Capacity Metric for Chip Heterogeneous Multiprocessors

Mwaffaq Naif Otoom

Dissertation submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

February 23, 2012
Falls Church, Virginia

Keywords: Capacity Metric, Chip Heterogeneous Multiprocessors, Multichannel Workloads, Workload Modes, Workload Specific Processors

Capacity Metric for Chip Heterogeneous Multiprocessors

Mwaffaq Naif Otoom

ABSTRACT

The primary contribution of this thesis is the development of a new performance metric, *Capacity*, which evaluates the performance of Chip Heterogeneous Multiprocessors (CHMs) that process multiple heterogeneous channels. Performance metrics are required in order to evaluate any system, including computer systems. A lack of appropriate metrics can lead to ambiguous or incorrect results, something discovered while developing the secondary contribution of this thesis, that of workload modes for CHMs – or Workload Specific Processors (WSPs).

For many decades, computer architects and designers have focused on techniques that reduce latency and increase throughput. The change in modern computer systems built around CHMs that process multi-channel communications in the service of single users calls this focus into question. Modern computer systems are expected to integrate tens to hundreds of processor cores onto single chips, often used in the service of single users, potentially as a way to access the Internet. Here, the design goal is to integrate as much functionality as possible during a given time window. Without the ability to correctly identify optimal designs, not only will the best performing designs not be found, but resources will be wasted and there will be a lack of insight to what leads to better performing designs. To address performance evaluation challenges of the next generation of computer systems, such as multicore computers inside of cell phones, we found that a structurally different metric is needed and proceeded to develop such a metric.

In contrast to single-valued metrics, Capacity is a surface with dimensionality related to the number of input streams, or channels, processed by the CHM. We develop some fundamental Capacity curves in two dimensions and show how Capacity shapes reveal interaction of not only programs and data, but the interaction of multiple data streams as they compete for access to resources on a CHM as well. For the analysis of Capacity surface shapes, we propose the development of a demand characterization method in which its output is in the form of a surface. By overlaying demand surfaces over Capacity surfaces, we are able to identify when a system meets its demands and by how much. Using the Capacity metric, computer performance optimization is evaluated against workloads in the service of individual users instead of individual applications, aggregate applications, or parallel applications. Because throughput was originally derived by drawing analogies between processor design and pipelines in the automobile industry, we introduce our Capacity metric for CHMs by drawing an analogy to automobile production, signifying that Capacity is the successor to throughput. By developing our Capacity metric, we illustrate how and why different processor organizations cannot be understood as being better performers without both magnitude and shape analysis in contrast to other metrics, such as throughput, that consider only magnitude.

In this work, we make the following major contributions:

- Definition and development of the Capacity metric as a surface with dimensionality related to the number of input streams, or channels, processed by the CHM.
- Techniques for analysis of the Capacity metric.

Since the Capacity metric was developed out of necessity, while pursuing the development of WSPs, this work also makes the following minor contributions:

- Definition and development of three foundations in order to establish an experimental foundation – a CHM model, a multimedia cell phone example, and a Workload Specific Processor (WSP).
- Definition of Workload Modes, which was the original objective of this thesis.
- Definition and comparison of two approaches to workload mode identification at run time; The Workload Classification Model (WCM) and another model that is based on Hidden Markov Models (HMMs).
- Development of a foundation for analysis of the Capacity metric, so that the impact of architectural features in a CHM may be better understood. In order to do this, we develop a Demand Characterization Method (DCM) that characterizes the demand of a specific usage pattern in the form of a curve (or a surface in general). By doing this, we will be able to overlay demand curves over Capacity curves of different architectures to compare their performance and thus identify optimal performing designs.

Dedication

… and my success (in my task) can only
come from Allah (God). In Him I trust,
and unto Him I look. (*Quran 11:88*)

Acknowledgments

First and foremost, I would like to thank my thesis advisor and mentor, JoAnn Paul, without whom this work would be impossible. You generously gave me time, effort, and opportunities to properly complete this work. Your insight has always inspired my thinking as a researcher, shaped my views on research and teaching, and helped me to focus my goals; something would greatly help me to excel in my next career. Thank you for being patient and encouraging me throughout tough times.

I would like to acknowledge my committee members: Greg Kulczycki, Tom Martin, Scott Midkiff, and Cameron Patterson for serving on my committee and for their valuable comments and feedback on the work.

I thank Virginia Polytechnic Institute and State University and its staff, especially Marija Telbis-Forster, Cynthia Robinson, Cynthia Hopkins, Anita Thacker, and Danielle Tillilie for their valuable help during my five-year study at VT.

This work was supported in part by the National Science Foundation through grant 0607934. Opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

Words cannot express my deepest gratitude to my parents, who motivated me to take up this challenge and helped me complete it with their love and support.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

AM: Arithmetic Mean

ASICs: Application Specific Integrated Circuits

ASP: Application Specific Processor

CAD: Computer Aided Design

CHMs: Chip Heterogeneous Multiprocessors

CLS: Chip Level Scheduler

CP: Control Processor

DCM: Demand Classification Method

DSP: Digital Signal Processor

EEMBC: The Embedded Microprocessor Benchmark Consortium

FPGAs: Field Programmable Gate Arrays

FSM: Finite State Machine

HCL: Heterogeneous Core Level (Design)

HDL: Hardware Definition Language

HM: Harmonic Mean

HMMs: Hidden Markov Models

GALS: Globally Asynchronous, Locally Synchronous

GCS: Global Chip State

GM: Geometric Mean

GPP: General Purpose Processor

ILP: Instruction Level Parallelism

ISA: Instruction Set Architecture

ISS: Instruction Set Simulator

MA: Multiple-Application

MESH: Modeling Environment for Software and Hardware (Simulator)

MIMD: Multiple Instruction, Multiple Data

MISD: Multiple Instruction, Single Data

MPSoC: Multiprocessor System on a Chip

MU: Multiple-User

NoC: Network on Chip

PE: Processing Element

RTL: Register Transfer Level (Design)

SA: Single-Application

SIMD: Single Instruction, Multiple Data

SISD: Single Instruction, Single Data

SMT: Simultaneous Multithreading

SoC: System on a Chip

SPEC: Standard Performance Evaluation Corporation (Benchmark Suite)

SU: Single-User

SUMA: Single-User, Multiple Application

U-A: User-Application (Taxonomy)

UP: Usage Pattern

WCM: Workload Classification Method

WM: Workload Mode

WSP: Workload Specific Processor

WSS: Working Set Size

Chapter 1

Introduction

If we knew what it was we were doing, it would not be called research, would it?

Albert Einstein

Performance metrics are required in order to evaluate any system, including computer systems. Latency and throughput are the two metrics commonly used to model performance of a computer system. Since the origins of computer design, computer architects and designers focus on techniques that reduce latency and increase throughput. The change in modern computer systems built around Chip Heterogeneous Multiprocessors (CHMs) that process multi-channel communications in the service of single users calls this focus into question. Modern computer systems are expected to integrate tens to hundreds of processor cores onto a single chip, often used by single users to access the Internet. The input to these systems is multi-channel, and the design goal is to integrate as much functionality as possible during a given time window. Latency assumes that reducing the execution time of individual tasks results in a better performing system. This is true so long as the system executes only one application at a time. Throughput assumes that the input demand is constant and that the output is a single stream. This, in turn, results in a single unit value that is described by its maximum, regardless of the amount of demand. While throughput can deal with multiprocessing via aggregation, it fails to model the performance of systems that process distinct, multiple heterogeneous channel inputs. Since it results in a single unit value, throughput does not model the impact of the type of demand on the capabilities of multiple processor cores that are heterogeneous. Further, since throughput assumes demand is invariant over time, it does not model the actual or useful work and is a function of supply only. Without the ability to compare designs of multicore computers that serve individual users, not only will the best performing designs not be found, but resources will be wasted and there will be a lack of insight as to what leads to better performing designs. We show that a structurally different metric is needed for the next generation of computer systems, such as multicore computers inside of cell phones.

The overall performance of a CHM that processes multichannel inputs must be understood as a collection of performance points that represent the variety of multichannel combinations the system can support. Accordingly, in this work, we develop a new performance metric, *Capacity*, which evaluates the performance of CHMs with multiple heterogeneous channels. Capacity is a surface with dimensionality related to the number of input streams, or channels, processed by the CHM. We develop some fundamental Capacity curves in two dimensions and show how Capacity shapes reveal interaction of not only programs and data, but also the interaction of multiple data streams as they compete for access to resources on a CHM. For the analysis of Capacity surface shapes, we propose the development of a demand characterization method in which its output is in the form of a surface as well. By overlaying demand surfaces over Capacity surfaces, we are able to identify when a system meets its demands and by how much. Using the Capacity metric, computer performance optimization is evaluated against workloads in the service of individual users instead of individual applications, aggregate applications, or parallel applications. We first introduce our Capacity metric for CHMs by drawing an analogy to automobile production, motivating that Capacity is the successor to throughput.

1.1 Automobile Analogy

CHMs have the potential to integrate tens to hundreds of heterogeneous processor cores onto single chips. Often they are used by single users in real time and in a wide variety of situations. The design goal is often to integrate as much functionality as possible for processing within a window of time, but also over a wide variety of anticipated processing scenarios [1]. Overall functionality is increasingly characterized as the processing of multichannel inputs, which result in multimodal situations. Multimodal situations are defined by the modeling of a variety of scenarios that arise from the intersection of user preferences and the data arrival from the Internet. The resultant system operates in different modes at different times and these modes must be recognized so that the system can respond to different loading situations.

The processing of multichannel input streams on multicore computers can be thought of as analogous to the production of multiple types of automobiles in a production plant composed of multiple manufacturing pipelines. This analogy also serves to illustrate how Capacity is a successor to throughput.

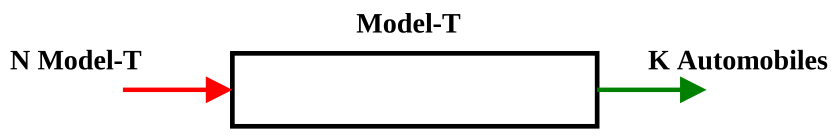

Figure 1: Batch Production

Automobiles were first produced in batches. In batch production, only one automobile would be assembled at a time. Figure 1 shows an automobile system example. The system receives a request of *N* Model-T automobiles and produces *K* automobiles in a given time interval. The design goal of these systems is to reduce the processing time of each product model. In computer systems, latency over one or more job types is an important performance metric for single application computers in which only one application is being executed at a time.

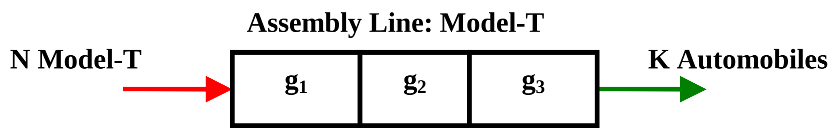

Figure 2: Single Assembly Line

Batch production was then replaced by assembly lines, which were first invented by Henry Ford. The motivation was to improve the throughput of the production system by dividing major tasks into smaller tasks that could be done simultaneously. Figure 2 shows a single automobile pipeline model. The pipeline is assumed to produce Model-T automobiles. The pipeline consists of three stages, g_j, where j is a stage sequence number. The input to the model is a demand for cars of type Model-T, and the output is the automobiles themselves. The pipeline throughput during a given time interval is defined by the duration of the bottleneck stage, or the slowest stage in the pipeline. In order to calculate the throughput of the system, all pipeline stages should be busy during the given time interval, which means that the input should not be the limiting factor. This analogy results in a single value for the throughput metric.

The design goal of assembly lines is to increase production throughput. The spilt of the production cycle into stages makes it possible to process multiple requests at the same time. In this example, while the execution time of individual models is similar to that of Figure 1, the throughput of this example system is three times that of the system in Figure 1, assuming all stages have equal cycle times. Next, we consider the performance of multiple heterogeneous assembly lines.

Figure 3 shows a heterogeneous two-pipeline plant – one pipeline produces Model-T automobiles and the other pipeline produces Model-A automobiles. Here, throughput is increased by implementing multiple assembly line designs. Multi-product scheduling on heterogeneous pipelines has been previously studied for automobile production [2]. The architecture of each pipeline is different, as would be expected in the production of different types of cars with different features. In Figure 3, they differ in terms of the number of stages and the duration of each stage; the *jth* stage on the line i is $g_{i,j}$. At the plant level, some coupling is expected between the overall production of the two types of cars. For example, the stage duration time of the Model-T pipeline could be different from that of Figure 2 due to starvation that arises because of the existence of other pipelines sharing the same inventories or workers.

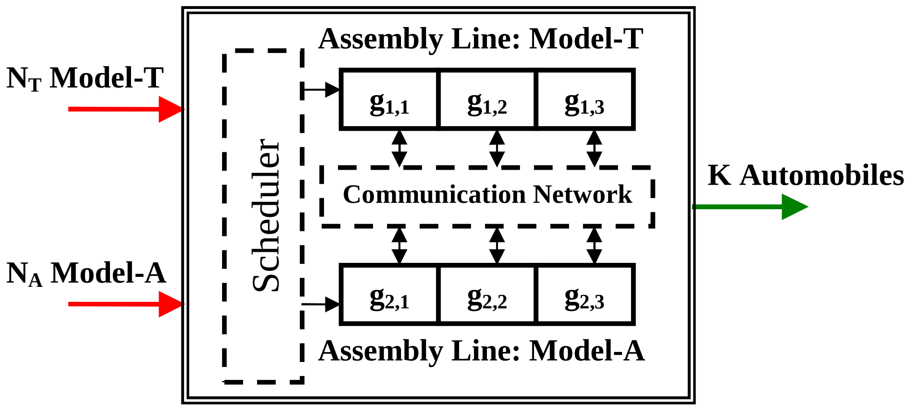

Figure 3: Multiple Assembly Lines Plant

In order to achieve the maximum production of the Model-T, production of the Model-A should be zero. This is the definition of throughput. Throughput of automobile assembly lines is found when individual production types are evaluated, resulting in a single unit of performance. This is analogous to performance evaluation of computer systems where computer architects evaluate performance using parallel programs. Each program can be multithreaded but runs individually, and so its overall rate of execution is evaluated. Similarly, microarchitects identify the maximum throughput, usually using a common work unit such as instructions per cycle. These maxima points can be used to compare different architectures or an average value can be generated.

Throughput is distilled into single score values using a common work unit. As a result, throughput can be shown as a straight line (a plane in higher dimensions). It connects the maximum production points of individual product models. This straight line represents the average production of this plant over a presumed common work unit for the heterogeneous production types. For example, the unit "automobiles" could be used instead of specific units for the Model-T and the Model-A. This average production results in a linear relationship as production of the Model-T and the Model-A are varied. This correctly models the production of completely independent channels in a decoupled system, but it does not capture the relationships of different production types as they access the common (shared) plant resources.

From the above discussion, there are three assumptions in which computer systems are evaluated and optimized for throughput. First is that demand is constant or maximum thus supply is the only limiting factor. The second assumption is that throughput results in a single output stream. This, in turn, results in a single unit value, described by its maximum, also regardless of the demand. Finally, work done by a production plant is not a function of time and is thereby not related to the input stream as a function of time. As a consequence of these assumptions, throughput does not model the useful work that can be done within a given time interval or the impact of the type of the product model on the capabilities of multiple lines(s) that are heterogeneous.

Traditionally, throughput could be used to model classes of parallelism because the similarity between elements of the parallel computation permitted evaluation to take place over a common work unit. However, in modern computer systems with multiple, diverse production streams that persist in the computer over long period of time, the differentiation in the types of production requires evaluation and analysis to consider the effects of workload heterogeneity. Next, we discuss our view of performance of the same plant system in Figure 3.

The performance of this plant can no longer be considered as distilled to a single number. To illustrate this concept, for the same plant system shown in Figure 3, consider Figure 4, in which we show the output of the plant as multiple heterogeneous channels. This plant can produce a wide variety of combinations of Model-Ts and Model-As in a given time interval. The production of one type of automobile affects the plant's ability to produce the other type of automobile. Characterization of the production capacity of this automobile plant as the maximum production of Model-Ts and Model-As requires a set of points, or a curve. Thus, there is no single-dimensional (single-unit, single variable) metric to model the Capacity of this plant because it now processes two input streams that share some common resources.

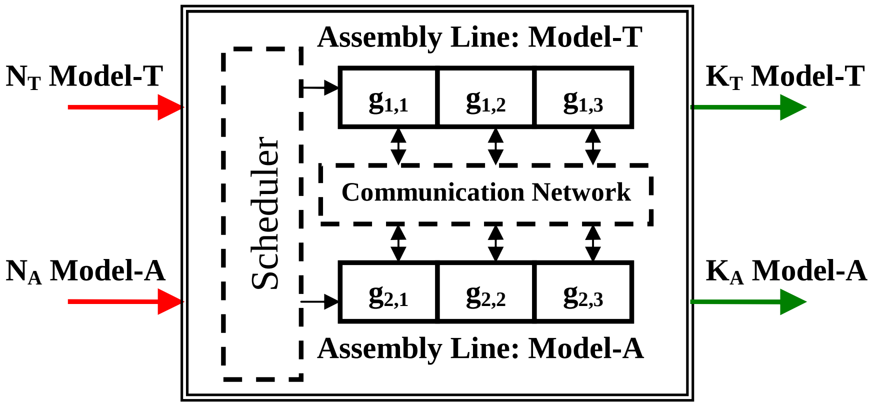

Figure 4: Our View of the Output of Multiple Heterogeneous Assembly Lines Plant

Because it lacks the ability to track interactions that occur in higher dimensions, throughput lacks cause and effect analysis, including, for example, the effects of combining different production types on the same plant. For instance, assuming that the supply of the plant does not increase, making more Model-Ts requires that resources be redirected from making the Model-A to making the Model-T. For a plant to increase the quantity of one model produced, production of the other model must be reduced. Here, production of the Model-A must be reduced in order to produce more of the Model-T. Further, throughput is presumed to be invariant over arbitrary time intervals, whereas, production can change with interval size as the relationships between types produced by a given plant can change with the period of time over which production is evaluated.

As a result of the above discussion, throughput cannot answer these questions:

Question 1: Can this plant satisfy the demand of N_T Model-Ts and N_A Model-As in so many days?

Question 2: For a specific increase in one of the supply features, which product models can take more relative advantage?

Question 3: How much of the production of Model-A must be reduced for a given increase in production of the Model-T?

Question 4: What new designs are better for different sets of production possibilities?

Each of the following questions has a parallel in the design of the CHMs that process workloads in the service of single users:

Question 1: Can this CHM design processes specific quantities of a set of tasks in a specific time window?

Question 2: What are the effects of adding architectural features on the production of specific demand streams of a computer system that simultaneously processes multiple output streams? This lends insight to designers about the cause and effect of design features in a CHM, to include processor choice, communications design and scheduler decisions.

Question 3: How much of one demand stream must be reduced for a given increase in the production of the other demand stream(s)?

Question 4: What CHM designs are better for different sets of application combinations and rates, or *workload modes*, and by how much, compared with other designs?

1.2 Chip Heterogeneous Multiprocessors (CHMs)

CHMs are increasingly being used to execute multichannel heterogeneous workloads, often in the service of single users. Multichannel inputs can be processed at different rates and in a variety of combinations. Single-user computers have a window of time during which the user would like to see a collection of jobs done. This window is tied to acceptable performance – something built into human perception, but also the way people now use their computers on the go. Within that window, we integrate as much functionality as we can. Latency, bandwidth, and recently throughput have served embedded systems and computer architecture design well for a long time because of the simple usage of computer devices at that time.

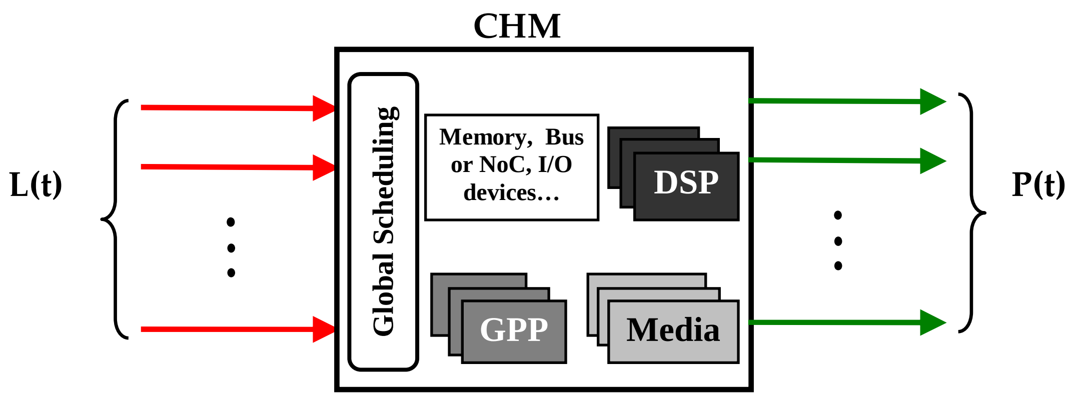

Figure 5: Input/Output of a Modern CHM

According to the Semiconductor Industry Association (SIA) [3], by the year 2015, future mobile devices will allow dozens of main processors and data processing engines to be placed onto an approximate area of an iPhone [4]. Furthermore, these processors will likely be heterogeneous [5], [6]. Heterogeneity in resources is being utilized not just to save power consumption but also to improve the performance of different demand types. Most of the current and potential processor types are classified into three categories in terms of specificity: General Purpose Processors (GPPs), Digital Signal Processors (DSPs), and Media processors. These categories fit the job types that dominate current mobile device applications such as speech recognition, mp3 streaming, video decoding, and text processing, etc. Future mobile computing devices have a the potential to integrate a network on chip (NoC) to connect the different internal components together [7]. Figure 5 shows a CHM integrating different processing, communication, and storage elements onto a single chip, as well as its multichannel input and output streams. Multichannel inputs can be processed at different rates and in a variety of combinations.

Also shown in Figure 5, at any given time, there is a set of demand streams, *L*, or channels. Each set potentially executes concurrently on a multicore processor; for example, while the user is surfing the Web, a Skype call arrives. The arrival of a new loading situation is identified as a new demand arrives. Note that the loading of the system is mainly a function of external timing, unlike uniprocessors which are sequenced by the speed at which an instruction stream is processed. Some demand streams may persist in the system indefinitely. Still others may not complete before the next demand (input) arrives. The interaction of the processing time with the external timing of inputs creates a workload which is a product of external timing from both the user and other inputs (such as the Internet) as well as internal processing. In modern computer systems with multiple, diverse production streams that persist in the computer over a long period of time, the differentiation in the types of production requires evaluation and analysis to consider the effects of demand heterogeneity. The performance of this system can no longer be considered as distilled to a single number.

Also shown in Figure 5 is the output of the system described as multiple heterogeneous streams. Each output stream is of a different type, thus the performance, *P*, of this system is defined as a wide variety of combinations of output streams in the given time interval, *I*. The production of one demand type affects the system's ability to produce the other types of demand.

The overall performance of the CHM that processes multichannel inputs must be understood as a collection of performance points that represents the variety of multichannel combinations the system can support.

1.3 Our Insight

Our metric is motivated as the successor to throughput via an analogy to automobile pipelines. We develop a Capacity metric as a curve (or a surface) with dimensionality related to the number of input streams, or channels, processed by the CHM. These curves give the designer the ability to understand the effects of adding architectural features on the performance of a computer system that simultaneously processes (produces) multiple output streams. Significantly, these curves do not result in linear relationships as demand for different production types varies. Distillation of rates of production over multiple production types to a common work unit – such as automobiles, instructions, tasks, or programs, results in a linear relationship as rates of production of more specific types within those categories varies. But if a straight line was superimposed on any of our Capacity curves, the real Capacity of the plants (designs) being examined will either underestimate the true Capacity of a given plant for a given set of production values, or it will overestimate what the plant is capable of.

We illustrate some fundamental Capacity forms and show how they may be used as the basis for evaluation and analysis of Capacity curves generated via measurement. Our Capacity curves are experimentally generated via measurement. Our overall goal is to motivate the development and use of the Capacity metric for performance evaluation of modern CHMs, specifically, we advocate investigation into how shapes of Capacity curves can be used to classify systems and identify how features of designs can be manipulated in order to change the shape of the Capacity curves. In order to achieve this, we show that Capacity surfaces (which we refer to as curves, as many of our experimental results are two-dimensional) have properties of both magnitude and shape, and each is required in order to understand workload performance. For example, we found the following types of Capacity curve shapes in our experimentation:

- Balanced – symmetric about a center point;
- Convex – favoring workload streams that are not mixed in type;
- Concave – favoring mixed workload streams over pure streams;
- Steep – representing a significant change as workload mix changes;
- Modal – including multiple local maxima, minima; and
- Independent – representing a decoupled system

By identifying these fundamental Capacity curve shapes of systems, we will be able to use Capacity curve's shape to analyze the significant features of a CHM. Future systems are anticipated to have many input streams, resulting in Capacity curves of higher dimensions that cannot be seen by graphical techniques. Therefore, a higher-dimension surface analysis will be done. For the analysis of Capacity surface shapes, we simply project these surfaces into their two-dimensional (2D) spaces. By doing this, we intend to facilitate analysis of these higher-dimension surfaces. We propose the development of a demand curve that characterizes a specific loading of a user. Then, we will compare the performance of different CHMs by overlaying a demand curve over their Capacity curves, identifying which design is better able to handle a specific demand.

1.4 Contributions Summary

By developing our Capacity metric, we illustrate how and why different processor organizations cannot be understood as being better performers without both magnitude and shape analysis, in contrast to other metrics, such as throughput, that consider only magnitude. The major contributions of this thesis are:

- Development of the Capacity metric as a surface or space with dimensionality related to the number of input streams, or channels, processed by the CHM.
- Techniques for analysis of the Capacity metric.

Since the need for the Capacity metric was discovered while pursuing the development of Workload Specific Processors, this work also makes the following minor contributions:

- Three foundations were developed in order to establish an experimental foundation – a CHM model, a multimedia cell phone example, and a WSP.
- The second is the definition of Workload Modes, which was the original objective of this thesis.
- The third is to develop, compare, and contrast two approaches to workload mode identification at run time; The Workload Classification Model (WCM) and another model that is based on Hidden Markov Models (HMMs).
- The fourth is to establish a foundation for the analysis of the Capacity metric so that the impact of architectural features in a CHM may be better understood. In order to do this, we developed a Demand Characterization Method (DCM) that characterizes the demand of a specific usage pattern in the form of a curve (or a surface, in general). By doing this, we will be able to overlay a demand curve over Capacity curves of different architectures to compare their performances.

1.5 Thesis Organization

Our work begins by discussing our observation that CHMs need to be designed and evaluated to workload modes. Then, we use an automobile production example to show how traditional metrics failed to evaluate the CHMs that process workload modes.

In Chapter 3, we develop three foundational models in order to establish an experimental foundation – a CHM model, a multimedia cell phone example, and a Workload Specific Processor. We contrast these foundations to those of the single-core computer design.

In Chapter 4, we summarize the relevant research on performance metrics. Since our Capacity metric was motivated by the need to evaluate WSPs, we also include a discussion of existing workload models and types of benchmarks. Finally, we discuss our MESH simulator that has been used to simulate the CHMs that process heterogeneous workloads, and show why existing HDL simulators failed to do this.

One of our secondary contributions in this thesis is the identification of workload modes for the purpose of real-time optimization, so we are focused on fast mode identification as well as the modeling of those modes that can be used to optimize system performance. In Chapter 5, we develop our WCM to identify workload modes at real time, and then extend it into the more complex HMM. We compare the two algorithms using our multimedia cell phone example.

To meet the central challenge of this thesis, the development of a new performance metric that can be used to evaluate the performance of the CHMs that process workloads, in Chapter 6 we introduce a descriptive, graphical, and denotational definition for our Capacity metric, contrast it with Pareto optimization, and show some basic forms of Capacity curves in 2D. Our ultimate goal is to motivate the development and use of the Capacity metric for performance evaluation of modern CHMs.

In Chapter 7, we advocate investigation into how information in the shape of the Capacity curves can be more significant than magnitude. We show how shapes of Capacity curves can be used to classify systems and identify how features of designs can be manipulated in order to change the shape of the Capacity curves.

The final chapter closes our thesis by summarizing the key contributions in this thesis and including future directions of research.

Chapter 2

Background

Somewhere, something incredible is waiting to be known.

Carl Sagan

The major contribution of this thesis is the development of a Capacity metric, the need for which was discovered when traditional metrics such as latency and throughput failed in pursuit of the thesis's original goal, which was designing single-user, multicore computers for patterns of workloads. Modern computers may potentially integrate tens to hundreds of heterogeneous processor cores onto single chips, or CHMs, often used in the service of single users. The input workload of these computers is becoming more situational, defined more by responding to a variety of situations, or *workload modes*, that arise from the intersection of user preferences and data arrival from the Internet. A workload mode is defined as a timed set of applications that simultaneously arrive at the system in a variety of combinations and at different rates. The sequence of workload modes exhibited by individual users may reveal a pattern. This opens up the possibility to design and evaluate CHMs used in the service of single users as WSPs. Toward that end, we found a need to rank the performance of different CHM designs that process workload modes in order to find optimal designs.

Traditional performance metrics such as latency and throughput failed to design and evaluate WSPs. Improving the latency of individual applications within workload modes is limited by user perception and preferences. More computing power does not always impact the user's ability to perceive it. Furthermore, when humans juggle sets of applications in which data is arriving in real time, they can only pay attention to so much information in a fixed amount of time. As a result, faster processing of some tasks is wasted on the user, the ultimate judge of performance. While throughput dominates parallel processing, it assumes that the relationship between work and the time interval in which it is measured is linear. In other words, increases in the time interval corresponds with increases in the amount of work that can be done in the same time period. This is representative only when the computer processes tasks in a batch style or independently on a general purpose processor or a homogeneous multiprocessor.

Since different CHMs are specialized for different sets of applications, depending on the computational requirements of these applications as well as the architectural features of the individual cores inside these CHMs, the performance of different CHMs may be ranked differently as different workload modes are being processed. In other words, one design may be ranked higher than another, but only over some of the anticipated set of workload modes for which it is being designed. Specific situations must be exposed in order to properly evaluate a design, requiring more information than single-valued metrics, such as throughput which models either fully loaded systems or averages, neither of which may occur in real designs. Distilling performance into single scores requires a function such as the mathematical means, as well as a common, short time, and work units. The determination of the appropriate average used in computer performance evaluation has long been controversial [8], especially because different means result in different design rankings. Short time units assume demand is continuous over the time interval in which performance is measured, resulting in an unreal performance. Because workload modes led us to the need to rank different designs, we found that it was impossible to rank them without a metric that simultaneously allows us to look at the different loading situations. As a consequence, the need for a new performance metric that shows how these workload modes can be used to evaluate the performance of different CHMs is necessary. In contrast to single-score metrics, our proposed metric shows performance as a surface with dimensionality related to the number of applications processed by the CHM. The surface includes all possible production modes of a CHM.

Next we will first discuss our observation that CHMs need to be designed and evaluated against workload modes. Then, we will show using an automobile production example demonstrating why traditional metrics failed to evaluate the CHMs that process workload modes.

2.1 Single-User Multicore Computers

Multicore, mobile devices are being widely used by single individuals for comprehensive wireless access to the Internet, often processing simultaneous applications. And yet, the design and evaluation of CHM computers have yet to be oriented toward workloads instead of toward benchmark suites, even parallel ones. Workloads differ from single applications distributed over multiple processors or the strictly timed applications that are consistent with traditional embedded systems because they arrive as sets that may overlap and which are distributed over time. In this thesis, we define these timed sets of applications as workload modes. The external timing of workload mode arrival is a primary factor in the potential to optimize. Thus, designing and evaluating CHMs for workload modes requires a fast runtime identification of persistent changes in loading that arise from patterns of use in single-user behavior, something we initially sought to develop in order to define and design WSPs. We found that existing performance metrics such as latency and throughput fail to properly rank CHM designs that process workload modes. We focus this chapter on why existing performance metrics cannot be used to rank the performance of this type of computing. We first introduce the motivation to design CHMs for patterns of workload modes.

2.1.1 Single-User Usage Patterns

In order to optimize CHMs according to the way individual users use their single-user computers, we first develop our definition of usage patterns through several examples.

Usage patterns are first introduced in [1]. Paul et al. introduced usage patterns or scenarios as a design methodology to help designers model and evaluate the performance of CHMs [1]. These scenarios model application’s functionality, concurrency, and arrival timing pattern. Gheorghita et al. [9] develop the concept of *system scenarios*. System scenarios are unique system behavior phases, called Run-time Situations (RTS), which have a similar cost. In order to identify these unique phases, a classification and clustering method based on the cost function is used. The overall use of these scenarios is to optimize the system at design-time, which in turn should result in performance gains at run rime. Thus, these scenarios need to be predicted at run time. In order to do this, Gheorghita et al. used the Markov-based predictor developed by Vandeputte et al. [10] .

In contrast with the work of Paul et al. [1], these scenarios cannot be generalized to all platforms because system scenarios are identified from the execution phases on specific platforms. This in turn complicates the design space exploration of embedded systems. Further, these scenarios are extracted from single program phases, so they do not model multiple co-executing programs. Still further, the cost of predicting these scenarios at run time is too expensive especially that their prediction also needs an on-line learning or calibration process. Finally, while the cost function that classifies scenarios can be anything, Gheorghita et al. focus only on energy consumption [9]. Thus, these scenarios are not an accurate representation of processing workloads.

While the usage pattern examples throughout this thesis are projections of future use, their implementation is real. Our projected usage patterns have proven accurate in scope, however, since the latest version of the iPhone allows a greater degree of application concurrency, to include Web browsing while speaking on the phone [11]. We will explore considerable work in real usage pattern profiling and discovery in section 4.4.

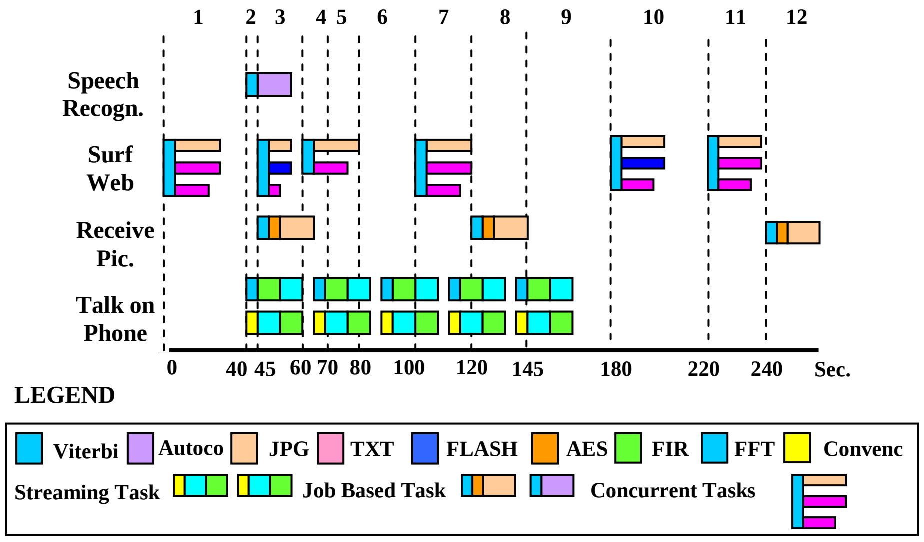

Figure 6: The First Usage Pattern

Figure 6 depicts a *usage pattern* in which the user uses the cell phone to call customers, receive pictures of products, talk on the phone, or check a Web page. The timeline is on the independent axis and applications are on the dependent axis. Each application consists of several tasks, summarized in the legend below Figure 6 and discussed later. Note that as time progresses, the user places different requirements on the cell phone system as different combinations of applications execute – a set of applications are run concurrently, driven by user intervention and the arrival of data from the Web. The task progression is also shown in Figure 6 (e.g. to receive a picture, the system must first perform a Viterbi decode, followed by rijndael decryption (AES) task, and finally perform JPEG decode). Because of task heterogeneity and concurrency, this usage model may benefit from execution on a heterogeneous multicore computer. Also note that aggregate models of workloads are not appropriate for such computer usage, especially when these usage patterns include external timing.

Aggregate workloads have been reasonable as long as they represented an approximation of computer usage, processed as single input streams on single processor computers. For batch execution systems in which the input is single stream, lower latency and higher throughput translate to improved performance because their performance is correctly evaluated using an aggregate model of their workload. In contrast, the variety of applications that execute on single-user computers that process heterogeneous multichannel input need to be executed on heterogeneous multicore processors; aggregate models of workloads are not appropriate. Situational models of workloads, however, are appropriate evaluation techniques for this kind of computer system. This type of situational loading with user saturation is unique to single-user, multicore computing, and results in the need to consider sets of workloads that are developed from usage patterns or the way single users use their computers.

Consider an individual workload mode, *wm*, which results from changes in patterns of inputs arriving over time. While external timing impacts the workload mode of the chip, workload modes are not distinctly presented to the system. In contrast to embedded systems in which external timing *specifies* system behavior, workload modes represent only the *potential* to optimize. This observation implies that changes in workload modes must be identified at runtime. From an architectural point of view, the distinction between a multiprocessor being in *wm* A vs. *wm* B is not significant unless there is the potential to optimize. Multicore computer design has yet to fully enter into a model in which evaluation, design, and optimization are oriented around single-user workload patterns that exhibit distinct situations.

Consider another usage pattern, shown in Figure 7. It involves a user listening to music and surfing the Web on the cell phone. When a phone call arrives, the "mp3 streaming" task is suspended by the usage program, although the "surf Web" and "send picture" tasks persist. Because of the different combinations of application sets, input data, deadlines, and constraints, this usage pattern may exhibit different system performance from the previous usage pattern, shown in Figure 6, causing the designer to tune the systems differently, despite each system executing the same set of applications.

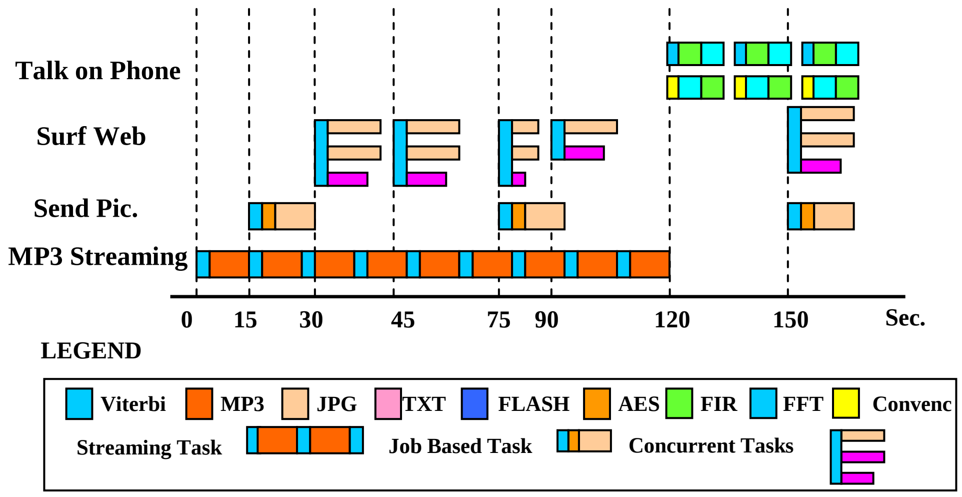

Figure 7: The Second Usage Pattern

Usage patterns provide not only applications and datasets to the system under test, but also arrival timing of workloads consistent with the perceived usage of the system. Usage patterns model the impact of overlapping data-dependent execution, fixed response times for streaming (periodic) inputs, and aperiodic arrival times of events, and can facilitate the evaluation of how well the system responds to the arrival of new applications while others persist in the system. In order to identify persistent changes in loading that arises from patterns of use in single-user behavior, we develop the basis to model and classify workload modes.

2.1.2 Workload Modes

A workload mode is a unique set of concurrently executing applications that includes external timing information that models user demand. Each workload mode opens up the possibility for tuning the system differently. It also models the effects of task interactions on the performance of a CHM. Workload modes are presented to the system in response to a mixture of timed external and untimed internal events. The external events result from user–computer interaction and data arrival. In Figure 6 and Figure 7, the vertical dashed lines identify individual workload modes, which are subsequently numbered at the top of Figure 6. At each dashed line, the load on the system is different enough to warrant consideration for chip level optimization. The internal events arise from changes in data that are being processed by existing applications as well as the execution completion of existing application(s). Since the timing of application execution completion is defined by the architecture, different architectures may generate different workload modes from the same usage pattern.

As discussed in the previous section, usage patterns are a sequence of workload modes. Workload modes need to be predicted in advance, so that optimal optimizations can be selected at design time. This in turn would also result in less identification overheads at run time. In order to do this, we exploit some design-time knowledge about workload modes in order to identify them at run time and to apply predefined optimal optimization profiles. This in turn results in a more efficient system in terms of both performance and energy. Later in this thesis, we present a method of workload mode identification that exploits some design-time knowledge and contrasts it with another method that is independent of design-time knowledge of workloads and uses HMMs.

Workload mode identification has to happen at both design time and run time. At design time, for each unique workload mode, we dedicate an optimization profile that stores the optimal system parameters such as frequency/voltage of each processor as well as where each task can be run. Other techniques such as Dynamic Voltage and Frequency Scaling (DVFS) [12], [13], [14] and processor power state management (on, off, or idle) can be used for reducing energy consumption. Further, different scheduling/mapping mechanisms can also be customized according to the identified workload mode.

At run time, once the workload mode is identified, using the same procedure at design time, one of the set of optimization profiles that have been already built at design time is chosen, depending on the identified workload mode. Intuitively, the optimization profile is not changed during execution unless a new workload mode arrives. More significantly, changing optimization profile incurs some overhead. Later we discuss that the amount of overhead depends on the sequence of these modes. Thus, there should be a tradeoff between the amount of incurred overhead and expected performance gains. Further, some workload modes may be important for chip optimization while others are not, and this difference is a direct consequence of the ability of the architecture to optimize for a workload mode. Later we conduct a cost—benefit analysis of our WCM and HMM models. One significant difference between our WCM and HMM is that our WCM is sup-optimal. In other words, it does not have to identify all workload modes, in contrast with HMM which has a very expensive learning phase that makes it capable of identifying most workload modes and thus is able to change optimization profiles all the time. We also discuss how identification and optimization overhead is crucial to chip optimization.

Since the concept of workload modes is similar to that of demands for automobiles in that they both arrive to the system as timed sets of heterogeneous content, we use the same automobile example described in Chapter 1 (Figure 4) to illustrate our thesis.

Table 1: Five Different Automobile Plant Configurations

Config.	**Model-T Assembly Line**		**Model-A Assembly Line**		
	Model-T	**Model-A**	**Model-A**	**Model-T**	**avg.**
C1	6	3	9	2	10
C2	4	8	13	1	13
C3	10	2	6	6	12
C4	6	3	6	3	9
C5	9	2	9	2	11

2.2 Automobile Example

The automobile plant, shown in Figure 4, consists of two assembly pipelines, one for the Model-T and one for the Model-A. Table 1 shows the production capacity of five different configurations for the same plant system during a day shift. In each configuration, we assume that the supply and human resources of its pipelines have changed relative to other configurations to produce more or less of the Model-T and Model-A products. These configurations range from fully homogeneous to fully heterogeneous designs to show how different plant configurations perform when different demands arrive at the system.

