Performance Prediction for Dynamic Voltage and Frequency Scaling

Rustam Raisovich Miftakhutdinov

High Performance Systems Group
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas 78712-0240

TR-HPS-2014-004
August 2014

This page is intentionally left blank.

The Dissertation Committee for Rustam Raisovich Miftakhutdinov certifies that this is the approved version of the following dissertation:

Performance Prediction for Dynamic Voltage and Frequency Scaling

Committee:

Yale N. Patt, Supervisor

Robert S. Chappell

Derek Chiou

Mattan Erez

Donald S. Fussell

Performance Prediction for Dynamic Voltage and Frequency Scaling

by

Rustam Raisovich Miftakhutdinov, B.S.E.E., M.S.E.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2014

Preface

> One of the biggest and most important tools of theoretical physics is the wastebasket.
>
> Richard Feynman

This dissertation is a culmination of three years of work, from Summer 2011 to Summer 2014. Although not a theoretical physicist, I spent much of that time true to the words of Richard Feynman—struggling to solve problems I ended up discarding. The time I spent on these discarded problems was, however, not in vain. Little by little, each one contributed to my broader understanding and, in the end, allowed me to formulate and solve the problem of this dissertation.

Many people have helped me along the way and I would like to acknowledge their contributions.

I would like to thank my advisor, Dr. Yale N. Patt, for the opportunity and advice he gave me and the patience he afforded me.

I would like to thank the rest of my doctoral committee, Dr. Robert S. Chappell, Dr. Derek Chiou, Dr. Mattan Erez, and Dr. Donald S. Fussell for their feedback on my research. I would like to thank Rob Chappell in particular for introducing me to my dissertation topic during my summer internship at Intel in 2011.

I would like to thank the many present and former members of the HPS research group that helped me in my research pursuits. Specifically, I would like to thank

- Eiman Ebrahimi, Onur Mutlu, and Francis Tseng for working with me on my research problems,
- José Joao for sacrificing his time to maintain the IT infrastructure of the research group,

- Rob Chappell, Chang Joo Lee, Hyesoon Kim, Onur Mutlu, Paul Racunas, and Santhosh Srinath for significant contributions to my simulation infrastructure, and

- Marco Alves, Eiman Ebrahimi, Faruk Guvenilir, Milad Hashemi, José Joao, Khubaib, Hyesoon Kim, Peter Kim, Ben Lin, Veynu Narasiman, Moin Qureshi, Aater Suleman, Francis Tseng, and Carlos Villavieja for sharing their research ideas and discussing mine.

I would like to thank Melanie Gulick and Leticia Lira for helping me navigate the administrative bureaucracy of the university.

Finally, I would like to thank Muawya Al-Otoom, Hari Angepat, Abhishek Das, Mark Dechene, Chris Fallin, Andy Glew, Min Jeong, Maysam Lavasani, Ikhwan Lee, Nikhil Patil, Mike Sullivan, Dam Sunwoo, Birgi Tamersoy, Gene Wu, and Dan Zhang for many intellectual discussions (research-related and otherwise).

Rustam Miftakhutdinov
August 2014, Austin, TX

Performance Prediction for Dynamic Voltage and Frequency Scaling

Publication No. _____

Rustam Raisovich Miftakhutdinov, Ph.D.
The University of Texas at Austin, 2014

Supervisor: Yale N. Patt

This dissertation proves the feasibility of accurate runtime prediction of processor performance under frequency scaling. The performance predictors developed in this dissertation allow processors capable of dynamic voltage and frequency scaling (DVFS) to improve their performance or energy efficiency by dynamically adapting chip or core voltages and frequencies to workload characteristics. The dissertation considers three processor configurations: the uniprocessor capable of chip-level DVFS, the private cache chip multiprocessor capable of per-core DVFS, and the shared cache chip multiprocessor capable of per-core DVFS. Depending on processor configuration, the presented performance predictors help the processor realize 72–85% of average oracle performance or energy efficiency gains.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

> Essentially, all models are wrong,
> but some are useful.
>
> George E. P. Box

Dynamic voltage and frequency scaling (DVFS) presents processor designers with both an opportunity and a problem. The opportunity comes from the multitude of voltage and frequency combinations, or *operating points*, now available to the processor. The operating point that maximizes performance (within a power budget) or energy efficiency depends on workload characteristics; hence, with DVFS, the processor can improve its performance or energy efficiency by switching to the best operating point for the running workload. The problem is to identify which operating point is the best at any given time.

