sum_{j=1}^{n} x_j$. The rest of the equation (4.6) evaluates the fairness of the allocation. If we take $\lambda = \frac{1-\beta}{\beta}$, equation (4.6) reduces to α-fairness. If we take $\beta \to \infty$, equation (4.6) gives proportional fairness.

In this experiment, we take $\beta = 2$ as in the previous work [66]. Taking the derivative of equation (4.6), we get

$$\frac{\partial f_{\lambda}(x)}{\partial x_j} \propto (1+2\lambda)\sum_{k}\frac{1}{x_k}x_j^2 - \sum_{k} x_k \tag{4.7}$$

The fairness function (4.6) is convex, so we can employ interior point method with the above gradient to optimize it. We implemented interior point method within the hadoop framework. The results are shown in figure 4.12 as IPM0 and IPM100. These two points correspond to two values, 0 and 100, for the efficiency weight $1 + 2\lambda$. Figure 4.12 shows the performance-fairness tradeoff that can be obtained by manipulating fairness function is limited. Notably, the LTL schedulers are located towards the northwest corner of the figure 4.12, which is a direction with more fairness and less turn-around time.

4.6.5 Simulation Results

In this experiment we extract workload pattern from the 2010 Facebook MapReduce trace [60]. We simulate 600 quad-core servers as G/M/c queues, the same model used in [39]. The trace provides job submission time and job size for 5894 jobs in a 24 hour period. Execution time for the tasks ranges from 10 minutes to an hour. The block size for Hadoop MapReduce is set to $64MB$, so we estimate number of tasks for each job as $taskQuantity = jobSize/64MB$. We assume the jobs are submitted by 10 users, one out of which has a bursty submission pattern. A max-throughput scheduler F selects next job to dispatch in the scheduling queue by:

$$min_{j \in schedQ} F(j) = min_{j \in schedQ} taskQuantity_j \tag{4.8}$$

The max-throughput scheduler prioritizes jobs of small sizes so that it maximizes the throughput of task execution. But this strategy may hurt the fairness of resource allocation and leave the bursty user in starvation. This is shown in figure 4.13. Figure 4.13.a displays the pattern of job submissions for each user. Figure 4.13.b shows the cumulative amount of jobs dispatched for max-throughput scheduler. Note the red line corresponds to the bursty user and the quantity of dispatched jobs remains zero.

One way to attain *max-min fairness* is round-robin, which is shown in figure 4.13. Round-robin scheduler enforces a tight form of fairness called *max-min fairness* [61]. The numbers of cumulative dispatched jobs increase at the same rate for all the users. In our experiment, the round-robin scheduler is 20% less efficient than max-throughput scheduler in terms of throughput.

The result of F+LTL scheduler is shown in figure 4.14. It can be seen the bursty user

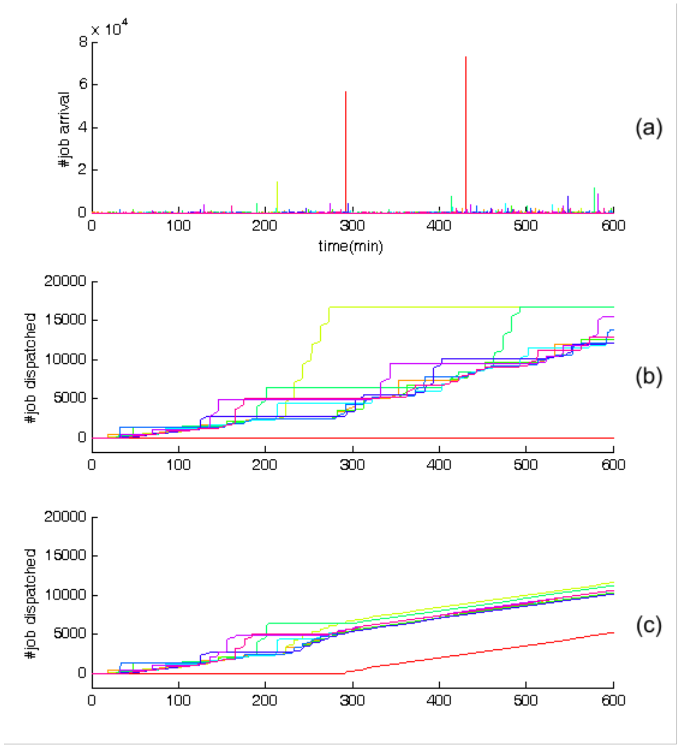


Figure 4.13: Max-throughput vs. round-robin. Round-Robin scheduler enforces *max-min fairness* with a throughput degradation (20%).

is no longer starved but allocated less bandwidth than others.