For illustration, consider configuration C1. Configuration C1 is a heterogeneous plant system, in which each pipeline is more specialized to produce a specific model type. The Model-T assembly line can produce six Model-T automobiles per hour, and can also be used to produce only three Model-A automobiles per hour. The Model-A assembly line can produce nine Model-A automobiles per hour, and can also be used to produce only two Model-T automobiles. Consider another configuration, C2. We use the same fixed plant resources to generate an alternative configuration. Here, the production of the Model-A assembly line has been improved by deploying more workers to produce 13 Model-A automobiles per hour, and can also be used to produce only one Model-T automobile per hour. Since the number of workers is fixed, the Model-T assembly line can now produce only four Model-T automobiles per hour, and can also be used to produce eight Model-A automobiles per hour. Configurations C4 and C5 are homogeneous, in which both pipelines produce the same amount of each model type. Note that configurations C4 and C5 use the same pipelines as configuration C1. Configurations C2 and C3 are heterogeneous configurations with different production capacities than that of the other configurations. Also note that we have chosen these configurations deliberately – the average number of models produced by each configuration is different. The average, *avg.*, is calculated using the arithmetic mean of the maximum production of each model type when produced individually. Later in this section we explore different types of mathematical means. Again, while this example is hypothetical, it is used for the purpose of illustration.

Since we are interested in the performance of different plant configurations that simultaneously process different rates of combinations of model types, we draw the response time of the five plant configurations for four production modes (Model-T, Model-A), shown in Figure 8. Shown on the dependent axis is response time, measured in hours, as the plant processes the demands that arrive from customer requests and configures itself for each different production mode. Each curve represents a different candidate configuration. Figure 8 shows curve fitting over each of the production modes. The curves differ not only by magnitude, but also in their shapes. These production variation curves show crossover points in which different configurations exhibit different performance as demands vary. If the curves did not cross, then there would be no significant difference in optimal performers due to production mode variation. But, because the curves do cross, the actual input to the plant system results in the identification of different configurations.

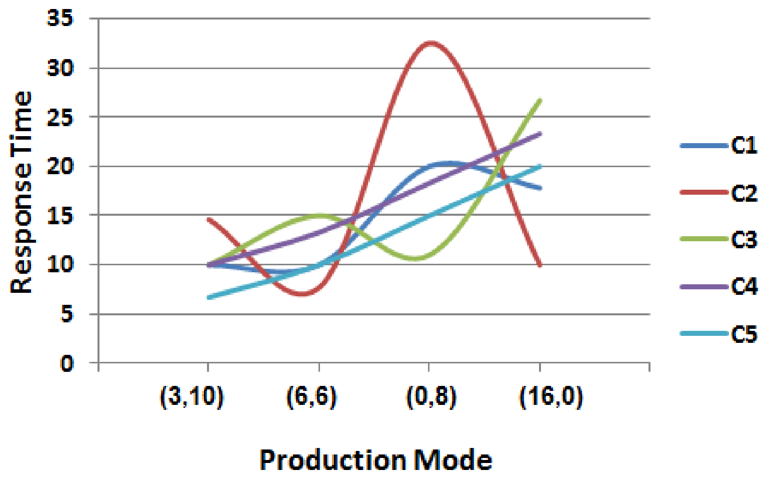

Figure 8: Performance of Different Configurations Processing Different Demands

Interestingly, homogeneous configurations (C4 and C5) behave linearly for the same set of demands in the order shown on Figure 8, while heterogeneous configurations (C1, C2, and C3) behave non-linearly as demand varies. For such a system, there is no clear optimal configuration because for different production modes of interest the rank of the optimal performers changes. As a result, composite performance metrics may not appropriately rank such systems. For example, since they do not include timing information and therefore do not cluster individual jobs, benchmark suites tend to aggregate the performance of individual jobs using the geometric mean. Computer architects opt to use the geometric mean to generate a composite score of performance because it balances performance, which is appropriate for general-purpose usage but not for systems that exhibit distinct situational usage.

Here, we further illustrate why traditional composite performance metrics fail to rank the performance of multiple heterogeneous resource systems that process multiple heterogeneous demand streams. Figure 9 illustrates how single-score performance metrics are generated. Figure 9(a) shows an automobile plant that at some point processes a demand of two heterogeneous product type streams, *T* and *A*, that arrive simultaneously. Note that this demand needs to be fulfilled in time window, τ. In order to evaluate the production throughput, *P*, of this plant during time window, τ, using a single score metric, an artificial procedure of two main steps is adopted. Figure 9(b) and Figure 9(c) show this procedure.

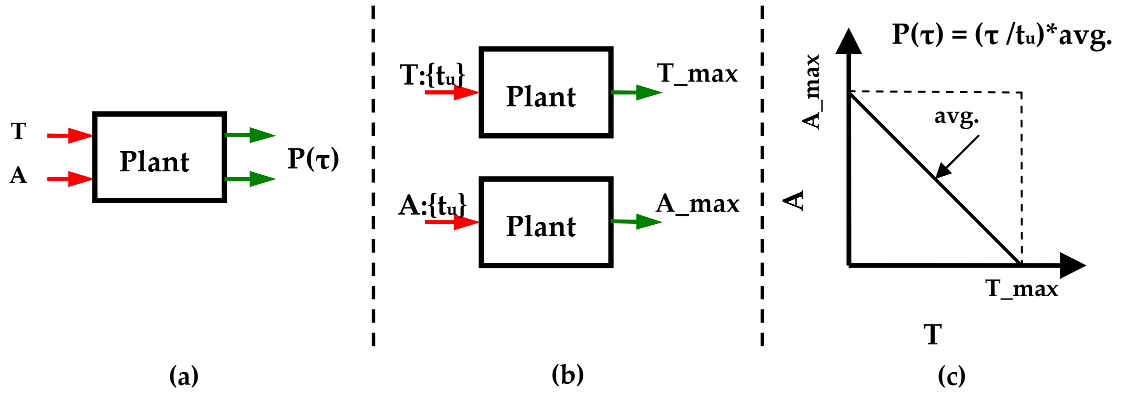

Figure 9: Distilling Performance into a Single Number

First, the two demand streams need to be separated, as shown in Figure 9(b), and evaluated independently over a relatively short time interval, t_u, such as a day, or even an hour. By selecting a short time interval over which production is being evaluated, demand is presumed to be non-zero and time-invariant over the entire time interval. Further, work is expressed using a common model type, such as automobiles, without distinction of the type of model. The output from this step is the maximum production of each demand type during the short time interval, t_u. Figure 9(c) shows the production throughput of this plant as a relationship between the maximum production of the two demand streams, *T_max* and *A_max*, expressed in terms of a common model type, automobiles.

Second, it is also assumed that the average area occupied by the two maxima points can be represented by a linear curve. To transform this linear relationship between the two maximum productions into a single, representative score of the plant performance functions, such as mathematical means, is needed. Note that since we use a common work unit, such as automobiles, we can average the throughput of different demand types.

Thus, computer architects use metrics such as Instruction per Cycle (IPC) or Instructions per Second (IPS) to evaluate throughput. Finally, since throughput is measured during a short time interval and that the model type is not taken into account, production throughput is presumed to be linear with time interval length. As a result, to compute the production throughput of this plant over the specified time window, τ, the average performance over the short time interval, t_u, is normalized to the length of the time window, τ, resulting in the same linear relationship.

Table 2: Optimal Configuration Ranking of Table 1

Config.	Production		Arithmetic		Harmonic		Geometric	
	P1	P2						
	(0,21)	(16,0)	AM	Rank	HM	Rank	GM	Rank
C1	17.8	20.0	18.9	3	18.8	4	18.8	4
C2	10.0	32.5	21.3	5	15.3	1	18.0	3
C3	26.7	11.0	18.8	2	15.6	2	17.1	1
C4	23.3	18.3	20.8	4	20.5	5	20.7	5
C5	20.0	15.0	17.5	1	17.1	3	17.3	2

The major mathematical means are the arithmetic mean (AM), geometric mean (GM), and harmonic mean (HM). The arithmetic mean has been used to summarize performance for time-based metrics, whereas the HM is used for rate-based metrics. The HM, when applied to a rate, is equivalent to calculating the total number of products divided by the total time. In contrast, the GM can be used for systems that balance the performance of different task types such as a general-purpose processor, but the behavior should first be normalized with respect to a specific configuration. Thus, standard benchmarks such as SPEC report performance values as speedup relative to a standard, reference machine, the SPECRatio. In this way, execution time is factored out of comparisons, or else applications that take longer would be given more weight in comparisons, but this time-based weighing is important to modeling how individual users perceive the response time of a system that processes workloads.

Table 2 shows the response time of the five configurations, shown on Table 1, when they process two production modes. These production modes were selected because they represent the maximum production of each model type when produced individually during a single day shift. Thus, some configurations would be able to produce these modes in a day shift while others cannot – they need more time. Table 2 also shows the rank of the average response time of each configuration, computed using three different averages: HM, GM, and AM. Note that using different mathematical means results in different ranks, as previously predicted. The HM ranks C2 as the best configuration because it is better suited to handle these demands, whereas the AM ranks C5 as the best configuration. The GM ranks C3 as optimal. When other design factors, such as scheduling in the case of heterogeneous resources, are considered, the rank is also expected to be different because the heterogeneity of assembly lines opens up the possibility for multiple scheduling options. Still, all configurations operate under a fixed area budget. Intuitively, this observation occurs because different loading situations differentiate configurations – the performance ranking of configurations changes from one loading situation to another, depending on the specificity of these configurations. Later in this section, we study the impact of scheduling on the rank of optimal performers.

Table 3: Optimal Configuration Ranking (More Production Modes)

Config.	**Production**						**Arithmetic**		**Harmonic**		**Geometric**	
	P1	**P2**	**P3**	**P4**	**P5**	**P6**						
	(0,8)	**(0,21)**	**(3,10)**	**(6,6)**	**(9,6)**	**(16,0)**	**AM**	**Rank**	**HM**	**Rank**	**GM**	**Rank**
C1	6.7	17.8	10	10	16.7	20.0	13.5	2	11.6	3	12.6	4
C2	3.8	10.0	7.7	14.6	20.0	32.5	14.8	4	9.3	1	11.9	2
C3	10.0	26.7	15.0	10.0	10.0	11.0	13.8	3	12.1	4	12.8	3
C4	10.0	23.3	13.3	10.0	16.7	18.3	15.3	5	13.9	5	14.6	5
C5	7.8	20.0	10.0	6.7	10.0	15.0	11.6	1	10.1	2	10.8	1

Interestingly, when more production modes are included, the ranking of the same mean changes. In Table 3, we included four more production modes. Note that the ranking of all means has changed relative to the ranking in Table 2. The problem of ranking different configurations in order to find the optimal performers has inspired us to come up with a new performance metric, Capacity.

Our Capacity metric views performance of the CHM that processes workload modes as a curve or surface with dimensionality related to the number of application types within a workload mode. In Figure 10, we apply this insight to the plant example. The Capacity curves, shown in Figure 10, show all feasible combinations of two automobile models produced concurrently during a day shift. Each curve represents a different candidate configuration. Shown on each axis is a product model. In order to increase the production of one model, production of the other model must be reduced. For example, production of automobiles of the Model-A type must be reduced to produce more Model-T automobiles. Thus, for a given increase in production of one model, Capacity shows how much of the other model(s) must be reduced. Production modes of interest can be overlaid on these curves. The shape and magnitude of a curve or a set of production modes on the curve (for multimodal curves) reveal some information about the production system. The analysis of these curves for the purpose of ranking CHMs that process workload modes is a major contribution of this thesis.

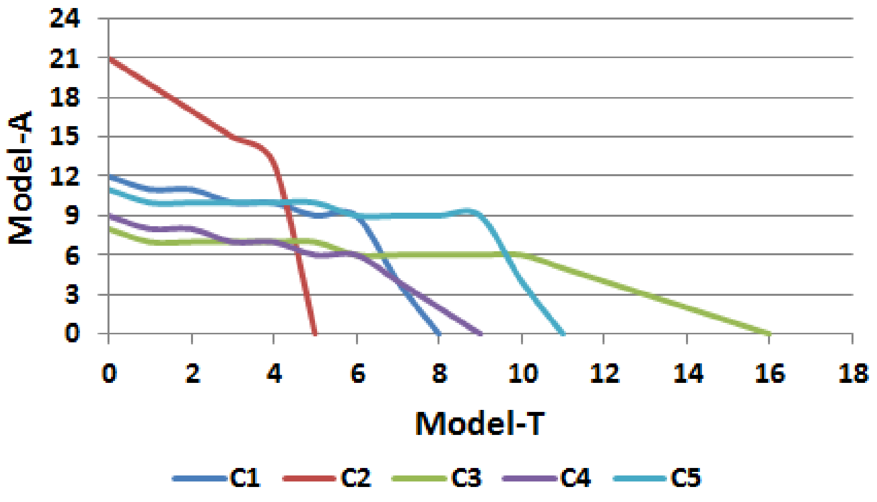

Figure 10: The Production Capacity Curves of the Five Configurations

Next consider another example in which we point out to another problem of using throughput. Table 4 shows three made-up configurations that consist of Model-T and Model-A assembly lines. Note that the production throughput of all configurations is the same when measured using the arithmetic mean of the maximum production of each model type individually. Configuration C1 is a homogeneous configuration in which both assembly lines, Model-T and Model-A, can produce eight Model-A automobiles during a day shift, and can also be used to produce only six Model-T automobiles. Configuration

C2 and C3 are heterogeneous, but they differ in the structure of the internal assembly lines. Each assembly line in C2 is analogous to a general-purpose processor in that it produces almost the same amount of products of each model type in unit time. In configuration C3, each assembly line is specified for specific model type.

Table 4: Three Different Plant Configurations

Config.	Model-T Assembly Line		Model-A Assembly Line		
	Model-T	Model-A	Model-A	Model-T	avg.
C1	6	8	8	6	14
C2	10	12	4	2	14
C3	10	4	12	2	14

Figure 11 shows the Capacity curve for the three configurations, discussed in Table 4. The Capacity curves show all feasible combinations of Model-A and Model-T models produced concurrently during a day shift; this is something that throughput cannot show. Each curve represents a different candidate configuration. Shown on each axis is a product model. Note that all Capacity curves have the same maximum points when considering the production of one model type at a time, which is the definition of throughput. Also note how the curves are not straight line and the shape varies as different rates and combinations of models arrive at the plant.

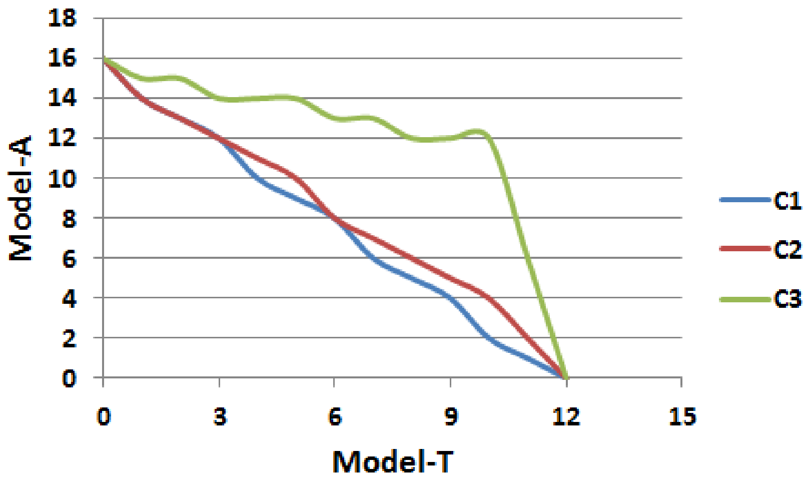

Figure 11: The Production Capacity Curves When Mapping to The Best Resource

Figure 12 plots the response time of each configuration in response to producing four different demands. Note that response times do not cross, thus there is no problem in ranking optimal performers using an average.

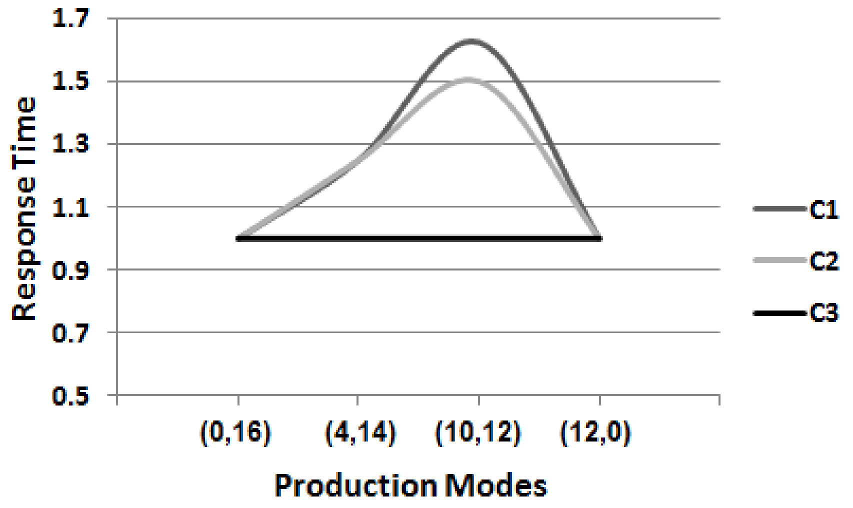

Figure 12: Performance of Different Configurations Processing Different Demands

Table 5 shows the performance ranking of the three configurations processing different production modes. Note that the ranking is the same, without distinction of the mathematical mean used, since there is no crossover in performance curves.

Table 5: Optimal Configuration Ranking of Table 4

Config.	Production				Arithmetic		Harmonic		Geometric	
	P1	P2	P3	P4						
	(0,16)	(4,14)	(10,12)	(12,0)	AM	Rank	HM	Rank	GM	Rank
C1	1.00	1.25	1.63	1.00	1.22	3	1.17	3	1.19	3
C2	1.00	1.25	1.50	1.00	1.19	2	1.15	2	1.17	2
C3	1.00	1.00	1.00	1.00	1.00	1	1.00	1	1.00	1

Since configurations C2 and C3 are heterogeneous, we study the impact of scheduling demands to resources other than the one that provides the best performance for the task type on the Capacity. Note that the first configuration C1 has no other scheduling options since it is a homogeneous configuration. Figure 13 shows the Capacity curves when mapping demands to a resource other than the one that provides the best performance for the model type. Interestingly, while the maximum production is the same as in Figure 11, intermediate production modes, especially for C3, are totally different. This is another problem of throughput. It does not show the impact of scheduling on different combinations of inputs because it evaluates them individually. Note that throughput assumes a linear relationship between the two maximum productions, similar to C1 and C2 curves which are not fully heterogeneous.

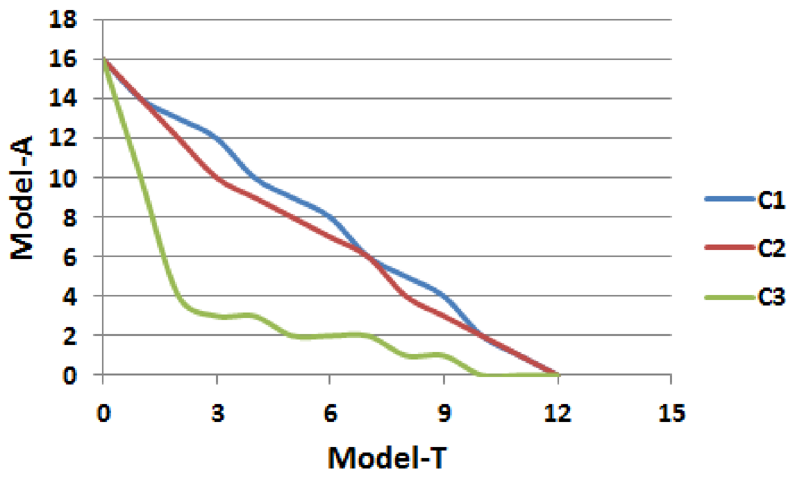

Figure 13: The Production Capacity When Mapping Randomly

Again, while all mathematical means result in the same ranking of the optimal configurations, the ranking has changed due to the change in scheduling. Now configuration C3 has become the worst among the three configurations, whereas previously it was the optimal configuration, as shown in Table 5.

In this chapter we illustrated the need for a new performance metric to evaluate the performance of the CHMs that simultaneously process heterogeneous outputs or workload modes. CHM evaluation for workload modes, in contrast with single applications or aggregate models of workloads, requires a new metric in order to find optimal designs for specific usage patterns, or WSPs. We showed that distillation of multi-type production over a given heterogeneous multi-pipeline plant (design) to a common work unit, an average, includes a two-fold problem. First, it precludes the designer from identifying cases in which the system has more Capacity than the average or cannot meet the predicted average production – where the non-linear curves rise above or fall below the straight line (average) that occurs when a common work unit for throughput is used. The other problem is that different mathematical means result in different rankings due to the fact that performance curves, that relate response time or throughput to a change in demands, do cross. The crossover points result due to the fact that different CHM designs are specialized for different demand types.

In the next chapters, we include both parts of the work, designing and evaluating CHMs as WSPs using traditional metrics such as latency and throughput and using our suggested metric, Capacity. Our overall goal is to show how processor resources affect the end user of mobile computing devices and, more specifically, point to where resources are wasted when resources targeted to old and incorrect models of performance. Next, we develop the foundation for our observation that CHM designs need to be evaluated for workload modes.

Chapter 3

Foundations

Research is creating new knowledge.
Neil Armstrong

In this chapter we develop three foundations that we used to develop our Capacity metric – a CHM model, a multimedia cell phone example, and a WSP. Each of these foundations has analogies to models that are used in single core design, but which are missing in multicore computer design. Part of our thesis is that WSPs more correctly capture what mobile computing devices do and that they result in categorically different approaches to processor design, something we initially sought to develop in more detail but which has become a secondary contribution to this thesis when that work resulted in the discovery that single valued metrics failed to properly rank optimal performing designs. WSPs are enabled by the presence of collections of individual user initiated job-style applications that arise as users interact with multicore, mobile devices. Evaluation of WSPs is what led us to develop the Capacity metric. Existing models for single-core computers, (i.e. those relying on an Register Transfer Level (RTL) and accompanying benchmarks) cannot be used to develop and evaluate our WSP model. RTL key design features do not capture those of CHM designs. And, individual benchmarks do not accurately model the real workload of single-user multicore, mobile devices. Thus, we needed to develop new examples that permit benchmarking of single-user mobile computing devices to replace traditional benchmarks, an equivalent model to RTL for the types of architectures that are projected to be in those devices, that is a CHM model, and finally a WSP model that is to CHMs is what an ISA is to an RTL.

3.1 Single-Core Computer Design

Existing models for single core computers such as Register Transfer Level (RTL) have been long dominant in computer systems design, however, they are inappropriate to model multicore systems for a three-fold reason: (1) the key design features of RTL design do not capture those of CHMs; (2) traditional benchmarks do not model the real workloads of single-user multicore, mobile devices; and (3) since CHMs include multiple-ISA cores, there is no single ISA at the H/S boundary of CHMs. Thus, Heterogeneous Core Level (HCL) design has become compelling, as the trend for computer systems is to integrate heterogeneous multicore designs. HCL design requires the definition of equivalent models to that of lower level designs, in addition to new hardware design languages and simulation tools [124].

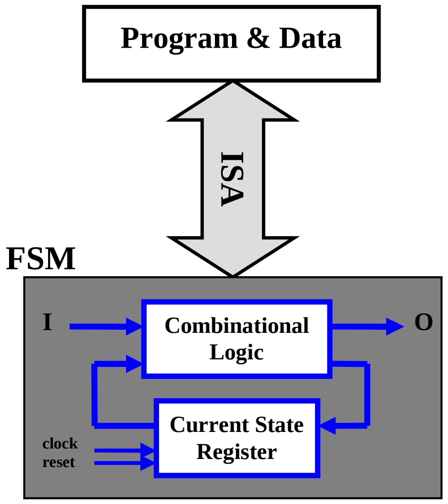

Figure 14: A Single Processing Element (PE)

As an analogy, consider the way a Processing Element (PE) is designed at the RTL. Figure 14 shows a single PE as a relationship between a Finite State Machine (FSM), program and data streams, and an Instruction Set Architecture (ISA). A conventional processor is an FSM that mainly consists of a state register and a combinational logic. The primary input to the FSM is a program stream and a data stream. In the combinational logic, the value of the state register is combined with the incoming data for computation. A state register may store data, an instruction pointer or the program counter. In response to the incoming data, the FSM can load the program counter. This allows dynamic change of execution sequence that makes FSMs programmatic designs, but still their execution is sequential. Computation in a FSM is triggered at the clock edges. In a synchronous FSM, all of the processing is defined to occur during the clock period. Thus, the processing time of the combinational logic can be determined a *priori*. In summary, RTL design consists of three key features:

- Computation that is done in combinational logic blocks;
- Communication that is done on simple wires; and
- State that is held in registers.

Further, two key features characterize the benchmarks of an RTL design:

- Individualized: benchmarks arrive to the system individually; and
- Untimed: benchmarks and their associated datasets are sequenced by the processor speed thus they do not include an external time reference.

An ISA specifies how the FSM layer is accessed by a software program. It includes a set of control, communication, and computation instructions that the underlying processor understands and supports. Thus, ISAs are developed after the processor model is designed, regardless what software is going to execute on this processor model. An ISA is accessible by the developer or the compiler, representing a contract between an FSM and the developer (or compiler).

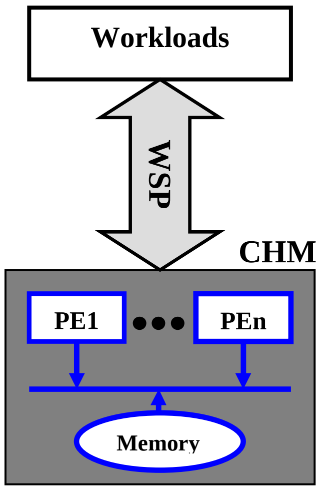

Figure 15: Our View of the Design at the HCL

So far we have shown that RTL design includes three foundational models – FSMs, benchmarks, and ISAs. Such models are missing at the HCL. Figure 15 depicts our view of the design at the HCL. In the remainder of this chapter, we develop these three foundations for HCL design – a CHM model, a multimedia cell phone example, and a WSP.

3.2 A CHM Model

Since key features of RTL design are inappropriate for HCL design, we first establish a baseline model of a CHM like that of an RTL for an ISA. Our model includes the key elements of computer design. Figure 16 shows our suggested CHM model. It shows a central control processor (CP) that coordinates all other resource processors, from R1,…, Rn. We establish a CHM that has some central control [15] so that the chip may respond to persistent changes in loading. Each resource processor has its own contribution to the overall Global Chip State (GCS), denoted as RSi. Consider GCS as logically global runtime state which provides information guiding Chip-Level Scheduling (CLS) decisions, and so the time it takes to gather GCS must be factored into the effectiveness of a chip-level scheduler. Examples of GCS are numbers of idle tasks in the system, the execution progress of running tasks, the availability of resources, and the current load on the memory and bus, etc. The state of each resource processor (RSi) must be transferred to the CP if it is to be used in the decision making of the CP in a given situation. Examples of RSi are the frequency and voltage level of the processor as well as the progress of running task(s), if any. Because of the lower communications costs within the resource processor relative to the communications costs within the whole chip, and also in order to reduce loading on the interconnection network, RSi (collectively the GCS) is kept local to the resource processor until global scheduling is triggered. The CP gathers the GCS and sends scheduling decisions to computational resources, which are the processing elements used to execute the assigned application functionality. Since the central controller coordinates all other system resources, this is the easiest control flow to realize.

Note that all application tasks are eligible to be placed at any of the processing resources. The main advantage of coordinating the system state in this way is the preservation of a sequential control flow scheme for programming the chip; this is largely a programmatic solution in which distinct blocks of software may be accelerated by execution on custom hardware or on many blocks in parallel, and often both. Many current CHMs fall into this category of design, in which a general-purpose processor serves as the central controller. The main disadvantage of this approach is that the central controller can be a bottleneck. The system can only execute as fast as the sequential fractions of software can execute on the CP. In general, the software executing on the central controller must be as simple as possible or the central controller must have sophisticated scheduling that tunes the application to the slave processors so that throughput can be maintained.

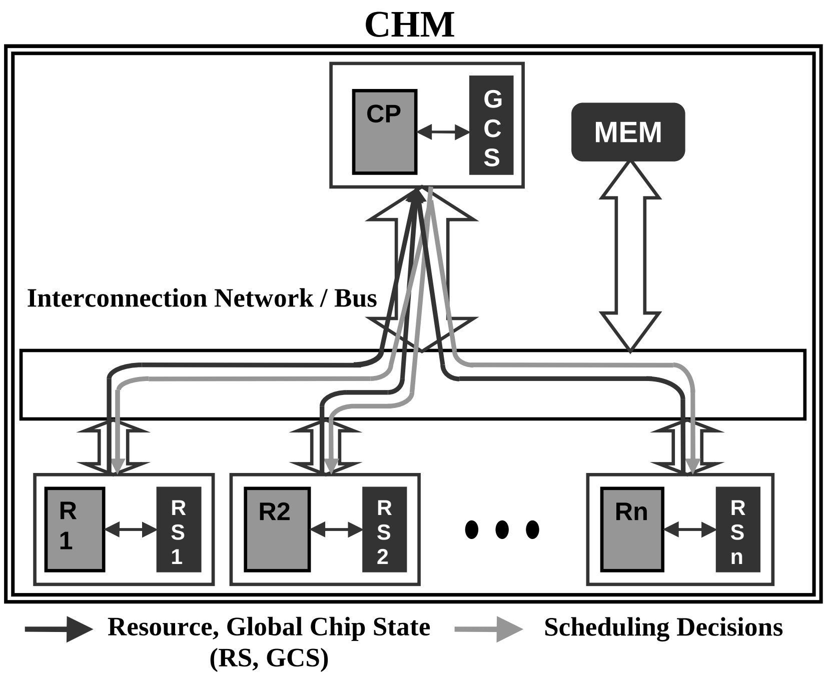
CHM
CP
GCS
MEM
Interconnection Network / Bus
R1
RS1
R2
RS2
Rn
RSn
Resource, Global Chip State (RS, GCS)
Scheduling Decisions

Figure 16: A CHM Architecture

Distributed control flow is also possible. In the case where the CHM operates on several distinct groups of applications, in which each group uses multiple programmable resources, it might be more advantageous to specify several controllers in the system, one per group of applications. This approach lowers the amount of system state passed to each controller since every controller receives only the state relevant to its decisions (i.e. the state relating to the application group the controller is managing). A fully distributed approach in which each processing element makes a decision about what to run next, can be useful in systems in which state sharing overhead greatly dwarfs the performance gains of making a global task mapping choice.

We view optimization for single-user workload modes as the basis of global, chip level optimization for CHMs. The optimization of the chip can be thought of as the need to optimize globally, and hence we model this using a global central controller. However, we model a state that leads to global decision making the GCS, as logically distributed among local processors, and the actual global scheduling could be distributed as well. In either case there will be overhead that synchronizes distributed parts of the global scheduling. We include that overhead in our model. Our model focuses on the simplest form of control, the centralized controller with the distributed state shown in Figure 15. We will use this architectural model later in the experimentation. Note that our model is conceptual; the actual implementation of our constructs may vary considerably in the actual architectures, but these variations will only affect physical parameters, such as the overhead costs.

Here, we summarize the key features of our CHM model:

- Computation: processors (R1,…,Rn) and the central processor (CP), having different ISAs, in contrast to FSMs in which computation is done in combinational logic blocks.
- Communication: an interconnection network or bus with different properties such as communication bandwidth and burst width, in contrast with FSMs in which communication is done on wires.
- State: global state (GCS) and local states (RSi) kept in memories, in contrast with FSMs in which state is held in regiesters.
- Triggering method: even-driven, time-triggered, or a mix of both, in contrast with FSMs in which triggering is done on the clock cycle.

The architecture of our CHM model is based upon a combination of a survey of existing devices and a projection of where we believe they are evolving. Computer systems are increasingly becoming multicore. When multiprocessing happens within a single die/processor, we call that a multicore architecture. Although the focus of our survey will be on CHMs, we will also investigate existing homogenous multicore processors, since it is likely that these types of processors will soon be extended to include heterogeneous cores with different ISAs. Below is a brief overview of existing homogenous and heterogeneous multicore architectures.

Most multicore designs for desktops, laptops, and servers have homogenous cores that all implement the same ISA. Examples of homogenous multicore processors include IBM’s POWER6 dual-core multithreaded processors [16], [17], AMD’s quad-core Opteron processors [18], Intel’s dual-core and quad-core Xeon processors [19], [20], Compaq’s Piranha 8-core research prototype processor, and Sun Microsystems’ T1 and T2 8-core multithreaded processors [21], [22]. An overview of early homogenous multicore designs is presented in [23]. Multicore designs with tens of processors, often referred to as many-core designs, have recently been introduced. Examples of many-core designs include Tilera’s Tile64 Processor for multimedia processing [24], AMD’s Radeon and Firestream processors for graphics processing [25], [26], NVIDIA’s GeForce and Tesla processors for graphics processing [27], [28], [29], and Intel’s Larrabee processor for visual computing [30]. Many-core processors often include programmable processors, along with on-chip interconnect, bus controllers, memory controllers, and cache. Although homogenous multicore and many-core designs offer some advantages over CHMs in terms of design reuse and programmability, they often cannot match the area, performance, and power benefits of CHMs [31]. Because of their heterogeneity feature, CHM processors can match each application to the core best suited to meet its performance demands, resulting in performance gains. CHMs also, compared with homogeneous multicores, would use a larger number of small low-power (and low-frequency) cores, thus it can be more area-efficient and significantly reduce processor power dissipation, especially when dynamic power management is adopted.

Because of their area, performance and power advantages, several designs for CHM processors have been developed. AMD’s Accelerated Computing Initiative [32] and Intel’s Tera-scale Computing Research Program [33], [34] acknowledge the benefits of CHM designs, but have not yet developed them. Most of these designs are targeted toward specific application domains, such as multimedia processing, graphics processing, and wireless communications. CHM processors include:

- Sandbridge Technologies SB3000 processor [35], which has four Sandblaster DSPs, an ARM9 application processor, on-chip memory, a DMA controller, and numerous peripherals for implementing a variety of handheld device functions.

- Texas Instruments DaVinci Digital Media System-on-Chip [36], which has a C64x DSP, an ARM9 processor, multimedia accelerators and co-processors, and numerous peripherals.
- Texas Instruments OMAP3440 Processor [37], which has an ARM Cortex™-A8 processor, Imagination Technologies' POWERVR SGX™ graphics core, an image signal processor, an image video audio accelerator, and several peripherals.
- Infineon's MuSIC Processor [38], which has an ARM processor, four SIMD DSPs, on-chip shared memory, a multi-layer bus interface, accelerators for wireless communications, and various peripherals.
- STI's Cell Processor [15], which has a Power Processing Element (PPE) with vector extensions, eight Synergistic Processing Elements (SPEs), an element interconnect bus, and on-chip memory and bus-interface controllers.

Each processor in the Sandbridge SB3000 has its own private L1 and L2 cache memories, making processors loosely coupled. SB3000 also includes a central processor, which is the ARM9 application processor that facilitates programming the other chip elements. The central processors usually are general-purpose processors and of different types than the other processing elements on chip. The application processor has a specific task, which is to control the functions of the other processing elements. Other cores can run any task since they are homogeneous. Thus, this mapping model is semi-static. The TI DaVinci is similar to the SB3000 in that its processors are loosely coupled, but it has no central processor. Each core processor in the DaVinci is specialized to do a specific job. In contrast, OMAP3440, MuSIC, and Cell processors share L2-cache memory, so their processor cores are tightly coupled. MuSIC and Cell processors are centralized architectures since they include a central processor that facilitates programming the other chip elements, while OMAP3440 is a distributed system in which there is no central controller. While the Cell and MuSIC processors have a central controller, this controller can be also used to execute other tasks that can also be executed in the other processing elements, thus its mapping model is partially dynamic. OMAP3440 is a dynmaic system since any task can run on any processor. In these systems, processor cores are connected via different bus technologies.

Table 6: CHM Processors Classification

System	*Processor Coupling*	*Programming*	*Mapping*	*Communication*	**Class**
SB3000	Loosely	Centralized	Semi Static	AMBA Bus	A
DaVinci	Loosely	Distributed	Static	Multilayer Bus	B
OMAP3440	Tightly	Distributed	Dynamic	System Bus	C
Cell	Tightly	Centralized	Semi Dynamic	Ring Bus	D
MuSIC	Tightly	Centralized	Semi Dynamic	Multilayer Bus	D
Trend	**Tightly**	**Centralized**	**Fully Dynamic**	**NoC**	**–**

Based upon the above discussion, we extracted four key features from these systems, processor coupling, programming model, processor-application mapping type, and processor interconnection technology. We classify the CHM systems in the above-mentioned list based upon these features. Table 6 summarizes these classes according to the selected features. Table 6 also includes the chip makers trend in designing CHMs. Note that multicore systems are increasingly becoming centralized to facilitate global programming models and thus to extremely exploit the parallelism feature of multicore designs. Therefore, some sort of global state is required to be saved in a shared memory. This global state makes processors tightly coupled. Dynamic mapping or scheduling has proven significant performance gains over static scheduling by allowing any task to execute on any processor. Future systems are expected to integrate high-speed networks to connect chip elements. As increasing transistor densities and larger die sizes enable increased integration, we envision future CHM processor systems with tens to hundreds of heterogeneous cores including general-purpose processors, digital signal processors, graphics processors, and/or multimedia processors, along with programmable hardware accelerators, fixed-function hardware, on-chip memory, high-speed interconnect networks, and a wide variety of peripherals. We used this trend to build our baseline CHM model, which we believe will be prevalent in future.

3.3 Multimedia Cell Phone Example

Since our WSPs are enabled by the presence of workloads (instead of individual benchmarks) that result from single users, we needed to develop new examples that permit benchmarking of CHMs. In contrast to existing benchmark suites, our examples include the arrival time of applications as well as some level of concurrency. We first develop our example – that is a multimedia cell phone example – and then we contrast it with existing benchmark suites.

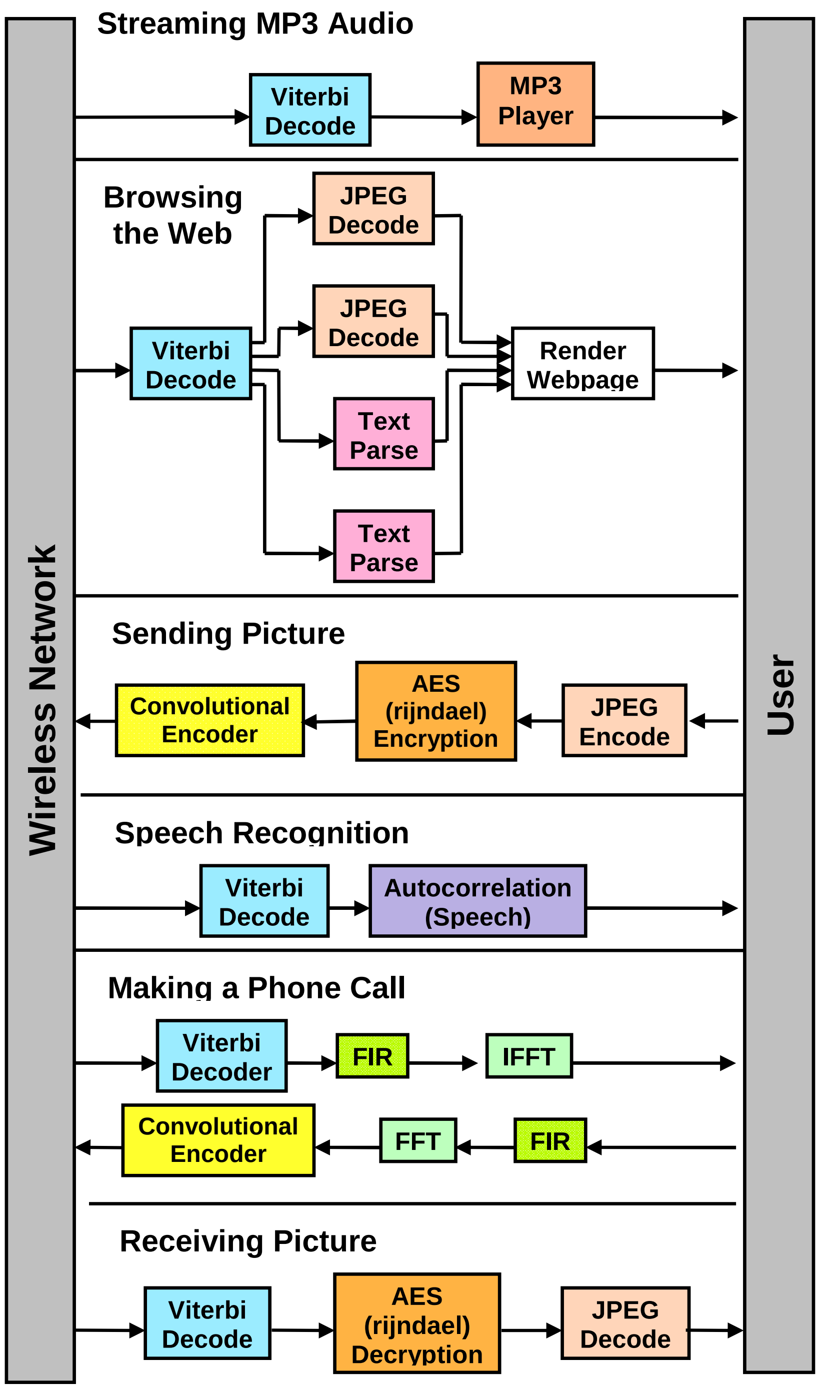

Figure 17: Six Cell Phone Applications and their Components

Modern mobile phones, or smart phones, are becoming increasingly ubiquitous. At the same time, these devices increasingly are including software features such as those in laptops and desktops. The cell phone example has been developed as the basis of a benchmark suite for CHM systems, and is first introduced by Paul et al. [1]. The cell phone is capable of browsing the Web, and has enhanced Human Computer Interaction features typical of many future – and even current – devices. Together, these make the resultant system a hybrid of an embedded system and a personal computing device.

We extended the cell phone example of Paul et al. [1] to include more functions such as speech recognition. Moreover, we further developed the implementation of some applications to preceisly capture their behaviors when executed on CHMs. Consider several applications that can run on a cell phone. The applications within the cell phone are able to exhibit a wide variety of behaviors, placing differing needs on the hardware. We define six possible cell phone applications: (1) surfing the Web, (2) making a phone call, (3) sending a picture, (4) receiving a picture, (5) speech recognition, and (6) streaming mp3, as shown in Figure 17.

Table 7: Our Benchmark Classification

Benchmark	*Type (Y)*	*Complexity (C)*	*Data Dependency (S)*
FIR/IFFR	Streaming	Low	Size
FFT/IFFT	Streaming	Low	Size
Viterbi	Streaming or Job-based	Low	Size
AES	Streaming or Job-based	Low	Size
MP3	Streaming	High	Size
Convenc	Job-based	Low	Size
Autocor	Job-based	Low	Content
JPEG	Job-based	Medium	Content
Text	Job-based	Low	Size
FLASH	Streaming or Job-based	High	Content

Each application consists of several tasks, shown in Table 7, extracted from the EEMBC benchmark [39]. For instance, “surfing the Web” takes on an arbitrary number of JPEG, text, and Macromedia Flash jobs that execute concurrently, but the system must first perform a Viterbi decode. The twelve different tasks, in Table 7, are grouped into classes. First, we distinguish between streaming and job-based application behavior. Second, we look at the application data dependency as it relates to run time. An important distinction between streaming and job-based categories is that job-based tasks can always use more computational power in order to reduce latency, in which the streaming tasks have no use for extra computational power once deadlines are met. An example of a streaming task is making a phone call. An example of a job-based task is taking a picture. Note that some tasks belong to two task classes depending upon the contexts in which they are used. For example, Viterbi is a streaming task when used to stream data as in a phone call. However, it is a job-based task when used in sending a picture. On an orthogonal axis, we consider data dependency. A data content dependent task exhibits a different run time when faced with input data of different content, but of the same size. An example of a data content dependent task is MPEG video compression, in which run time is dependent on the content of the picture that is encoded. In contrast, data size dependent task run times vary with the amount of data applied to the input. Many encryption algorithms' run times are not dependent on the content of the file to be encrypted, but rather only on the file size.