One way to solve this problem is to equip the processor with a *performance predictor*, a mechanism capable of predicting what processor performance would be at any operating point. So equipped, a processor can improve its performance by periodically switching to the operating point predicted to yield the highest performance for the running workload. To improve energy efficiency in the same way, the processor would also need a power consumption predictor; however, since power prediction is simple if accurate performance prediction is available,[1] in this dissertation we focus primarily on performance predictors.

A performance predictor consists of

[1] Section 3.4.4 describes a power prediction scheme that relies on performance prediction.

1. a mathematical model that expresses processor performance as a function of the operating point and workload characteristics and

2. hardware mechanisms that measure these workload characteristics.

The model could be either

- *mechanistic*, that is derived from an understanding of how the mechanism (in our case, the processor) works, or

- *empirical*, that is based purely on empirical observations.

In this dissertation, we focus on DVFS performance predictors based on mechanistic models, which are more valuable than empirical models from a researcher's point of view. Most importantly, mechanistic models of processor performance advance our understanding of the major factors that drive processor performance, whereas empirical models, at best, merely show that performance is predictable without revealing why. Note that the assumptions underlying a mechanistic model may still be (and, in our case, often are) based on empirical observations.

This dissertation proves the feasibility of designing good DVFS performance predictors based on mechanistic models. We measure the goodness of a DVFS performance predictor by how well the predictor can guide DVFS; that is, how much of the benefit obtained by a hypothetical oracle predictor can the real predictor realize. In short, this dissertation proves the following thesis:

> A performance predictor comprised of a mechanistic model and hardware mechanisms to measure its parameters can guide dynamic voltage and frequency scaling well enough to realize most of the benefit obtained by an oracle predictor.

To prove this thesis, we design and evaluate DVFS performance predictors for three processor configurations: the uniprocessor, the private cache chip multiprocessor, and the shared cache chip multiprocessor. In Chapter 3, we develop a performance

predictor for the uniprocessor and use it to guide chip-level DVFS to improve energy efficiency. In Chapter 4, we develop a performance predictor for the private cache chip multiprocessor and use it to guide per-core DVFS to improve performance within a power budget. In Chapter 5, we show that the performance predictor for the private cache chip multiprocessor also works with a shared cache, explain why, and propose a more robust mechanism tailored for the shared cache configuration.

As we follow this path and develop new and more accurate DVFS performance predictors in the chapters ahead, we shall see two main points of focus emerge:

1. *Realistic memory systems.* We take care to consider the major features of modern memory systems:

   - variable DRAM request latencies resulting from DRAM timing constraints and DRAM scheduler queuing delays,
   - the commonly used stream prefetcher which may greatly increase DRAM bandwidth demand, and
   - the prioritization of demand (instruction fetch and data load) requests in DRAM scheduling.

   We pay particular attention to these details of modern memory systems because, as we shall soon see, memory system behavior largely determines the performance impact of frequency scaling—the same performance impact we want the processor to be able to predict.

2. *Performance impact of finite bandwidth.* As a result of our focus on the details of modern memory systems, we show that the commonly used stream prefetcher may lead to DRAM bandwidth saturation—an effect ignored by prior DVFS performance predictors. We design our DVFS performance predictors to take DRAM bandwidth saturation into account. For the uniprocessor, we model how finite bandwidth may limit processor performance. For the chip multiprocessor, we model how finite bandwidth, shared among the cores, may limit performance of some of the cores but not the others.