In fact, we can achieve a continuous trade-off between max-throughput and round-robin scheduler. In figure 4.15, the blue curve shows the average throughput (#jobs per second) for all users. The green curve shows the percentage of the total dispatched jobs that belong to the bursty user. Max-throughput scheduler achieves 3.5 tasks/s and 0% (starvation) in the graph (the diamonds). Round-robin scheduler achieves 2.8 tasks/s and 5.4% (the circles). By changing the parameter m, a designer can choose any operating point suitable.

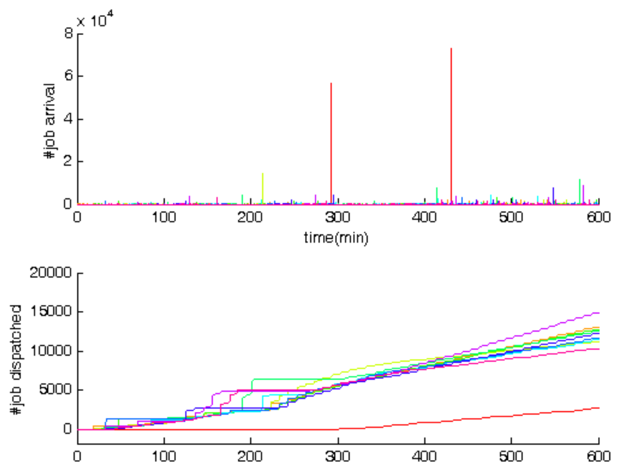


Figure 4.14: The max-throughput scheduler is augmented with starvation-free LTL constraint ϕ.

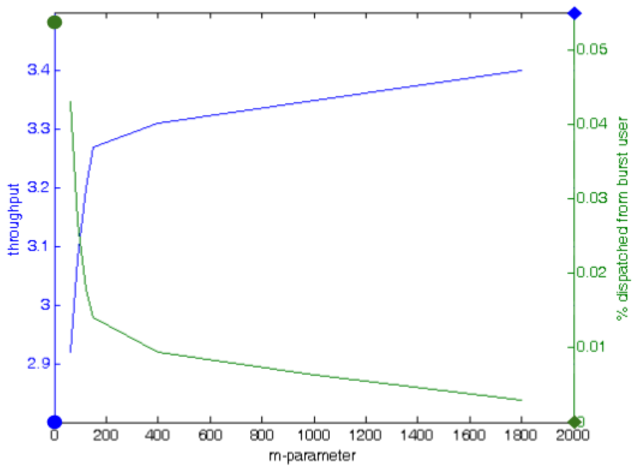


Figure 4.15: Throughput-fairiness trade-off by modulating m. A continuous trade-off curve is achieved between max-throughput scheduler (diamonds) and round-robin scheduler (circles).

5. CONCLUSION

In this dissertation, we visited three resource management problems for computation hardware. We first propose a methodology and algorithmic techniques for joint gate implementation selection and adaptivity assignment of adaptive circuit design. Experimental results show that our method can substantially reduce adaptivity overhead. Secondly, we propose new algorithms for TDM-based NoC QoS with consideration of power-efficiency. Our SAT approach solves QoS time slot allocation problem with $2\times \sim 3\times$ success rates compared to a previous work. Our path-based ILP approach further takes into consideration the network link capacity and buffer size optimization. It provides solutions that cost about 25% less energy dissipation than an iterative greedy heuristic and are near to the optimal solutions, and is order of magnitude faster than the optimal method. In the third scenario, we demonstrate a new framework of the resource management, which leverages the expressiveness of Linear Temporal Logic in decision constraints and the LTL constraints is seamlessly integrated with conventional objectives through reinforcement learning. This new approach is validated for datacenter power management and multi-tenant job scheduling. Simulations are performed on Google and Facebook traces with comparison with previous works. The results show that our approach is superior in balancing multiple conflicting objectives.