Now, we perform a survey of existing benchmark suites and show why they fail to optimize CHMs, in contrast with our example. The SPEC CPU [40] suite includes scientific and engineering application benchmarks that are used mainly to characterize workloads of general-purpose computing [41], [42], [43]. SPEC compares the performance of a system (in term of both speedup and rate) to a reference machine – this normalization is eventually used to average the performance of different applications using the geometric mean. Two problems exist in SPEC benchmarks. First, SPEC is not geared toward embedded computing and multimedia applications. The other problem is that SPEC is not able to measure and model power consumption.

In contrast with SPEC, MediaBench [43] benchmark geared toward multimedia and communications systems [43], [44]. The problem of MediaBench is two-fold. First, it focuses on batch execution systems where individual programs are evaluated. The other problem is that MediaBench focuses only on multimedia applications and not embedded systems, in general.

In contrast with both SPEC and MediaBench, EEMBC and MiBench benchmarks are geared toward embedded systems [42]. There are differences between MiBench and EEMBC benchmarks. MiBench is a free benchmark, in contrast with EEMBC which requires a paid membership to have access to the testing suites [42]. Similar to the other benchmark suites, these two benchmark suites focus on batch execution systems [45].

In summary, benchmarks are executed one a time, which is a major problem. This implies that there is no external time reference for the arrival of these programs.. Another problem is that the data associated with these programs are also untimed. Modern embedded systems applications can often be described as workloads consisting of numerous I/O streams, and parallel applications that enter and leave the system at different times and in a bursting fashion, rather than in terms of a system with constant loading or programs executing in isolation. Table 8 summarizes existing benchmarks based on key features. Also, it includes our definition of workloads that arise from the interaction between individual users and modern mobile devices. The definition of workloads motivates us to develop a new performance metric, Capacity.

Table 8: Existing Benchmarks Summary

Benchmark	*Concurrency*	*Timed datasets*	*Timed applications*	*Target*
SPEC	Multiple	Untimed	Untimed	General purpose
MediaBench	Single	Untimed	Untimed	Multimedia
MiBench	Single	Untimed	Untimed	Traditional Embedded
EEMBC	Single	Untimed	Untimed	Traditional Embedded
Our Example	**Multiple**	**Timed**	**Timed**	**Modern Embedded**

Several workload models have been proposed for use in multiprocessor systems, and are discussed thoroughly in section 4.3. Aside from benchmarks, researches have implemented numerical, statistical, and stochastic models on measured data to come up with accurate workload models. More interestingly, user behavior graphs have been developed to capture user interactions at higher levels. However, not all of these models are tractable for single-user multicore computers. Individual benchmarks do not represent real workloads. Although user behavior graphs seem to be a good choice for emergent workloads, they do not model the arrival time of commands. Trace driven models incur overhead in terms of instrumentation setup and the amount of collected data. Stochastic models are not scalable while the number of workload modes and processors increases. Statistical and numerical models are static models, in contrast to the behavior of users, and may yield misleading results when using the wrong model.

We will use our multimedia cell phone example to artificially form different usage patterns, discussed in Chapter 2, which can be used later in this thesis as benchmarks to evaluate CHM designs.

3.4 Workload Specific Processors (WSPs)

There is no ISA for CHMs that bridges CHMs to the way the system is used and will be programmed. We suggested WSPs to do that. Thus, WSPs are to CHMs what an ISA is to an RTL. We first perform a survey of existing computing categories based on their performance evaluation models, showing how our WSPs result in categorically different approaches to processor design.

Computing can broadly be classified into its dominant research communities, characterized by traditional Computer Architecture and Computer Aided Design. Computer Architecture has focused on general purpose computing of the kind found in desktops and laptops, while Computer Aided Design has focused on Application Specific Integrated Circuits (ASICs) and Embedded Computing. Each community has offshoots, but the broad goals of these communities can be classified. In Computer Architecture, optimization of the design is to a pre-fixed and agreed upon set of benchmarks, and in Computer Aided Design, each design is highly unique and customizable. We describe the assumptions inherent to each of these categories below, in order to illustrate how WSPs differ. The WSP model of performance evaluation is later discussed in Figure 18. We develop our WSP model by contrasting it to general purpose processing (GPP) and application specific processing (ASP) models. Our definitions of GPP and ASP models show the foundations of existing performance evaluation techniques the way they are designed, not the way they are used.

Figure 18 illustrates the differences between the three performance optimization models; the GPP, the ASP and the WSP. Figure 18 shows that the inputs are conceptually presented to the GPP sequentially so that applications do not overlap and thus there is no contention between them. The processor (or the compiler) generates the sequence of execution by identifying interdependences between individual program inputs. Figure 18 also shows that each application starts execution after the previous application completed execution, in which a downward arrow indicates the beginning of an application and an upward arrow indicates the end of an application. Each input represents a different type of application. Each application type takes a different execution time, *E(App$_i$)*, to complete. Evaluating the performance of all application types produces a set of performance values that can be normalized to a reference architecture as speedup, and then averaged to produce P_i. This model is appropriate for batch execution, but does not accurately describe the performance of interactive workloads [46]. Even in the case of multithreaded GPPs, the input to the system is still presented as a single demand stream. That also leads to the use of average performance values. In the uniprocessor world, SPEC [40] is a popular set of benchmark applications; SPLASH-2 [47] is the analog in the multiprocessor world.

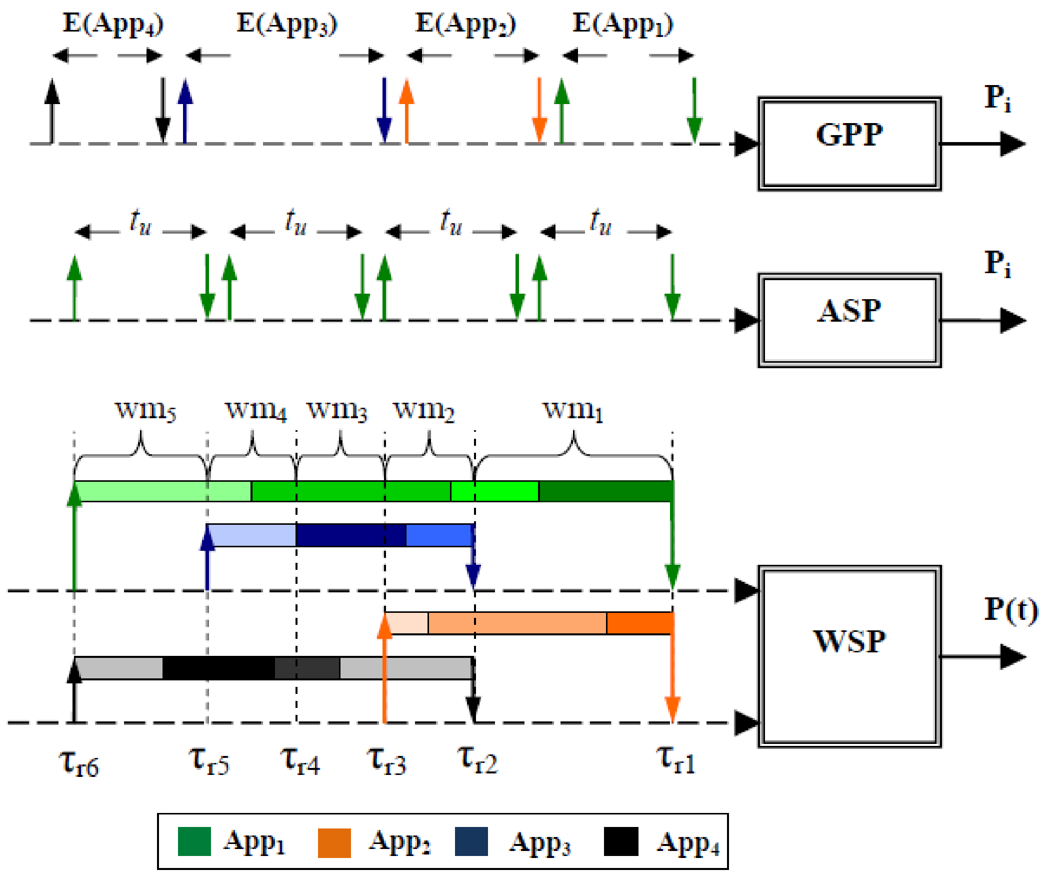

Figure 18: Application Arrival Timing of Different Performance Optimization Models

Next, we consider real-time embedded systems that use Application Specific Processors (ASPs). Figure 18 illustrates that the input is an application or a set of applications that persist in the system over time and that are considered to arrive periodically. Thus, real-time embedded systems are designed given upfront specifications of the applications it will carry out. These systems are designed to meet real-time demands within a period, t_u, and the execution time, P_i, of the application or the set of applications has to be less than the deadline, t_u, usually leaving some slack time. Increasing the computation power of these systems is wasteful when the deadlines are met unless the designer wishes to add more functionality to the existing system functionalities [46]. Thus, today's cell phones do not fall into the traditional real-time embedded systems category. Our new evaluation model (WSPs) addresses the way processors are used in this design.

Figure 18 also illustrates our WSP model. At any given time there is a set of applications. These applications are presented to the system in response to a mixture of timed, external and untimed, internal events. The external events result from user-computer interaction and data arrival. The internal events arise from changes in data that are being processed by existing applications. Thus, the arrival of new application(s) or data, a significant change in the data being processed by existing application(s), or the departure of an existing application(s) defines a new mode of operation. For instance, App_1 and App_2 arrived at the system at time τ_{r1}, resulting in workload mode wm_1. Later on, at time τ_{r2}, App_3 and App_4 arrived while App_1 and App_2 were still executing. This results in a different workload mode, wm_2. Thus, wm_2 at time τ_{r2} now has four applications: App_1, App_2, App_3, and App_4. In this example, there are four workload modes generated due to application arrival and departure. wm_4 was generated due to a significant change in the data being processed by App_1.

In Figure 18, the change in the input data is denoted by using different color contrasts to show different phases of execution. For example, the processing of an MPEG file that has been previously downloaded can result in different processing requirements that can persist in the system long enough to result in a significant program phase. Thus, while the MPEG application persists in the system for a long period of time, optimization of the whole system requires consideration of the data being processed. Also, webpages have different time granularities, during which the content updates [46], [48]. These updates may result in on-chip optimizations. Kumar, Tullsen and Jouppi divided individual program execution into phases that exhibit different behaviors due to the change in the input data during execution [49].

Figure 18 also shows that time is continuous and that some jobs may persist in the system indefinitely while others may not complete before the next input arrives. For instance, App_1 arrived at τ_{r1} and departed at τ_{r5}. Meanwhile, applications App_3 and App_4 arrived and finished execution. Again, the downward arrow indicates the beginning of an application and an upward arrow indicates the end of an application.

Paul et al. established the foundation for the observation that emerging computers have a categorically different relationship with users by the development of a new taxonomy that is called the U-A (User-Application) Taxonomy [46]. The U-A Taxonomy extends the most widely known taxonomy in computing, that is Flynn’s Taxonomy [50], which categorizes computer architecture into Single Instruction, Single Datastream (SISD), Single Instruction, Multiple Datastream (SIMD), Multiple Instruction, Single Datastream (MISD), and Multiple Instruction, Multiple Datastream (MIMD). The focus of Flynn’s Taxonomy is on the structure of the computer and not the objectives of the computer. Thus, an additional classification scheme for computing that considers the objective of the computer seems necessary. The U-A Taxonomy is proposed as a way of distinguishing the objective of the computer from its structure.

The most common computing structures from Flynn’s Taxonomy that are realized are SISD and MIMD. In the U-A Taxonomy, the authors define four new classes of computing and further subdivide them according to their realization as SISD or MIMD computers, resulting in eight classes of computers. A Single-User (SU) computer is designed with the presumption that only one person will be using the computer at any point in time. In contrast, a Multiple User (MU) computer is designed to satisfy the needs of multiple users who share the computer services at the same time. A Single Application (SA) computer is designed to execute a single application at a time. When the single application is complete, the computer moves on to the next application. In contrast, a Multiple Application (MA) computer is designed to execute multiple applications at the same time; multiple applications are considered to be executing in a MA system concurrently, even though the concurrency can be achieved in a variety of ways.

Our WSP model falls under the SUMA-MIMD classification. The SUMA-MIMD computer has the objective to execute multiple applications concurrently on a multiprocessor in the service of a single user. Examples of SUMA-MIMD machines include personal computers and laptops with multiple cores. Because personal computing devices are increasingly designed to meet portability demands (on size and power consumption), SUMA-MIMD computers are starting to be realized as heterogeneous multiprocessors. SUMA-MIMD computers represent a class of computer for which performance is not always dominated by latency and throughput over the application set. The most obvious way to see this is that the user's perceptions are inherently limited, so that more computing power does not always impact the user's ability to perceive it. For example, when humans juggle sets of applications in which data is arriving in real time, they can only pay attention to so much information in a fixed amount of time. As a result, faster processing of some tasks is wasted on the user, who is the ultimate judge of performance. Thus, performance must be evaluated in terms of its impact on individual users.

Figure 19 illustrates the definition of a WSP. At any given time, there is a set of applications. Each application potentially executes concurrently on a multicore processor.

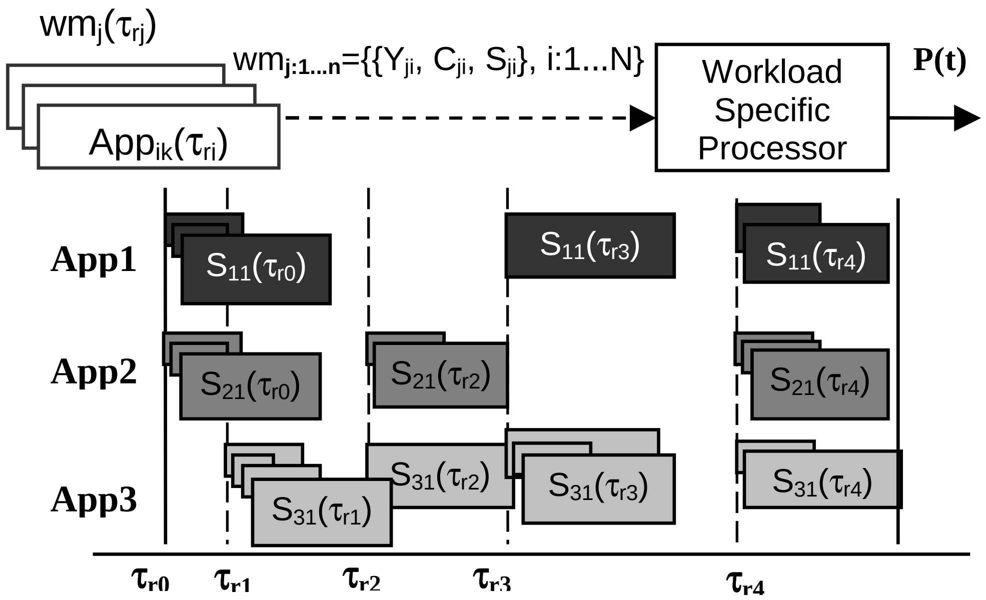

Figure 19: Workload Definition of WSPs

We use time to identify both the arrival of applications as well as their persistence in the system as new applications arrive. Workload mode wm_1 at time τ_{r1} has n_1 applications of m_1 application types, wm_2 at time τ_{r2} has n_2 applications of m_2 application types, and so on. Further, each application type at τ_{ri} consists of s instances, and the cumulative summation of these instances at any given time τ_{ri} equals n_i. Thus, the loading of the system is a function of external timing, unlike uniprocessors, which are sequenced by the speed at which an instruction stream is processed. Furthermore, each user application consists of several processor tasks that have different characteristics, such as Type, Y, Complexity, C, and Data Dependency, S, discussed in Table 7. When executed, each task in a mode wm_i processes associated input data, S, from the set. The input data sets also change with time. Some applications may persist in the system indefinitely while others may not complete before the next input arrives.

From an optimization perspective, this results in a multimodal system; the system operates in different modes at different times and these modes must be recognized so that the system can respond to different loading situations. In this thesis, we model overlapping demands as workload modes and show how the identification of workload modes is critical to effective optimization of WSPs. We show why they must be identified using models of the likely behavior of the system, in which these models are derived from single-user models. Next, we discuss how prior work in performance metrics and workload analysis has not focused on single-user usage patterns that result in workload modes that include heterogeneity, concurrency, and timing features.

Chapter 4

Prior Work

If there is any one secret of success, it lies in the ability to get the other person's point of view and see things from that person's angle as well as from your own.

Henry Ford

Since the primary contribution of this work is the development of the Capacity metric, in this chapter we first summarize existing performance metrics used in computer systems, discussing why they fail to properly evaluate the performance of CHMs that process heterogeneous multi-channel inputs. Because our Capacity metric is graphical and includes the effects of global resource sharing, similar to metrics used in networks, we also include a discussion of metrics in computer and communication networks. The development of our Capacity metric was motivated by the need to evaluate WSPs, so we then include a discussion of existing workload models and types of benchmarks and why these models and benchmark suites are inappropriate to evaluate CHMs. Finally, our Capacity metric needed to simulate workloads on CHMs, so we discuss our MESH simulator that has been used in that regard and show why existing HDL simulators failed to achieve this goal.

4.1 Performance Metrics

Considerable prior work exists in metrics and evaluation of workloads and multicore computing. Although prior work has called upon the need to develop new performance metrics [51], none has considered breaking out the heterogeneity of workloads or has modeled heterogeneity impact on the performance of a CHM that processes a heterogeneous multichannel workload.

Alameldeen and Wood discuss the need for simple work-based metrics [52]. However, suggested metrics accurately predict performance only if the unit of work is representative of the entire system’s functionality, which is not the case when multichannel inputs implicitly carry different units of work. Kumar et al. evaluate the throughput of CHMs by increasing the demand for each application type until it reaches the maximum production [49]. The overall throughput is then evaluated by finding the average score of the individual throughput scores. But averaging implies a functional overlay of the way different demand streams will be combined that can obscure the variety of situations a real system can encounter. Skadron et al. pointed out that average values used for many current systems that run multiple processes simultaneously (such as average IPC) can lead to misleading performance results because they factor out performance differences that result due to the bursting behavior of current systems [51]. Hill and Marty develop a speedup hardware model of multicore systems to complement Amdahl’s software model, but assume that the input is a single channel [53]. SPECRate is also a single valued metric [54].

Prior work has also examined the throughput of Simultaneous Multithreading (SMT) and CHM architectures using both parallel programs and multiprogrammed workloads [55], [56], [57], [49]. Both parallel and multiprogrammed workloads provide thread-level parallelism, but they differ in how threads compete for global chip resources. Threads of multiprogrammed workloads do not share memory references; this results in more accesses to the cache memory. Furthermore, these threads have different phase patterns such as ILP and memory accesses, causing interference in branch predictors. Kumar et al [49] statically mapped each thread type to a specific core, thus multiprogrammed workloads are less likely to vie for the same processing elements on chip. Thus, evaluating computer architectures using multiprogrammed workloads is different from our evaluation for Capacity metric because the number of threads of different types is not fixed to the number of cores on chip and the execution time is limited by a time window that is perceived by the user. While parallel applications [58], [59], [60] may have the benefit of sharing the caches and branch predictors, they test SMT differently. In contrast with the multiprogrammed workloads, all threads in a parallel application have similar execution patterns. As a result, threads in a parallel application may create bottlenecks in these resources. Traditionally, throughput could be used to model these classes of parallelism, because the similarity between elements of the parallel computation permitted evaluation to take place over a common work unit. However, in modern computer systems with multiple, diverse production streams that persist in the computer over long period of time, the differentiation in the types of production requires evaluation and analysis to consider the effects of workload heterogeneity.

Existing performance metrics can be broadly classified as either time-based or work-based metrics. Table 9 summarizes examples of each class. Time-based metrics, such as latency and response time, are used to optimize the computer systems in which the input is sequential, or SUSA computers. In the new era of computing in which computer systems are increasingly integrating multicore chips in the service of single users, work-based metrics, such as throughput and bandwidth, are more appropriate to optimize these systems. In other words, the design goal of single-user multicore computers, or SUMA, is to process as much work as possible during a window of time. None of existing work-based metrics has considered breaking out the heterogeneity of workloads or has modeled heterogeneity impact on the performance of a CHM that processes a heterogeneous multi-channel workload. For computer systems that need to be optimized for both time and work metrics such as super computers, power consumption and heat dissipation become a major concern. In contrast, for the systems used in some domains such as banking the designer (and the customer) worries about the security, reliability and availability of the system more than anything else.

Table 9: Taxonomy of Existing Performance Metrics

Work	**Time**	
	Matters	**Doesn't Matter**
Matters	Power, Power per Area, and Utilization, etc	Throughput, and Bandwidth, etc
Doesn't Matter	Latency, and Response Time, etc	Reliability, Availability, Security, and Scalability, etc

Through recognition of the architectural coupling of different demand streams in multicore computing, but at the same time the separation of the performance of each type of demand stream from the perspective of the user, we establish the foundation for analysis of CHMs that service single users, or SUMA computers. With the Capacity metric, we set up the possibility of analyzing CHMs so that resources will not be wasted as resource loading, or workloads, can be executed by the computing device, but these effects might never be seen by the user.

4.2 Performance Modeling of Computer and Communication Networks

There is a similarity between our Capacity metric and performance metrics in networking:

- Some performance metrics in networking are graphical. For instance, considerable work has been done on network optimization using the network load metric, which is a graphical metric. The shape of the network load metric curves reveals significant information that helps in load balancing management; and
- Both measure the effects of global resource interactions on the performance of a system. Global resource interactions that take place while routing a packet, represented as an overhead, are widely modeled and studied in networking.

Thus, we review the literature of performance metrics in networking to point out similarities and differences with our work, signifying the potential of our Capacity metric to be used in communication networks as well.

Performance analysis has an important role in networking because it provides estimates of network behavior in terms of different metrics. Depending on the metrics of interest, designers can select network architectures and protocols. Performance modeling and evaluation can be done using either simulation or analytical models. Simulation is based upon measurement, similar to the way we generate our Capacity curves, while analytical models are based on a representation of a system via a model. Both need to model performance in terms of a metric.

Two primary metrics are used in the performance evaluation of mobile ad hoc networks: packet delivery rate and end-to-end delay. These metrics are mainly used to compare different ad hoc routing protocols. The work of [61] uses the packet delivery rate and end-to-end delay metrics as the two primary metrics for evaluating VoIP communication. These metrics are analogous to the throughput and latency metrics used in computer systems. Other metrics such as jitter, flow length, and packet overhead are also important metrics that can be derived from the primary metrics. All of these metrics are single-valued metrics. Each metric is typically obtained via averaging all metric values obtained from the monitored flows. Recently, two directions of research have been taken in the development of performance metrics of ad hoc mobile networks. In the first direction, different types of performance metrics are distilled into a single-valued index. In the other direction, performance is broken down into its components instead of using averaging. This requires a graphical means to view performance. Here, we discuss both directions.

The work of [62] defines an *index* for measuring the performance of ad hoc networking protocols. This performance index distills different performance values that result from different metrics into a single value that can be used to describe the overall system performance. In order to do this, Ajbar and Perkins develop a statistical model that averages the performance of four metrics: packet delivery ratio, end-to-end delay, jitter, and packet overhead [62]. While this approach makes comparison easier, but it may lead to incorrect results because it factors out the performance differences between systems when evaluated against different metrics.

The work in [63], [64] find that distilling performance into a single-valued score via averaging conceals the relationships between performance metrics and system parameters. Moreover, it often leads to misleading conclusions about system performance. Note that different routing flows from senders to receivers have different characteristics [65], [66]. As a result, Yuen and Yates [63] present a graphical representation of performance results. This graphical representation can be used to reveal the different relationships between different performance metrics and system parameters. This in turn gives insight about what parameters lead to changes in performance. While this type of work identifies the significance of cause and effect analysis in performance evaluation through the use of graphical metrics, it does not consider the different types of work the system is producing and the impact of the work type on the performance of a system.

Considerable work in networking has considered the impact of routing overhead on the performance of networking systems [67], [68], [69], [70]. While overhead modeling in networking considers the size and type of the packet: control or data, it does not consider the heterogeneity of the content of the original files being sent through the network.

While much of the work on the performance evaluation of ad hoc networks has focused on routing performance, other work has considered power constraints that limit routing performance [71]. In general, power is derived from two primary metrics: packet delivery rate and delay. This is analogous to the way we view performance; in order to correctly model power consumption, Capacity must be accurately modeled first.

4.3 Workload Analysis

The development of accurate workload models of single-user multicore devices was required for the development of our Capacity metric. Our Capacity metric accounts for the effects of these features of workloads, which should also be considered in modeling the workload of CHMs:

- Heterogeneous concurrency: heterogeneous inputs arrive the systems as sets of concurrent applications or workload modes;
- Arrival time: timed arrival of workload modes and their associated data. Workload modes occur due to either external events that occur due to user—computer interaction or internal events that arise due to changes in the data;
- Global resource interactions (represented as overhead), such as schedulers, memory, communication resources, etc.: This is a result of the above two features; the overlap between different sets of concurrent, heterogeneous applications makes global resource interaction more significant to performance.

There is much prior work on workload modeling, but none includes the above list of features when modeling workloads of single-user, mobile devices and uses them as the foundation of performance optimization of CHM architectures.

Considerable prior work has recognized the significance of workload characterization for accurate performance evaluation [72], [73], [74], [75]. Previous research on workloads characterization has focused on constructing workloads using either existing benchmarks [49] or realistic data collection [76]. Exposing the system to a collection of benchmarks, either one at a time or collectively, does not construct a real workload. On the other hand, previous work that used realistic data by observing a specific system for some period of time has focused on the number of requests, session duration, number of accessed pages, etc. These parameters do not take into consideration much more vital characteristics of the request itself, such as the complexity of the request, the number of simultaneous instances of each request type, or the time granularity of the request that may be limited by single-user access patterns, etc. Some previous work used statistical models to represent workloads [77], but these aggregate instances do not permit modeling of individual usage patterns. Finally, most of the previous work has focused on server workloads [78], [79]. Next, we elaborate more on different types of workload analysis, including existing workload modeling and workload characterization techniques.

4.3.1 Workload Models

We group existing workload models in the literature review into three categories; Benchmarks, Graphs, and Traces.

A. Benchmarks

Previous research on workloads has focused on constructing representative workloads using existing benchmarks [49]. Joshi et al. propose an approach to identify similarities between programs according to microarchitecture-independent characteristics [80]. Hoste et al. use program similarities to predict programs' performance in advance [81]. In contrast, Vandierendonck et al. develop techniques to identify differences between benchmark programs [82]. However, exposing the system to a collection of benchmarks, either one at a time or collectively, does not construct a real workload. When models of single-user usage patterns are used, distinct modes of operation emerge that do not emerge from a random collection of interacting programs.

In Chapter 3 we concluded that existing benchmark suites such as SPEC, MiBench, MediaBench, and EEMBC represent the workload of a batch style execution system. Thus, they do not model concurrency and external timing of input arrival. Because existing benchmark suites focus only on single program executions, existing benchmark suites cannot be used to evaluate the performance of the CHM that process timed sets of concurrent heterogeneous applications.

B. Graph-Based

Some existing workload models can be used to model application parallelism. One popular representation of application parallelism is task graphs [83], [84]. These graphs are composed of nodes that represent functions and arcs that show the dependencies among these functions.

D. Ferrari [85] introduce user behavior graphs. The nodes of a user behavior graph represent different types of user requests. The arcs associated with probabilities represent the sequences of user requests. If overlap option was chosen by the user, the user can issue as many requests as needed without having to wait for the completion of the existing one. Thus, these requests are simultaneously executing on the system.

User behavior models are hierarchical. In other words, high-level workloads will be translated into a stream of low-level workloads. For that, Calzarossa et al. define a layered framework for the modeling of user behavior models [86]. Markov chain models can be used to identify dependencies between user requests [87], [88], [76], [89]. Later in this thesis, we show how Markov chains are too costly for recognition of workload modes in single-user devices. Although user behavior graphs seem to be a good choice for emergent workloads, they do not model the arrival time of requests.

C. Trace-Driven or Measurement

Both benchmarks and graphs are a representation of system workloads that may include a large margin of error. Thus, another type of workload models is based on the collection of traces via measurement. Traces are used to identify the key characteristics of a real execution behavior. To capture a real behavior, accurate instrumentation need to be developed. Further, to capture more characteristics about the execution behavior there is a need to insert a large number of checkpoints simply such as print statements into the system. Still further, because the collected data are usually large there is also a need for a tradeoff analysis between collection overhead and the accuracy of the model. Moreover, appropriate techniques have to be developed, so that it can be applied to ensure the quality of the derived workload model [90], [91]. While they are more accurate models of workloads, trace driven models incur overhead in terms of instrumentation setup and the amount of collected data. Next, we survey some workload characterization approaches that can be used in that regard.

4.3.2 Workload Characterization Approaches

There are several workload characterization approaches that can be used to generate workload models. We classify them into three groups; numerical, statistical and stochastic.

A. Numerical

One numerical technique used in workload characterization is clustering [91]. Clustering is used to reduce the volume of collected data by discovering patterns in data. Two primary clustering algorithms are: k-means and hierarchical clustering [92]. As discussed earlier, Joshi et al. used both clustering approaches to identify similarities between programs [80]. Calzarossa and Serazzi [93] apply numerical fitting techniques to model the fluctuations in job arrival patterns. Both clustering and fitting techniques are needed to identify representative arrival patterns.

B. Statistical

Numerous studies have focused on modeling workloads by collecting workload traces or realistic data [94] and developing statistical models to fit these data based on specific attributes of workloads [77]. However, these approaches fail if the data is not stationary [95]. For example, Chiang et al. found that collected data may not have the same pattern during different times in the year [94]. S. Hotovy finds that workloads are also different at different installations and as users learn better how to use the system [96]. Moreover, previous work has focused on less significant system characteristics such as the number of requests, session duration, and the number of accessed pages, etc. These parameters do not take into consideration much more vital characteristics of the request itself, such as the complexity of the request, the number of simultaneous instances of each request type, or the time granularity of the request.

Descriptive statistics has been widely used to characterize workloads [73], [97], [74], [98]. This model targets batch and interactive systems [91]. For the workload of personal computers, general descriptive statistics are used to identify user behavior [99]. But these aggregate instances do not permit modeling of individual usage patterns. Moreover, most of these studies model only the static characteristics of the workload. To model the dynamic characteristics of the workload, graphical and mathematical methods are used. Thus, statistical and numerical models are static models, in contrast with the behavior of users, and may yield misleading results when using the wrong model.

C. Stochastic

The work of [100] builds a model of instructions' sequences of individual jobs using a Markov chain model. Markov chains are composed of states that represent different instructions and arcs that represent the possibility of instruction transition or sequences. Stochastic models, based on Markov chains, are used by G. Haring [101] to represent task interactions. Another domain that stochastic models are used in is the identification of phases that are used to represent the execution behavior of a program. A computer program is as a sequence of phases that differ in terms of processing, communication, memory access patterns. Carlson et al. [102] analyze the execution profile of an application in order to identify its phases. The execution profile is defined as a sequence of clusters of periods that have roughly uniform processor utilization and separated by sudden changes in processor utilization. Waheed and Yan [103] find by studying computational fluid dynamics applications that the characterization of phases can be used to develop tuning methodologies In general, stochastic models are not scalable while the number of workload modes and processors increases – something we discuss thoroughly in the next chapter.

By focusing on emergent single-user multicore computers, we differentiate from all other workload approaches in which workloads are derived from stochastic models of arrival. Usage patterns of single individuals are far more tractable than workload models that arise due to the interaction of many individuals vying for access to limited resources. We focus on building a classification scheme whereby architectures can be categorically optimized to classes of individual user access patterns.

In general, these techniques can be used to develop a single-user model of usage patterns, which then result in a new form of benchmarking of the form shown in Figure 6. One purpose of this thesis is to show why this new form is necessary for emerging CHM computers, which are used by single individuals for processing and communications with other computers. We do so by focusing on the need to optimize to significant changes in workload patterns, or workload modes, and show how this optimization is not likely to be effective in the absence of more sophisticated forms of benchmarks that are suited for emerging WSPs.

4.3.3 A Summary of Workload Modeling Techniques

We summarize the above discussion of workload modeling and characterization techniques. We classify these techniques based on two orthogonal factors: (1) task concurrency and type, and (2) arrival timing. The intersection between the values of these factors results in a different performance evaluation model. Table 10 summarizes our classification.

Table 10: Performance Evaluation Models Classification

Timing / ***Concurrency / Type***		***Un-timed***	**Timed**
Single		Benchmarks, Traces	Stochastic, Statistics
Multiple	Homogeneous	Benchmarks, Behavioral Graphs	Stochastic
	Heterogeneous	Benchmarks, Behavioral Graphs	Workload Modes

Benchmarks are presented to the system in an untimed manner. Further, they are presented to the system individually or concurrently (multiple copies of the same benchmark submitted to the system). Recently, prior work has considered heterogeneous, concurrent benchmarks. The number of running benchmarks matches the number of cores and thus these benchmarks are statistically mapped to these cores. While behavioral graphs are interesting because they profile the user behavior, which may expose the system to concurrent tasks, they lack the inclusion of arrival time information. Only stochastic processes can model the arrival time, but these times are not real, they are based on statistical analysis of collected data or traces. Further, there is not a stochastic model that models heterogeneous concurrency.

Another level of classification that can be considered is when the type of architecture is included. Architectures can be classified as uniprocessors or multiprocessors, similar to the concurrency and heterogeneity features of the software. Multiprocessor architectures are further classified as being either homogeneous or heterogeneous. None of the above techniques have also considered the overhead of maintaining a heterogeneous set of processors that are especially used to process a heterogeneous multi-channel input. As a result, we needed to develop a workload model that includes all of these features and effects that can be then used to evaluate CHMs.

4.4 User Profiling and Usage Pattern Discovery

Perhaps the greatest potential of the Capacity metric is to use it in conjunction with user profiling. In this way, the resultant Capacity shapes can be used to identify where performance of the CHM, in effect, is wasted on the user, especially characteristics of an individual class of user who uses their mobile device in very different ways from other classes of users. This will ultimately result in caps placed on Capacity curves/shapes, in which, above a certain shape, performance of the CHM is wasted. At this point the computer designer can use information developed in the analytical model in order to provide performance to the individual user that will actually have an effect on the user. Further, since our Capacity metric is used to model the performance of CHMs optimized to single-user usage patterns, usage pattern discovery is key. While we are not modeling real usage patterns, we also survey methods of usage pattern discovery for the sake of completeness.

We have previously shown that user access patterns of webpages can result in single CHM architectures with significant performance differences [104]. There are two primary user profiling approaches. Explicit profiling is done through explicit input from the users themselves in the form of questionnaires, interviews, and polls, etc in order to capture their interests [105]. Implicit profiling, or indirect profiling, is done by observing the navigation patterns of users using client or server logs. User profiling commonly uses machine learning techniques to discover potential interests and thus be able to create useful patterns in the profiles [105].

Once data is collected, the pattern discovery process starts. Several methods from different fields such as statistics, data mining, machine learning, and pattern recognition can be used to do pattern discovery. Kosala and Blockeel [106] classify the purpose of Web usage mining to either understanding the user behavior so that the Web site can be optimized accordingly [107], [108], [109] or to improve the website’s effectiveness

[110]. Web personalization to user preferences has been done using artificial intelligence techniques in [108], [109].

It is essential to ensure that accurate usage profiles are established in order to improve the efficiency of design choices. Hence, there is a need to gather as much information as possible pertinent to a user's interaction with the system in order to identify behavioral patterns of users that could occur. In future work, we will investigate these and other techniques in order to create clusters of user profiles. For the purpose of this thesis, while our usage scenarios are artificially made-up, their implementation is real, which we think is enough to show the potential of our new performance metric.

4.5 The Modeling Environment for Software and Hardware (MESH)

The development of WSPs needed to simulate the performance (or the Capacity) of CHMs that process workloads result from different usage patterns of our multimedia cell phone example. Performance simulation of CHMs requires the manipulation of:

1. Threads instead of instructions;
2. Multiple-ISA processing elements instead of functional units;
3. Memories instead of state registers;
4. Complex interconnection networks instead of simple, message-passing channels or even wires;
5. Chip level (global) dynamic schedulers instead of dataflows; and
6. Timing that results from the intersection of discrete events running on heterogeneous processing elements instead of global clock cycles

The above list of design elements requires the simulation tool to manipulate:

a. Global-local state preservation and coordination;
b. The overhead of global resource interactions (such as communication and memory contention, congestion, and synchronization);
c. Task migration between heterogeneous processing elements; and
d. Different triggering methods

Further, the exploration of these design elements needs:

- Quick design space exploration and
- Easy debugging

Here, we first perform a survey of existing simulation methodologies, languages, and tools. Then, we show why each tool fails to simulate CHMs. Finally, we include a discussion of our MESH simulator and contrast it with existing simulators.

Cycle accurate and instruction accurate simulators have been used for performance evaluation of CHMs. As computers integrate more processing elements, ISSs cannot be used to simulate the performance of CHMs, not just because they are slow, but also because they cannot manipulate the hardware and software features mentioned in the above list. Table 11 summarizes the list of existing tools reviewed, classified as performance specification vs. performance evaluation. Those tools that are classified as performance evaluation they also include performance specifications as well, usually conducted by a third party tool or language.

Table 11: System Level Performance Languages and Tools

Specification	***Evaluation***
VHDL, Verilog	SpecC
NI LabVIEW FPGA	SystemC
JHDL	Ptolemy
JHDLBits	ARMn
JBits	GEMS

The Hardware Description Languages (HDLs), such as VHDL and Verilog, have been long used to describe systems at the RTL. They are only used to describe the hardware and not the combination of hardware and software. Further, they are too detailed which limits fast design space exploration. For Field Programmable Gate Arrays (FPGAs), there are tools that combine software code with hardware code to create a functional design. Examples of these tools are NI LabVIEW FPGA [111], Java-based structural Hardware Description Language (JHDL) [112], [113], JBits [114], and JHDLBits [115]. However, these tools are not mature; they still lack modeling some design features. Further, they are not much different from traditional HDLs because they still simulate registers, wires, and gates, etc. They do not describe high-level evaluation of workloads executing on heterogeneous multicore chips.

System-level modeling languages such as SpecC, Ptolemy, and SystemC are built upon existing programming languages such as C and C++, so they do not have dedicated compilers. They use the compilers built for pre-existing programming languages. These languages have different aspects of system modeling. SpecC is used in Computer Aided Design (CAD) community. It uses communication refinement for performance modeling and to link different abstraction levels together [116]. The goal of SpecC is to eventually plug its modeled design to automated refinement and synthesis tools [116]. One of the major drawbacks of SpecC is that it limits user extendibility feature [117].

Ptolemy II is a java-based tool that is developed by the Ptolemy Group [118]. It supports heterogeneous behavioral hierarchy through an actor-oriented design in which actors are software components that execute concurrently. The semantics of a model are defined by a software component that exists at each hierarchy layer (called the director) and implements a model of computation. However, Ptolemy II is used in the embedded software and synthesis community [116], [119]. Thus, in order to be used in a different domain, the designer needs to use more instructions.

In contrast, SystemC allows extensions of the language to be created by users [116], [119]. SystemC's goal is to allow quick design space exploration by facilitating the creation of designs with different variants [120]. However, there is a three-fold major drawback to using SystemC:

- SystemC is generally used with an HDL where SystemC models the testbench and the HDL simulator models the system itself.
- Debugging; errors in SystemC code will not be found until runtime during testing. Further, these errors are C++ errors; they are not related to the modeled system itself, making debugging complicated and time consuming.
- SystemC models are even more complex because performance analysis procedures and the underlying communications structure have to be specified by the user.

For these reasons, SystemC and the other system level languages are inappropriate to simulate CHMs that process heterogeneous, multichannel inputs.

Recently several simulation tools targeting multicore systems have been developed such as GEMS [121] and ARMn [122]. Both simulators are cycle-accurate. The GEMS was initially developed to simulate multiprocessors in database and Web servers. The ARMn is targeting single-ISA (that is ARM) multicore systems. In summary, design languages and simulation tools described in this section are inappropriate to model hundreds of heterogeneous multicore architecture variations and evaluate their performance in a timely manner.

Because existing simulators failed to simulate the performance of CHMs that process workloads, we used our MESH simulator to model our cell phone example in the experiments of Chapters 5 through 7. The Modeling Environment for Software and Hardware (MESH) permits performance and power evaluation when threads execute on sets of heterogeneous resources under a variety of custom schedulers [123], [124]. MESH explores CHM design above the level of the ISS, where designers deal with threads, processors, and memories instead of instructions, functional units, and registers.

MESH enables designers to evaluate the performance effects of design trade-offs in the numbers and types of processors and communications mechanisms, scheduling decisions, and software tasks (arrival time and complexity) on the overall performance of CHM systems. MESH has also previously proven its accuracy and speedy simulation [125], [126].

MESH increases simulation performance by emulating target system performance using annotations inserted within the code. These annotations are generated by executing the same application code on the real platform to capture data dependent execution. This approach is called execution-driven simulation with cross-profiling or back-annotation. It has been widely used for traditional multiprocessor simulation [127], as well as for simulation of CHM systems [128] [124]. These annotations capture the computational complexity of a program’s code. Once the annotation region is reached, the simulation kernel determines the physical timing needed to execute the code of this region using the computational complexity values in these annotations. Each piece of code has to be annotated only once. This allows speeding up the design space exploration. This process can be automated using profiling tools [128].

MESH is also capable of evaluating the effects of bus and memory contention on overall system performance. Once a shared resource access is accessed, an annotation must be inserted to determine contention [124].

The process starts with a program's source code. Then, the designer must identify the locations in the code in which annotations need to be inserted. Annotations need to capture the control flow of a program [124]. Therefore, annotations are usually inserted at the end of every loop and in every path of the branch.

Schedulers use "consume calls" to resolve the logical threads of the software to the physical threads of the chip resources. "Consume calls" represent the complexity of a program fragment. A program fragment is defined as the granularity of software in a simulation. Therefore, fragments can be as large as the threads or as fine as the instructions. In general, fragments are finer than individual threads and coarser than individual instructions. If fragments are too coarse, accuracy suffers, and if fragments are too fine, simulation time suffer. Significantly, the annotation of "consume values" for program fragments is flexible, and this in turn makes design space exploration faster. However, the accuracy of these values is an important part of creating a good model.