These points of focus are the reason that the DVFS performance predictors we develop significantly outperform the state-of-the-art and deliver close to oracle gains in energy efficiency and performance.

Chapter 2

Background

> Learning without thought is labor lost;
> thought without learning is perilous.
>
> Confucius

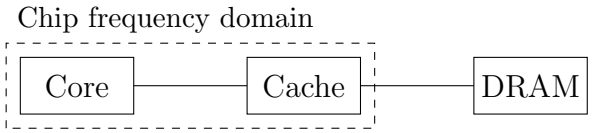
In this chapter, we present some background information on dynamic voltage and frequency scaling, the general approach of performance prediction, notational conventions, DRAM, and stream prefetching. We shall rely on this background information to explain our performance predictors in Chapters 3, 4, and 5.

2.1 Dynamic Voltage and Frequency Scaling

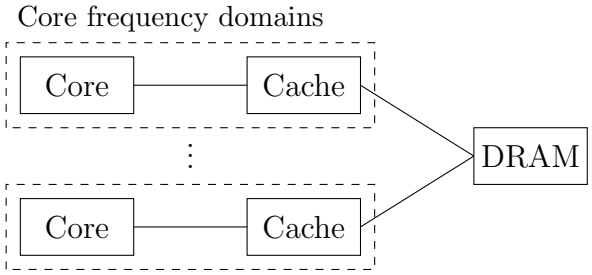
Dynamic voltage and frequency scaling (DVFS) [5, 27] allows the processor to change the supply voltage and operating frequency (the combination of which we call an *operating point*) of the whole chip or its parts at runtime. Voltage and frequency are generally scaled together because higher frequencies require higher voltages. Generally, performance increases at most linearly (but often sublinearly) with frequency whereas power consumption increases roughly cubically with frequency. [57]

The variation in performance impact of DVFS is due to variation in workload characteristics. For example, if a workload accesses off-chip memory often, the performance impact of DVFS is sublinear because DVFS does not affect off-chip memory latency. On the other, if a workload never misses in the cache, the performance impact of DVFS is linear with frequency.

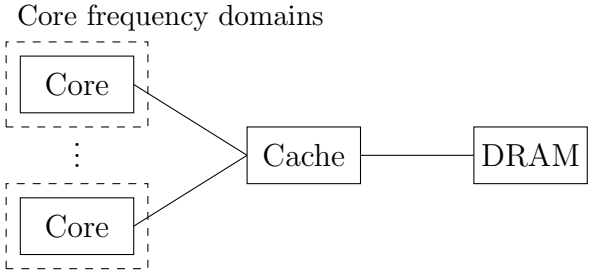
This variation in performance impact of DVFS means that the optimal operating point for the processor is workload-dependent. Whatever the target metric



(a) Uniprocessor addressed in Chapter 3



(b) Private cache CMP addressed in Chapter 4



(c) Shared cache CMP addressed in Chapter 5

Figure 2.1: Processor and DVFS configurations addressed in this dissertation

(such as energy, performance, or energy-delay product), DVFS endows the processor with the capability to dynamically adjust its operating point in order to improve that target metric. This capability gives rise to the central problem of this dissertation: that of choosing the optimal operating point at runtime.

Traditionally, DVFS has been applied at the chip level only; recently, however, other DVFS domains have been proposed. David et al. [16] propose DVFS for off-chip memory and Intel's Westmere [45] supports multiple voltage and frequency domains inside the chip. In this dissertation, as illustrated in Figure 2.1, we focus on reducing energy consumption using chip-level DVFS (Chapter 3) and improving performance within the power budget using per-core DVFS (Chapters 4 and 5).

2.2 Performance Prediction

Performance prediction is a way to improve processor performance or energy efficiency by adapting some adjustable parameter of the processor to better suit workload characteristics. As discussed in Chapter 1, a performance predictor generally consists of a) a mathematical model that expresses performance as a function of the adjustable parameter and some workload characteristics, and b) hardware mechanisms that measure the necessary workload characteristics.