REFERENCES

[1] J. W. Tschanz, J. T. Kao, S. G. Narendra, R. Nair, D. A. Antoniadis, A. P. Chandrakasan, and V. De, “Adaptive body bias for reducing impacts of die-to-die and within-die parameter variations on microprocessor frequency and leakage,” *IEEE Journal of Solid-State Circuits*, 2002.

[2] M. Agarwal, B. C. Paul, M. Zhang, and S. Mitra, “Circuit failure prediction and its application to transistor aging,” in *Proceedings of the IEEE VLSI Test Symposium*, 2007.

[3] S. V. Kumar, C. H. Kim, and S. S. Sapatnekar, “Mathematically assisted adaptive body bias (ABB) for temperature compensation in gigascale LSI systems,” in *Proceedings of Asia and South Pacific Design Automation Conference*, 2006.

[4] X. Liang, G.-Y. Wei, and D. Brooks, “Revival: a variation-tolerant architecture using voltage interpolation and variable latency,” *IEEE Micro*, 2009.

[5] K.-N. Shim and J. Hu, “Boostable repeater design for variation resilience in VLSI interconnects,” *IEEE Transactions on VLSI Systems*, 2013.

[6] K.-N. Shim, J. Hu, and J. Silva-Martinez, “Dual-level adaptive supply voltage system for variation resilience,” *IEEE Transactions on VLSI Systems*, 2013.

[7] K. M. Brownell, A. D. Khan, G.-Y. Wei, and D. Brooks, “Automating design of voltage interpolation to address process variations,” *IEEE Transactions on VLSI Systems*, 2011.

[8] C. R. Lefurgy, A. J. Drake, M. S. Floyd, M. S. Allen-Ware, B. Brock, J. A. Tierno, J. B. Carter, and R. W. Berry, “Active guardband management in Power7+ to save energy and maintain reliability,” *IEEE Micro*, 2013.

[9] S. H. Kulkarni, D. M. Sylvester, and D. T. Blaauw, “Design time optimization of post-silicon tuned circuits using adaptive body bias,” *IEEE Transactions on Computer-Aided Design*, 2008.

[10] C. Zhuo, D. Blaauw, and D. Sylvester, “Variation-aware gate sizing and clustering for post-silicon optimized circuits,” in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, 2008.

[11] M. Mani, A. Singh, and M. Orshansky, “Joint design time and post-silicon minimization of parametric yield loss using adjustable robust optimization,” in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 2006.

[12] A. Agarwal, D. Blaauw, and V. Zolotov, “Statistical timing analysis for intra-die process variations with spatial correlations,” in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 2003.

[13] Y. Kunitake, T. Sato, and H. Yasuura, “A replacement strategy for canary flip-flops,” in *Proceedings of the IEEE Pacific Rim International Symposium on Dependable Computing*, 2010.

[14] M. Guthaus, N. Venkateswaran, C. Visweswariah, and V. Zolotov, “Gate sizing using incremental parameterized statistical timing analysis,” in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 2005.

[15] D. Sinha, N. V. Shenoy, and H. Zhou, “Statistical timing yield optimization by gate sizing,” *IEEE Transactions on VLSI Systems*, 2006.

[16] A. Srivastava, D. Sylvester, and D. Blaauw, *Statistical analysis and optimization for VLSI: timing and power*. 2006.

[17] H. Chang and S. S. Sapatnekar, “Statistical timing analysis under spatial correlations,” *IEEE Transactions on Computer-Aided Design*, 2005.

[18] A. Lu, H. He, and J. Hu, “Proximity optimization for adaptive circuit design,” in *Proceedings of the 2016 on International Symposium on Physical Design*, 2016.

[19] C. P. Chen, C. C.-N. Chu, and D. F. Wong, “Fast and exact simultaneous gate and wire sizing by Lagrangian relaxation,” *IEEE Transactions on Computer-Aided Design*, 1999.

[20] H. He, J. Wang, and J. Hu, “Collaborative gate implementation selection and adaptivity assignment for robust combinational circuits,” in *Low Power Electronics and Design, 2015 IEEE/ACM International Symposium on*, 2015.