Chapter 5

Workload Mode Identification Models

It doesn't matter how beautiful your theory is, it doesn't matter how smart you are. If it doesn't agree with experiment, it's wrong.
Richard P. Feynman

The initial objective of this work was to design and evaluate CHMs around our definition of *workload modes* that arise from single-user usage patterns. A central challenge for this type of design is how to model and identify the workload modes at real time, otherwise the systems do not have the ability to optimize to distinct situations as they are encountered. Thus, we analyzed and evaluated two modeling techniques, a Workload Classification Model (WCM) and a Hidden Markov Model (HMM), against their ability to permit dynamic optimization when specific workload modes were encountered. We include experimentation on our multimedia cell phone example and in so doing demonstrate why this work resulted in the need to develop the Capacity metric.

5.1 Workload Classification Model

WCM extends prior work in microarchitecture-independent workload characterization for individual benchmarks [129] by including timed concurrent applications. Our initial procedure is:

Step 1: Characteristic Extraction. Each task type, *Y*, mentioned in Section 2 has a different working data set, *S*, and complexity, *C*, classified into three relative classes: low, medium, and high. Working data sets are further classified based upon their relative frequencies of update: cached and non-cached. Characteristic extraction is conducted in an adhoc manual manner by the system designer at design time.

Step 2: Forming **wm** *Vector*. Step 1 resulted in *m* different workload mode characteristics. **wm**=[$w_1,\ldots,w_j,\ldots,w_m$] represents the workload characteristics vector of length m, characterizing each workload mode wm, in which w_j is a workload characteristic.

Step 3: Forming **M** *Matrix.* Each usage pattern has n workload modes. For example, the usage pattern in Figure 1 has 12 workload modes. **M** is the generated matrix of dimension $n \times m$ that represents a usage pattern, in which rows are **wm** vectors and columns are the number of instances, s, of each workload characteristic, w, in **wm**.

Step 4: Forming **R** *Vector.* We suggested normalized weights, r, that correspond to the task complexity. For instance, baseline JPEG was assigned higher weights than text since JPEG has higher complexity than Text. Different JPEGs have different weights due to data content. Further, cached JPEG has lesser complexity than non-cached JPEG. Generally, r takes values in the range $m \geq r \geq 1$.

Normalized complexity weights are generated by dividing the cycle budget of each application by the maximum cycle budget needed by an application in the system. We extend Sim-Profile [130] of SimpleScalar to extract the number of cycles for each application. Sim-Profile profiles each application when it runs on a different processor type.

Step 5: Assigning **R** *Vector Values to* **M** *Matrix.* Matrix **M** is then multiplied by weight vector **R**. We normalize each column (characteristic) in the matrix by subtracting the mean value and dividing it by the standard deviation. The goal of the normalization is to put all characteristics on a common scale. Then, we linearly combine the absolute values in each row (workload mode vector **wm**).

This algorithm is used to identify workload modes at runtime, exploiting some design-time knowledge. The algorithm characterizes the workload of each mode individually without considering the prior workloads that might exist in the system before the arrival of the current mode. We gathered the data used by the WCM by observing the workloads of real-time usage patterns of the single-user cell phone example. The workloads are composed of sets of applications that differ in terms of datasets and arrival times and are therefore sufficient for the purpose of illustration.

These usage patterns are represented by a data array in the format of [time, application(s)], in which time is the arrival time of application(s). The data array is then used as an input to a simple execution engine that reads the data array. Each workload mode is defined at the arrival time (**wm** vector). The engine identifies all workload modes in the data array and starts filling the **M** matrix by computing the number of instances of each application in different workload modes. Then it calls the profiler for each application to compute the complexity of each application; this forms the **R** vector. Finally, it assigns the complexities to Matrix **M**.

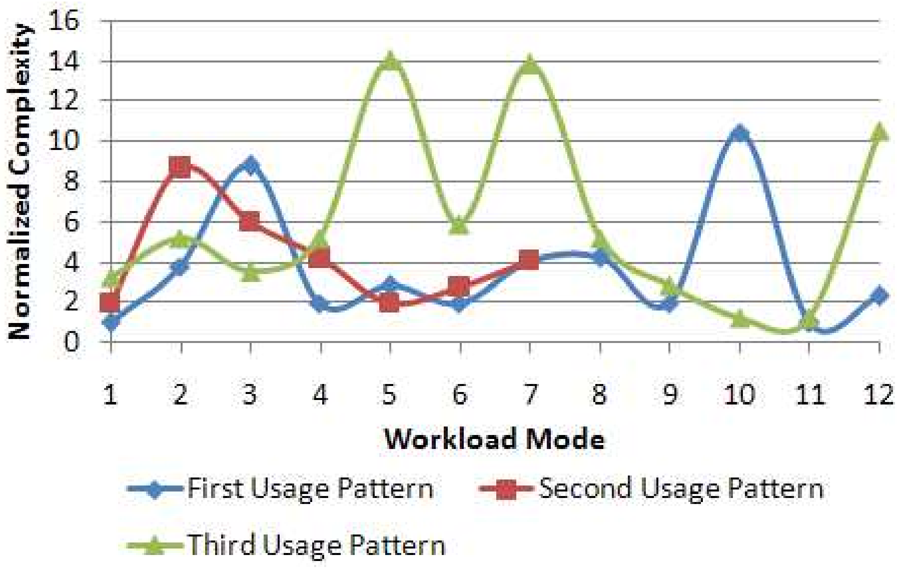

Figure 20: Workload Mode Characterization

The number of distinguishable workload modes from a system is too large. Therefore, to avoid the complexity of handling all of them at run time, several workload modes are clustered. This clustering is based on the execution cost of the workload modes. We build a vector table that includes all workload modes identified at design time as well as those identified at run time. Based on the values of the Global Chip State (GCS), a simple lookup in the vector table of workload modes can be carried out to find the optimal optimization profile for the identified workload mode.

Figure 20 shows the normalized average workload complexity for each workload mode in three usage patterns (Figure 6, Figure 7, and Figure 22 are the first, second and third usage patterns respectively). Each of the first and third usage patterns has 12 modes, while the second usage pattern has only seven modes. Workload modes are clustered into a set of unique modes. These normalized complexities are considered as workload mode unique IDs that can later be used to identify workload modes at run time. For instance, modes 5, 7 and 12 in usage pattern 3 and mode 10 in usage pattern 1 have relatively the most complex workloads, while modes 1, 2, 3, 4, 6, 8, and 9 in usage pattern 3, modes 2, 5, 7, 8, and 12 in usage pattern 1 and modes 3, 4, 6, and 7 in usage pattern 2 have the lowest workload complexity. High complexity workloads have a higher number of jobs and more of type streaming while low workloads have fewer jobs and more of types like text processing.

5.2 Hidden Markov Based Model (HMM)

Our initial WCM model assumed that there was sufficient distance between the time intervals between modes so sequence does not matter. However, if model sequence matters, this assumption is invalid and a more complex model that cannot take advantage of design-time modeling of workload modes is required. HMMs are one such model. HMMs is a useful framework for modeling state transitions in a variety of areas such as speech recognition and bioinformatics [134]. Computer architects also use HMM in workload modeling of different computer systems [131], [132], [133]. Figure 21 shows a simple three-state diagram to capture the sequence relationship between workload modes. Here, we assume that current state depends on only the previous state, but not earlier states, or modes. If sequence matters, our initial model must be modified with three additional steps (using the notation similar to Rabiner's [134]), thus forming an HMM:

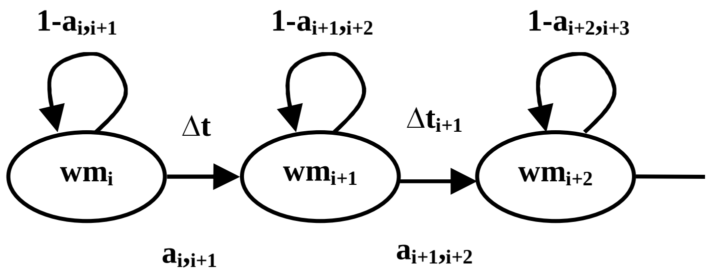

Figure 21: A Simple Three-State HMM Diagram

- *Step 3.a: Determining the **A**, **B** and **π** Matrices.* **A =** matrix of transition probabilities a_{ij} for moving from mode wm_i to mode wm_j in one time step; **B =** observation probabilities of observing a symbol while being in mode wm_j; and **π** = the initial mode probability distribution.

- *Step 3.b: Determining the inter-modes duration probability Matrix* **X**. This step enhances HMM to be used in our problem space in which a mode's inter-arrival time is unpredictable due to the arrival of external event. Steps 3.a and 3.b could be done through user profiling.
- *Step 3.c: Running an HMM re-estimate (learning) algorithm to adjust the HMM parameters (**A**, **B**, **π**) based on the forthcoming workloads*. The Baum-Welch algorithm is one of the popular algorithms used to do this. It uses the forward-backward procedure to re-estimate the parameters of the model as soon as new data are available.

Our enhanced HMM model results in a workload mode transition table in which the next workload mode can be predicted based on the current workload mode or chip mode and the values of the GCS. The data in the workload mode transition table is updated at run time using a learning algorithm. In the next section, we experimentally evaluate the use of the WCM and HMM models used to identify persistent changes in the system loading, modeled as workload modes.

5.3 Experiments

Our goal in the experiments is to compare the use of WCM and HMM through the illustration of a single-user CHM with Internet access. While our example is small, it includes a representative set of applications that can be modeled with arrival times that cause overlap and are differentiated by type, as discussed previously. Our main goal is the comparison of the use of single-user workload mode modeling with application-independent optimization in a CHM.

We start by describing the experimental setup, then describe three sets of experimental results: one that examines the impact of mode sequence on modeling, one that describes how WCM and HMM differ in terms of overhead, and one that evaluates several candidate architectures for optimal performance.

5.3.1 Example: Our Multimedia Cell Phone

Our cell phone example has been developed as a basis of a benchmark suite for heterogeneous multicore systems [1] and is discussed previously in section 3.3. The example is modeled using The Modeling Environment for Software and Hardware

(MESH) simulator, which is discussed in section 4.5. We test the example system using differnet usage patterns. Examples of usage patterns used in our experiments are in Figure 6, Figure 7, Figure 22, Figure 23, and Figure 24. For comprehensive testing purposes, we include three additional usage patterns, which also use the tasks shown in Table 7. Note that usage patterns, in general, differ from each other in terms of (1) the duration of the pattern, (2) the number of workload modes in the pattern, (3) the inter-arrival times of workload modes within a pattern, (4) the number and complexity of applications in each workload mode, and finally (5) the sequence of workload modes. The five usage patterns developed and used in our experiments depict some of these differences.

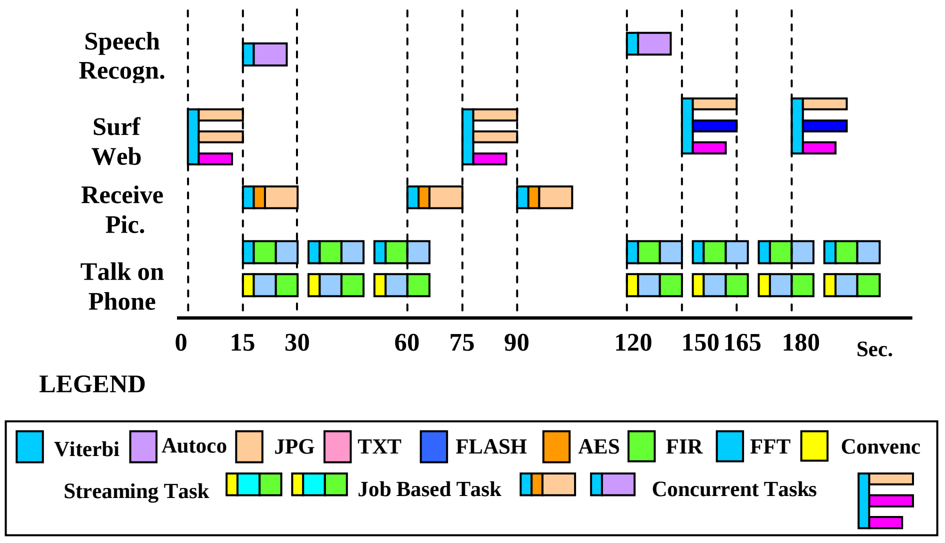

Figure 22: The Third Usage Pattern

Figure 22 shows a usage pattern in which the user is talking on a phone with travel agencies while browsing the Web for cheap flight tickets. Note that travel websites include search functionalities and contain a large number of photos and Macromedia Flash files. Further, the user enabled the speech recognition feature to look up phone numbers. Note that as time progresses, the user places different requirements on the systems as different combinations of applications execute.

Consider another usage pattern, timeline for which is shown in Figure 23. In this usage pattern, the user is talking on a phone with a friend while sharing pictures of her pets. This usage pattern exercises the system differently from the previous usage patterns, requiring encoding and decoding of multiple images at the same time. Finally, consider the usage pattern shown in Figure 24. In this usage pattern, the user is listening to music, surfing social networking websites, and exchanging pictures with a friend.

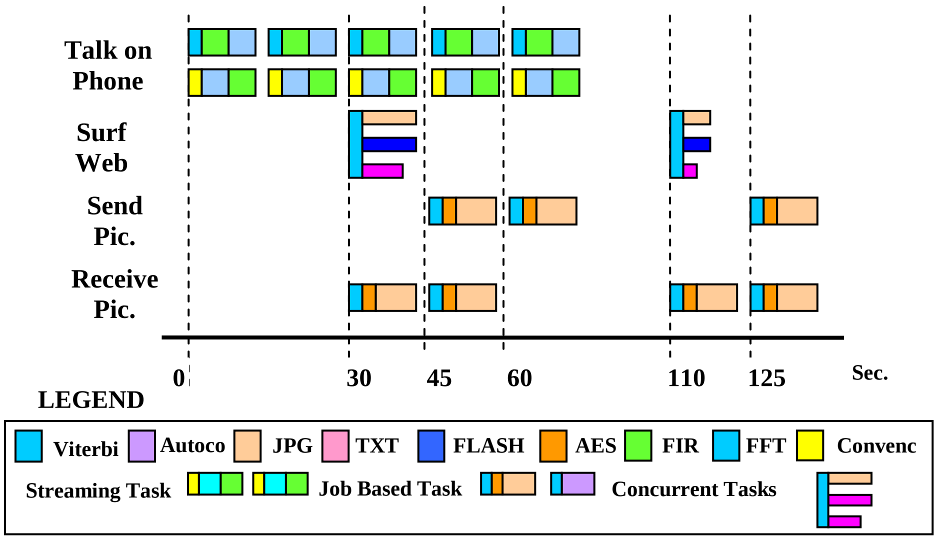

Figure 23: The Fourth Usage Pattern

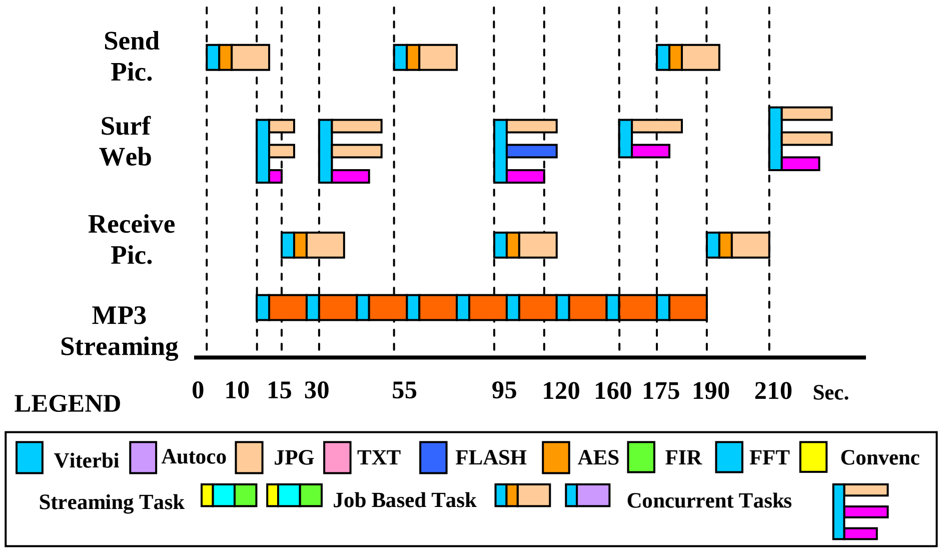

Figure 24: The Fifth Usage Pattern

Because of the different set of application sets, input data, deadlines and constraints, different usage patterns may exhibit different system performance, capturing the strengths and weaknesses of each architecture. Our five usage patterns are sufficient to show that single-user future CHMs open up the possibility to be designed as WSPs, but we agree that a complete evaluation of a CHM may include tens of different test usage patterns executing on the same architecture.

5.3.2 Architecture Modeling

Three different processors (AMD K6-2E+, Philips PNX1700, and ADSP-BF533) were chosen for our experiments to model the potential diversity of the computational capabilities, power consumption, and area requirements of cores in even simple heterogeneous multicore systems. Based upon the job types that dominate virtually all cell phone applications, it seems reasonable to select three categories of processor types to simulate: PNX1700 is the Media processor, ADSP Blackfin533 is the DSP, and the AMD K6-2E+ is the GPP. The ADSP-BF533 is used because it is highly optimized for DSP applications, such as Viterbi and Convolutional. The PNX1700 is highly optimized for media applications such as MP3 and JPEG encoding and decoding. The AMD K6-2E+ is a general-purpose processor that executes all applications fairly well. Although it consumes more power, the GPP is the second best choice for image files, which is interesting because the DSP barely outperforms the GPP for movie files. Both the DSP and GPP are significantly faster for text processing than the Media processor, and the GPP is the fastest.

The GPP is the worst processor in terms of size and power consumption. It is four times larger than the media processor and two times larger than the DSP processor. In terms of power consumption, the GPP consumes five times more power than the Media processor and almost 20 times more power than the DSP processor. Thus, the quantity of GPP processors available in the system and the tasks to be scheduled for the processor should be limited in order to consume less energy. Further, the Media processor is essential for a mobile device due to its small size and excellent performance for image and movie files. The only disadvantage for the Media processor is its inability to deal well with text processing. The area and power consumption for these processors were derived from information available from [135], [136], [137]. These processors are inadequate for use in a mobile device such as a cell phone because of area and power requirements. However, they are adequate to represent the key relative system-level design trade-offs for fixed-area devices, because they are consistent with each other. Lack of access to proprietary information makes a detailed examination of the processors used in mobile computing devices impossible.

Our target implementation is a single-chip heterogeneous multiprocessor with a fixed area budget to be populated by four categories: Media Processors (M), Digital Signal Processors (D), General Purpose Processors (G), and Chip-Level Cache (C) with a 1024K cache set aside for data processing. The Central Processor (CP), or controller, is used to run the identification and scheduling algorithms. These sets produce a total of 16 different architectures, as described in Table 12. eCacti [138] was used to determine the cache area and cache power consumption based on the selected cache size, block size and technology. We assumed a 65-nm manufacturing technology. Also, we assumed the power consumption of a processor in an idle state is 20% of its active power consumption.

Table 12: Modeled Architectures

	Architecture	**Central Processor**		**Architecture**	**Central Processor**
1	3G	G	**9**	4M, 4D	D
2	6D	D	**10**	6M, 3D	D
3	12M	M	**11**	8M, 2D	D
4	1G, 4D	G	**12**	10M, 1D	D
5	2G, 2D	G	**13**	1G, 2M, 3D	D
6	1G, 8M	G	**14**	1G, 4M, 2D	D
7	2G, 4M	G	**15**	1G, 6M, 1D	D
8	2M, 5D	D	**16**	2G, 2M, 1D	D

The Chip Level Scheduler (CLS) used in these experiments permits tasks to execute on processors other than the one that provides the best performance for the task type. Thus, note that all the application tasks are eligible to be placed at any of the processing resources. For a multiple-ISA architecture, this requires separately compiled copies of the same task available for the different processor types. Despite the requirement for more memory storage, prior work shows that this type of programming of heterogeneous multicore chips may result in better performance than static scheduling [139]. When the CLS is triggered, all tasks are candidates for redistribution on the processor resources.

Furthermore, this CLS permits task migration between cores. We do not focus on task migration due to preemption that arises from QoS priorities, such as in streaming. Rather, we focus on the migration of tasks to better performing resources when they become available. Because tasks, shown in Table 7, are small, independent tasks and task migration in CHMs typically requires complex approaches, we model migration by simply restarting task execution on the new processor. This requires task state rollback. Our prior work implemented task migration with rollback in the MESH kernel and shows advantages of it [140].

5.3.3 Overhead Modeling

A central question in computer design is: when does performance overcome the cost of the overhead and how will this relationship scale? Since we seek to understand of the impact of design-time knowledge of workloads on the design of the architecture (discussed in the next section), we include experimental data for the impact of overhead on mode identification.

First we analytically model the overhead of both WCM and HMM. We use the Big-O notation for time and space complexities to describe the behavior of both WCM and HMM. We also assume that the basic arithmetic operation on numbers take O(1) constant time that is the uniform cost model of time and space. For example, in step 5, Matrix **M** of size n*m is multiplied by Matrix **C** of size m*1. Matrix multiplication requires three for-loops resulting in time complexity of n*m*m or $O(n*m^2)$. The space complexity from creating the results matrix of size n*1 is O(n).

The learning and searching phases of the HMM require algorithms of three orders of magnitude of time complexity and two orders of magnitude of space complexity. Elliot et al. have shown that the HMM can be enhanced to give better predictions at the expense of increasing the space and time complexity of the learning algorithm to $O(n^4)$ [141]. Therefore, the most expensive step in HMM is the learning phase. It requires algorithms of at least $O(n^3)$ of time complexity and $O(n^2)$ of space complexity. However, the most expensive step in the WCM is step 5. Matrix multiplication requires time complexity of $O(n*m^2)$ and space complexity of O(n). Generally speaking, compared with HMM, our

WCM is two orders of magnitude faster and requires an order of magnitude less memory. Thus, HMM is significantly more complex than the WCM model, which points in the direction of the necessity for design-time workload models. Next, we experimentally evaluate overhead. We experimentally model the overhead due to: (1) gathering global chip state, (2) evaluating and sending scheduling decisions, (3) bus contention, (4) cache memory misses, and (5) task migration.

The architecture used connects processors via a 32-bit wide 200-Mhz bus. In order to capture the effects of contention on the CLS overhead and overall system performance, the usage of the system bus was carefully modeled. Since the bus is designed to primarily move large continuous amounts of data between the processors and the memory, it operates within a burst mode. The burst mode allows a processor to gain bus access for multiple cycles, allowing the arbitration for the bus to occur much less frequently.

Our simulations perform bus arbitration every 50 bus cycles. The bus is used every time a streaming application is running by periodically sending out a burst of data to the processor that needs it. The bus is also used every time that one task completes its work and must pass on its output data to a task residing on a different processor. Finally, the bus is used every time the CLS must gather GCS or send the task mapping decision out from the CP. Bus contention not only affects the latency of individual tasks, it also interferes with the CLS. We take all of this into account when analyzing the CLS overhead.

Even though the scheduling task has the highest priority on the bus, it still has to wait up to 50 cycles for the current bus burst to complete. Since our bus operates in burst modes of 50 cycles, we make an assumption that one burst is enough to transfer the GCS of one processor to the scheduling master (50 cycles * 32 bits = 200 bytes). Therefore, the overhead of gathering or sending information for one scheduling decision (measured in processor cycles) is:

*((Number of processors * 50 bus cycles) + Bus contention delay of 50 bus cycles).*

To ensure fair arbitration, the bus uses a simple lottery-based arbitration strategy outlined in [142] which is implemented within a hardware arbiter. Scheduling decisions are given highest priority, since they do not use the bus for very long, but can be a bottleneck to other work being done in the system. Next is streaming applications, making them more likely to gain the bus in the case of contention. We additionally differentiate the phone call application as more important than other streaming applications, due to its high quality of service requirement. Finally, streaming applications have higher priority than job-based applications.

In MESH, global resource contention is modeled via annotations containing shared resource accesses, as discussed earlier in section 4.5 [124].

5.3.4 Workload Mode Sequence

First, we experimentally verify that workload mode sequence has impact on chip-level optimization. We ask whether the optimization of the system to a previous chip mode has an impact on its ability to identify and respond or optimize to a current workload mode. A major reason for using HMM is because it models the effect of sequence, and WCM is not useful if workload mode sequence must be included in the runtime model. We measure the overall response time of the system as performance-optimizing overhead. Optimization overheads are affected by the GCS value that includes the previous mode, or chip mode. Thus, we look at the CLS overhead caused by different workload mode sequences, for a total of 25 in all.

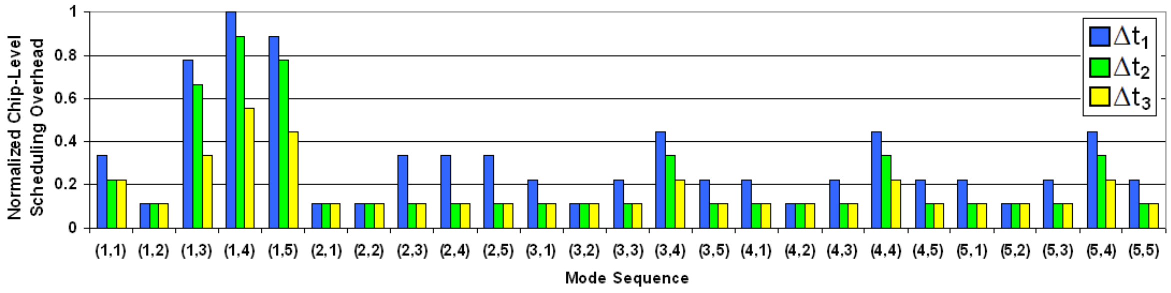

Figure 25: Chip-Level Scheduling Overhead

We chose five different workload modes from Figure 6. These workload modes represent five different workload categories: (a) concurrent job-based tasks (mode 1), (b) one job-based task (mode 12), (c) one streaming task (mode 4), (d) concurrent job-based and streaming task (mode 3), and (e) one job-based and one streaming task (mode 8). We selected a sequence length of two, which is the minimum required to show the impact of mode sequence on CLS overhead. This results in 25 different sequence combinations. Further, three different arrival times of inter-modes were chosen; one second (Δt_1), five seconds (Δt_2), and ten seconds (Δt_3). Finally, we ran these experiments using chip architecture 13 from Table 12.

Figure 25 shows the CLS overhead for the 25 different mode sequences, shown on the independent axis. On the dependent axis are the overhead values normalized to the sequence with the highest overhead. Figure 25 shows that:

- *Mode sequence affects scheduler overhead.* First note that the inter-arrival time of modes matters to the overhead required to optimize a multicore architecture from one mode to the next. For instance, consider mode sequence (1,3). Overhead decreases as the inter-arrival time gets longer. In general, longer inter-arrival times ($\Delta t3$) lead to less CLS overhead since longer inter-arrival times typically mean less execution overlap and thus there is less opportunity for the scheduler to optimize. However, mode sequence alone sometimes invalidates this first-order conclusion. For instance, consider modes 3 and 4. Here, optimization overhead decreases when inter-arrival time of mode sequence (3, 4) increases from five seconds to ten seconds, but it does not in mode sequence (4, 3).
- *Scheduler overhead is dependent on both task type and arrival time*. When new streaming tasks arrive to the system that is occupied by job-based tasks, the overhead incurred is larger because, in this case, arriving tasks have higher priorities than existing tasks resulting in preemption or even migration of running tasks and that incurs overhead and bus contention. An example is mode sequence (1, 4) in which a user started a phone call only one second after opening Gmail. This demonstrates that attempts to optimize require anticipation of how long the system is likely to remain in a specific mode, or else optimization can obstruct overall performance. This also demonstrates the impact of QoS requirements on chip-level optimization.

Since mode sequence matters, it would seem that a complex model must be used to identify modes, such as HMM. However, if design-time knowledge of architecture can be used to eliminate the impact of mode sequence, then it is possible that the simpler and potentially more efficient WCM model can be used.

5.3.5 Overhead and Performance

Table 13 and Table 14 show the average workload mode identification overhead for both identification models, WCM and HMM. Three different usage patterns were applied to these models. We look at both the time overhead in terms of simulation cycles and the space overhead in terms of bytes.

Table 13: WCM Identification Overhead

	WCM - Identification											
Arch.	UP1		UP2		UP3		UP4		UP5		Average	
	Time (Giga Cycles)	Space (Kilo Bytes)	Time (Giga Cycles)	Space (Kilo Bytes)	Time (Giga Cycles)	Space (Kilo Bytes)	Time (Giga Cycles)	Space (Kilo Bytes)	Time (Giga Cycles)	Space (Kilo Bytes)	Time (Giga Cycles)	Space (Kilo Bytes)
1	0.20	14.30	0.10	10.10	0.14	11.90	0.19	12.10	0.09	9.10	0.14	11.50
2	0.18	13.50	0.09	9.63	0.13	11.50	0.16	12.10	0.09	8.73	0.13	11.09
3	0.15	12.20	0.08	8.91	0.10	10.00	0.12	11.10	0.08	8.11	0.10	10.06
4	0.21	14.60	0.11	10.50	0.15	12.20	0.19	13.20	0.09	9.30	0.15	11.96
5	0.23	15.20	0.11	10.60	0.15	12.20	0.20	13.10	0.10	8.80	0.16	11.98
6	0.20	14.00	0.09	9.32	0.12	10.90	0.19	13.50	0.08	8.32	0.13	11.21
7	0.21	14.50	0.10	10.10	0.14	12.00	0.18	12.30	0.09	8.90	0.14	11.56
8	0.22	14.90	0.11	10.50	0.15	12.10	0.20	14.10	0.10	9.30	0.15	12.18
9	0.23	15.10	0.11	10.40	0.15	12.30	0.21	13.30	0.09	9.70	0.16	12.16
10	0.23	15.00	0.11	10.40	0.14	12.00	0.22	13.90	0.09	9.50	0.16	12.16
11	0.20	14.30	0.10	9.95	0.14	11.70	0.19	12.70	0.09	7.85	0.14	11.30
12	0.18	13.60	0.09	9.59	0.11	10.50	0.16	12.90	0.09	8.49	0.13	11.02
13	0.26	16.30	0.12	10.90	0.16	12.80	0.25	13.60	0.10	9.80	0.18	12.68
14	0.26	16.20	0.11	10.60	0.15	12.50	0.23	15.10	0.10	8.80	0.17	12.64
15	0.23	15.30	0.10	10.20	0.14	12.00	0.22	13.30	0.09	8.90	0.16	11.94
16	0.26	16.10	0.12	11.00	0.16	12.60	0.24	14.60	0.10	10.10	0.18	12.88

The simulation cycles consist of the bus and the CP cycles. Table 13 and Table 14 illustrate the amount of identification overhead variation between the two models for different usage patterns of our cell phone example executing on different architectures.

Table 13 and Table 14 show that the workload mode identification using WCM is, on average, 34 times faster than using HMM. Further, the identification using WCM is on average 83% more space efficient than using HMM. As a consequence of reducing the space overhead by 83%, the cache memory can also be reduced by 83%. Here, because the working data set of an identification model is called frequently, every time a workload mode starts, we assume that the working data set of an identification model fits into cache. Again, the selected cache area is four times an individual Media processor's area. For instance, three more Media processors can be included on the chip when using WCM. Thus, the system improves not only due to faster mode identification, but because there is more space left over to process the actual workloads.

Table 14: HMM Identification Overhead

	HMM - Identification											
Arch.	UP1		UP2		UP3		UP4		UP5		Average	
	Time (Giga Cycles)	Space (Kilo Bytes)	Time (Giga Cycles)	Space (Kilo Bytes)	Time (Giga Cycles)	Space (Kilo Bytes)	Time (Giga Cycles)	Space (Kilo Bytes)	Time (Giga Cycles)	Space (Kilo Bytes)	Time (Giga Cycles)	Space (Kilo Bytes)
1	9.97	99.90	2.47	49.80	4.86	69.70	9.82	98.10	2.23	46.70	5.87	72.84
2	7.89	88.80	2.06	45.40	4.16	64.60	7.71	87.70	1.90	44.30	4.74	66.16
3	5.39	73.50	1.51	38.90	2.43	49.30	5.13	72.20	1.33	37.80	3.16	54.34
4	10.90	104.00	2.90	53.90	5.24	72.40	9.10	91.00	2.64	52.70	6.16	74.80
5	12.90	114.00	3.00	54.80	5.39	73.50	11.10	100.00	2.83	53.30	7.04	79.12
6	9.36	96.80	1.80	42.50	3.37	58.10	9.19	94.70	1.63	41.30	5.07	66.68
7	10.60	103.00	2.47	49.80	4.93	70.30	8.80	103.00	2.33	47.70	5.83	74.76
8	11.70	108.00	2.90	53.90	5.16	71.90	10.10	97.00	2.57	52.10	6.49	76.58
9	12.40	112.00	2.85	53.40	5.55	74.60	11.30	99.00	2.58	52.60	6.94	78.32
10	12.30	111.00	2.85	53.40	4.93	70.30	10.20	94.00	2.66	51.70	6.59	76.08
11	10.10	101.00	2.34	48.40	4.57	67.60	8.70	89.00	2.17	46.60	5.58	70.52
12	8.21	90.60	2.02	45.00	2.90	53.90	8.03	89.00	1.91	44.10	4.61	64.52
13	17.00	130.00	3.37	58.10	6.51	80.70	14.30	117.00	3.12	57.50	8.86	88.66
14	16.80	130.00	3.00	54.80	5.80	76.20	15.10	121.00	2.81	52.90	8.70	86.98
15	13.20	115.00	2.61	51.10	4.93	70.30	11.70	101.00	2.39	49.80	6.97	77.44
16	16.10	127.00	3.49	59.10	6.15	78.40	14.40	109.00	3.29	58.20	8.69	86.34

Table 15 shows the optimization overhead in terms of simulation cycles for both models. Again, we consider the same usage patterns and architectures in Table 12. The results in Table 15 follow the same results trend in Table 13 and Table 14.

WCM shows consistently less optimization overhead than HMM through all architectures and usage patterns. In our cell phone example, using WCM decreases the optimization overhead by on average 49% compared with HMM. While the optimization overhead is not directly related to the identification models, the higher identification overhead results in workload mode overlap. This, in turn, results in greater GCS and thus more time to gather, process, and schedule. In other words, the identification process is the input to the optimization process.

Table 15: WCM vs. HMM Optimization Overhead

Arch.	WCM Optimization Overhead (Mega Cycles)					HMM Optimization Overhead (Mega Cycles)				
	UP1	**UP2**	**UP3**	**UP4**	**UP5**	**UP1**	**UP2**	**UP3**	**UP4**	**UP5**
1	0.342	0.89	0.1	0.502	0.729	0.513	0.133	0.15	0.42	0.087
2	0.444	0.167	0.192	0.213	0.107	0.666	0.251	0.289	0.541	0.193
3	0.612	0.304	0.336	0.58	0.266	0.918	0.457	0.504	0.788	0.343
4	0.413	0.146	0.17	0.377	0.104	0.619	0.22	0.255	0.509	0.156
5	0.389	0.118	0.137	0.308	0.089	0.584	0.177	0.205	0.489	0.123
6	0.549	0.234	0.274	0.491	0.111	0.824	0.351	0.411	0.731	0.299
7	0.449	0.169	0.201	0.399	0.156	0.674	0.253	0.301	0.601	0.186
8	0.501	0.205	0.237	0.436	0.178	0.751	0.307	0.355	0.633	0.247
9	0.549	0.233	0.276	0.501	0.143	0.824	0.35	0.414	0.756	0.278
10	0.588	0.262	0.301	0.521	0.201	0.882	0.394	0.452	0.693	0.303
11	0.599	0.278	0.328	0.526	0.213	0.898	0.417	0.492	0.802	0.378
12	0.605	0.294	0.322	0.517	0.243	0.908	0.442	0.483	0.872	0.389
13	0.505	0.183	0.215	0.433	0.142	0.758	0.274	0.323	0.645	0.204
14	0.548	0.207	0.244	0.488	0.167	0.822	0.31	0.366	0.732	0.267
15	0.557	0.228	0.268	0.478	0.199	0.836	0.342	0.401	0.799	0.277
16	0.455	0.153	0.177	0.403	0.107	0.682	0.23	0.265	0.61	0.149

This raises the possibility that the architecture can have an impact on the method used to identify modes, that the architecture can permit the use of the simpler WCM model because it can respond to individual modes faster and thus eliminate the impact of sequence. We examine this next.

5.3.6 Overall Performance

We applied both identification models (WCM and HMM) individually to three different usage patterns (workloads) executing on different architectures of Table 12. Figure 26, Figure 27, and Figure 28 show the normalized average values of the average response time, energy consumption, and overhead of the five usage patterns for all architectures in Table 12. The normalized value is calculated by dividing each value in a single performance metric (overhead, response time, and consumed energy) by the highest value in the same metric. Thus, lower normalized response time, consumed energy, and overhead values mean lower actual values. Consumed energy is calculated as the summation of power consumed by each processor during active and idle time [123].

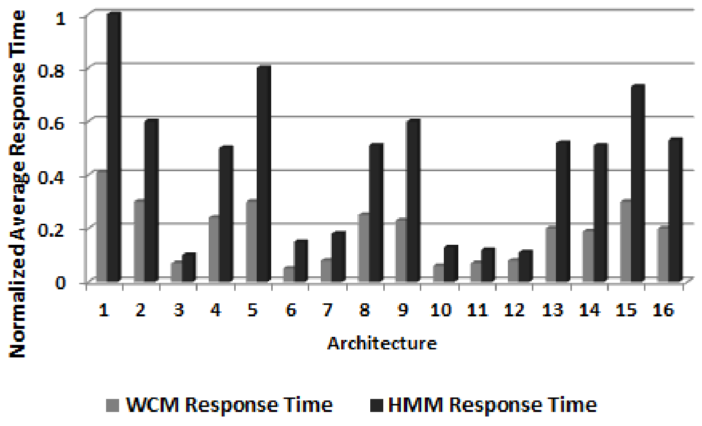

Figure 26: The Normalized Average Response Time

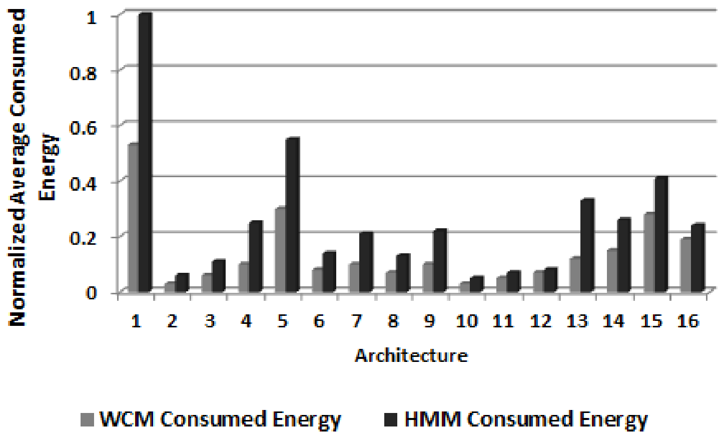

Figure 27: The Normalized Average Consumed Energy

Optimal use of the WCM model requires the ability to leverage design-time knowledge of the individual usage patterns, so that a sequence-independent model can be used. However, we found that even sub-optimal use, when WCM does not identify all optimization points, can outperform the use of the more costly HMM. Furthermore, we found that different architectures performed significantly different so that future single chip multicore architectures should be designed to models of design-time single-user workloads.

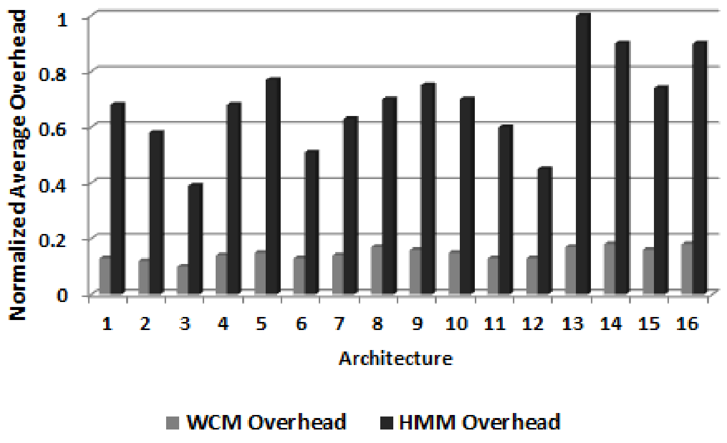

Figure 28: The Normalized Average Overhead

On average, over all architectures, the use of the WCM decreased application response time by 45% while overall system performance improved 191%. While the maximum system performance achieved is 256% by architecture 12, the minimum is actually 158% achieved by architectures 5, 9, 13, 14, 15, and 16. Further, the energy consumed is decreased by an average of 56%. The maximum energy decrease achieved is 82% by architecture 12. The minimum is 42% achieved by architectures 13 and 9.

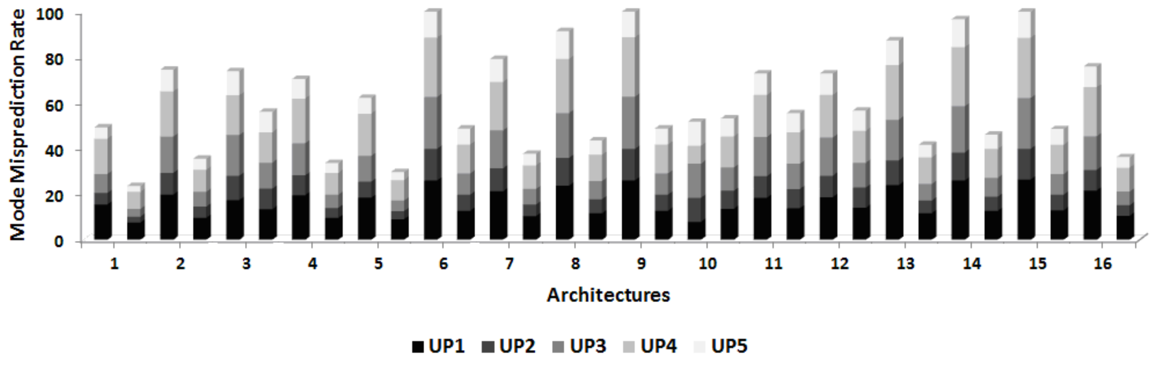

Figure 29: Workload Mode Misprediction Rate

Next, we investigate more insight about the relationship between overhead and performance. The time overhead introduced by switching between different optimization profiles may cause undesired side effects in a system, such as degrading the system throughput, the amount of work done between two consecutive workload modes may negatively impacted due to the time overhead needed to switch to another optimization profile such as frequency/voltage scaling of processors.

Figure 29 shows the workload mode misprediction rate of the WCM and the HMM identification models of the five usage patterns for all architectures in Table 12. The first column is for the WCM and the second column is for the HMM. The misprediction rate value is calculated by dividing the number of identified workload modes by the total number of workload modes. Note that each architecture has different total number of workload modes. As can be seen from Figure 28, there is a difference in misprediction rate between the usage patterns and between architectures. The second usage pattern (UP2) has the lowest misprediction rate among the five usage patterns. This is because UP2 has relatively larger average inter-arrival time intervals between workload modes and workload modes are more regular in their arrival, giving more time to identify these workload modes. Further, usage patterns with fewer workload modes make identification faster and more accurate.

The computing power of different architectures is also a significant factor in the accuracy of the identification model. This, in turn, also has a significant impact on the performance of usage patterns. The computing power of architectures defines the interval between workload modes as well as the speed of executing the identification model. Note in Figure 29 that architectures 5 and 10 have relatively lower misprediction rates in all usage patterns. Comparing this observation with the results in Figure 26, Figure 27, and Figure 28, it shows that the performance of these architectures is relatively better than other architectures, though this is not general. For instance, note that architecture 6 has a higher WCM misprediction rate while it results in better performance values.