The processor uses the performance predictor to control the adjustable parameter as follows:

1. For one interval, the performance predictor measures workload characteristics needed.

2. At the end of the interval, the predictor feeds the measured workload characteristics into the performance model.

3. The performance model estimates what the processor performance would be at every available setting of the adjustable parameter.

4. The processor changes the adjustable parameter to the setting predicted to maximize performance (or another target metric) and the process repeats for the next interval.

All performance predictors considered in this dissertation operate in this fashion.

Note the assumption implicit in these steps: the workload characteristics measured during the previous interval are expected to remain the same during the next interval. We make this assumption because we focus on predicting performance given known workload characteristics; predicting what the workload characteristics in the next interval would be is the problem of phase prediction—a very different problem [29, 30, 74, 87] which lies outside the scope of this dissertation.

Performance prediction has been used by prior work to dynamically control DVFS [10, 13–15, 18, 22, 37, 49, 61, 71], the number of running threads [82], shared cache partition [2, 66, 67], prefetcher aggressiveness [51], core structure sizes [19], DRAM bandwidth partition [53–55, 66, 67], and choice of core type in an asymmetric CMP [56, 86]. We present a more detailed overview of prior work on performance prediction in Chapter 6.

Two prior DVFS performance predictors are of particular interest to us: *leading loads* [22, 37, 71] and *stall time* [22, 37]. These works are the only previously proposed DVFS performance predictors based on a mechanistic model. We describe these predictors in detail in Section 3.1.

2.3 Notational Conventions

Performance prediction for DVFS and the associated mathematical models require nontraditional notation to express performance and frequency.

Traditionally, performance of a single application is expressed in instructions per cycle (IPC); however, this unit is inappropriate when chip or core frequencies are allowed to change. Therefore, in this dissertation, the fundamental measure of performance of a single application is instructions per unit time (IPT).

In addition, the mathematical models of performance under frequency scaling turn out to be easier to express not in terms of IPT and frequency, but rather in terms of their reciprocals. Specifically, we shall deal mostly with time per instruction (TPI) rather than IPT and cycle time rather than frequency.

2.4 DRAM

As we shall see later on, DRAM latency and bandwidth are important factors in processor performance, particularly under frequency scaling; thus we provide a brief overview of DRAM below.

Modern DRAM systems [31, 58] are organized into a hierarchy of *channels*, *banks*, and *rows*.[1] Each channel has a data bus connected to a set of banks. Each bank contains many rows and can have a single row open at any given time in its *row buffer*. All data stored in DRAM is statically mapped to some channel, bank, and row.

To access data stored in DRAM, the DRAM controller issues commands to close (or "precharge") the open row of the relevant bank, open (or "activate") the row mapped to the data needed, transferring the row to the bank's row buffer, and read or write the data over the data bus. Subsequent requests to the same row, called *row hits*, are satisfied much faster by data bus transfers out of or to the row buffer.

Modern DRAM controllers typically prioritize row hit requests, demand (instruction fetch and data load) requests, and oldest requests. Row hit prioritization [70] exploits row locality in data access patterns to reduce DRAM access latency and better exploit DRAM bandwidth; demand prioritization helps shorten the latency of core stalls caused by demand accesses.

Much of this dissertation deals with the impact of limited DRAM bandwidth on processor performance; hence, we take a closer look at the three DRAM bandwidth constraints:

1. *Row open bandwidth.* The "four activate window" (FAW) constraint limits the rate at which rows of a channel may be opened by allowing at most four row opens ("activates") in any window of t_{FAW} consecutive DRAM cycles (where t_{FAW} is a DRAM system parameter).

2. *Bank bandwidth.* The latencies needed to open, access, and close rows of a bank limit the rate at which the bank can satisfy DRAM requests.

[1] There are other elements of DRAM organization, such as *ranks* and *chips*; however, a description of these is not necessary to understand the performance impact of DRAM under frequency scaling.