[21] M. Ozdal, C. Amin, A. Ayupov, S. Burns, G. Wilke, and C. Zhuo, “An improved benchmark suite for the ISPD-2013 discrete cell sizing contest,” in *Proceedings of the ACM International Symposium on Physical Design*, 2013.

[22] A. R. Agnihotri, S. Ono, and P. H. Madden, “Recursive bisection placement: Feng Shui 5.0 implementation details,” in *Proceedings of the ACM International Symposium on Physical Design*, 2005.

[23] R. Marculescu, U. Y. Ogras, L.-S. Peh, N. E. Jerger, and Y. Hoskote, “Outstanding research problems in NoC design: system, microarchitecture, and circuit perspectives,” *IEEE Transactions on Computer-Aided Design*, 2009.

[24] H. He, G. Yang, and J. Hu, “Algorithms for power-efficient quality-of-service in application specific network-on-chips,” in *Low Power Electronics and Design, 2014 IEEE/ACM International Symposium on*, Aug 2014.

[25] A. Hansson, K. Goossens, and A. Rădulescu, “A unified approach to constrained mapping and routing on network-on-chips architectures,” in *Proceedings of the IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2005.

[26] S. Stuijk, T. Basten, M. C. W. Geilen, A. H. Ghamarian, and B. D. Theelen, "Resource-efficient routing and scheduling of time-constrained network-on-chips communication," *Elsevier Journal of Systems Architecture*, 2008.

[27] Z. Lu and A. Jantsch, "TDM virtual-circuit configuration for network-on-chips," *IEEE Transactions on VLSI Systems*, 2008.

[28] R. Stefan and K. Goossens, "A TDM slot allocation flow based on multipath routing in Network-on-Chips," *Elsevier Journal on Microprocessors and Microsystems*, 2011.

[29] M. Schoeberl, F. Brandner, J. Sparsø, and E. Kasapaki, "A statically scheduled time-division-multiplexed network-on-chips for real-time systems," in *Proceedings of the IEEE/ACM International Symposium on Networks-on-Chip*, 2012.

[30] G. Yang, H. He, and J. Hu, "Resource allocation algorithms for guaranteed service in application-specific NoCs," in *Proceedings of the IEEE International Conference on Computer Design*, 2013.

[31] J. Hu and R. Marculescu, "Energy-aware communication and task scheduling for network-on-chips architectures under real-time constraints," in *Proceedings of Design, Automation and Test in Europe Conference*, 2004.

[32] I. Walter, E. Kantor, I. Cidon, and S. Kutten, "Capacity optimized Network-on-Chips for multi-mode System-on-Chips," in *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 942–947, 2011.

[33] D. Rhodes, R. Dick, and K. Vallerio, "Task graph for free." http://ziyang.eecs.umich.edu/ dickrp/tgff/.

[34] J. Hu and R. Marculescu, "Energy and performance-aware mapping for regular Network-on-Chips architectures," *IEEE Transactions on Computer-Aided Design*,

vol. 24, pp. 551–562, Apr. 2005.

[35] H. Wang, X. Zhu, L. Peh, and S. Malik, "Orion: a power-performance simulator for interconnection networks," 2002.

[36] A. Gandhi, M. Harchol-Balter, and M. A. Kozuch, "Are sleep states effective in data centers?," in *Green Computing Conference, 2012 International*, 2012.

[37] C. B. Bash, C. D. Patel, and R. K. Sharma, "Dynamic thermal management of air cooled data centers," in *Thermal and Thermomechanical Proceedings 10th Intersociety Conference on Phenomena in Electronics Systems.*, 2006.

[38] F. Ahmad and T. Vijaykumar, "Joint optimization of idle and cooling power in data centers while maintaining response time," in *ACM Sigplan Notices*, 2010.

[39] V. Kontorinis, L. E. Zhang, B. Aksanli, J. Sampson, H. Homayoun, E. Pettis, D. M. Tullsen, and T. S. Rosing, "Managing distributed UPS energy for effective power capping in data centers," in *Computer Architecture, 39th Annual International Symposium on*, 2012.

[40] C. Baier and J. Katoen, *Principles of model checking*. MIT press Cambridge, 2008.