For the HMM however, the reduction of the misprediction rate is quite substantial. On average, the HMM identification model can reduce the number of mispredictions by more than 46%. Figure 29 also shows that our WCM model does not perform well compared with the HMM model for many usage patterns, while it results in better performance. The reason for this is that the HMM improves the prediction rate at the expense of applying optimizations by requiring more time for identification. Thus, the sup-optimal WCM is more efficient than HMM.

Another metric that can be used to compare the two models is cache misses. Our MESH simulator can be used to identify and compute cache misses. In order to do this, we need first to use an instruction set simulator (ISS) to generate full memory address traces. These traces are used to generate metrics that describe the memory behavior of individual program fragments. These metrics are annotated into the original source code, which is executed on the hardware architecture using the MESH simulator. The simulator can then find the cache hits and misses for each architecture when used to execute heterogeneous concurrent applications. This approach is introduced by Pieper et al. [143]; it is a derivative of stack distance histograms. Prior work [143] shows that MESH has a simulation time two orders of magnitude faster than cycle-accurate simulators and can accurately model cache configurations in CHMs.

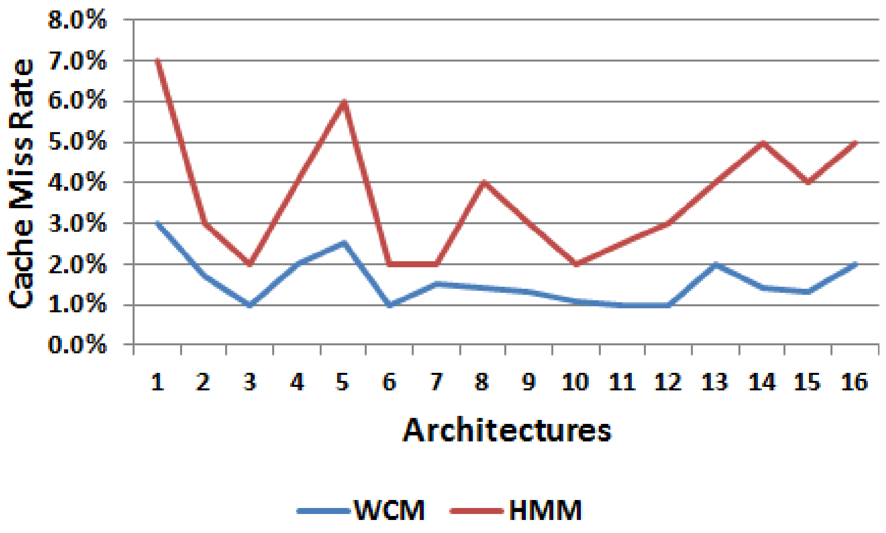

Figure 30: Cache Miss Rate

Figure 30 shows the average cache miss rate of the WCM and the HMM identification models of the five usage patterns for all architectures in Table 12. The cache miss rate is calculated by dividing the number of cache misses to the total number of memory references. Note that Figure 30 shows that the WCM model, on average, performs better than the HMM-based model. This is because the HMM requires more memory and this, in turn, results in more cache misses, the relationship between the data size and cache misses. Intuitively, note that Figure 30 shows that architectures with lower cache miss rates perform better (see Figure 26, Figure 27, and Figure 28).

5.4 Analysis

Overall, the costs of the HMM could not be overcome because HMM is, on average, 36 times more costly than WCM. We found that it is better to miss optimization points using WCM rather than to incur the cost of finding all possible optimization points using HMM. Given our experimental results, in this section we expand our scalability analysis of each approach by including the impact of CHM architectures. We also verify our analysis.

Heterogeneity is desirable because it can be used to exploit differences in task types, potentially retiring individual tasks within a workload mode sooner. However, completion of a task within a workload mode also presents the possibility for optimization of the CHM for the remaining applications in the mode. Heterogeneous architectures result in more potential optimization points within a workload mode than homogenous architectures due to a much wider variety of task completion times. Thus, the impact of executing WCM vs. HMM on a CHM can potentially be even greater than the Big-O analysis predicts because of the need to identify internal optimization points within workload modes.

We define candidate workload modes (N), which arise due to both external changes in the input stream and internal events on a given architecture as:

$$N = \sum_{i=1}^{N^{UP}} \rho_i$$

in which N^{UP} is the number of workload modes in a usage pattern (*UP)* and ρ_i is the number of internal events within a workload mode that arises due to the ability of the hardware to process the mode. For example, the UP in Figure 1 has 12 modes, but the execution of that usage pattern on different CHMs can result in different numbers of candidate modes.

We observe that different higher degrees of multitasking, multiprocessing, and heterogeneity tend to be generational, and our analysis is focused on the impact of workload modes on computing as it moves into the next era.

We begin with a homogeneous multiprocessor system that processes sets of heterogeneous workloads. This can be considered present-day computing, in which multi-core personal computers process multimedia applications from the Internet, in which homogeneous groupings can be exploited. Each group of homogeneous tasks within application sets represents internal optimization points. Thus, for workload mode *i* in this system, $\rho_i = H_i$, in which $H_i > N^P$. Here, N^P is the number of processors on chip and H_i is the number of groups of homogeneous tasks within the workload mode *i*. The completion of each task grouping represents the potential to optimize remaining tasks within the workload mode.

Next, we consider a heterogeneous workload that is presented to a CHM system. Note that this type of system, which many consider to be the future of computing [144], results in far more candidate workload modes due to the heterogeneity in the architecture. However, this same heterogeneity can also benefit overall potential performance while satisfying space and power constraints [49]. Here, all tasks and all processor resources have the potential to be heterogeneous, even though some subsets of tasks and processors may be homogeneous. Still, the completion of any single task in the system can create the potential to optimize. For example, a group of homogeneous tasks may be scheduled to execute on a single processor in sequence instead of waiting for an array of processors. Tasks may be migrated to better performing resources when they become available, even when there are fewer tasks in the system than available processor resources. Thus, ρ_i, in the worst case, equals all of the tasks in a mode, times all of the processors in the system; $\rho_i = N_i^T * N^P$, in which N^T is the number of tasks within a single workload mode.

Note that as the number of processors in a CHM increases, overhead is expected to increase with heterogeneity due to increased potential optimization points, but also decreases if there are more processors overall, as N_i^T decreases if more processing power can retire tasks sooner. Thus, we expect to see overhead increase with different classes of heterogeneity, but reduce slightly as the number of processors increases. We also expect this effect to be more dramatic for HMM than for WCM, since HMM is already orders of magnitude more complex than WCM, irrespective of the internal optimization points that arise due to heterogeneity of the CHM.

We computed the number of candidate modes for each architecture in our experimentation, and compared the predicted overhead of both the WCM and HMM models. We classified architectures by degrees of heterogeneity and ordered them by the total number of processors within those classes. The results are shown in Figure 31 and Figure 32, which compare the predicted and measured overhead trends of WCM and

HMM. We grouped the architectures in Table 12 on the independent axes as A, B, and C: (A) architectures are fully homogeneous (one processor type), (B) architectures are semi-heterogeneous (two processor types), and (C) architectures are fully heterogeneous (all three processor types). Note that architectures of each group are rearranged to show the increase in the number of processors. The overhead is expressed in orders of magnitude. Our estimations focus only on the usage pattern shown in Figure 6.

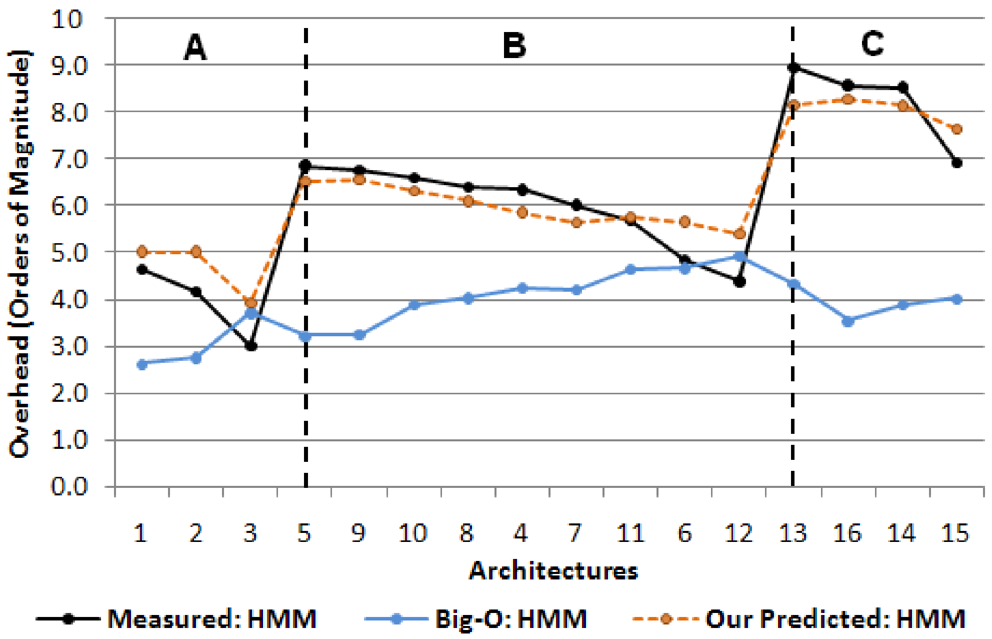

Figure 31: Measured, Analytical and Our Predicted HMM Overhead

Figure 31 and Figure 32 confirm that our analysis that includes internal modes is more accurate than the Big-O analysis of section 5.3 that considers only external workload modes. As architecture heterogeneity increases (from class A to class C), the incurred overhead when using HMM increases exponentially, while WCM overhead increases only linearly. It also confirms that as the number of processors increases within a group, overhead decreases since the number of internal optimization points decreases as task groups are retired sooner. Compared with the measured overhead, our prediction model has only a 4% margin of error, while the Big-O model has 36%.

Overall, we note that as computing moves toward CHMs with large numbers of processors of many different types, there exists a compelling need to design to models of single-user workload modes.

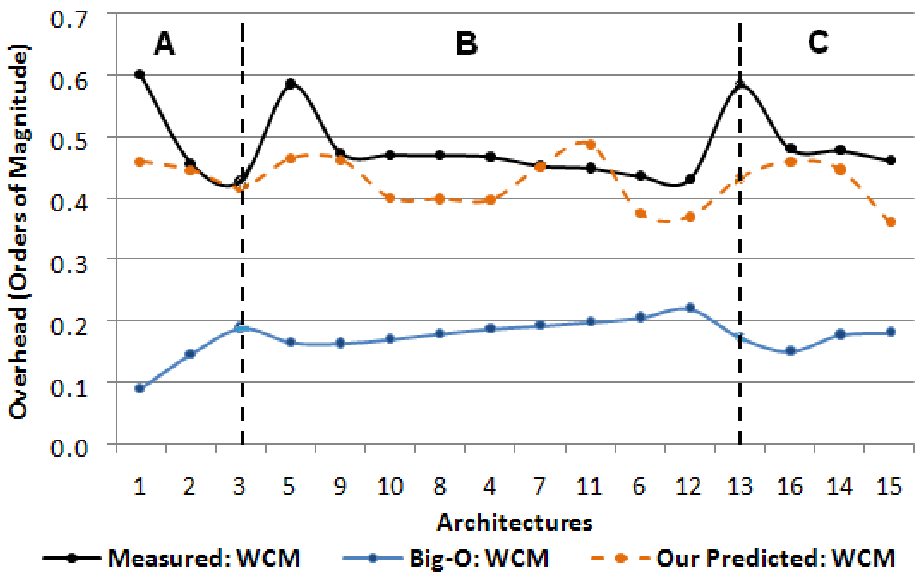

Figure 32: Measured, Analytical, and Our Predicted WCM Overhead

5.5 Summary

As we move into the next era of computing in which the complexity of applications and architectures grows, computers need to be optimized to workload modes that arise from single-user usage patterns. In this chapter, we aimed at designing WSPs by defining and identifying workload modes and optimizing CHMs to these modes at real time. Toward that end, we faced the following key challenge: which design is better, given the processing times of all workload modes executing on each design? Therefore, there was a need to rank these designs in order to identify WSPs. For that purpose, we used traditional single-valued performance metrics (specifically, response time and power consumption) to compare the performance of different designs.

Here, we summarize some observations that put our optimal design ranking in question:

1. Interestingly, for the same architecture, increasing the demand for one task type in a workload mode may result in a better overall performance. This is because the overhead generated by this specific combination of tasks (arrived during a specific time and sequence in the usage pattern) is smaller. The implication of this is that by using single-valued metrics, we preclude these effects from the designer. The overhead of identifying and processing workload modes has the

most interesting observations in this chapter. There are several factors that contribute to the magnitude of overhead. The factors that are related to the type of workload modes are mainly the degree of concurrency (number of tasks), the degree of heterogeneity (types of tasks within a workload mode), the granularity of workload modes arrival time, and the sequence of workload modes.

2. Further, for the same architecture, different optimal design ranking results when processing different workload modes. Thus, by using single-valued metrics, we factor performance differences of different workload modes out. This also means not just that the best performing design may not be the best performer for all workload modes in a usage pattern but, more interestingly, may not meet all of them within the specified time interval.
3. While we used the geometric mean to come up with a single-valued score, we expect the other mathematical means to result in additional different optimal design rankings due to the fact that performance varies as different workload modes are being processed, as discussed in Chapter 2. This variation occurs not only because CHMs are specialized for different types of workload modes, but more significantly because of the different values of overhead incurred due to the arrival of a different combination of work types.
4. When it comes to architectures, multiprocessing, heterogeneity, and other architectural features such as communication and cache sizes all of these define the magnitude of overhead.

The most interesting question to ask is: which design feature leads to better or worse performance? Again, single-valued metrics preclude this cause and effect analysis from the designer. It also precludes the effects of the interactions of multiple architectural features.

In the next chapters, we use our Capacity metric to identify these WSPs. In Chapter 6, we define our Capacity metric, and in Chapter 7 we show how and why the optimal design ranking of the architectures in this chapter is inaccurate in contrast with our Capacity metric.

Chapter 6

Capacity Metric

Be sure you put your feet in the right place, then stand firm.

Abraham Lincoln

In order to properly evaluate and rank CHM designs that process multiple, heterogeneous channels, the main contribution of this thesis is the introduction of a descriptive, graphical, and denotational definition of our Capacity metric. We previously introduced the Capacity metric by analogy with automobile plant production. Here, we contrast our Capacity with Pareto optimization and show some basic forms of Capacity curves. Thus, we motivate the continued development and use of the Capacity metric for performance evaluation of modern CHMs. Specifically, we advocate investigation into how shapes of Capacity curves can be used to classify systems and identify how features of designs can be manipulated in order to change the shape of the Capacity curves.

6.1 Definition

Capacity is a surface that shows the production feasibility of combinations of different types of outputs for a given plant over some interval of time. When two types of outputs are considered, Capacity results in a curve. If the production of the different types of outputs does not vary with each other over time, then the Capacity curve results in a collection of rates of production. The Capacity graph shows the maximum amount of one type that can be obtained for any specified production level of the other type(s), given the resources available as well as the way the different production types compete for those resources. Mathematically, a Capacity curve can be defined as a function in two variables:

$$\{T|A \textit{ or } A|T\text{: } P(\tau)\}$$

in which T is the production variable (or function) of Model-T, A is the production variable (or function) of Model-A, P is a specific production plant design, and τ is a given time interval in which we measure Capacity by collecting the production combination of different output types: Model-T and Model-A. Note that the production of Model-T, T, is given by the value of the production of Model-A, A, and vice versa. Later in this section, we define the function for a Capacity curve.

According to this mathematical denotation, Capacity curves are generated for a specific plant design during a time window using measurement. In other words, the production of one model type is measured for different levels of production of the other model type(s) for the same design. The measurement includes the overhead of sharing resources between different output streams.

6.1.1 An Illustrative Example

For illustration, consider the plant example in section 1.1. Assume that the Model-T assembly line can produce four Model-T automobiles per hour, and can also be used to produce only two Model-A automobiles per hour. The Model-A assembly line can produce six Model-A automobiles per hour, and can also be used to produce only two Model-T automobiles. We generate all production possibilities of combinations of the Model-T and Model-A outputs for this plant. Figure 33 shows the Capacity of this plant during a time window of one hour in the form of a curve.

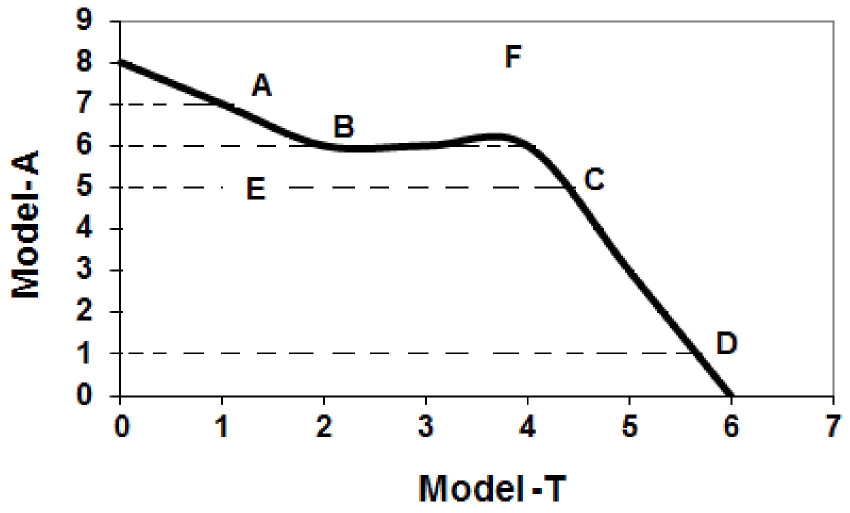

Figure 33: A Capacity Curve

A Capacity curve shows all possible combinations of two automobile models that can be produced simultaneously during a given time window. For a plant to increase the quantity of one model produced, production of the other model must be reduced. Here, production of automobiles of Model-A must be reduced in order to produce more of Model-T. Capacity represents how much of the Model-A must be reduced for a given increase in production of the Model-T.

Assuming that the supply of the plant does not increase, making more Model-Ts requires that resources be redirected from making Model-A to making Model-T. All points on the Capacity curve are efficient. Hence, all points on the curve (such as A, B, C, and D) are points of maximum productive efficiency. In other words, no more output can be achieved from the given inputs. All points inside the curve (such as E) are feasible but productively inefficient, and all points outside the curve (such as F) are unfeasible with the given resources and time window and thus unattainable. Note that point E is dominated by point A or point C, whichever is the model of interest. The user can choose between combinations or points on the Capacity curve. Point A is when Model-A is prioritized, point D is when Model-T is prioritized, in the middle of the curve (such as point C) an intermediate mix is achieved, and so forth.

The example used above represents one form of Capacity. It represents a disparity in the heterogeneity of assembly lines in producing the two models. That is, as a plant specializes more and more in one model, the cost of producing that model increases because we are using more and more resources that are less efficient in producing it. The cost of producing successive units of the Model-T will increase as resources that are more and more specialized in for Model-A production are moved into Model-T production. If costs are constant, a straight-line Capacity curve is produced. This case reflects a situation in which assembly lines are not specialized (or homogeneous in their production of different models) and can be substituted for each other with no added cost. Products requiring similar resources will have an almost straight Capacity curve, and therefore almost constant costs. The Capacity curve would appear bowed toward the origin, with costs falling as more is produced of each respective model. Here greater specialization in producing successive units of a model drives down its cost. Later on in this paper, we investigate some of Capacity forms in computer systems.

Therefore, Capacity curves can represent how assembly line specialization that favors production possibilities of one model, assume the Model-T, shifts the curve more toward the Model-T axis, biasing production possibilities in that direction. Similarly, if one model makes more use of one assembly line and if the Capacity of that assembly line improves faster than other assembly lines, improvement possibilities might be biased in favor of that model.

If the productivity or supply of assembly lines increases, the plant's Capacity to produce both models increases. This increase ideally results in an outward Capacity curve. Conversely, a limitation in the production of the assembly lines, like the lack of supply in inventories, might move the Capacity curve inward, reflecting a reduction in a plant's total productive Capacity. In the experiments section, we find more relationships between design features and Capacity curves.

6.1.2 Capacity Function

The reduction in the production of a given model represents a cost. This cost is measured in the ratio of the number of units of the second model reduced for the production of one or more units of the first model. In the context of Capacity, cost is directly related to the shape of the curve. Unless the curve is a straight line, the cost varies all along the curve. In Figure 33, producing more units of the Model-T, in general, results in a reduction of the number of units of the Model-A produced. Interestingly, at point B, the plant can produce more of the Model-T without any reduction in the production of the Model-A; thus the cost is zero. Therefore, Capacity represents how much of the Model-A must be reduced for a given increase in production of the Model-T. This ratio of costs can be determined using curve slopes at different points on the curve.

As shown in Figure 33, as the production of Model-T rises (or falls) the production of Model-A may fall (or rise). The production of one variable may determine the production of the other, but there is no explicit formula for one in terms of the other such that A=*f*(T) *or* T=*f*(A). This is because of the fact that our definition of Capacity may result in a multi-valued function. Thus, the Capacity curve can be represented by an implicit function:

$$f(T,A) = 0$$

The curve described by f(T,A) = 0 has tangent line at the point (*ti*, *ai*) given by the equation $A = a_i + A'(t_i)\ .\ (T - t_i)$, in which $A'(t_i)$ solves the equation given by differentiating the original equation with respect to T, treating A as a function of T and using the standard rules, and finally substituting $A = a_i$, $T = t_i$.

The slope, $A'(t_i)$, at a point (t_i, a_i) is given by the partial derivative of the one model production with respect to the other model production. The slope numerically describes the rate at which output of one model can be transformed into output of the other. Again, unless the curve is a straight line, the rate varies all along the curve resulting in a multiple-valued function. Capacity curves are generated by measurement. But with sufficient experiments using them to model systems, they also have the potential to form the basis of analytical analysis. One reason for the variable cost rate along the curve is that heterogeneous resources include quantization effects such as workers being unable to finish a product in the remaining time of their shift. Further, integrating different quantities of different model types onto heterogeneous production resources incurs a different amount of overhead that impacts the measured production due to starvation that arises because of the existence of other pipelines sharing the same inventories. Interestingly, Capacity determines the cost of integrating more of one model type on the production of the other model type(s), which is something that cannot be displayed using a single performance score. Various numerical methods exist for solving implicit functions. Later on in this chapter, we introduce our own analysis.

Here, we used only a two-model example for simplification, necessary for graphical analysis. If one model is of primary interest, all others can be represented as a composite model. In addition, the example can be generalized to the *n*-model case using n-D analysis. The tradeoff between the production of different types of commodities in an economic system has been investigated using Pareto optimality [145]. Next, we contrast our definition of Capacity with the concept of Pareto optimality.

6.2 Capacity vs. Pareto

We show how Capacity is different than Pareto using the same automobile plant example. We first define Pareto optimality for the above automobile plant example consisting of two assembly lines, L_T and L_A, which produce two automobile models, Model-T and Model-A. The outputs of the two automobile models that may be produced by the plant:

$$K_A = A(L_T^A, L_A^A)$$
$$K_T = T(L_T^T, L_A^T)$$

in which L_i^j is a pipeline originally designed to produce automobiles of type *i* but can also be used to produce automobiles of type *j*. A and T production functions can be simply thought of as the allocation of the assembly lines to produce different quantities of different model types and are assumed to be increasing. In other words, allocating more assembly lines to produce specific model types increases the production of that model.

Given an initial combination of output models, a change in the allocation of input models to resources that makes at least one output model better off without making any other outputs worse off is called a Pareto improvement. A combination is defined as a Pareto efficient or Pareto optimal when no further Pareto improvements can be made. Thus, for the production of the two automobile models to be Pareto efficient we require that we cannot increase the production of one model such that more of one is produced without giving up some of the other. This can alternatively expressed as:

$$\mathit{Max}\ A(L_T^A, L_A^A)$$
$$\text{such that } T(L_T^T, L_A^T) \geq T^*$$

in which T^* is the level of production of that model which must not be reduced. This is a multi-objective optimization problem that can be solved mathematically using the method of Lagrange. By doing this, we get from this analysis that the slope of both curves, A and T, must be equal at a Pareto efficient point. Each curve represents a relationship between the production of one type and an optimization function such as utility of labor or capital. Note that in order to achieve the maximum of function A, T^* should be zero; this is the definition of throughput in the assembly line production of automobiles in which individual production types are evaluated, resulting in a single unit of performance. This is analogous to performance evaluation of computer systems in which computer architects evaluate performance using parallel programs. Each program can be multithreaded but they run individually and so their overall rates of execution are evaluated. Similarly, microarchitects identify the maximum throughput, usually using a common work unit, such as instructions per cycle. These maximums can be used to compare different architectures or an average value can be generated. In contrast to our Capacity metric, throughput is distilled into single score values using a common work unit. The effects of combining different production types on the same plant and cause and effect analysis are missing from throughput. Further, throughput is presumed to be invariant over arbitrary time intervals. In contrast, Capacity surfaces can change with interval size as the relationships between types produced by a given plant can change with the period of time over which production is evaluated.

For illustration, consider the plant example in Figure 4. We use the same fixed plant resources to generate an alternative configuration. Here, assume that the production of Model-T assembly line has been improved by deploying more workers to produce six Model-T automobiles per hour, and can also be used to produce only two Model-A automobiles per hour. Since the number of workers is fixed, the Model-A assembly line can now produce only four Model-A automobiles per hour, and can also be used to produce only two Model-T automobiles.

Figure 34 portrays the differences between Pareto efficiency and our Capacity metric. Figure 34(a) shows the Pareto curve as a relationship between design configurations such as the production capabilities of individual assembly lines and the production of the two automobile models. Each point on the curve is Pareto efficient and represents a different plant configuration. Therefore, the performance of each plant design is distilled into one single point (K_A, K_T). Note that the performance metric in Pareto is automobiles, so this point can be averaged in order to come up with a single score for each plant design. Interestingly, since each design is represented by only one single production value, the output curve is concave, assuming the gradual re-allocation of workers from Model-T to Model-A results in a gradual decrease in the production of Model-T while Model-A increases. Figure 34(a) also shows the mechanism behind Pareto optimality. It shows that each model type is individually produced by plant configuration, *Pi*, and the output is represented using a common type that is automobiles.

In contrast, Figure 34(b) shows Capacity curve as a relationship between the production of the two automobile models. In this case, each plant configuration is represented by a separate curve. Figure 34(b) includes all production possibilities of the same design for two different plant configurations. Note in Capacity curves, different sets of production possibilities or modes have different magnitudes and curve types: convex, concave, or a straight line, in contrast to the Pareto production curve in Figure 34(a), which shows a regular shape over all production combinations. This is due to the fact that Capacity includes all production possibilities in the same design. It includes the cost of increasing the demand for one model type on the production of the other model type(s). And this requires that both model types simultaneously compete for plant resources, as shown in the bottom part of Figure 34(b).

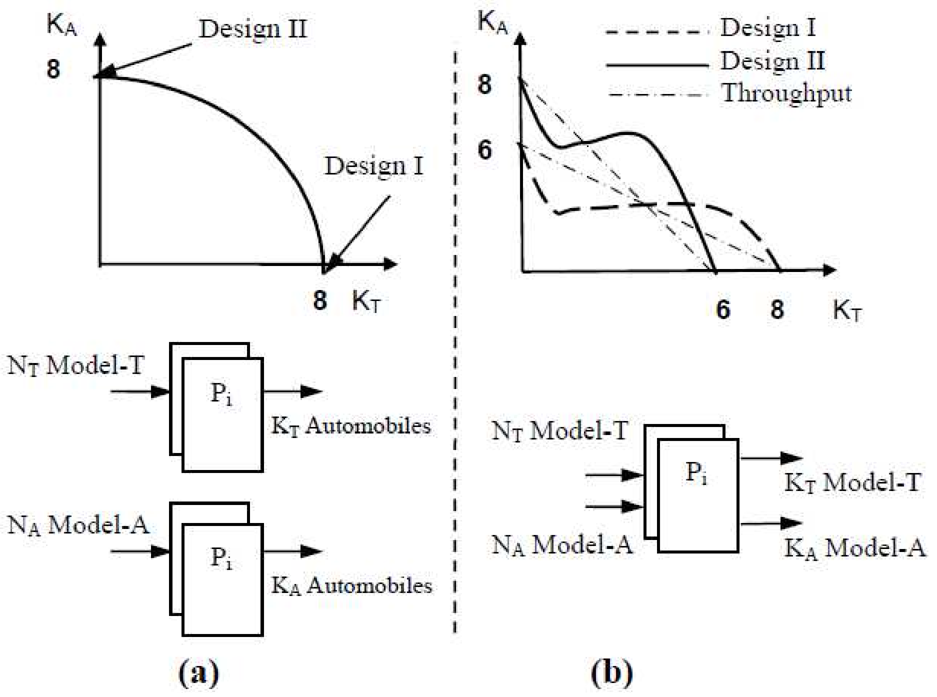

Figure 34: (a) Pareto Optimization vs. (b) Capacity

Also note that on the curves of Figure 34(b) we contrast Capacity curves with throughput. Throughput is shown as a straight line (a plane in higher dimensions). It connects the maximum production points of individual models. This straight line represents the average production of this plant over a presumed common work unit for the heterogeneous production types. For example, the unit "automobiles" could be used instead of specific units for Model-T and Model-A. This average results in a linear relationship as production of Model-T and Model-A are varied. This correctly models the production of completely independent channels in a decoupled system, but it does not capture the relationships of different production types as they access the common (shared) plant resources. In the experiments section, we show many directly measured Capacity curves that do not result in linear relationships, as demand for different production types varies. Accordingly, for a given set of production values, any linear relationship will either underestimate the true Capacity of a given plant, or it will overestimate what the plant is capable of.

In summary, Capacity includes the relationship between different production types that can be produced by a given plant or design. Thus, it tells which design is better at what range of production possibilities (or modes) and by how much. For example, the production of Design I is two times that of Design II when an intermediate mix of Model-Ts and Model-As is needed. If the production of the Model-T is prioritized, Design I achieves 33% more production than Design II, while Design II achieves 33% more production when the Model-A is prioritized, whereas Pareto models the impact of (design) variables on what is presumed to be a prefixed relationship between models being produced. Thus, it only tells which design is better at a specific combination of production types. Further, it cannot answer the question: can this plant satisfy the demand of N_T Model-Ts and N_A Model-As in so many days, since it does not show the breakdown of performance for different combinations of multichannel input streams?

Further, Capacity includes the effects on production of overhead, or resource sharing, as different production types compete for resources. Thus, Capacity curves lend insight into the cost of combining different production types on a common set of resources. Capacity shapes reveal the interaction of not only programs and data, but the interaction of multiple data streams as they compete for access to resources on a CHM. Eventually analytical techniques will be developed so that designers may better understand the origins of those inefficiencies and how designs may compensate for them. Next, we develop some fundamental forms of the Capacity metric in order to illustrate how Capacity shapes can be used to better understand design for multi-type production.

6.3 Capacity Form

In this section, we introduce some fundamental forms of the Capacity metric. Ultimately the Capacity metric will have its greatest potential when Capacity curves and surfaces can be developed and compared analytically. Consider some fundamental forms of the metric in Figure 35, in which Capacity measures the performance of two demand streams, or channels, JPEG and, text processing, which might represent elements on webpages being processed by a single user on a mobile device. On the independent axis is the number of JPEG images (JPEGs) to be processed and on the dependent axis is a number of text files. Note that the dependent and independent axes are interchangeable, one production, *P*, can be considered to be a function of the other, or *P(text)=f1(P(JPEG))* and *P(JPEG)=f2(P(text)).* The functional relationship between the maximum processing potential of different input channels describes the Capacity of a CHM. We describe some idealized forms that this functional relationship can take on for single and dual core CHMs that process the two channels of text and JPEG.

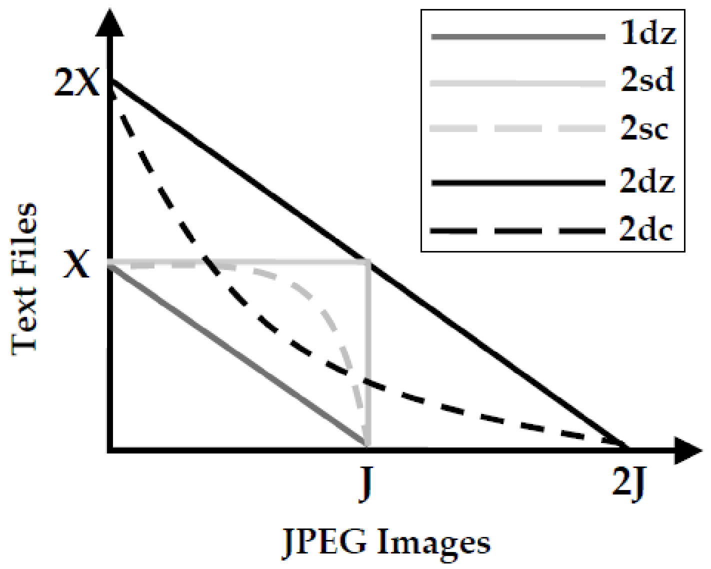

Figure 35: Fundamental Forms of Capacity Curves

First, consider the Capacity of a single core (*uP*) system that ideally processes unlimited combinations of text and JPEG; there are no quantization effects and there is no penalty for switching from processing one type of input to the other. The Capacity of this ideal *uP* is shown on Figure 34 as *1core_dynamic_zero_overhead (1dz)*, which is a straight line, formed between two maximums. The maximum of each type is the throughput when the core is always processing only that type. The production of the two demand streams on the idealized single core design is related by a constant, *P(text) = C*P(JPEG)*, in which *C = max(JPEG)/max(text)= J/X*. This constant-value relationship is a consequence of the idealized resource sharing and quantization effects as the system switches between processing different combinations of inputs.

Next, consider a homogeneous two-core processor in which tasks are mapped to restricted cores, for example, one core processes only text and the other processes only JPEG images, and this processing is completely decoupled. Figure 34 shows the Capacity for this system, *2core_static_decoupled (2sd)*, as two straight lines that are parallel to each axis and terminate at a point, *(max(JPEG), max(text)) or (J,X).* This point is the maximum production of each core for each input type, which is the same as the processing potential of the single core on that type in the absence of any demand for processing of the other type of input. The lines are independent of each other because there is no relationship between the production of JPEG and text because each has 100% of its own processor resource and there is no sharing of any resources in this ideally decoupled system. This system has exactly double the Capacity of the single *uP* system. It has two cores of the same type compared with the single *uP* system and the total area in the box doubles that of the straight line of the single core system. In each of these systems, throughput can be used to model the essential features of the overall system Capacity. In the *1dz* system, the two streams can be combined and averaged. In the decoupled system, throughput can be used to model the maximum production of completely independent channels. The Capacity of real systems, however, must include the effects of sharing resources such as communications networks, schedulers, and memories as multiple cores come together to form a system.

A more realistic Capacity curve is *2core_static_coupled (2sc)*, in which input channels are presumed to be statically mapped to processor resources, but there is some cost due to resource sharing. This shape asymptotically approaches the ideal boxed area of the two-core decoupled system of *2sd*, and the measure of the penalty, or overhead, due to coupling is captured in how far the curve deviates from the ideal boxed shape. Several works in communication systems address the identification of the curve knee [146]. Resource sharing can be considered to be analogous to communications sharing. Processing of one channel type is a function of the actual demand for the processing of the other channel type, however, in this case, the relationship does not result in a straight line. Rather, a convex curve is shown, in which there is a single mode and all effects of resource sharing can be considered to be lumped into this single mode for the purposes of this illustration. For now, the presumption of homogeneity in both the cores and the effects of resource sharing on the different input types serve to keep this analysis simpler, as it results in symmetric curves.

Next, consider another idealized CHM in which the input streams can be dynamically mapped to either of two processor resources as the curve, *2core_dynamic_zero_overhead (2dz)*. Again, the production of the two demand streams can be related to each other by the constant, C. Now, the Capacity of the two-core system can be considered to be double the Capacity of the single-core system (*1dz*), which is once again a highly idealized situation. The combination of the two cores is capable of processing double the demand of each input type taken individually, and all points in between. As scheduling and resource sharing becomes less ideal, a system such as *2core_dynamic_coupled (2dc)* occurs. Here, the system can process maximums for each input type, which are the same as for the idealized two-core system. But now, as different combinations of inputs are considered, the system shows Capacity less than the ideal *2dz* system because of the effects of sharing resources in the coupled CHM. A single modal, concave shape is shown for the *2dc* curve, reflecting a maximum amount of overhead penalty when the different input streams are balanced. This reflects penalties due to task migration; the same design feature that facilitates greater potential to process situations in which the inputs are less balanced, at the extremes, produces its greatest penalty when the inputs are most balanced.

The curves of most interest are the *2sc* and *2dc* curves because they include the effects of system coupling. The *2sc* curve is convex while the *2dc* curve is concave. Also, the maximum values that can be processed for unbalanced loads differ considerably. There is information in both the values and the shapes of the curves. This information can be related to some observations about the overall system organization. As analytical understanding of the shapes of Capacity curves progresses, we expect to find an additional relationship between system characteristics and the sizes and shapes of Capacity curves and discover what design features lead to: different extremes for unbalanced loads, convex and concave shapes for Capacity curves, multiple modes as multiple overheads interact, and asymmetry for heterogeneous cores and overheads.

These will be the basis of future analytical reasoning about Capacity curves. In the remainder of this thesis, we show how Capacity curves can be analyzed and how experimental generation of the Capacity metric can be used to compare different CHMs. In the experiments section we quantify the amount of performance gains that can be expected over that of traditional design.

6.4 Capacity Metric Analysis

As illustrated in section 6.3 in which fundamental shapes of the Capacity metric are discussed, the overall magnitude Capacity metric differs according to the flexibility of the scheduler. In general, the more chip-level schedulers have the flexibility to schedule tasks on any resource in the system, the higher the magnitude of the Capacity curve will be when loads are unbalanced. Thus, magnitude can be thought of as a means of measuring scheduler flexibility. The overall shape of the curve or surface, whether it is more convex or concave, can be thought of as a measure of the way the system handles load balance. In general, the more convex a curve or surface is, the more the system is suited to handle balanced loads and the more concave a curve or surface is, the more it is suited to handle unbalanced loads. If a curve or surface is more linear, it can approach an ideal system. However, this can also reflect a system that is lightly loaded, since it can so easily handle any load that is presented to it.

The 2-D Capacity curves shown in section 6.3 all had only single modes – there was only one peak in the curves. However, in general, Capacity curves or surfaces will have multiple modes. We show examples of multimodal curves in the experiments section. These modes result when there is a cancellation effect of one design feature with another. For example, burst width can facilitate an unbalanced system, while a certain processor core type can facilitate a balanced system, but these effects do not cancel out perfectly. Thus, there may be multiple peaks, or modes, in the Capacity curve or surface. Modeling and understanding these effects are some of the most important concerns of single-user multicore designs, since maximum performance is often only achieved when designers have better understanding of how specific design decisions interact. Finally, the slope defines the cost of increasing the demand for one model type on the production of the other model type(s). This is an important feature that gives the designer the ability to trade off not just between design choices but also between the usage preferences. In addition to these features, we will explore the potential for others.

6.5 Experiments

We used The Modeling Environment for Software and Hardware (MESH) simulator [124], discussed in section 4.5. Further, we used the same processors described in section 5.3.2. Since overhead is an important factor in the evaluation of multiprocessor designs, and we included it in our models, we used our model of overhead in section 5.3.3.

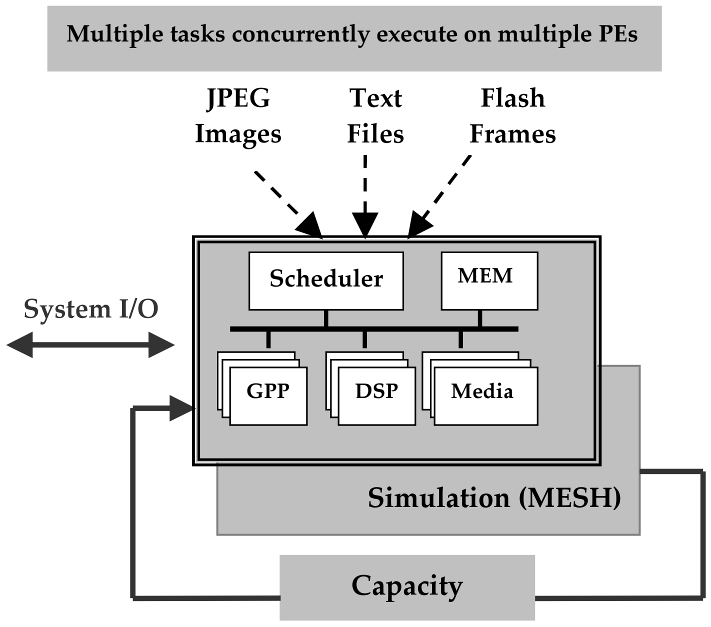

Figure 36: Experimental Setup Summary

Within a fixed area, we generated four different architectures as combinations of number of processors, type of processors and L2-cache sizes. Our model parameters are: (1) four different CHM architectures (Arch1: 2G, 4D, 6M, 128K of cache; Arch2: 1G, 4D, 6M, 192K of cache; Arch3: 2G, 2D, 6M, 192K of cache; and Arch4: 2G, 4D, 3M, 192K of cache) (2) three different burst widths [(B): 16-, 32- and 64-byte], (3) three different communication bandwidths [(C): 0.4GByte/sec; 0.96GByte/sec; and 1.2GByte/sec], and (4) two chip-level schedulers: A dynamic scheduler that maps tasks to the best available processors, permitting task migration, and a static scheduler that maps each task to a specific processor(s) specified at design time does not allow task migration. We modeled these schedulers in MESH. Further, we modeled three types of demand streams: JPEG, Text and Frames, each with different working set sizes. Finally, Capacity is measured during two time intervals, *I*: half second and one second. In general, the time interval will affect the shape of the Capacity curves, since the balance of processing is subject to change. The total number of design choices is 72. Figure 36 summarizes all design features.

The way we determine the time intervals during which Capacity is measured is by running each task type on all architectures. The maximum time needed by a task type is the selected time interval. Since architectures have different performance for different demand modes, different architectures may have different time intervals.

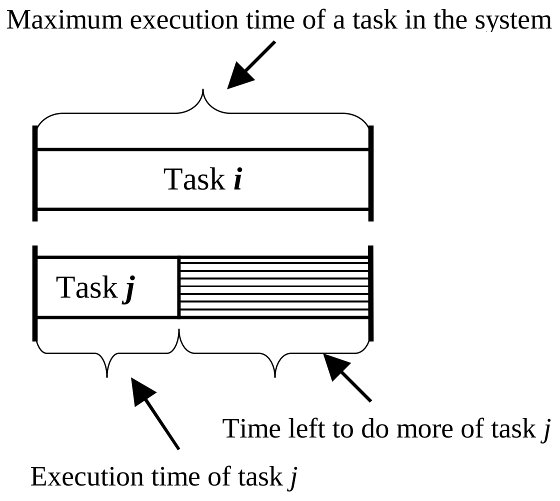
Maximum execution time of a task in the system
Task *i*
Task *j*
Time left to do more of task *j*
Execution time of task *j*

Figure 37: Normalized Execution Time

Figure 37 shows the concept behind our approach. It shows that for a specific architecture, the maximum time spent in a task is that of task *i*. Other tasks on the system, such as task *j*, complete the execution in a shorter time than that of task *i* leaving a slack time. Thus, the system can do more of the same task type during this slack time, as its demand is less than its supply, or the system is not being used to its full Capacity within that time interval. This approach is a type of execution normalization.

6.5.1 Results and Discussions

In this section, we include and discuss several sets of experimental results. We first focus only on two inputs, JPEG and text. We generated all Capacity curves of all designs, totaling 72 different CHM designs. We show some of these curves to give insight about the cause and effect of different design parameters on the system Capacity. While viewing these curves, we classify them based on patterns.

a) *Burst width feature*

Figure 38 shows two Capacity curves of a CHM differentiated by increasing the bus burst width from 16B (B1) to 32B (B2). The CHM consists of 2G, 4D, and 6M processors and 128K of L2-cache memory. Each processor has its own L1-cache memory. In this CHM, as the bus's burst width is changed from 16B (B1) to 32B (B2), the system changes from favoring balanced loads to unbalanced loads, similar to the difference between the concave and convex shapes of the *2dc* and *2sc* curves discussed previously. The B2 curve is the more ideal shape for the overall design goals since task migration is included, but the burst width impeded this result unless it was great enough.

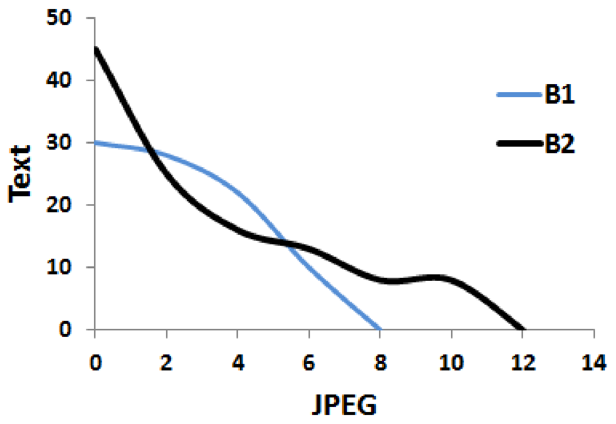

Figure 38: Capacity When Burst Width Increases

b) *Processor type and number features*

Figure 39 shows two Capacity curves for two different architectures (Arch2 and Arch3). In Arch3, we increased the number of GPP and Media processors, while reducing the number of DSPs. Note that adding this feature favors text and JPEG but not their mix.

The slope of the Capacity curve of Arch2 drops faster than that of Arch1. DSP processors have medium performance for processing both text and JPEG, thus they perform better than GPP and Media when a mix of text and JPEG is being produced.

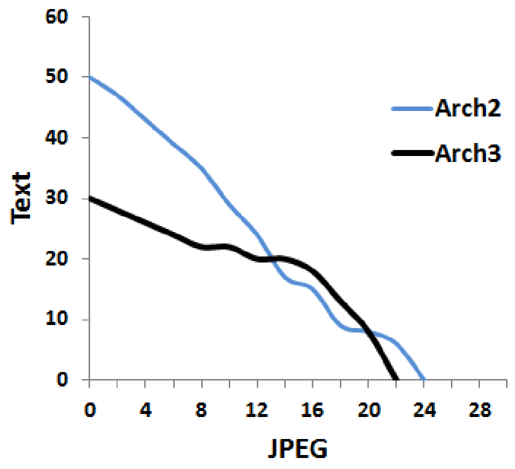

Figure 39: Capacity When More GPP and Media Added and DSPs Reduced

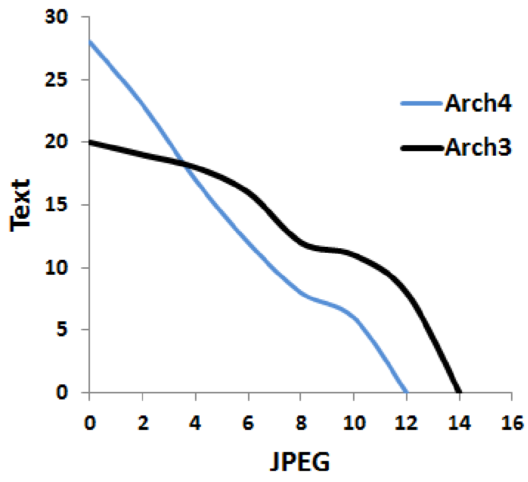

Figure 40: Capacity When More Media Processors Added and DSPs Reduced

In Figure 40 we show the effects of increasing the number of Media processors while reducing the number of DSPs. Figure 39 shows two Capacity curves for architectures Arch3 and Arch4. Compared with Arch4, the number of Media processors in Arch3 has increased by two times while reducing the number of DSPs by the half. Note that adding this feature favors JPEGs but not text because Media processors perform better than

DSPs in JPEG processing but worse in text processing. Note that the two curves differ not just in terms of magnitudes, but also in shapes. The slope of the Capacity curve of Arch3 drops slower than that of Arch4 resulting in a convex shape.

c) Cache size feature

Figure 41 shows the Capacity of the CHM of Figure 38 when the cache size increases by 1.5 times. As a consequence, the chip area has increased. Figure 41 shows too little improvement in Capacity because the selected working set sizes of the text and JPEG tasks are small. However, this feature favors JPEGs more because their processing can be considered to be internally I/O bound more than text processing.

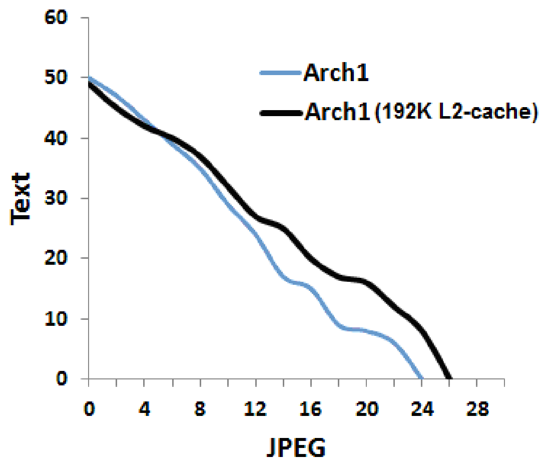

Figure 41: Capacity When Cache Size is Doubled

Figure 42 shows two Capacity curves for two different architectures (Arch1 and Arch2) differentiated by increasing the cache size from 128K to 192K at the expense of the number of processors on chip. Interestingly, the Capacity of both architectures to produce only text files is the same then the curves diverge significantly. In general, Arch2 shows major performance improvement because the selected size of working data sets of tasks is larger. Thus, the increase in the cache size helps in improving the performance. Note that in the case of producing text only, increasing the cache size does not result in any improvement because the working set size of text files is already smaller than JPEGs. Again, increasing the cache size favors the production of task types with larger working data sets.

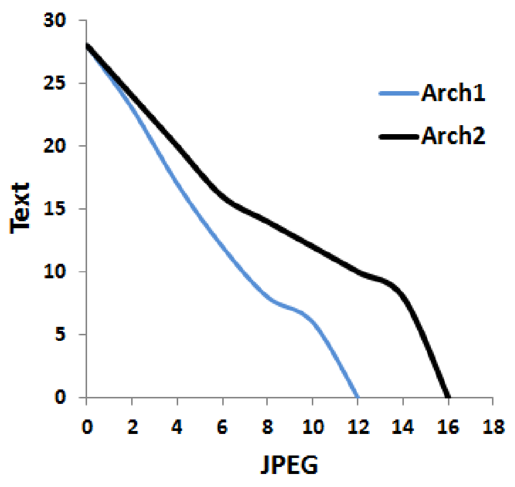

Figure 42: Capacity When Cache Size Increased (Less Processors)

d) Communication bandwidth feature

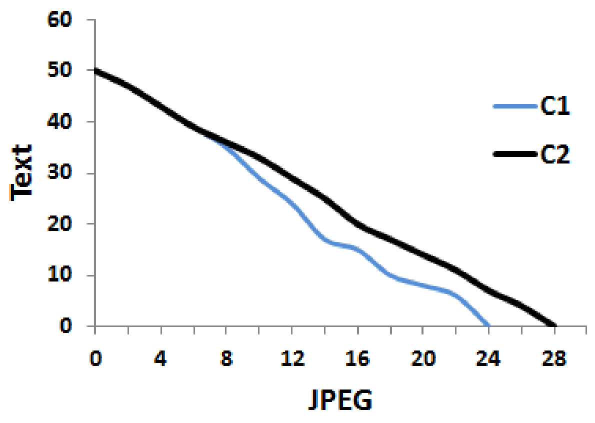

Figure 43: Capacity When Communication Bandwidth Increased

Figure 43 shows the Capacity of the CHM of Figure 38 in which the overall communication bandwidth increases. C1 is 0.4GB/s and C2 is 1.2GB/s. In each case, the system approaches the straight line of an idealized system even though the processing of input channels is coupled via competition for global resources such as schedulers and memories. The two curves in Figure 42 overlap at the beginning and then diverge. This asymmetry results because communications Capacity affects JPEG more than text. With respect to a given amount of text processing, JPEG processing can be considered to be internally I/O bound more than text is with respect to a fixed amount of JPEG processing.

However, if the region of interest over which the two CHMs are being evaluated is limited to the maximum number of text and a small number of JPEG inputs, the two systems do not differ. In general, Capacity curves will be asymmetric. The asymmetry arises from the heterogeneity of both the input and the architecture.

e) Multiple feature interactions

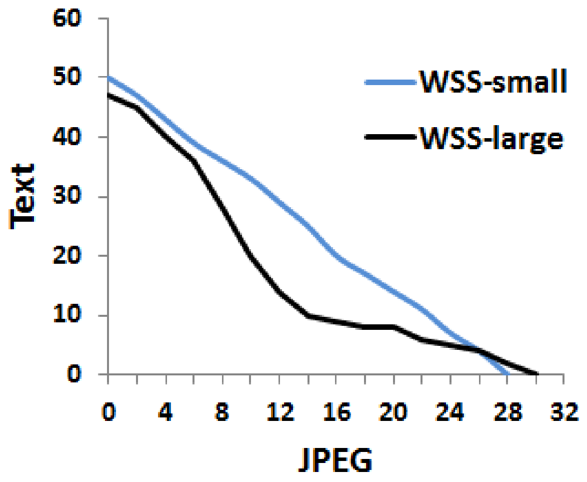

Figure 44: Capacity (Larger vs. Smaller WSS)

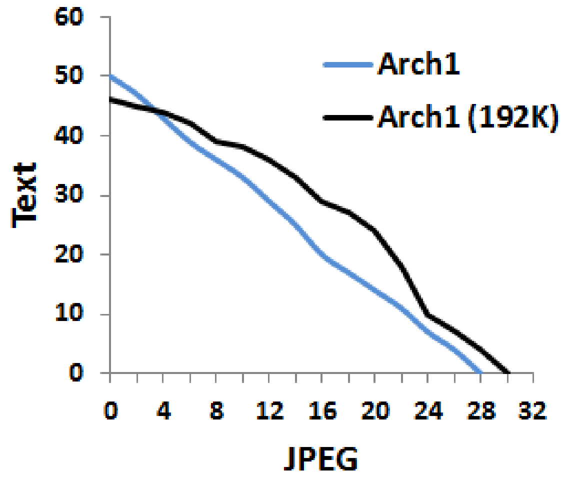

Figure 45: Capacity When Cache Size Increased (Same Processors)

In Figures 44 through 46, we use the same system from Figure 42 in which the overall communication bandwidth increased from C1 is 0.4GB/s to C2 is 1.2GB/s. The Capacity curve of the system approaches the straight line. Here, we picked this example to show how different architectural features interact. By changing other features, we show how the Capacity curve shape changes from being a straight line.

In Figure 44, we used larger working data sets for both JPEG and text tasks. We increased the working set size until we started to see the shape of the Capacity curve changed from a straight line to a convex. This drop in the Capacity is caused because of the overhead generated due to using larger working data sets. The size of the working data sets determines the space and time delays in buses as well as memories. Note that this change only affected the intermediate mix of jobs.

In Figure 45, we increased the cache size by 1.5 times, while keeping the same processors on the chip. The shape of the Capacity curve changed from a straight line to a concave. This increase in the Capacity is caused because of the increase in the cache memory size. Intuitively, increasing the cache memory size increases the production of each task type. Interestingly, note that this change affected only the intermediate mix of jobs. Since cache size increased, we are now able to store the data of both types in the cache memory.

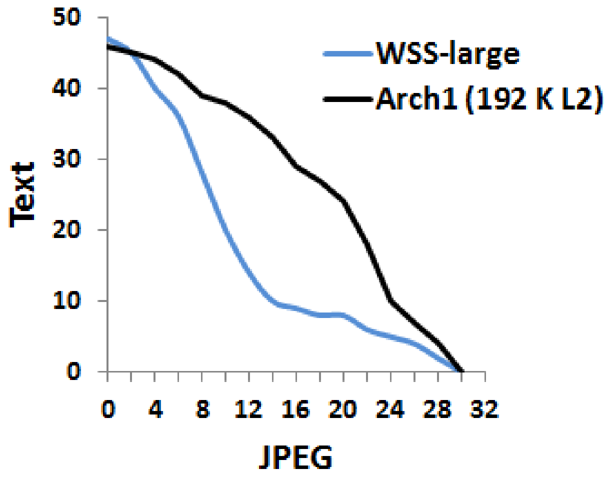

Figure 46: Capacity When Cache Size Increased (Larger WSS)

Using a different view, in Figure 46 we combined the Capacity curves of Figure 44 and Figure 45 in one figure. We can see that the system has changed from favoring balanced loads (when processing larger working data sets) to unbalanced loads (when the cache memory has increased.) For such a system, the designer not only can analyze causes and effects, but can also find solutions to existing problems in the system.

f) Scheduling feature

Figure 47 and Figure 48 show two Capacity curves using two different chip level schedulers: dynamic and static. The CHM consists of 2G, 4D, and 6M processors and 128K of L2-cache memory. The two figures are differentiated by the selected working set size; it is smaller in Figure 47 and larger in Figure 48.

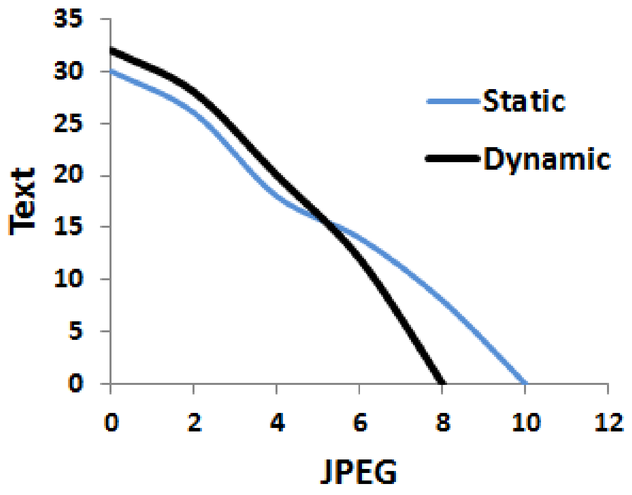

Figure 47: Capacity of Different Schedulers (Smaller WSS)

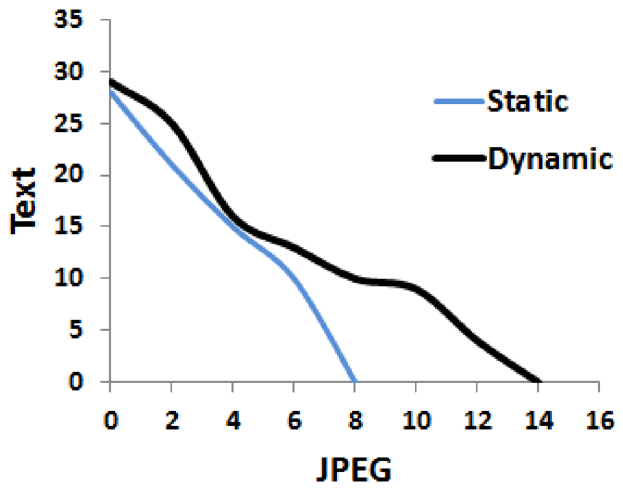

Figure 48: Capacity of Different Schedulers (Larger WSS)

Figure 47 shows that both schedulers resulted almost in the same Capacity curves, because the working set size is small, which in turn makes task migration occur less. In contrast, when using larger working size sets, the dynamic scheduler outperforms the static scheduler especially when more JPEGs are produced, as shown in Figure 48. Note that other results (shown in Appendix A) show that communication bandwidth, cache size, and burst width significantly impact the performance of dynamic schedulers.

g) Time window interval

Figure 49 shows the Capacity when the time window in which we measure Capacity is doubled. Note that doubling the time window does not result in double Capacity because of the overhead generated due to resource sharing and running heterogeneous concurrent tasks. Again, increasing the window of time favors homogeneous inputs. The heterogeneity of inputs incurs more processing, communication, and storage memory overhead.

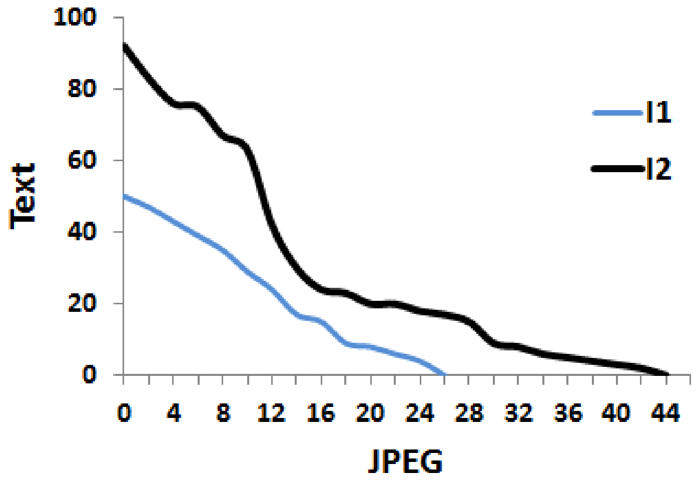

Figure 49: Capacity When the Time Window Is Doubled

6.5.2 Experiments Summary

The experimental results, shown in this section, represent different forms of Capacity curves. These curves give the designer the ability to understand the effects of adding architectural features on the performance of a computer system that simultaneously processes multiple output streams. Significantly, these curves do not result in linear relationships as demand for different production types varies. Distillation of rates of production over multiple production types to a common work unit, such as automobiles, instructions, tasks or programs, results in a linear relationship as rates of production of more specific types within those categories varies. But if a straight line were to be superimposed on any of our Capacity curves, the real Capacity of the designs being examined would either underestimate the true Capacity of a given design for a given set of production values, or it would overestimate what the plant is capable of.

For each design, Pareto optimization or throughput can only show the maximum production of individual tasks in the form of a vector or single value that is generated by taking an average. For example, when throughput is evaluated using a common work unit, such as Instructions per Second (IPS), it can lead to erroneous insight. For example, in Figure 38, if the maximum production of each type is only considered, the Capacity curve of B2 is better than that of B1. Note this is not true when we consider specific combinations of tasks that might be most important to the end user of the design. For example, for an intermediate mix of tasks, the curve of B1 shows better Capacity than does B2.

Here, we summarize some observations from our experiments about the effects of different architectural features:

- *Burst Width*: increasing the burst width favors unbalanced loads. Narrower widths give multiple heterogeneous tasks a more equal chance to share the communication media. Otherwise some tasks may occupy the communication media for a longer time without the need for that long time, impeding the execution of other tasks. Wider widths favor homogeneous tasks because they have same communication and memory access patterns. Other factors such as the size of the working set of the same task type, also determine the optimal value of the burst width. Thus, there is an optimal value of the burst width for each workload type.
- *Communication Bandwidth*: increasing the communication bandwidth ideally results in greater system Capacity. However, it also represents an overhead in terms of time, area and power consumption. Thus, increasing the communication bandwidth only makes sense when the gained Capacity

overcomes the incurred overhead. For instance, the production of tasks that require less I/O accesses takes less advantage from increasing the communication bandwidth.

- *Processor Number and Type*: increasing the number of processors of a specific type favors the type of tasks that take more advantage of this processor type, shifting the Capacity curve toward the axis of that task type. This results in different shapes of the Capacity curve: a convex shape occurs when processor types do very well for both task types, and a convex shape occurs when processor types favor different task types. The magnitude of the curve depends on the number of processors on chip. However, note that increasing the number of processors incurs more overhead due to the increased complexity of synchronizing and controlling these processors, in addition to other costs such as the die area and power consumption, etc.
- *Cache Memory Size*: increasing the cache memory size ideally results in better system Capacity. Its advantage is more obvious for those tasks that have larger sets of working data and greater I/O requirements. Since we model a fixed-area chip area, increasing the cache memory comes at the expense of reducing the number of processors. This trade-off analysis results in different Capacity curve shapes.
- *Scheduler Type*: dynamic schedulers for CHMs generally outperform static schedulers because they allow task migration and dynamic decisions in response to the system loading and previous and current chip state. Note that the making of dynamic decisions and task migration incur overhead that has an inverse effect on the system's Capacity. Dynamic schedulers often result in multimodal Capacity curve shapes because of this incurred overhead.
- *Time Interval*: because of the effects of overhead incurred because of the interaction of multiple heterogeneous inputs vying for heterogeneous global resources, doubling the time interval may result in less double Capacity, in contrast with the assumption of throughput in which work and time have a linear relationship. The ratio of the Capacity to the time interval length depends on several factors including the type of workload. In other words, for

a system with a homogeneous input, the ratio approaches to be linear, resulting in an almost straight-line Capacity curve shape, similar to the concept of throughput. In general, the increase in the time interval length favors unbalanced loads, resulting in a convex shape.

Interestingly, note that when multiple architectural features interact, multimodal shapes result. While in this chapter we focused on JPEGs and text task types, we include in Appendix A one more task type, that is GIF. Our goal is to show how the type of input changes the system's Capacity and thus be able to classify these effects into patterns. For the sake of completeness, in Appendix B we show the Capacity of this three-input channel system as a surface, although the way we analyze the Capacity surfaces is by holding one input constant at a time. For instance, a 3-D surface can always be reduced to a collection of 2-D curves. In general, an n-D surface can always be reduced to a collection of (*n-1*)-D surfaces, which can then be analyzed as collections. In Appendix C, we include the values of overhead incurred due to different combinations of inputs, and explain why multimodal Capacity curve shapes result.

In the next chapter, we demonstrate the usefulness of our metric through a Capacity shape analysis. We show that information in the shape of the Capacity curves can be more significant than magnitude.

Chapter 7

Shape Analysis

Concentrate all your thoughts upon the work at hand. The sun's rays do not burn until brought to a focus.

Alexander Graham Bell

Capacity is distinguished from other performance metrics because it emphasizes shape as well as magnitude. In this chapter, we show that information in the shape of the Capacity curves and surfaces can be more significant than magnitude is.

7.1 Why Single-Valued Performance Metrics Fail

In this section we show how attempts to distill performance to pure magnitude, or single-valued metrics, can result in ambiguity or identification of incorrect optimal designs.

7.1.1 Survey of Single-Valued Performance Metrics

We first perform a survey of some popular, single-valued metrics that have been used to compare curves that consist of discrete points:

(a) The average of *all* data points on the curve. The average is measured in terms of the model type on the dependent axis,

(b) The maximum production of each model type. This also includes the average of the maximum production of each model type, using a common output type,

(c) The area under the curve, measured in terms of the model type on the dependent axis, and

(d) Pair-wise (point-to-point) comparison or least square method. The pair-wise method cannot be used in our case because the lengths of Capacity sets (or more precisely, the numbers of values in multisets) of different designs are unlike; we include it for the sake of completeness.

Our goal is to show how our Capacity metric succeeds when single-valued performance metrics fail in evaluating the performance of the CHM that processes a multi-channel heterogeneous input. Thus, we examine how our metric compares to the metrics in items (a), (b) and (c) of the above list. We consider our metric to be more successful when it properly ranks designs from optimal performers to lesser performers. Admittedly, such comparisons can be somewhat arbitrary and subject to interpretation. In fact, we demonstrate this when we show that conventional means result in ambiguity, which is what prompted us to develop our new metric. Thus, we use workload analysis to show how some designs fit certain workloads better than others and how those designs cannot be identified in rank order using single-valued metrics.

7.1.2 Comparison Analysis

We first generate the Capacity curves for the designs that we have discussed in Table 12. Then we apply the different single-valued performance metrics in order to rank the performance of these designs. For the purpose of this section, we initially select the Capacity curves for only three designs. Three designs are sufficient for our purposes, because they show how single-valued performance metrics fail to identify optimal designs. We measure the Capacity of these designs to produce JPEG images and text files, which are examples of data-size dependent and data-content dependent execution, respectively [1].

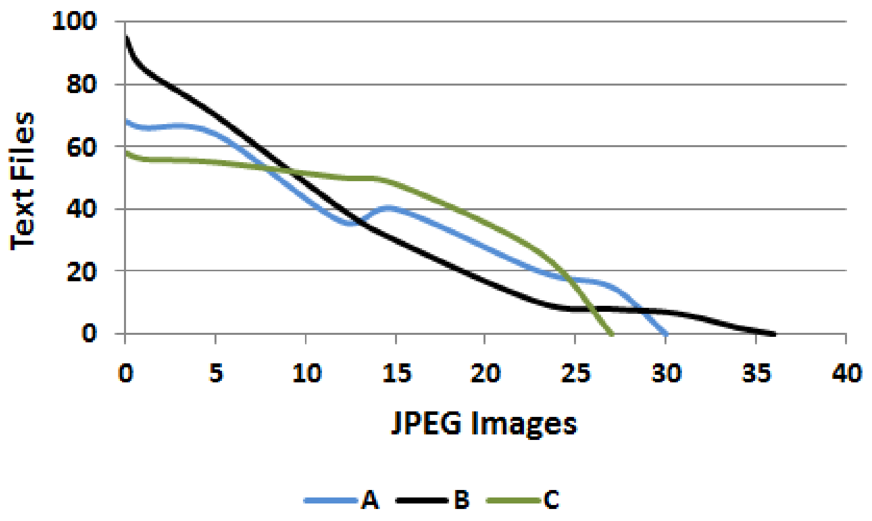

Figure 50: The Capacity Curves of Three Different Designs

Figure 50 shows the three Capacity curves as a relationship between the production of JPEG images and text files. On the independent axis of the figure is the production of JPEGs and on the dependent axis is the production of text files. Design A refers to design 4 in Table 12, which consists of one General Purpose Processor (GPP) and four DSP processors. Design B refers to design 7, which consists of two GPPs and four Media processors. Finally, Design C refers to design 2, which consists of six DSP processors. More discussion about the architecture of these designs is included Chapter 5.

Next, we compare the Capacity metric to the single-valued performance metrics by asking which metric identifies the best performing design. Here, we define the best performing design as the design that has the greatest Capacity during a specific time unit amongst all designs on hand. This also implies that the best performing design is the one that has the shortest response time to process a set of workload modes.

A. Averaging All Points

The first method is to average all points (or supported demands) on the Capacity curve of each design and then compare the average of different curves. These points represent the production rate of JPEGs and text files. Since we cannot simply average the production of JPEGs and text files because they are of different complexities, we must instead average the time required to produce these demands. Note that the required time to produce any demand on the Capacity curve is the time interval during which Capacity is measured. Hence, we select the maximum production demands amongst all three designs. For instance, the maximum production of JPEGs when we can produce 15 text files is 50, as shown in Figure 50. Design C can produce this demand during the specified time interval, while Design A and Design B need more time to produce the same demand. Thus, for a design, these maximum demands may be produced during the time interval or need more time to be produced. Then, for each design, we average the execution time of these maximum demands on that design.

Note that the best design varies as curves cross. Figure 50 shows crossover points in which different designs exhibit different performance as demands vary. The crossover points result because different CHM designs are specialized for different demand modes. If the curves did not cross, then there would be no significant difference in optimal performers due to demand mode variation. But, because the curves do cross, the actual input to the system results in the identification of different optimal designs.

Averaging requires the use of means. There are three primary mathematical means: Arithmetic Mean (AM), Harmonic Mean (HM), and Geometric Mean (GM), previously discussed in Chapter 2, in which we also showed the circumstances in which AM, HM, and GM can be used to average speeds (Demands/Time). HM is applied to equal demands, but with different execution time, whereas AM is used when demands are different but execution time is equal. The GM can be used for both time-based and rate-based behavior, but the behavior should first be normalized with respect to specific machines. Since the design goal of CHMs is to optimize the response time of individual demand modes, the HM is the most appropriate mean. In this case, a better performing design is one that does more in a given interval, which the way users experience modern is computing. But, since the usage of the appropriate mean has long been controversial in computing [8], [51], and since the geometric mean is by far the most common method used in the evaluation of single-core architectures, we include all means in our analysis.

Table 16: Response Time for all Production Modes

Demand Mode	**Response Time**		
	A	B	C
(0,95)	1.65	1.00	1.8
(1,85)	1.6	1.00	1.7
(5,70)	1.65	1.00	1.85
(12,50)	1.55	1.90	1.00
(15,48)	1.4	1.75	1.00
(23,26)	1.45	1.85	1.00
(27,15)	1.00	1.65	1.70
(30,7)	1.6	1.00	1.75
(32,5)	1.65	1.00	1.60
(34,2)	1.55	1.00	1.65
(36,0)	1.45	1.00	1.70

Table 16 includes the response time of each design when used to produce different demands. Note that some designs can produce a demand mode during one time unit (the optimal design for this demand), while others need more time to complete this demand. The time interval over which a response is measured is an important consideration in a metric for CHMs, and its impacts will be discussed in more detail later. Table 17 shows the rank of the average response time of each design, computed using the three different means (AM, HM, and GM).

Note that using different means results in different ranks, as previously predicted. HM ranks Design A as the best architecture because it is better suited to handle the increase in demands, since HM ranks architectures based on their ability to execute more demands per time unit (rates), whereas AM and GM rank Design B as the best architecture. The arithmetic and geometric means rank different performers as optimal, even though they rank same architectures at the top. Thus, single-valued means already result in ambiguity in ranking optimal performing designs, and this was discovered in the work on workload modes as presented in section 2.2. Without a proper means of ranking optimal performing designs, it did not make sense to continue pursuing that work in more depth.

Table 17: Optimal Design Ranking

Architecture	***Arithmetic***		***Harmonic***		**Geometric**	
	Mean	Rank	Mean	Rank	Mean	Rank
A	1.54	2	1.20	1	1.45	3
B	1.25	1	1.25	2	1.10	1
C	1.70	3	1.40	3	1.30	2

In summary, the traditional mathematical means result in *ambiguous* rankings of optimally performing designs. Averaging, in general, precludes the designer from identifying cases in which the system has more Capacity than the average or cannot meet the desired production.

B. Bounded Area

One of the best known techniques in curve comparison, for both discrete and continuous functions, is area difference. Mathematically, the area under a curve is calculated using the definite integral. Thus, the area bounded between two curves can also be found using the integral of the difference between their functions. Because our Capacity curves are measured and not computed and fitting these data using a function is an approximate that may include a significant error margin, we use numerical approaches to calculate the integral of discrete data.

Numerically, the trapezoidal rule is a well-known technique for calculating the definite integral. The trapezoidal rule works by approximating the region under the curve. It is one of a family of formulas for numerical integration called Newton–Cotes formulas. Simpson's rule is also another member of the same family. Newton–Cotes formulas differ only in terms of their computation speed.

When the trapezoidal rule is applied on the three Capacity curves shown in Figure 49, all curves have the same area, therefore their area difference is zero. This can be seen by inspection. Thus, according to this metric, all designs are equivalent.

The area difference between curves is zero because curves cross, allowing areas under each curve to be alike. This also means that this metric collectively cancels out the effects of design specialization and does not show which design is really better. While integral is feasible at higher dimensional surfaces, this metric also results in ambiguous results.

C. Maxima Points

The third single-valued metric we consider is the maximum production of each model type, or maxima points. According to this metric, Design B is the optimal design and Design C is the worst deign.

This metric is similar to the throughput metric in which the performance of different task types is individually evaluated and then averaged. Also note that some other designs (not shown) may have equal maximum points even though they are vastly different when they process an intermediate mix of work types. For instance, consider two designs that are only different in terms of the adopted scheduling policy. Schedulers may have the same performance in terms of Capacity when they process a homogeneous input, while they differ when they process a heterogeneous input; we have shown an example of this in Figure 13. Thus, this metric also leads to ambiguous results.

7.1.3 Summary

We have shown how single-valued performance metrics lead to either incorrect ranking or ambiguity. All three different single-valued metrics resulted in different rankings. Harmonic mean, which is the most appropriate metric according to our design goal, found that Design A is the optimal design, in contrast to the other mathematical means.

Whereas, the maxima points' metric found that Design B is the optimal design. On the other hand, the bounded area metric found that all designs are equivalent. The major reason behind this ambiguity is that curves cross, resulting in crossing areas that are in favor of specific designs over others. Single-valued metrics factor out these areas of comparison.

Designs A and B prefer the production of unbalanced loads. This is more obvious in Design B, in which the system performs very well when producing only text or JPEGs. As the demand increasingly becomes an intermediate mix of text and JPEGs, Capacity degrades (i.e. a fewer number of jobs are produced). Design A performs better than Design B when a mix of tasks is produced, but it performs very differently from Design B when it is processing only text or JPEGs. In contrast, Design C prefers the production of balanced loads. As the demand is becoming more heterogeneous, the Capacity gradually decreases.

The only metric that describes the performance of these designs to produce different types of jobs, as we just described, is the curve shape. The shape of the Capacity curve of Design A and B is a convex, while it is a concave for Design C. The shape of the Capacity curve of Design A is a convex with less regularity than that of Design B. The abnormality is originated from the effects of overhead on the system Capacity. Note that the curve of Design A at some production region produced less than it should ideally have because of the overhead generated when having a specific combination of jobs. One reason for that is the overhead of task migration or memory and communication contention. Later on we include a deep analysis of the effects of overhead in creating multimodal curve shapes.

Single-valued performance metrics fail to find optimal designs because they do not consider the relationship between the system's production capabilities of specific workload modes and match them to the actual demands placed on the system. We bridge this gap by developing an algorithm to characterize demands in the form of a curve and overlay these curves over the Capacity curves. When Capacity curves match demand curves, they are optimal designs, regardless of their magnitude.

7.2 Our Approach: Demand-Capacity Matching

In order to show the usefulness of our Capacity metric, we first need to characterize the demand for a system. Because the demand for modern CHMs is not, itself, single-valued, characterization of demand needs to be in the form of workloads, or distributed sets of concurrent jobs that result in collections of distinct loading situations.

7.2.1 Demand Curve Generation

In order to generate demand curves, we first develop an algorithm to identify all possible demands in a specific usage scenario. The identified demands are then used to draw a demand curve.

A. Demand Collection Method (DCM)

We use an enhanced version of the Workload Characterization Method (WCM), discussed in Chapter 5, to generate all possible demands exposed by a single-user usage scenario. Demand generation is conducted in an adhoc manual manner by the system designer.

Step 1: Type Extraction. Each task type, Y, represents a dimension on the curve. Tasks with different working data sets, S, and complexities, C, can be considered different types of tasks.

Step 2: Forming **dm** *Vector.* Step 1 resulted in m different task types within a demand mode (**dm**). **dm**=$[Y_1,...,Y_j,...,Y_m]$ represents the demand vector of length m, in which Y_j is a task type.

Step 3: Forming **M** *Matrix.* Each usage scenario has n demand modes. The number of demand modes is a function of the length of the time window in which demands are characterized as well as the sequence of demands. **M** is the generated matrix of dimension $n \times m$ that represents a usage scenario, in which rows are **dm** vectors and columns are the number of instances, s, of each job type, Y, in **dm**.

Step 4: Appends **M** *Matrix.* Step 3 is repeated for different time window's length as well as for different sequences of demands.

In order to convert the generated demand modes into an n-d graph, matrix **M** is represented in the form of a multiset. Each set represents a demand mode (or vector) that is shown as a point on an n-d space. A *multiset* is similar to a set except for allowing duplicate values to exist. Since our demands are unordered and duplicates may exist, we chose to use a multiset type of collections.

Note that our DCM algorithm is similar to the WCM algorithm, but for different applications. While WCM was used to identify workload modes at runtime, utilizing some design-time knowledge, the DCM is used to generate all possible demands exposed by a usage scenario. We should also point out that the terms workload modes and demand modes are interchangeable. While we previously published [147] using the term workload modes for demands placed on the CHM, we have evolved and now more correctly distinguish this as demand modes, since others have claimed the term workload modes for different purposes, such as power optimizations [131]. Thus, we developed the term demand modes, while defining our Capacity metric.

B. Time Window Selection and Demand Sequence

One key part in our DCM algorithm is the inclusion of time, something that reflects a real workload. In order to do this, we divide the usage scenario into equal time intervals during which demand modes need to be collected. Here, the time interval is selected by running each scenario on all architectures. The maximum time needed by a demand mode is the selected time interval. Since architectures have different performance for different demand modes, different architectures may have different time intervals. This is analogous to the way we specify the time interval during which we measure Capacity, as discussed in section 6.5.1. This approach is a type of execution normalization. By doing this, we define a sliding window over our scenario during which we collect demands. For different sliding window sizes we expect different demands to arrive during this window.

Another key part in our DCM is the sequence of demands. By evaluating the overhead generated due to different mode sequences, we have shown in Chapter 5 that the sequence of demand modes matters to system performance; for different mode sequences different amounts of overhead is incurred. Therefore, when demands are collected we consider all demand sequences for different time intervals. The number of sequences is *n!*, in which *n* is the number of modes in the scenario. For instance, the scenarios shown in Figure 51 and Figure 52 have six modes each. Next, we discuss these scenarios and use our DCM to generate all demands exposed by these scenarios.

C. Example: Usage Scenarios

Here, we discuss two usage scenarios. While our scenarios are fictional, the data collected from these scenarios is real. In both scenarios, the mobile device is used only to surf webpages.

Figure 51 depicts a usage scenario of a teenager. The timeline is on the independent axis and webpages are on the dependent axis. Each webpage consists of several tasks that arrive the system in the form of timed sets. These tasks are summarized in the legend below the figure. This usage scenario models an example input of modern computers in which multiple applications possibly run concurrently and arrive at different times. Note that as time progresses, the user places different requirements on the system as different combinations of applications execute. A set of applications are run concurrently, driven by user intervention and the arrival of data from the Web. The task progression is also shown in the same figure (e.g. to download a picture, the system must first perform a Viterbi decode and then JPEG decode).

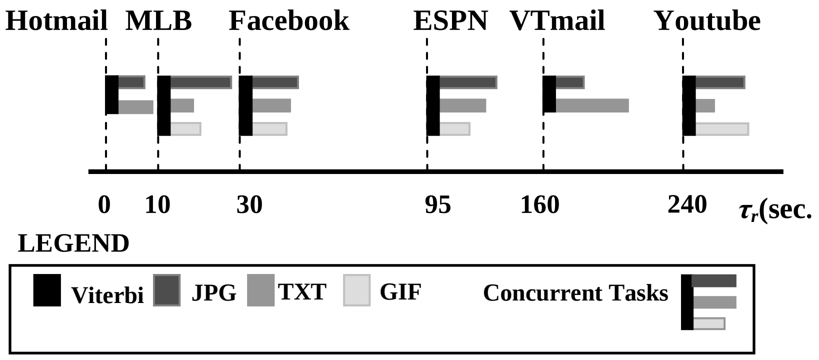

Figure 51: A Teenager Scenario

Consider another usage scenario of a businesswoman, shown in Figure 52. Because of the different combinations of application sets, input data, deadlines, and constraints, this usage scenario may exhibit different system performance from the previous usage scenario of Figure 51, causing the designer to tune the system differently, despite each system executing the same set of applications. These differences come from the type of webpages surfed by the user as well as the arrival timing of jobs. For instance, the user in the scenario of Figure 51 uses his or her mobile device to access social networks and entertainment webpages. This type of webpage has different requirements than that of the second scenario, in which the businesswoman accesses financial webpages most of the time. These differences do not just exist in the size and complexity of content, type of content, and number of jobs, but also in the arrival timing of data that affects the response of the system to the frequency of changes in the input stream [48].

While our scenarios consist of webpages only, they can be considered representative workloads of other applications. For instance, sending and receiving text and image messages in our multimedia cell phone example, discussed in Chapter 3, behave the same. We used only webpages because other cell phone applications are periodic, such as processing voice packets and MP3 frames. Since these applications are periodic, their demand is constant and therefore it is not interesting to include them. The Capacity metric is most useful in systems that have collections of data-size-dependent and data-content-dependent execution.

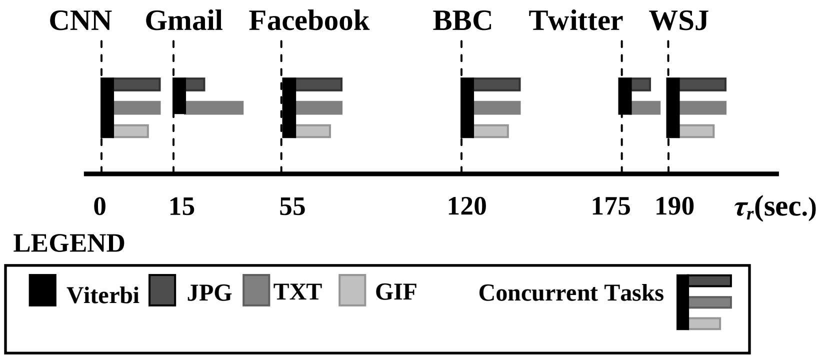

Figure 52: A Businesswoman Scenario

Using our DCM algorithm, we generate all demands of the scenario in Figure 50. Since the demands consist mainly of three job types (text, JPEG, and GIF), the DCM results in a 3-D demand space. The data used by our algorithm were extracted using an online tool (Webpage Analyzer) [148]. For simplification, we project this 3-D space into its 2-D spaces. Future systems are anticipated to have many input streams, resulting in demand curves of higher dimensions that cannot be seen by graphical techniques. By projecting an n-D space into its 2-D spaces, we facilitate analysis of these higher-dimension spaces. Initially, we have computed three time intervals during which we collect demands by running this scenario onto the three designs described in section 7.1. We have also considered all mode sequences.

In Figures 53 through 55, we show demands in 2-D. Again, these 2-D spaces are the projection of a 3-D space. On each plane, we show the set of demand modes generated due to this specific usage scenario. Some of these demands are aligned vertically. This means that for the same value of the independent variable there may be multiple values of the dependent variable. This is because demands for some type of jobs may be repeated during a specific scenario but in different combination with other job types. For instance, consider a demand that consists of two text files and five JPEGs, and another demand that consists of two text files and four JPEGs. For the same amount of text files, there are different amounts of JPEGs. This repetition may also occur in more than one variable in a demand that consists of more than three variables. The red line curve represents the maximum demand modes. For each value on the independent axis, there is only one unique maximum correspondent value on the dependent axis. We fit these maximum values using a curve that simply connects these values. We refer to this curve as the maximum production curve.