[41] D. Sadigh, E. S. Kim, S. Coogan, S. S. Sastry, and S. A. Seshia, "A learning-based approach to control synthesis of markov decision processes for linear temporal logic specifications," in *53rd IEEE Conference on Decision and Control*, 2014.

[42] A. Gounaris, "Probabilistic model checking at runtime for the provisioning of cloud resources," in *6th International Conference on Runtime Verification*, 2015.

[43] K. Johnson, R. Calinescu, and S. Kikuchi, "An incremental verification framework for component-based software systems," in *Proceedings of the 16th International ACM Sigsoft symposium on Component-based software engineering*, 2013.

[44] M. M. Bersani, D. Bianculli, S. Dustdar, A. Gambi, C. Ghezzi, and S. Krstić, "Towards the formalization of properties of cloud-based elastic systems," in *Proceedings of the 6th International Workshop on Principles of Engineering Service-Oriented and Cloud Systems*, 2014.

[45] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli, "Dynamic quality-of-service management and optimization in service-based systems," *IEEE Transactions on Software Engineering*, 2011.

[46] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *23rd International Conference on Computer Aided Verification*, 2011.

[47] J. Esparza and J. Křetínskỳ, "From LTL to deterministic automata: A safraless compositional approach," in *Computer Aided Verification*, 2014.

[48] M. Y.Vardi, "Automatic verification of probabilistic concurrent finite state programs," in *Foundations of Computer Science, 26th Annual Symposium on*, 1985.

[49] C. J. C. H. Watkins, *Learning from delayed rewards*. PhD thesis, University of Cambridge England, 1989.

[50] L. Eyraud-Dubois and H. LarchevÃłque, "Optimizing resource allocation while handling sla violations in cloud computing platforms," in *Parallel Distributed Processing, IEEE 27th International Symposium on*, 2013.

[51] C. C. Lin, P. Liu, and J. J. Wu, "Energy-aware virtual machine dynamic provision and scheduling for cloud computing," in *Cloud Computing, 2011 IEEE International Conference on*, 2011.

[52] L. Chen, H. Shen, and K. Sapra, "Distributed autonomous virtual resource management in datacenters using finite-markov decision process," in *Proceedings of the*

ACM Symposium on Cloud Computing, 2014.

[53] D. A. Patterson, "The data center is the computer," *Communications of the ACM*, 2008.

[54] "Google. http://www.google.com/about/datacenters/efficiency/internal/."

[55] Facebook, "Hacking conventional computing infrastructure. http://opencompute.org/."

[56] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google cluster-usage traces: format+schema," *Technical Report, Google Inc., Mountain View, CA, USA*, 2011.

[57] Q. Zhang, M. F. Zhani, S. Zhang, Q. Zhu, R. Boutaba, and J. L. Hellerstein, "Harmony: Dynamic heterogeneity-aware resource provisioning in the cloud," in *Distributed Computing Systems, IEEE 33rd International Conference on*, 2013.

[58] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proceedings of the Third ACM Symposium on Cloud Computing*, 2012.

[59] "Apache hadoop. http://hadoop.apache.org/."

[60] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, "The case for evaluating MapReduce performance using workload suites," in *2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, July 2011.

[61] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European conference on Computer systems*, 2010.

[62] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, “Dominant resource fairness: Fair allocation of multiple resource types.,” in *USENIX Symposium on Networked Systems Design and Implementation*, 2011.

[63] R. Jain, D. M. Chiu, and W. R. Hawe, *A quantitative measure of fairness and discrimination for resource allocation in shared computer system.* Eastern Research Laboratory, Digital Equipment Corporation Hudson, MA, 1984.

[64] J. Mo and J. Walrand, “Fair end-to-end window-based congestion control,” *IEEE/ACM Transactions on Networking*, vol. 8, pp. 556–567, Oct 2000.

[65] F. P. Kelly, A. K. Maulloo, and D. K. Tan, “Rate control for communication networks: shadow prices, proportional fairness and stability,” *Journal of the Operational Research society*, 1998.

[66] C. Joe-Wong, S. Sen, T. Lan, and M. Chiang, “Multiresource allocation: Fairness-efficiency tradeoffs in a unifying framework,” *IEEE/ACM Transactions on Networking*, 2013.