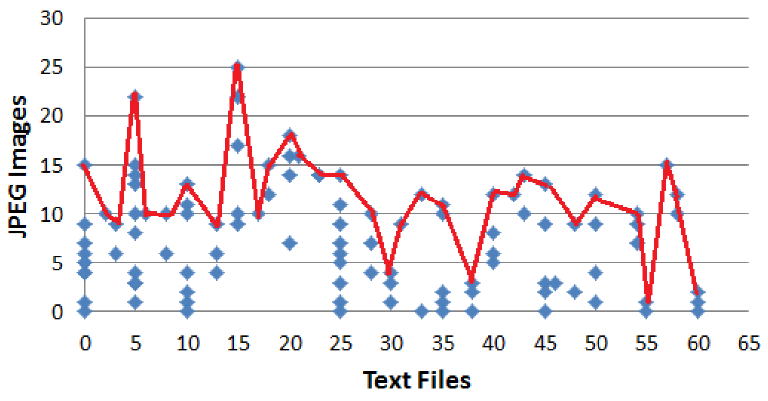

Figure 53: Text-JPEG Capacity Plane

Our final goal is to identify a demand curve, similar to the concept of Capacity curves. The demand curves are analogous to how an ideal system would carry out the actual, required demand, a system with perfect Capacity to meet required workloads in all situations the system is expected to encounter, with no waste. In order to do this, having all possible demands the system needs to service, we can generate different types of curves such as the maximum and minimum demand curves. The demand curve is the target workload limit that the system must meet.

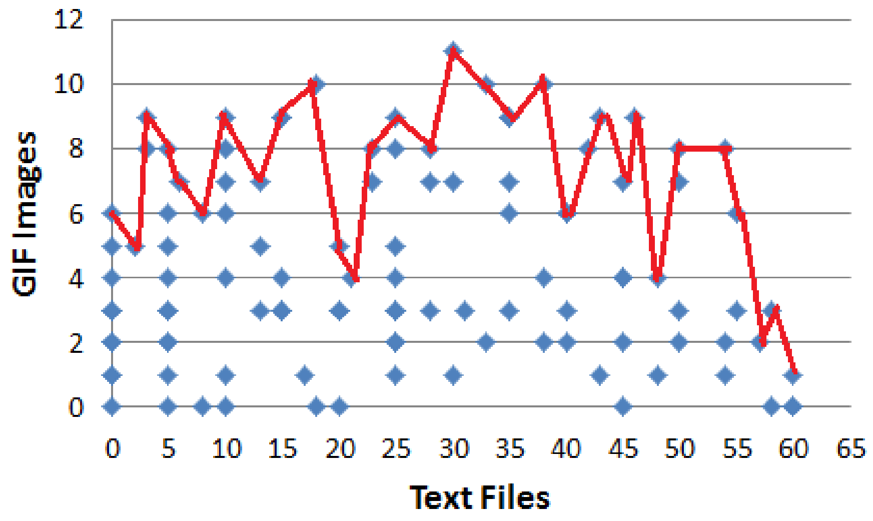

Figure 54: Text-GIF Capacity Plane

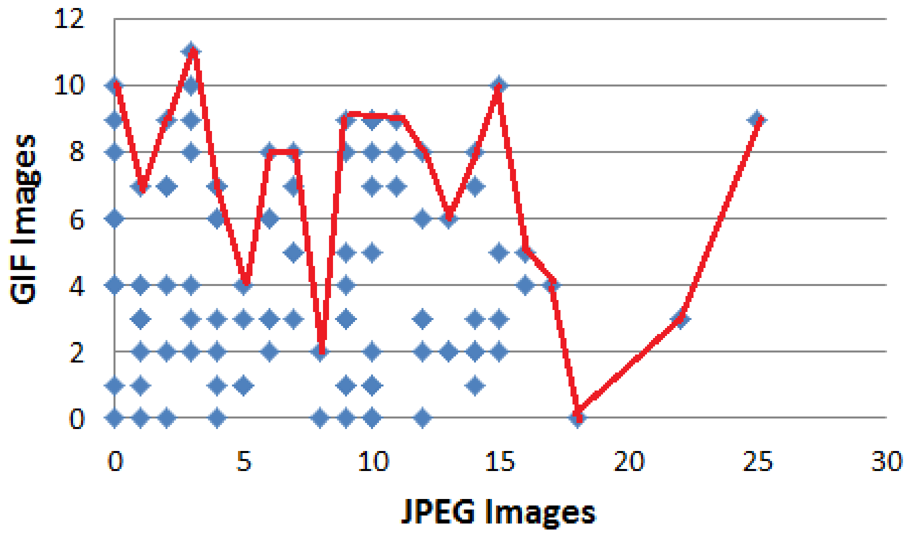

Figure 55: JPEG–GIF Capacity Plane

Depending on the design goal of the system, the designer may create different types of demand curves. For instance, the designer may choose to design a system to meet the most frequently repeated demands, demands that users may be most interested in, or demands that exhibit a high access to caches. For the purpose of illustration, we next extend our DCM algorithm to generate another type of demand curves, which is the occurrence frequency curve.

D. Occurrence Frequency Curve

One alternative is the design to the most frequently repeated demands, or the occurrence frequency curve. There are different ways to identify the occurrence frequency curve from the set of demands that we have. One approach we adopt is to use a weighted version of our DCM algorithm to identify clusters of demands. Each cluster is represented by only one demand mode on the curve.

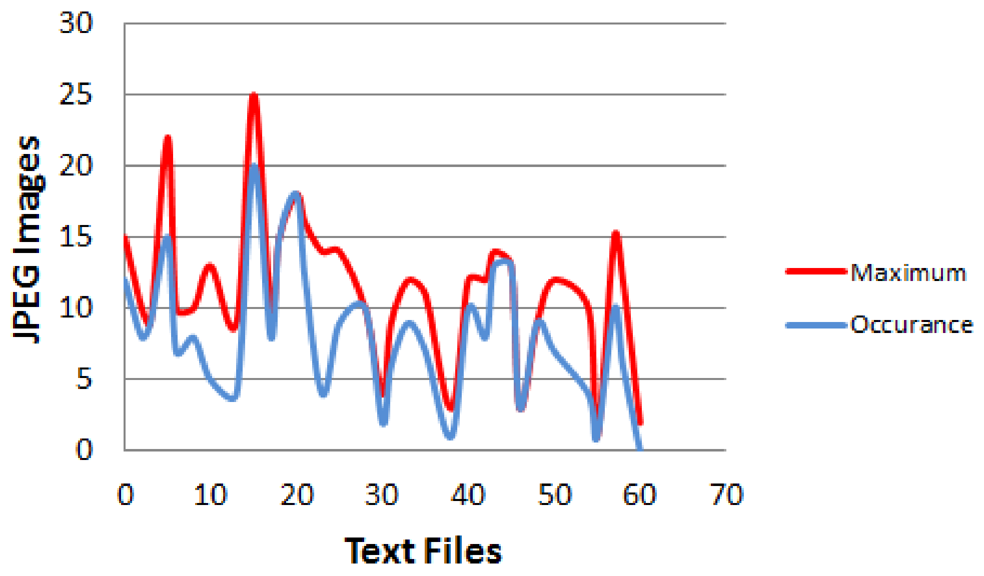

Figure 56: Maximum Demand vs. Occurrence Curves

For that purpose, normalized complexity weights are generated by dividing the cycle budget of each application by the maximum cycle budget needed by an application in the system. We extend Sim-Profile [130] of SimpleScalar to extract the number of cycles for each application. Sim-Profile profiles each application when it runs on a different processor type. We enhance our DCM algorithm by including two more steps:

Step 5: Forming **C** *Vector*. We suggested normalized weights, *c*, that correspond to the task complexity. These values are extracted again using the modified version of the Sim-Profile.

Step 6: Assigning **C** *Vector Values to* **M** *Matrix.* Matrix **M** is then multiplied by weight vector, **C**. We normalize each column (task type) in the matrix by subtracting the mean value and dividing it by the standard deviation. The goal of the normalization is to put all task types on a common scale. Then, we linearly combine the absolute values in each row (demand mode vector **dm**).

Demands with equal complexities are clustered together. The clusters with a larger number of elements (demands modes) can then be selected. Figure 56 shows the same curve of Figure 51 but is overlaid over the occurrence frequency curve. While curves intersect in some demand modes, they diverge in others. Again, the most appropriate curve depends on the system design goal.

7.2.2 Curves Match Fit

The final step in our approach is to match Capacity curves to demand curves. There are several techniques in the literature that can be used to match curves, especially in image processing and pattern recognition [149], but because we project both Capacity and demand surfaces into 2-D curves, matching 2-D curves is graphically applicable.

The match fit should ensure that all demand modes are covered by the Capacity curve (the design can support all demand modes), but at the same time, the distance between the two curves should not be too far because this excessive Capacity may come at the expense of power consumption and waste in system resources.

7.3 Experiments

The purpose of these experiments is to show how our Capacity metric, in combination with demand curves, succeeds to identify optimal designs when single-valued performance metrics led to ambiguity or identification of wrong designs. The formation of demand curves emerged from the observation that the shape of the Capacity curve has information about the loading preference of a design (i.e. the demands that it can support). In our experiments, we also aim at analyzing the effects of overhead on the Capacity curve shape. We conclude with some discussion.

7.3.1 Experimental Setup

We use the same experimental setup in section 5.2. The only change is that instead of using the scenarios, discussed in section 5.1, as the benchmarks to evaluate the performance of the designs, summarized in Table 12, we use the set of scenarios discussed in section 7.2.1. As discussed earlier in this chapter, the reason for this change is because the scenarios of section 5.1 were generated according to our multimedia cell phone example, discussed in section 3.3, which includes applications that have constant demands. The most interesting applications are those that have different demand requirements at different times.

7.3.2 Results

We first use the same example in Figure 50 to show how our Capacity metric leads to different results. In order to do this, we overlay the same Capacity curves shown in Figure 50 over the demand curves generated from our scenarios, discussed in section 7.2. The optimal design is the one with a Capacity curve that matches the demand curve. In Figures 57 through 59, we overlay the Capacity curve of Design A, Design B, and Design C, respectively, over both maximum and occurrence frequency demand curves.

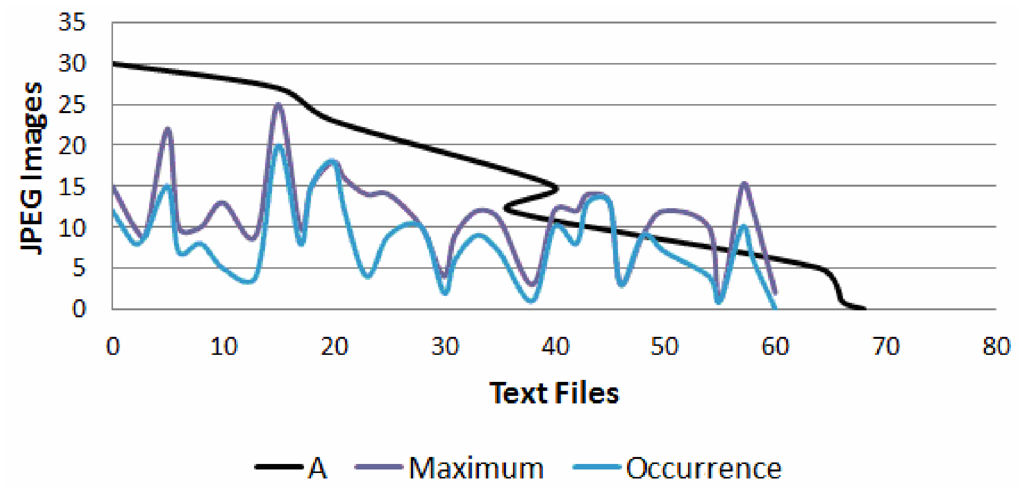

Figure 57: Overlay Demand Curves Over Design "A's" Capacity Curve

First note that our demands are balanced loads of both job types. As expected, Figure 57 and Figure 58 show that Design A and Design B prefer unbalanced loads, loads that are mostly of one type of jobs, while they cannot produce some mix of text and JPEG files, especially those that have more jobs of both types. This is because Design A has a GPP processor that is the best fit for producing text files and four DSPs that is the second best fit for the production of JPEGs. Design B is even better in producing unbalanced loads because it has two GPPs and eight Media processors that are the best performing processors for text files and JPEGs, respectively. As a result, Design B is a worse fit than Design A because the demand curve includes more balanced loads. In general, both designs could not match any of the demand curves.

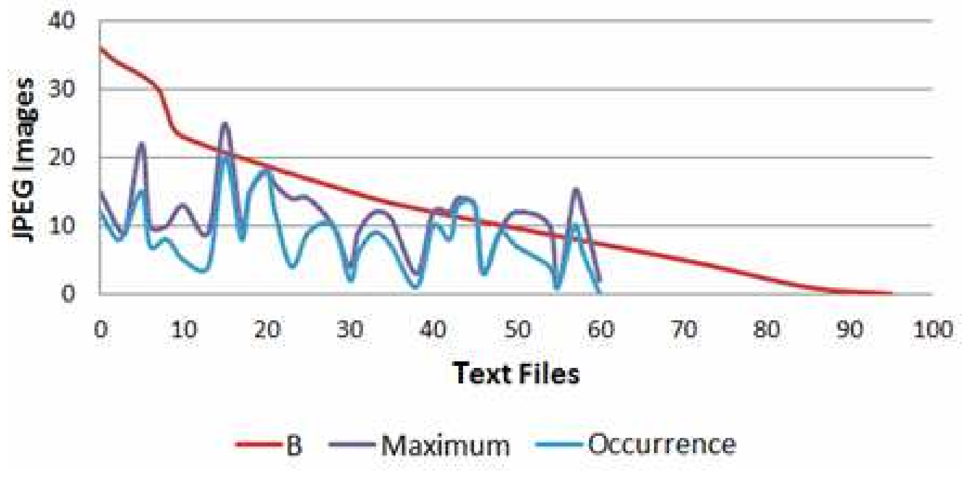

Figure 58: Overlay Demand Curves Over Design "B's" Capacity Curve

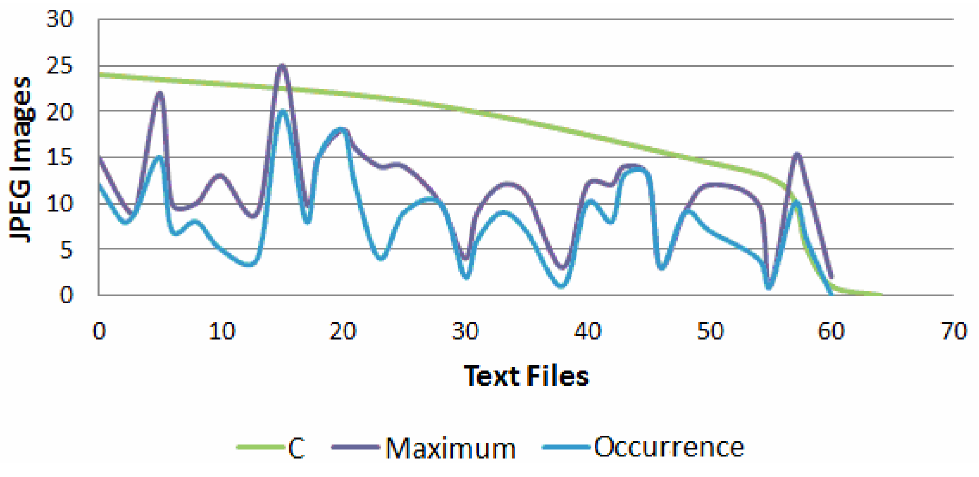

Figure 59: Overlay Demand Curves Over Design "C's" Capacity Curve

In contrast, Figure 59 shows that Design C was able to cover all demands of the occurrence curve and almost all demands of the maximum curve. Note that Design C performs better when the system receives balanced loads. This is because it has six of DSP processors that are the best performing processors for a mix of JPEGs and text files. The DSP outperforms Media processors in processing text files and outperforms GPPs in processing JPEGs.

Back to our discussion in section 7.1, in which we have shown that single-valued metrics lead to different rankings, using the HM, which is the most appropriate metric according to our design goal, we found that Design C is the worst design, similar to the ranking of the arithmetic mean. Using the geometric mean, we found that Design C is the second worst design of the three designs. The maxima points' metric also found that Design C is the worst design. Finally, the bounded area metric found that all designs are equivalent. In contrast, for specific loading situations, our Capacity metric found that Design C is the optimal design.

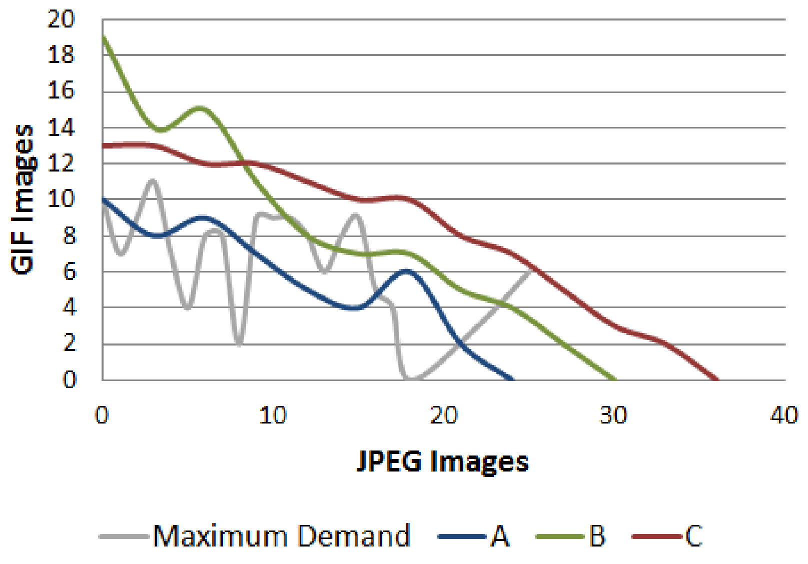

Figure 60: JPEG–GIF Capacity Plane

Now consider other demand and Capacity planes. Figure 60 shows Capacity and demand curves as a relationship between the production of JPEGs and GIF images. On the independent axis is the production of JPEGs and on the dependent axis is the production of GIF images. Note that we show in this figure the maximum demand curve only. Also note that we include all Capacity curves of Design A, Design B, and Design C on the same figure.

Now consider the demands generated from another usage scenario shown in Figure 56 (a businesswoman scenario). Again, we used our DCM method to generate all demands exposed to the system due to this scenario. Then we fitted these demands using the maximum demand curve. Similar to Figures 56 through 58, in Figures 60 through 62 we overlay the capacity curves of Design A, Design B, and Design C over the demand curve generated by the businesswoman scenario. Our goal from this experiment is to show how optimal design ranking changes as received demands change. Here, we view only one plane of the demand surface: the JPEG–text plane. Note that by looking at the maximum demand curve itself, you can see that it characterizes an unbalanced system loading when most demands are either JPEGs or text files. This is intuitive because the content of the webpages browsed in this scenario is either text (such as the content of Twitter, WSJ, or Gmail) or image (such as the content of Facebook, CNN, or BBC).

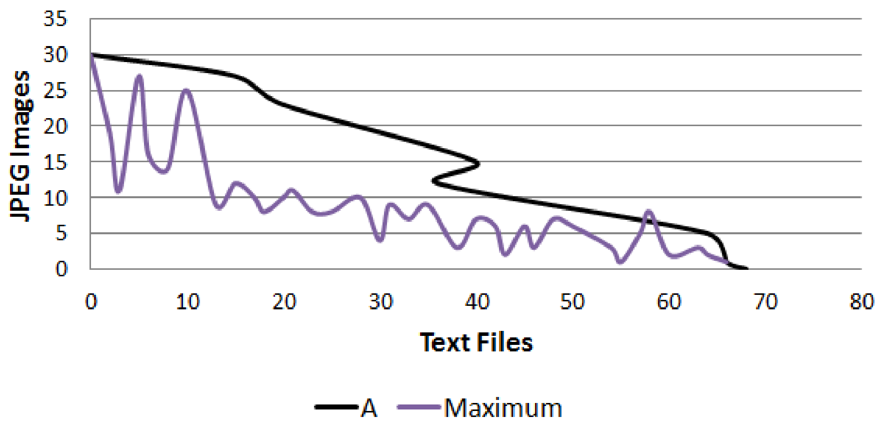

Figure 61: Overlay Demand Curves Over Design "A's" Capacity Curve

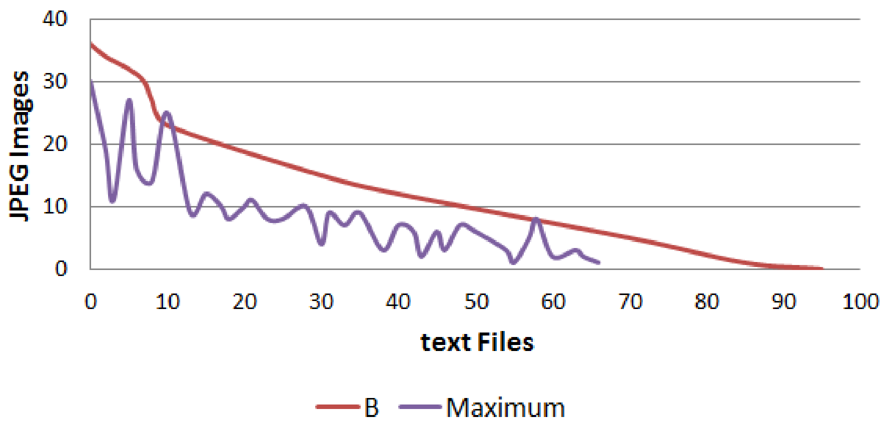

Figure 62: Overlay Demand Curves Over Design "B's" Capacity Curve

Figure 61 and Figure 62 show that the Capacity of Design A and Design B almost fits all demands. The only difference between these designs is that Design A has an excessive Capacity that is not needed by this type of demand. This excessive Capacity may cause an overhead on the system, such as more power consumed.

Figure 63 shows that Design C is the worst design because it does not fit all demands, while for some demand combination it has an excessive Capacity. Thus, the optimal design for this type of demand is Design B. Interestingly, the Capacity curve of Design B is a convex that favors unbalanced loads; this in turn matches the type of demand.

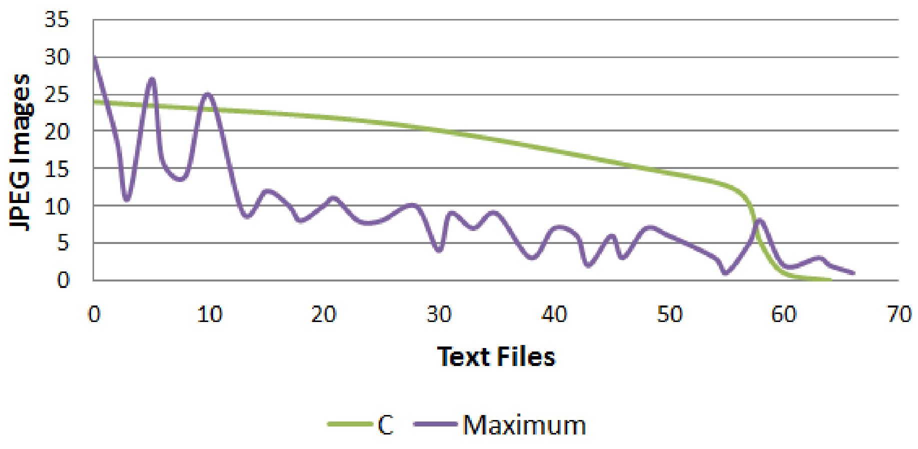

Figure 63: Overlay Demand Curves Over Design "C's" Capacity Curve

Here, we find four interesting observations from these experiments aside from which design fits the demand curve more.

- The Capacity curve (JPEG–GIF plane) of Design C is a concave that approaches the shape of a straight line. This is because Design C is a homogeneous processor (it has six DSP core processors) and both demand types (JPEGs and GIF) are of the same type of jobs that share some common processing requirements in, for example, the way they are migrated, access shared resources, etc. Thus, production of different job types can be replaced for each other at almost the same rate.
- Second, the shape and magnitude of the curves on one plane are not the same as on other planes. For instance, when producing JPEGs *only*, Design A and Design B produced fewer JPEGs in the JPEG–GIF plane compared with the

production of JPEGs in the JPEG–text plane. In contrast, Design C produced more JPEGs in the JPEG–GIF plane, again compared with the production of JPEGs in the JPEG–text plane. The presence of the ability to process an additional input stream results in an overhead, even in the absence of data on that input stream.

- Third, demand curves also have different shapes that reveal the type of the system loading. This in turn makes shape and magnitude matching between Capacity and demand curves possible.
- Finally, the Capacity curves (JPEG – GIF planes) of Design A and Design B are asymmetric (or not ideal), exhibiting multimodal shapes. The asymmetry is caused when multiple overheads interact due to the existence of heterogeneous cores that process a combination of heterogeneous inputs. Overhead occurs when multiple processors are physically or logically joined in the formation of a system and include, for example, communications (busses and networks), scheduling, protocols, memory sharing, I/O sharing, and caching. Next, we analyze the effects of overhead on the Capacity of the system.

7.3.3 Overhead Effects

Overhead has long been thoroughly studied in computer architecture in the essence of performance facilitation. While a primary goal has been to reduce or hide the effects of overhead, future successful CHM design may well require more exposure of the effects of overhead in specific situations. Examples for that are: the inclusion of more caches at the expense of other system resources, increasing the communication bandwidth at the expense of area and power consumption, and developing more computationally complex schedulers in terms of processing time and memory space. We have studied in Chapter 5 the cost of overhead on system optimization. We have also modeled overhead in section 5.2, including different elements that contribute to the overall system overhead.

Figure 64 shows both the Capacity curve of Design A and the overhead curve generated, while measuring the same Capacity curve. Each curve has different scale. The scale for the Capacity curve is on the left, while the scale of the overhead is on the right.

We use normalized values of overhead to show how overhead varies along with different combinations of demands.

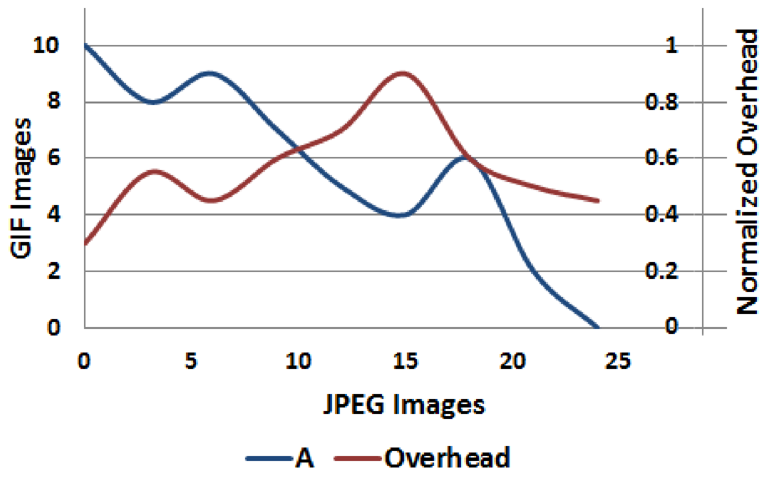

Figure 64: Capacity vs. Overhead

Note here several interesting observations:

- Overhead increases when different types of input exist. For instance, the overhead incurred due to optimizing the system to demands that are homogeneous (either JPEGs or GIF images) is relatively lower than that of other demands. Optimizing different types of jobs concurrently costs more because they require more complex optimization in addition to exposing different requirements on the processing, communication and storage resources.
- The cost of optimizing the system to produce only JPEGs is more than the cost of optimizing the system to produce only GIF images.
- Increasing the number of jobs on the system increases the incurred overhead. Note that as we are producing more JPEGs, the overhead increases.
- More interestingly, for some combination of demands, overhead increases exponentially because the system at these points suffers from communication and memory contention due mainly to task migration. Note that the choice of doing task migration at these points hurts the system's Capacity because of the incurred overhead; the gained performance did not overcome the increase in the overhead.

Note that the incurred overhead differs for the same set of tasks when architectures change. For instance, Design C incurs less overhead (shown in Appendix C) compared with Design A and Design B, especially when processing a homogeneous input, because it is a homogeneous processor.

7.3.4 Experiments Discussion

Here, we summarize some observations from our experiments:

- Optimal design ranking is a function of the type of input demands. Our initial goal was to show how future multicore mobile devices have to be designed to workloads that result from specific usage scenarios. In pursuing this goal, which we partially fulfilled, we found that single-valued performance metrics lead to ambiguity because they do not include the nature of workloads in their calculations. By developing the Capacity metric, we were able to integrate it with the definition of single-user usage scenarios in order to identify optimal designs for these scenarios.
- Different combination of demand types in the same scenario have different optimal design ranking. This is intuitive and falls under the category of the previous point in this list.
- Overhead is a key feature in design. We have also shown in these experiments how overhead affects the shape of the Capacity curve and the system performance overall. A solid model of overhead modeling is needed in addition to the Capacity curves in order to make the right optimization decisions.
- Power, area, and resource utilization should be explicitly considered. They are already implicitly considered because there are limits to the numbers and types of processors considered in our examples. It may happen that several designs match the same demand, the optimal design now is the one that consumes less power and utilizes system resources more efficiently, for example.

In this chapter we illustrated why single-valued performance metrics fail to evaluate the performance of the CHMs that simultaneously process heterogeneous outputs or demand modes. In contrast, our Capacity metric was able to find optimal designs for specific usage patterns, or Workload Specific Processors (WSPs).

Chapter 8

Conclusions and Future Research Directions

As for accomplishments, I just did what I had to do as things came along.
Eleanor Roosevelt

In this work we introduced a Capacity metric to evaluate the performance of Chip Heterogeneous Multiprocessors (CHMs) that simultaneously process heterogeneous outputs. Our metric is motivated as the successor to throughput via an analogy to automobile pipelines. Our Capacity curves are experimentally generated via measurement and are generally not linear when two input streams are processed. Thus, we showed that distillation of multi-type production over a given design (plant) to a common work unit, an average, precludes the designer from identifying cases in which the system has more Capacity than the average or cannot meet the predicted average production, in which the non-linear curves rise above or fall below the straight line (average) that occurs when a common work unit for throughput is used.

8.1 Summary of Contributions

We make several contributions in this work to identify optimal designs or WSPs. We classify these contributions into primary and secondary ones. Our primary contribution is the development of the Capacity metric.

Our metric is motivated as the successor to throughput via an analogy to automobile pipelines that produce single types of cars as compared with the Capacity of plants that produce multiple types of cars, using multiple pipelines. We developed the Capacity metric as a curve in two dimensions (or a surface in higher dimensions) with dimensionality related to the number of input streams, or channels, processed by the CHM. These curves give the designer the ability to understand the effects of adding architectural features on the performance of a computer system that simultaneously processes (produces) multiple output streams. Significantly, these curves do not result in linear relationships, as the demand for different production types varies. Distillation of rates of production over multiple production types to a common work unit, such as automobiles, instructions, tasks or programs, results in a linear relationship as rates of production of more specific types within those categories varies. But if a straight line was superimposed on any of our Capacity curves, the real Capacity of the plants (designs) being examined will either underestimate the true Capacity of a given plant for a given set of production values, or it will overestimate what the plant is capable of.

We illustrated some fundamental Capacity forms and showed how they may be used as the basis for evaluation and analysis of Capacity curves generated via measurement. Our overall goal is to motivate the development and use of the Capacity metric for performance evaluation of modern CHMs; specifically, we advocate investigation into how shapes of Capacity curves can be used to classify systems and identify how designs features can be manipulated in order to change the shape of the Capacity curves. In order to achieve this, we showed that Capacity surfaces have properties of both magnitude and shape, and each is required in order to understand workload performance.

This work also makes the following secondary contributions that made our primary contribution possible:

1) Three Foundations at the Chip Heterogeneous Core Level

We developed three foundations that we used to develop our Capacity metric: a CHM model, a multimedia cell phone example, and a WSP. Existing models for single core computers (i.e. those relying on an Register Transfer Level (RTL), Instruction Set Architectures (ISAs), and accompanying benchmarks) cannot be used to develop our Capacity metric because: (1) RTL key design features do not capture those features of CHM designs, (2) individual benchmarks do not accurately model the workload of single-user mobile devices, and (3) there is nothing such an ISA for CHMs. Thus, we needed to develop:

a) A multimedia cell phone example that permits the benchmarking of single-user mobile computing devices, to replace traditional benchmarks,

b) A CHM model equivalent to RTL for the types of architectures that are projected to be in future mobile devices, and
c) A WSP. WSPs are to CHMs is what an ISA is to an RTL. Part of our thesis is that WSPs more correctly capture what mobile computing devices do and that they result in categorically different approaches to processor design. WSPs are enabled by the presence of collections of individual user initiated job-style applications that arise as users interact with multicore mobile devices.

2) Workload Modes Definition and Identification

We defined workload modes, derived from models of single-user access patterns, as a means of design orientation and performance optimization for future single-user CHMs. Through experimentation on our cell phone example and additional analyses, we found that a workload classification model that leverages workload models of user patterns can outperform the more costly Hidden Markov Model (HMM). Our ultimate goal is to show how the design of multicore architecture to models of single-user usage patterns and their associated workload modes will be necessary as the complexity of applications and architectures grow in future designs. Thus, we advocate WSPs as a new means of orienting single-user CHMs.

We also developed a Workload Classification Model (WCM) to identify workload modes at runtime. We analyzed and evaluated the WCM and contrasted it to a HMM. We included experimentation on our cell phone example, illustrating how WCM is, on average, 34 times more time efficient and 83% more space efficient than HMM is, while improving overall performance by an average of 191% and being, on average, 56% more energy efficient. We found that even sub-optimal use of WCM can outperform HMM, further supporting the need for design-time workload models.

3) Capacity Metric Analysis Foundation

In addition to the development of the fundamental Capacity curve shapes, we established a foundation for analysis of the Capacity curves so that the impact of architectural features in a CHM may be better understood. In order to do this, we developed a Demand Characterization Method (DCM) that characterizes the demand of a specific usage pattern in the form of a curve (or a surface in general). By doing this, we were able to overlay demand curves over Capacity curves of different architectures to compare their performance and thus identify optimal performing designs.

8.2 Future Research Directions

This research focused on the development of a new performance metric, Capacity, to correctly evaluate the performance of the CHMs that process workloads. The overall goal is to identify optimal designs for specific usage patterns that result from individual users accessing multicore mobile devices; the basis we used to define these designs was WSPs. One of the key challenges in this research is the scalability of our metric with the growing number of inputs and processing elements. Here, we summarize two key challenges for future work.

8.2.1. Multi-Dimensionality Analysis

The Capacity metric results in a surface with n dimensions modeling the interaction of n multichannel streams as they compete for resources. Surface analysis needs to be done in order to extract features in a feature vector. Feature analysis is necessary in order to give insight to designers about the cause and effect of design features in a CHM to include processor choice, communications design, and scheduler decisions. Future systems are expected to have many channels of inputs, and graphical comparisons of the Capacity metric will not scale beyond three dimensions. While the breakdown of performance into multichannel input streams is required in order to understand how real user demand affects the design of CHMs, this breakdown must ultimately result in a means by which numerical analysis can be used as a basis of comparison for performance.

8.2.2 User Profiling

Since Capacity metric was originally emerged from the need to optimize CHMs to workload modes that result from individual users using multicore mobile devices, single-user usage profiling is essential for discovering real usage patterns or workloads. In this way, Capacity curves can be used to identify for each user where performance does not meet, exactly matches, or excessively meets his needs and by how much. In Chapter 4, we performed an extensive survey of existing methods on user profiling that can be used in that regard.

Appendix A

Capacity Curves

In Chapter 6, we focused on two input types, JPEG and text. Here, we analyze more results of our Capacity metric by including a third input type, GIF images. Thus, we show how different architectural features interact with the type of input by analyzing the Capacity curve's shape and magnitude. Our goal is to have more insight about the effects of these features and to be able to classify these effects into patterns.

Here, we summarize some interesting observations from our experiments by comparing the results in this Appendix to those of Chapter 6. We mainly compare the Capacity curves of GIF–text, JPEG–GIF, and JPEG–text. The JPEG–text curves are shown in Chapter 6 and the GIF–text and JPEG–GIF curves are shown in this Appendix. Our observations give more insight about the effects of the input type on the Capacity.

- *Burst Width*: JPEG–GIF inputs took the most advantage from increasing the burst width, compared with JPEG–text and GIF–text. Increasing the burst width favors unbalanced loads. Thus, since GIF and JPEG input types have approximately similar communication and memory access patterns, wider widths favor this type of inputs more.
- *Communication Bandwidth*: JPEG–GIF inputs took the most advantage from increasing the communication bandwidth as well because their processing requires more I/O accesses. Because both input types have approximately similar performance on different processor types, the shape of the curve approaches a straight line.
- *Processor Number and Type*: for the JPEG–GIF inputs, changing the number and type of processors, results almost in the same Capacity curve shape because the selected processor types perform well for both task types. The minor change in the shape happens because of the different overhead values

that result from the interaction of different architectural features. However, the magnitude of the curve changes according to the number of processors.

- *Cache Memory Size*: increasing the cache memory size has shown more advantage for JPEG–GIF because their processing requires more I/O accesses. However, because GIFs are animated, they took more advantage when cache size increased, making the Capacity curve shift toward the GIF axis.
- *Scheduler Type*: dynamic scheduling matters only when we have heterogeneous inputs. Because JPEG–GIF inputs are more homogeneous than GIF–text and JPEG–text inputs, dynamic scheduling shows less advantages for the JPEG–GIF, compared with the static scheduler. Also note that the incurred overhead that is shown in the form of the multimodality of the Capacity curve shapes is greater in the GIF–text and JPEG–text inputs because they are heterogeneous.
- *Time Interval*: the ratio of the Capacity to the time interval length has increased in the JPEG–GIF plane compared with the GIF–text and JPEG–text planes. As pointed out before, this ratio increases for unbalanced loads. Since GIF and JPEG have approximately similar performance on the different processor types on chip, the ratio approaches being linear, resulting in an almost straight-line Capacity curve shape.

Again, note that when multiple architectural features interact, multimodal shapes result.

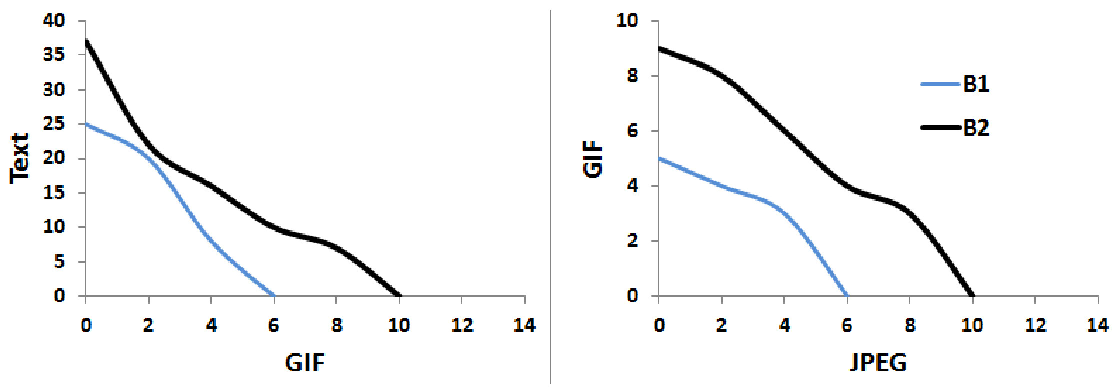

Figure 65: Capacity When Burst Width Increased

- *Burst width feature*

Figure 65 shows two Capacity curves of a CHM differentiated by increasing the bus burst width from 16B (B1) to 32B (B2). The CHM consists of 2G, 4D, and 6M processors and 128K of L2-cache memory. In the GIF–text Capacity curve, as the bus's burst width is changed from 16B (B1) to 32B (B2), the system changes from favoring balanced loads to unbalanced loads. Whereas, in the JPEG–GIF Capacity curve the architecture favors balanced loads. Also note that doubling the burst width results almost in the double Capacity.

- *Processor type and number features*

Figure 66 shows two Capacity curves for two different architectures (Arch2 and Arch3). In Arch2, we increased the number of GPP and Media processors, while reducing the number of DSPs. Note that adding this feature favors the mix of text and GIF but not their mix. The slope of the Capacity curve of Arch2 drops faster than that of Arch1. DSP processors have medium performance for processing both text and GIF thus they perform better than GPP and Media when a mix of text and GIF is being produced. On the other hand, both architectures show almost same performance when processing JPEG–GIF; this is because both types of jobs require the same processing, but differ in term of memory requirements. GIF images are animated thus they need more memory accesses.

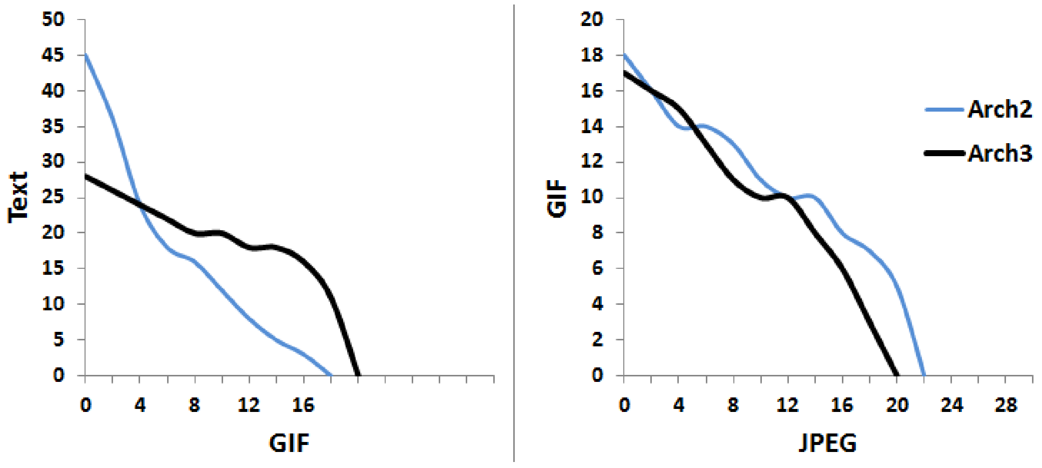

Figure 66: Capacity When More GPP and Media Added and DSPs Reduced

In Figure 67 we show the effects of increasing the number of Media processors while reducing the number of DSPs. The figure shows two Capacity curves for architectures: Arch3 and Arch4. Compared with Arch4, the number of Media processors in Arch3 has increased by two times, while reducing the number of DSPs by half. Note that adding this feature favors GIFs and text since Media processors perform better than DSPs in GIF and text processing. Note that the two curves differ not just in terms of magnitudes, but also shapes. The slope of the Capacity curve of Arch3 drops faster than that of Arch4, resulting in a convex shape.

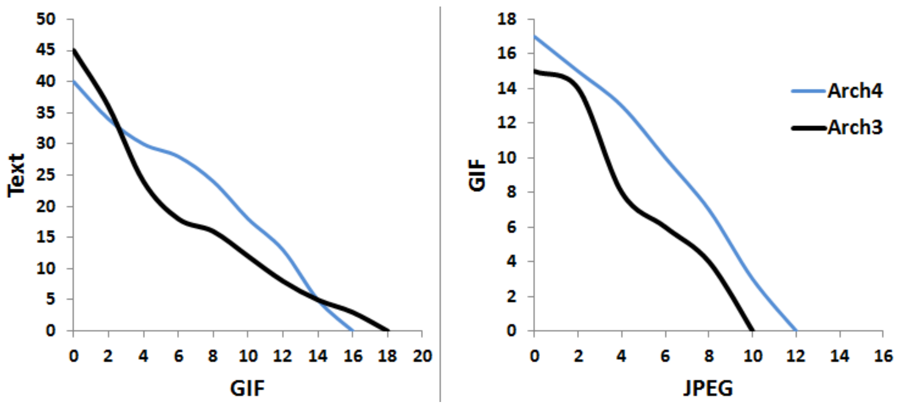

Figure 67: Capacity When More Media Processors Added and DSPs Reduced

For the JPEG–GIF Capacity curve, again both processor models perform the same for both job types, but JPEGs have less memory requirements because they are static images.

- *Cache size feature*

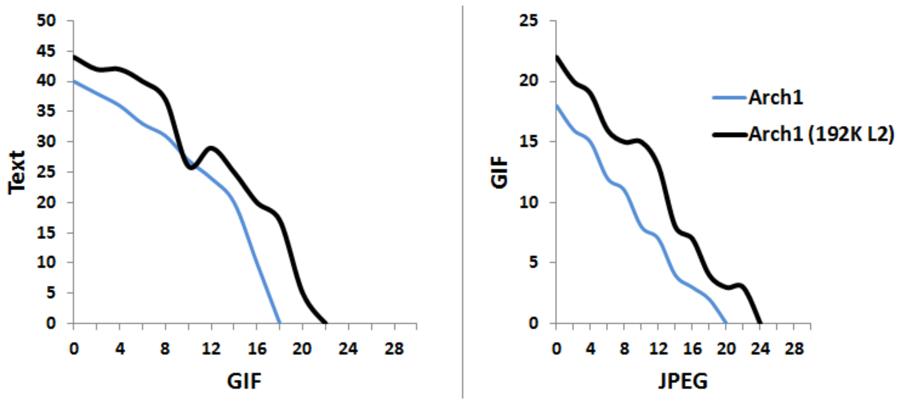

Figure 68: Capacity When Cache Size Is Increased (Same Processors)

Figure 68 shows the Capacity of the CHM of Figure 65 when the cache size increases by 1.5 times. As a consequence, the chip area has increased. The figure shows too little improvement in Capacity because the selected working set sizes of the text, GIF, and JPEG tasks are small. However, this feature favors GIF and JPEGs more since their processing can be considered to be internally I/O bound more than text processing.

Figure 69 shows two Capacity curves for two different architectures (Arch1 and Arch2) differentiated by increasing the cache size from 128K to 192K at the expense of the number of processors on the chip. Interestingly, the Capacity of both architectures to produce only text files is the same then the curves diverge significantly. In general, Arch2 shows major performance improvement because the selected size of working data sets of tasks is larger. Thus, the increase in the cache size helps in improving the performance. Note that in the case of producing text only, increasing the cache size does not result in any improvement because the working set size of text files is already smaller than GIFs. Again, increasing the cache size favors the production of task types with larger working data sets.

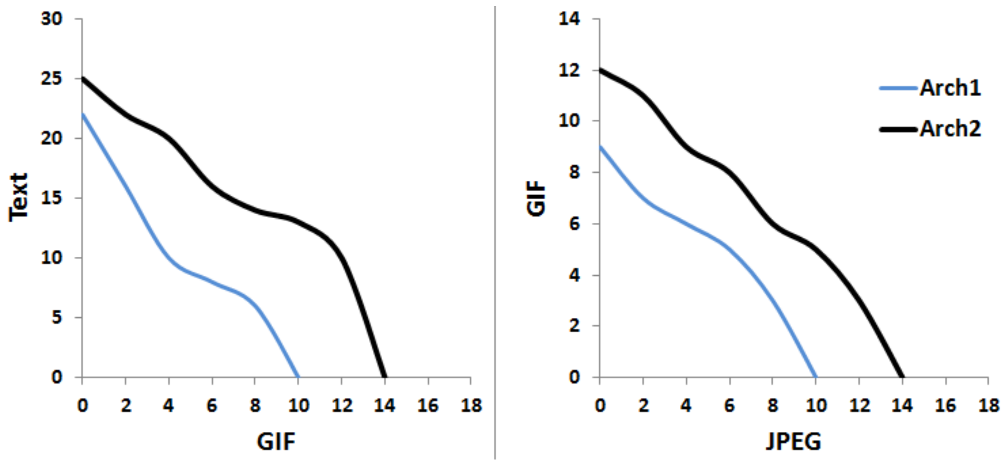

Figure 69: Capacity When Cache Size Increases (Less Processors)

For the JPEG–GIF Capacity curve, capacity has increased almost equally for both tasks because again they perform equally when executed on different architectures; only GIFs need more memory accesses because they are animated.

- *Communication bandwidth feature*

Figure 70 shows the Capacity of the CHM of Figure 37 when the overall communication bandwidth increases. C1 is 0.4GB/s and C2 is 1.2GB/s. The two curves in the GIF–text Capacity curve overlap at the beginning and then diverge. This occurs because communications Capacity affects GIF more than text. With respect to a given amount of text processing, GIF processing can be considered to be internally I/O bound more than text is with respect to a fixed amount of JPEG processing.

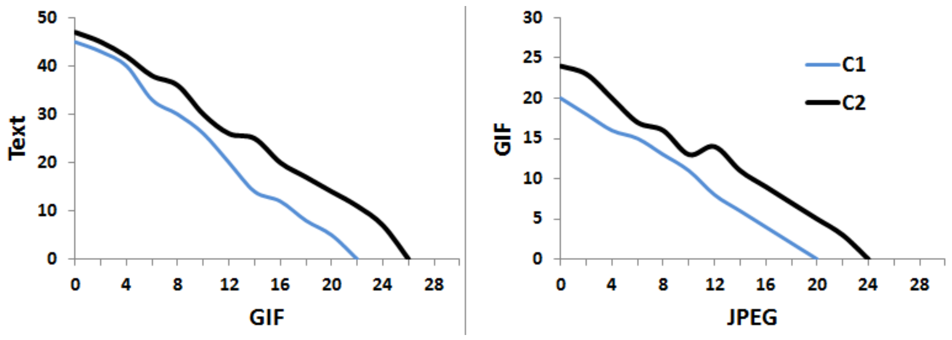

Figure 70: Capacity When Communication Bandwidth Increased

- *Scheduling feature*

Figure 71 and Figure 72 show two Capacity curves using two different chip level schedulers: dynamic and static. The CHM consists of 2G, 4D, and 6M processors and 128K of L2-cache memory. The two figures are differentiated by the selected working set size; it is smaller in Figure 71 and larger in Figure 72.

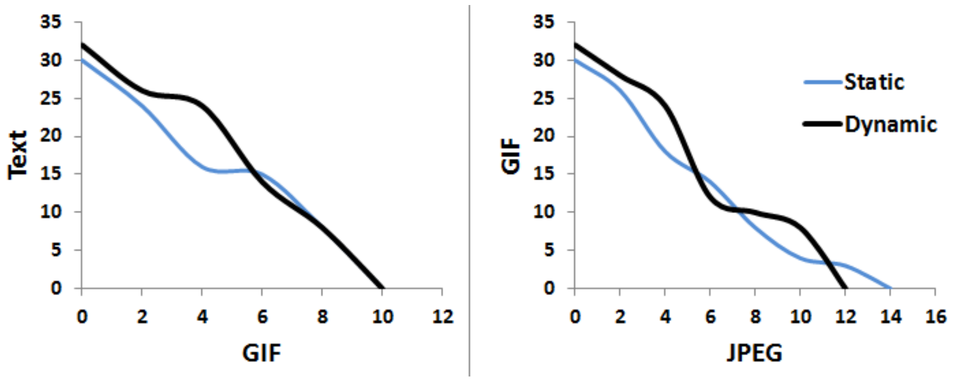

Figure 71: Capacity of Different Schedulers (Smaller WSS)

Figure 71 shows that both schedulers resulted in almost the same Capacity curves, because the working set size is small, which in turn makes task migration occur less. In contrast, when using larger working size sets, the dynamic scheduler outperforms the static scheduler, as shown in Figure 72. Note that dynamic schedulers result in multimodal shapes because of the incurred overhead.

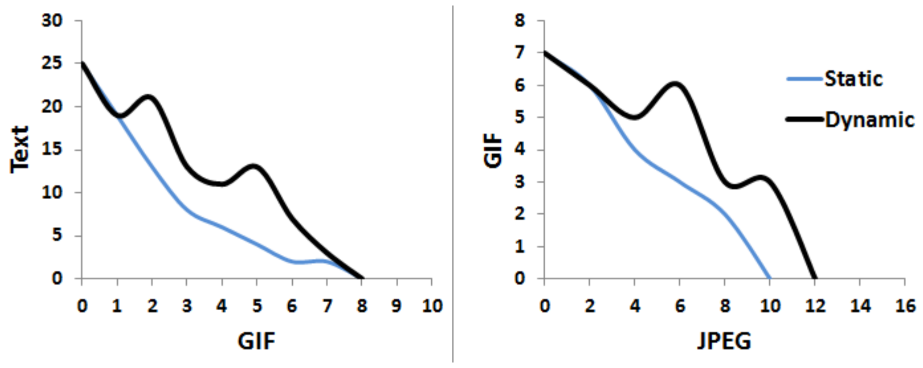

Figure 72: Capacity of Different Schedulers (Larger WSS)

- *Time window interval*

Figure 73 shows the Capacity when the time window in which we measure Capacity is doubled. Note that doubling the time window does not result in double Capacity because of the overhead generated due to resource sharing and running heterogeneous concurrent tasks. Again, increasing the window of time favors homogeneous inputs. The heterogeneity of inputs incurs more processing, communication and storage memory overhead. Note that for the JPEG–GIF Capacity curve, doubling the time interval results almost in double capacity even for the mixes of jobs. This is because both tasks perform almost the same on all processors.

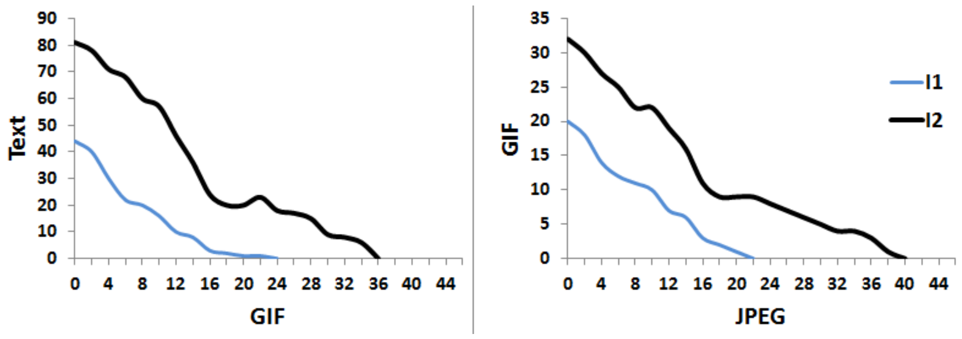

Figure 73: Capacity When the Time Window Doubled

Appendix B

3-D Capacity Surfaces

When a CHM is evaluated for its ability to process n channels of input, an n-D Capacity surface results. The way we analyzed an n-D Capacity surface in this thesis is by holding one input constant at a time. In that way, dimensionality can be reduced. For instance, a 3-D surface can be always reduced to a collection of 2-D curves. In general, an n-D surface can always be reduced to a collection of (*n-1*)-D surfaces, which can then be analyzed as collections. For the sake of completeness, we extend our example of Chapter 6 to include three input channels: JPEG, text, and Flash frames. Since 3-D surfaces can be graphically analyzed, we view the Capacity of this example by looking at the shape of the surface. We use this example to point out how our analysis of high dimensionality is effective.

Future systems are expected to have many channels of inputs, and graphical comparisons of the Capacity metric will not scale beyond three dimensions. While the breakdown of performance into multichannel input streams is required in order to understand how real user demand affects the design of CHMs, this breakdown must ultimately result in a means by which numerical analysis can be used as a basis of comparison for performance. Sammon Mapping [150] can be used to transform higher dimensional spaces into smaller ones. Kiviat diagrams [151] can be used to facilitate depiction of the Capacity metric in higher dimensions. In computer architecture, little research has been done on visualization of simulation results, because computer architects usually distill the performance of different architectures into single values. Tools that can be used to explore an n-dimensional performance data [152] aim at the clustering and reduction of multidimensional performance data [129].

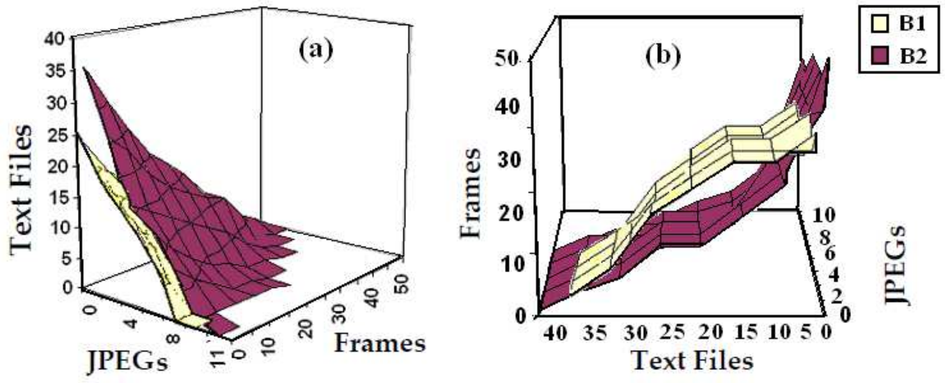

Figure 74: 3-D Capacity Surfaces

Figure 74 shows two different views of two 3-D Capacity surfaces that result when the same heterogeneous CHM is differentiated by increasing the bus burst width from 16B (B1) to 32B (B2) and is evaluated for its ability to process one more demand stream, or frames such as those found in Macromedia Flash. Note that the shapes of the curves on the JPEG–text plane are not the same as the previous Capacity curves, even in the absence of the frames channel. The presence of the ability to process an additional input stream results in overhead, even in the absence of data on that input stream. The convex and concave shapes in the Text-Frames plane, shown in Figure 74(b), suggest that the system changes from favoring balanced loads to unbalanced loads, with respect to frames and text. But Figure 74(a) suggests that the processing of text with respect to a variety of combinations of JPEG and frames takes on a more idealized, linear, shape, regardless of burst width. The JPEG–frames plane (not included) shows asymmetry in system Capacity curves. All of the shapes suggest that multiple factors contribute to a less than ideal shape, overall. In general, the more factors that affect the interaction of multichannel inputs, the less regular the resulting Capacity curves will be and the more modalities will contribute to those irregularities. While 3-D surfaces can be graphically analyzed, our analysis of higher dimensionality by breaking an n-D surface into a collection of its 2-D curves is more applicable.

Appendix C

Overhead Modeling

We have studied in Chapters 5 through 7 the effects of overhead on system optimization. Here, we include a detailed analysis of overhead. We include the different values of overhead incurred due to different combinations of inputs, and provide reasoning for why multimodal shapes result. We divided overhead into its main elements modeled in section 5.2: (1) chip level scheduling that includes collecting the GCS and making and sending scheduling decisions, (2) contention in memories and buses, (3) cache misses, and (4) the overhead of task migration.

Table 18: Overhead of the System in Figure 64

Workload Mode (JPEGs, GIFs)	Overhead (Mega Cycles)			
	Chip Level Scheduling	Global Resource Contention	Cache Misses	Task Migration
(0,10)	0.342	0.1	0.118	0.213
(3,8)	0.512	0.17	0.304	0.436
(6,9)	0.444	0.192	0.264	0.308
(9,7)	0.433	0.236	0.346	0.501
(12,5)	0.489	0.237	0.367	0.68
(15,4)	0.549	0.274	0.89	0.502
(18,6)	0.512	0.276	0.233	0.491
(21,2)	0.501	0.237	0.205	0.399
(24,0)	0.449	0.201	0.169	0.377

Table 18 shows the overhead for the Capacity curve of Figure 64. Note how these values incurred due to the arrival of a specific combination of demands are tied to the shape of the Capacity Curve on Figure 64.

Bibliography

[1] J.M. Paul, D.E. Thomas, and A. Bobrek, “Scenario-Oriented Design for Single-Chip Heterogeneous Multiprocessors,” *IEEE Transactions on VLSI Systems*, 14(8), pp. 868-880. 2006

[2] Y. Zhang, P.B Luh, K. Yoneda, T. Kano, and Y. Kyoya, “Mixed-model Assembly line Scheduling using the Lagrangian relaxation technique,” *IIE Transactions*. pp.125-134, 2003

[3] Semiconductor Industry Association: http://www.sia-online.org/

[4] ITRS Roadmap. System Drivers report of 2006

[5] **“**A D&T Roundtable: Are Single-Chip Multiprocessors in Reach?, ” *IEEE Design and Test of Computers*, 18(1), pp. 82-89, Jan/Feb, 2001

[6] W. Wolf. “How many system architectures?” *IEEE Computer*, pp. 93 -95. 2003

[7] L. Benini and G.D. De Micheli. “Networks on chips: A new SoC paradigm,” *IEEE Computer*. 35(1), pp. 70–78. Jan. 2002

[8] J.E. Smith. “Characterizing Computer Performance with a Single Number,” *Communications of the ACM*, 31(10), pp.1202-1206, 1988

[9] S.V. Gheorghita, M. Palkovic, J. Hamers, A. Vandecappelle, S. Mamagkakis, T. Basten, L. Eeckhout, H. Corporaal, F. Catthor, F. Vandeputte, and K.D. Bosschere, “System-scenario-based design of dynamic embedded systems, ” *ACM TODAES,* 14(1), pp. 1-44, 2009

[10] F. Vandeputte, L. Eeckhout, and K. DE Bosschere. “A detailed study on phase predictors, ” *International Euro- Par Conference*, J. Cunha and P. Medeiros, Eds. LNCS, vol. 3648. Springer, pp. 571–581. 2005

[11] http://www.multipixeldevelopment.com/iphone.php

[12] N.K. Jha. “Low power system scheduling and synthesis, ” *DAC*. pp:259–263. 2001

[13] D. Shin and J. Kim, “Optimizing intra-task voltage scheduling using data flow analysis,” *ASPDAC*. pp. 703–708. 2005

[14] B.H. Meyer, J.J. Pieper, J.M. Paul, J.E. Nelson, S.M. Pieper and A.G. Rowe, “Power-Performance Simulation and Design Strategies for Single-Chip Heterogeneous Multiprocessors,” *IEEE Transactions on Computers*, 54(6), pp. 684-697. 2005

[15] M. Gschwind, H.P Hofstee, B. Flaches, M. Hopkins, Y. Watanabe and T. Yamazaki. “Synergistic Processing in Cell's Multicore Architecture,” *IEEE Micro*, 26(2), pp.10-24, 2006.

[16] R. Kalla, S. Balaram, and J. M. Tendler, “IBM Power5 Chip: A Dual-core Multithreaded Processor,” *IEEE Micro*, 24(2), pp. 40-47, Mar-Apr 2004.

[17] B. Stolt, Y. Mittlefehldt, S. Dubey, G. Mittal, M. Lee, J. Friedrich, and E. Fluhr, “Design and Implementation of the POWER6 Microprocessor,” *IEEE Journal of Solid-State Circuits*, 43(1), pp. 21-28, Jan. 2008

[18] J. Dorsey, S. Searles, M. Ciraula, S. Johnson, N. Bujanos, D. Wu, M. Braganza, S. Meyers, E. Fang, and R. Kumar, “An Integrated Quad-Core Opteron Processor,”*IEEE International Solid-State Circuits Conference*, pp. 102-103, Feb. 2007

[19] S. Rusu, S. Tam, H. Muljono, D. Ayers, J. Chang, B. Cherkauer, J. Stinson, J. Benoit, R. Varada, J. Leung, R. D. Limaye, S Vora, “A 65-nm Dual-Core Multithreaded Xeon® Processor With 16-MB L3 Cache,” *IEEE Journal of Solid-State Circuits*, 42(1), pp. 17-25, Jan. 2007

[20] Quad-Core Intel® Xeon® Processor 5400 Series, 2008. Available from ftp://download.intel.com/products/processor/xeon/dc54kprodbrief.pdf.

[21] A.S. Leon, K.W. Tam, J. L. Shin, D. Weisner, and F. Schumacher, “A Power-Efficient High-Throughput 32-Thread SPARC Processor,” *IEEE Journal of Solid-State Circuits*, 42(1), pp. 7-16, Jan. 2007

[22] U. G. Nawathe, M. Hassan, K. C. Yen, A. Kumar, A. Ramachandran, and D. Greenhill, "Implementation of an 8-Core, 64-Thread, Power-Efficient SPARC Server on a Chip, " *IEEE Journal of Solid-State Circuits*, 43(1), pp. 6-20, Jan. 2008.

[23] D. Geer, "Chip Makers Turn to Multicore Processors," *IEEE Computer*, 38(5), pp. 11-13, May 2005

[24] D. Wentzlaff, P. Griffin, H. Hoffmann, B. Liewei, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown, and A. Agarwal, "On-Chip Interconnection Architecture of the Tile Processor," *IEEE Mico*, 27(5), pp. 15-31, Sept.-Oct. 2007

[25] AMD, *ATI Radeon™ HD 4800 Graphics Technology – Overview*, 2008. Available at http://ati.amd.com/products/radeonhd4800/index.html.

[26] AMD, *AMD FireStream™ 9250*, 2008. Available at http://ati.amd.com/technology/streamcomputing/product_firestream_9250.html

[27] J. Montrym and H. Moreton, "The GeForce 6800," *IEEE Micro*, 25(2), pp. 41-51, March-April 2005

[28] D. Luebke and G. Humphreys, "How GPUs Work," *IEEE Computer*, 40(2), pp. 96-100, Feb. 2007

[29] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro*, 28(2), pp. 39-55, March-April 2008

[30] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan "Larrabee: A Many-core x86 Architecture for Visual Computing," *ACM Transactions on Graphics*, 27(3), pp. 18:1–18:15, Aug. 2008

[31] M. Pericas, A. Cristal, F. J. Cazorla, R. Gonzalez, D. A. Jimenez, and M. Valero, "A Flexible Heterogeneous Multi-Core Architecture," *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pp. 13-24, Sept. 2007

[32] AMD, "The Future is Fusion. The Industry-Changing Impact of Accelerated Computing," AMD Whitepaper, 2008. Available from http://www.amd.com/us/MarketingDownloads/AMD_fusion_Whitepaper.pdf.

[33] Intel, "Tera-scale Computing Research Program," 2008. Available at: http://techresearch.intel.com/articles/Tera-Scale/1421.htm.

[34] M. Azimi, N. Cherukuri, D. N. Jayasimha, A. Kumar, P. Kundu, S. Park, I. Schoinas, and A. S. Vaidya "Integration Challenges and Tradeoffs for Tera-scale Architectures," *Intel Technology Journal*, pp. 173-184, 11(3), August 2007

[35] M. J. Schulte, J. Glossner, S. Jinturkar, M. Moudgill, S. Mamidi, and S. Vassiliadis, "A Low-Power Multithreaded Processor for Software Defined Radio," the *Journal of VLSI Signal Processing*,43(2-3), pp. 143 – 159, June 2006

[36] Texas Instruments, "DaVinci™ Technology Background and Specifications," 2007. Available from http://focus.ti.com/lit/an/sprt401a/sprt401a.pdf.

[37] S. Madhavapeddy and B. Carlson, "OMAP™ 3 Architecture from Texas Instruments Opens New Horizons for Mobile Internet Devices," Texas Instruments, Whitepaper, 2008. Available from http://focus.ti.com/pdfs/wtbu/ti_mid_whitepaper.pdf.

[38] U. Ramacher, "Software-Defined Radio Prospects for Multistandard Mobile Phones, " *IEEE Computer*, 40(10), pp. 62-69, Oct. 2007

[39] EEMBC: http://www.eembc.org/. Accessed March 2011

[40] SPEC: Standard Performance Evaluation Corporation, 2008. SPEC's benchmarks and published results, http://www.spec.org/benchmarks.html#web, accessed Sep. 2010..

[41] S. Pieper, J.M. Paul, and M. Schulte, "A New Era of Performance Evaluation, " *IEEE Computer*, 40(9). pp: 23-30. 2007

[42] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A free, commercially representative embedded benchmark suite", *Proceedings from 4th Workshop on Workload Characterization*, pp. 1-12, 2001

[43] C. Lee, M. Potkonjak, and W. Mangione-Smith, “MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems”, *International Symposium on Computer Architecture (ISCA)*, pp. 330-335, 1997

[44] B. Bishop, T. Kelliher, and M. Irwin, “A Detailed Analysis of Mediabench*”, In Proceedings of IEEE Workshop on Signal Processing Systems*, pp. 448-455, 1999

[45] J.M Paul, D.E. Thomas, and A. Bobrek, “Benchmark-Based Design Strategies for Single Chip Heterogeneous Multiprocessors”, *Proceedings from International Conference on Hardware Software Codesign*, pp 54-59. 2004

[46] J.M. Paul, M. Otoom, M. Somers, S. Pieper and M. J. Schulte: “The Emerging Landscape of Computer Performance Evaluation,” *Advances in Computers*, 46, pp. 235-280. 2009

[47] S. Woo, M. Ohara, E. Torrie, J. Sing, and A. Gupta, “The SPLASH-2 Programs: Characterization and methodological considerations,” *ISCA*, pp. 24–36. 2005

[48] M. Otoom and J.M. Paul. “Holistic Design and Caching in Mobile Computing,” *CODES+ISSS* , pp. 115-120, 2008

[49] R. Kumar, D.M. Tullsen, and N.P. Jouppi. “Core architecture optimization for heterogeneous chip multiprocessors, ” *PACT*, pp. 23 – 32, 2006.

[50] M. Flynn, “Some Computer Organizations and Their Effectiveness, ” *IEEE Trans. Computers*, C-21, pp. 948. 1972

[51] K. Skadron, M. Martonosi, D.I. A, M.D. Hill, D.J. Lilja, and V.S. Pai. “Challenges in Computer Architecture Evaluation,” *IEEE Computer*, pp. 30-36, 2003

[52] A.R. Alameldeen and D.A. Wood. “IPC Considered Harmful for Multiprocessor Workloads,” *IEEE Micro*, 26(4), pp. 8–17, 2006

[53] M.D. Hill and M.R. Marty. “Amdahl's Law in the Multicore Era,” *IEEE Computer*, pp.33-38, 2008

[54] A. Carlton, “CINT92 and CFP92 Homogeneous Capacity Method Offers Fair Measure of Processing Capacity,” *SPEC Newsletter,* 4(2), pp. 1-4. 1992. www.spec.org/cpu92/specrate.txt. Accessed Nov. 2010

[55] D. M. Tullsen, S. Eggers, J.S. Emer, H.M Levy, J.L Lo, and R.L Stamm, “Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor, ” *ISCA*, pp: 191–202, 1996

[56] D. M. Tullsen, S. J. Eggers, and H. M. Levy. “Simultaneous multithreading: Maximizing on-chip parallelism, ” *ISCA*, pp: 392–403, 1995

[57] R. Kumar, D.M. Tullsen, P. Ranganathan, N.P. Jouppi, and K.I Farkas, “Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance, ” *ISCA*, pp: 64-75, 2004

[58] J. Lee-jay Lo. “Exploiting Thread-Level Parallelism on Simultaneous Multithreaded Processors, ” *PhD thesis*, Uni. of Washington, 1998

[59] J. P. Singh, J. L. Hennessy, and A. Gupta. “Scaling parallel programs for multiprocessors: Methodology and examples, ” *IEEE Computer*, 27(7). pp:42–50, 1993

[60] L. Liu, Z. Li, and A.H. Sameh. “Analyzing memory access intensity in parallel programs on multicore, ” *ICS,* pp: 359-367. 2008

[61] Zhang, H. Y., Homer, J. P., Einicke, G., and Kubik, K. K. T., “Performance comparison and analysis of voice communication over ad hoc network,”. *1st IEEE International Conference on Wireless Broadband and Ultra Wideband Communications (AusWireless'06)*, Australia (2006).

[62] I. Ajbar, and D. Perkins, “Evaluating Mobile Ad Hoc Networks: A Performance Index and Statistical Model ,” *Proceedings of 16th International Conference on Computer Communications and Networks, 2007. ICCCN 2007,* pp: 284 – 290. 2007

[63] W.H. Yuen, and R.D. Yates, “Inter-relationships of performance metrics and system parameters in mobile ad hoc networks,” *MILCOM*, pp: 519 – 524, 2002

[64] G. Holland and N. Vaidya, “Analysis of TCP performance over mobile ad hoc networks,” *Proceedings of the fifth annual ACM/IEEE international conference on Mobile computing and networking.* August 1999.

[65] J. Broch, D.A. Maltz, D.B. Johnson, Y.C. Hu, and J. Jetcheva, “A performance comparison of multi-hop wireless ad hoc network routing protocols,” *Proc. Mobicom*, pp.85-97, 1998

[66] C.E. Perkins, E.M. Royer, S.R. Das, and M.K. Marina, “Performance comparison of two on-demand routing protocols for ad hoc networks,” *IEEE Personal Communications* , 8(1) , pp.16-28, 2001

[67] C. A. Santivanez, B. McDonald, I. Stavrakakis, and R. Ramanathan, “On the scalability of ad hoc routing protocols,” *Proc. of IEEE INFOCOM*, 2002

[68] N. Zhou and A. A. Abouzeid. “Routing in ad hoc networks: A theoretical framework with practical implications,” *Proc.of IEEE INFOCOM*, pp: 1240– 1251, March 2005

[69] W. Xianren, H. R. Sadjadpour, and J. J. Garcia-Luna-Aceves, “Routing overhead as a function of node mobility: Modeling framework and implications on proactive routing, ” *Proc. Of IEEE International Conference Mobile Adhoc and Sensor Systems*, pp: 1–9, Oct. 2007

[70] P. Jacquet and A. Laouiti, “Overhead in mobile ad-hoc network protocols, ” INRIA Research report 3965, 2000

[71] E. Royer and C-K. Toh, “A review of current routing protocols for ad hoc mobile wireless networks,” *In IEEE Personal Communication*, 6,pp: 46- 55, April 1999

[72] D. G. Feitelson, “Metric and workload effects on computer systems evaluation”. *Computer* 36(9), pp. 18–25, Sep 2003

[73] D. Ferrari, “Workload characterization and selection in computer performance measurement”. *Computer* 5(4), pp. 18–24, Jul/Aug 1972

[74] A. K. Agrawala, J. M. Mohr, and R. M. Bryant, “An approach to the workload characterization problem”. *Computer* 9(6), pp. 18–32, Jun 1976

[75] M. Calzarossa and G. Serazzi, “Workload characterization: a survey”. Proc. *IEEE* 81(8), pp. 1136–1150, Aug 1993

[76] S-H. Chiang and M.K. Vernon, “Characteristics of a large shared memory production workload,” *JSSPP*. pp. 159–187, 2001

[77] A.M. Law and W.D. Kelton. *Simulation Modeling and Analysis*. McGraw Hill, 3rd ed., 2000.

[78] R. Stets, K. Gharachorloo, and L.A. Barroso, “A detailed comparison of two transaction processing workloads,” *WWC-5*. pp. 37- 48, 2002

[79] A. Ramirez et al. “Code layout optimizations for transaction processing workloads,” *ISCA*, pp. 155-164, 2001

[80] A. Joshi, A. Phansalkar, L. Eeckhout and L. John. “Measuring Benchmark Similarity Using Inherent Program Characteristics,” *IEEE Transactions on Computers*, 55(6), pp. 769-782, 2006

[81] K. Hoste, A. Phansalkar, A. Georges and L. John. “Performance prediction based on inherent program similarity,” *PACT*, pp. 114-122. 2006

[82] H. Vandierendonck and K. De Bosschere. “Experiments with Subsetting Benchmark Suites,” *Workshop on Workload Characterization*, pp. 55-62. 2004

[83] K. Sevcik. “Characterization of Parallelism in Applications and Their Use in Scheduling,” *ACM SIGMETRICS Conf.*, pp. 171-180, 1989

[84] S. Majumdar, D. Eager, and R. Bunt. “Characterization of Programs for Scheduling in Multiprogrammed Parallel Systems,” *Performance Evaluation*, 13(2). pp:109-130, 1991

[85] D. Ferrari. “On the Foundations of Artificial Workload Design,” *ACM SIGMETRICS Conf.*, p: 8-14, 1984

[86] M. Calzarossa, G. Haring, G. Kotsis, A. Merlo, and D. Tessera. “A hierarchical approach to workload characterization for parallel systems,” In B. Hertzberger and G. Serazzi, editors, *High Performance Computing and Networking, LNCS,* 919, pp. 102–109. Springer, 1995

[87] D. Menasce, V. Almeida, R. Fonseca, and M. Mendes. “Resource management policies for e-commerce servers,” *ACM SIGMETRICS Performance Evaluation Review*, pp. 27-35. 2000

[88] M. Calzarossa, R. Marie, and K. Trivedi. “System performance with user behavior graphs,” *Performance Evaluation*, 11, pp.155–164, 1990

[89] C. Chen. “Structuring and visualizing the www by generalized similarity analysis,” *Hypertext and Hypermedia*, pp. 177-186, 1997

[90] R. Hofmann, R. Klar and B. Mohr, “A. Quick, and M. Siegle. Distributed Performance Monitoring: Methods, Tools, and Applications,” *IEEE Trans. on Parallel and Distributed Systems*, 5(6). pp. 585-598, 1994

[91] M. Calzarossa, L Massari, and D. Tessera. “Workload Characterization Issues and Methodologies,” *Performance Evaluation*. pp:459-481. 2000

[92] A. Jain and R. Dubes. *Algorithms for Clustering Data*, Prentice Hall, 1988.

[93] M. Calzarossa and G. Serazzi. “A Characterization of the Variation in Time of Workload Arrival Patterns,” *IEEE Trans. on Computers*, 34(2). pp.156-162, 1985

[94] S-H. Chiang and M.K. Vernon. “Characteristics of a large shared memory production workload,” *JSSPP*. pp. 159–187, 2001

[95] D.G. Feitelson and D. Tsafrir. “Workload sanitation for performance evaluation,” *ISPASS*. pp:221-230. 2006

[96] S. Hotovy. “Workload evolution on the Cornell Theory Center IBM SP2,” *Job Sched. Strat. for Parallel Proc.*, pp. 27–40, Springer-Verlag LNCS 1162, 1996

[97] K. Sreenivasan and A.J. Kleinman. “On the Construction of a Representative Synthetic Workload,” *Communications of the ACM*, 17(3). pp.127-133, 1974

[98] G. Serazzi. “A Functional and Resource-oriented Procedure for Workload Modeling,” In F.J. Kylstra, editor, *PERFORMANCE*, pp. 345-361. North-Holland, 1981

[99] M. Zhou and A.J. Smith. “Tracing Windows95,” *Technical Report No. UCB/CSD-99-1037*, Computer Science Division, UC Berkeley, November 1998

[100] A.K. Agrawala and J.M. Mohr. “A Markovian Model of a Job,” *CPEUG*, pp. 119-126, 1978

[101] G. Haring. “On Stochastic Models of Interactive Workloads.” In A.K. Agrawala and S.K. Tripathi, editors, *PERFORMANCE*, pp. 133-152. North-Holland, 1983

[102] B.M. Carlson, T.D. Wagner, L.W. Dowdy, and P.H. Worley. “Speedup Properties of Phases in the Execution Profile of Distributed Parallel Programs,” In R. Pooley and J. Hillston, editors, *Modeling Techniques and Tools for Computer Performance Evaluation*, pages 83-95. 1992

[103] A. Waheed and J. Yan. “Workload Characterization of CFD Applications Using Partial Differential Equation Solvers,” *Workshop on Workload Characterization in High-Performance Computing Environments*, 1998

[104] M. Somers and J.M. Paul. “Webpage-Based Benchmarks for Mobile Device Design,” Proc. of the *Asia and South Pacific Design Automation Conference.* pp. 795-800, Jan. 2008

[105] M. Spiliopoulou. “Data mining for the web,” In Proc. Of *Principles of Data Mining and Knowledge Discovery*, pp. 588-589. 1999

[106] R. Kosala and H. Blockeel, “Web Mining Research: A Survey, ” *SIGKDD Explorations*, 2(1). pp:1-15, 2000

[107] P. Langley, “User modeling in adaptive interfaces,” *International Conference on User Modeling*, pp. 357-370, 1999

[108] M. Perkowitz and O. Etzioni, “Adaptive web sites: an AI challenge,” *Int. Joint Conf. AI*, pp. 16--23, 1997

[109] M. Perkowitz and O. Etzioni, “Adaptive Sites: Automatically Learning From User Access Patterns,” *WWW*, poster no. 722, 1997

[110] M. Spiliopoulou, C. Pohle, and L. C. Faulstich, “Improving the effectiveness of a Web site with Web usage mining,” *WebKKD*, 1836/2000, pp:142-162. 2000

[111] http://www.ni.com/fpga/

[112] A. Poetter, J. Hunter, P. Athanas, and C. Patterson, “JHDLBits: The Merging of Two Worlds,” in *14th International Workshop on Field-Programmable Logic and Applications (FPL).* 2004.

[113] B. Hutchings, P. Bellows, J. Hawkins, S. Hemmert, B. Nelson, and M. Rytting, “A CAD Suite for High-Performance FPGA Design,” *in Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM*), pp. 12–24, 1999

[114] Xilinx, Inc., “JBits 3.0 SDK for Virtex-II.” http://www.xilinx.com/labs/projects/jbits/.

[115] A. Poetter, P. Athanas, and C. Patterson, “JHDLBits,” *in International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA’04)*, 2004

[116] M. Fujita and H. Nakamura, “The Standard SpecC Language,” *Proceedings from the International Symposium on System Synthesis (ISSS)*, pp. 81-86. 2001

[117] R. Domer, D. Gajski, and A. Gerstlauer, “SpecC Methodology for High-Level Modeling,” *IEEE/DATC Electronic Design Processor (EDP) Workshop*, 2002

[118] Ptolemy Group. Ptolemy II Website: http://ptolemy.eecs.berkeley.edu/ptolemyII/.

[119] H. Patel, S. Shukla, and R. Bergamaschi, “Heterogeneous Behavioral Hierarchy for System Level Designs”, *Proceedings from Design Automation and Test in Europe*, pp. 565-570. 2006

[120] I. Yarom, and G. Glasser, “SystemC Opportunities in Chip Design Flow,” *In Proceedings of the International Conference on Electronics, Circuits, and Systems (ICECS). 2004*

[121] M.K. Martin , D. J. Sorin , B. M. Beckmann , M. R. Marty, M. Xu , A. R. Alameldeen , K.E. Moore , M.D. Hill, and D.A. Wood, “Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset, ” *ACM SIGARCH Computer Architecture News*, 33(4), 2005

[122] http://embedded.eecs.berkeley.edu/mescal/forum/4.html

[123] J.M. Paul, D.E. Thomas, A.S. Cassidy. “High-Level Modeling and Simulation of Single-Chip Programmable Heterogeneous Multiprocessors,” *ACM TODAES*, 10(3), pp. 431-461. 2005

[124] A. Bobrek, J.M. Paul, and D.E. Thomas. “Stochastic Contention Level Simulation for Single Chip Heterogeneous Multiprocessors.,” *IEEE Trans. on computers*. pp: 1402-1418. 2010

[125] J.M. Paul, A. Bobrek, J.E. Nelson, J.J. Pieper, and D.E. Thomas. “Schedulers as Model-Based Design Elements in Programmable Heterogeneous Multiprocessors,” *DAC*, pp. 408-411, 2003

[126] A. Bobrek, J.J. Pieper, J.E. Nelson, J.M. Paul, and D.E. Thomas. “Modeling Shared Resource Contention Using a Hybrid Simulation/Analytical Approach,” *DATE*, pp. 1144-1149, 2004

[127] R. Covington, J. Jump, and J. Sinclair. “Cross-profiling as an efficient technique in simulating parallel computer systems,” *Computer Software and Applications Conference*, pp. 75–80, 1989

[128] J. R. Bammi, W. Kruijtzer, L. Lavagno, E. Harcourt, and M. T. Lazarescu. “Software performance estimation strategies in a system-level design tool,” *CODES*, pp. 82–86, 2000

[129] K. Hoste and L. Eeckhout. “Microarchitecture-Independent Workload Characterization,” *IEEE Micro*, 27(3), pp. 63-72, 2007

[130] SimpleScalar: www.simplescalar.com. Accessed on Jan. of 2011.

[131] R. Marculescu, D. Marculescu, and M. Pedram. “Composite sequence compaction for finite-state machines using block entropy and high-order Markov models,” *ISLPED*, pp. 190-195, 1997

[132] Y. Tan and Q. Qiu. “A Framework of Stochastic Power Management Using Hidden Markov Model,” *DATE*, pp. 92-97, 2008

[133] J. Lau and S. Schoenmackers, and B. Calder: “Transition Phase Classification and Prediction,” *HPCA*, pp.278– 289, 2005

[134] L.R. Rabiner. “A tutorial on hidden Markov models and selected applications in speech recognition,” *IEEE*, 77(2), pp. 257-285, 1989

[135] Analog Devices, www.analog.com/en/epProd/0,,ADSP-BF533,00.html. Accessed on June 2010

[136] PNX17xx Series, www.nxp.com/pip/PNX17XX_SER_N_1.html. Accessed on June 2010

[137] AMD-K6 Series, www.amd.com/epd/processors/6.32bitproc. Accessed on June 2010

[138] M. Mamidipaka, and N. Dutt. “eCacti: An Enhanced Power Estimation Model for On-chip Caches,” *Technical Report #04-28, UCI*, 2004

[139] J.M. Paul and B.H. Meyer. “Amdahl's Law Revisited for Single Chip Systems,” *IJPP*, 35(2), pp. 101-123, 2007

[140] F.R. Johnson and J.M. Paul. “Interrupt Modeling for Efficient High-Level Scheduler Design Space Exploration,” *ACM TODAES*, Vol. 13, No. 1, pp. 1-22, 2008.

[141] R.J. Elliot, L. Aggoun, and J. Moore. “Hidden Markov Models, Estimation and Control,” *Springer-Verlag*. New York. 1995

[142] K. Lahiri, A. Raghunathan, and G. Lakshminarayana. “LOTTERYBUS: a new high-performance communication architecture for system-on-chip designs,” *DAC*, pp. 15–20, 2001

[143] J. J. Pieper, A. Mellan, J.M. Paul, D.E. Thomas, and F. Karim. “High Level Cache Simulation for Heterogeneous Multiprocessors,” *DAC*, pp. 287-292, 2004

[144] “The Future of Computing, According to Intel”. Available online: http://www.technologyreview.com/business/19432/page1/. Jan. 2011.

[145] B.R. Binger and E. Hoffman, *Microeconomics with Calculus*, Addison-Wesley, second edition, 1998

[146] P.C Hansen and D.P. O'Leary. **“**The Use of the L-curve in the Regularization of Discrete Ill-posed Problems,” *SICOMP*, 14(6), pp. 1487 – 1503. 1993

[147] M. Otoom and J.M. Paul. “Workload Mode Identification for Chip Heterogeneous Multiprocessors,” *International Journal of Parallel Programming*, pp: 184-224. 2012.

[148] Web Analyzer: http://www.websiteoptimization.com/services/analyze/

[149] M. Cui, J. Femiane, J. Hu, P. Wonka, and A. Razdan, “Curve matching for open 2D curves,” Pattern Recognition Letters. 30(1). pp. 1-10. 2009

[150] J.W. Sammon jr. “A Nonlinear Mapping for Data Structure Analysis,” *IEEE TC*, C-18(5), pp. 401-409, 1969

[151] T. Conte, “Insight, not (random) numbers,” Talk at *ISPASS*, 2005

[152] K. Huck and A. Malony, “PerfExplorer: A Performance Data Mining Framework For Large-Scale Parallel Computing,” *SC*, pp. 41, 2005