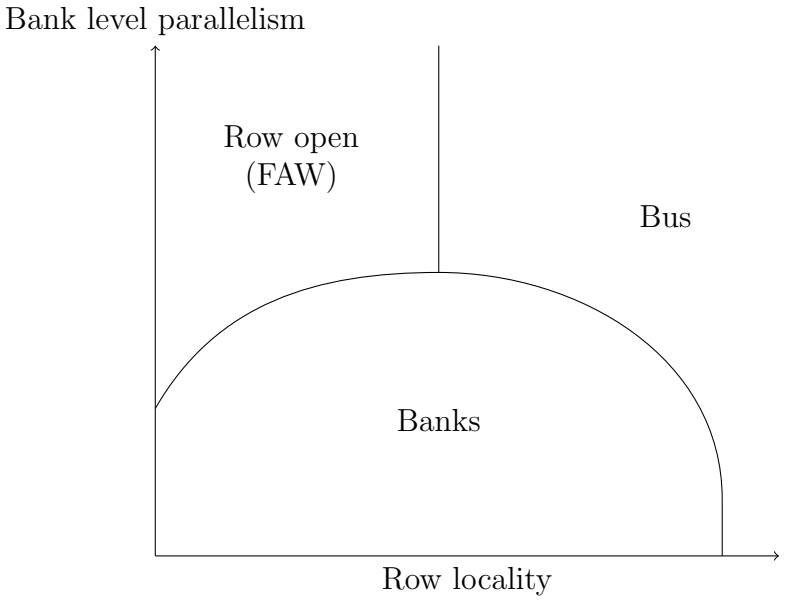


Figure 2.2: Qualitative relationship between row locality, bank level parallelism, and the dominant DRAM bandwidth bottleneck

3. *Bus bandwidth.* The time needed to transfer data over the bus and the overhead of changing the bus direction (read to write and vice versa) limit the rate at which the channel can satisfy DRAM requests.

Which of the three bandwidth constraints dominates depends on two parameters of the DRAM access stream: *row locality*, the number of row hits per row open, and *bank level parallelism*, the number of banks accessed simultaneously. Figure 2.2 shows the qualitative relationship between these two parameters and the dominant bandwidth bottleneck. The figure shows that

- the DRAM bus is the dominant DRAM bandwidth bottleneck when the DRAM access patterns exhibit large row buffer locality,
- the row open bandwidth is the dominant DRAM bandwidth bottleneck when the DRAM access patterns exhibit little row buffer locality but high bank level parallelism, and

- the DRAM banks are the dominant DRAM bandwidth bottleneck when both row buffer locality and bank level parallelism are relatively small.

As we show experimentally in Sections 3.3.1 and 4.1, a stream prefetcher (common in modern processors and described below) uncovers enough row locality to make the DRAM bus the major bandwidth bottleneck.

2.5 Stream Prefetching

Stream prefetchers are used in many commercial processors [6, 28, 47] and can greatly improve performance of memory intensive applications that stream through contiguous data arrays. Stream prefetchers do so by detecting memory access streams and generating memory requests for data the processor will request further down stream.

Figure 2.3 illustrates the high level operation of a stream prefetcher for a single stream of last level cache demand accesses to contiguous cache lines. The prefetcher handles a demand stream by progressing through three modes of operation:

1. *Training.* The prefetcher waits until the number of demand accesses fitting a stream access pattern crosses a *training threshold*.

2. *Ramp-up.* For every stream demand access, the prefetcher generates several prefetch requests (their number is the *degree*), building up the distance between the prefetch stream and the demand stream.

3. *Steady-state.* Once the distance between the prefetch stream and the demand stream reaches some threshold (the *distance*), the prefetcher tries to maintain that distance by sending out a single prefetch stream request for every demand stream request.

The prefetcher aims to generate prefetch requests early enough for them to bring data from DRAM before the corresponding demand requests are issued. Those prefetch

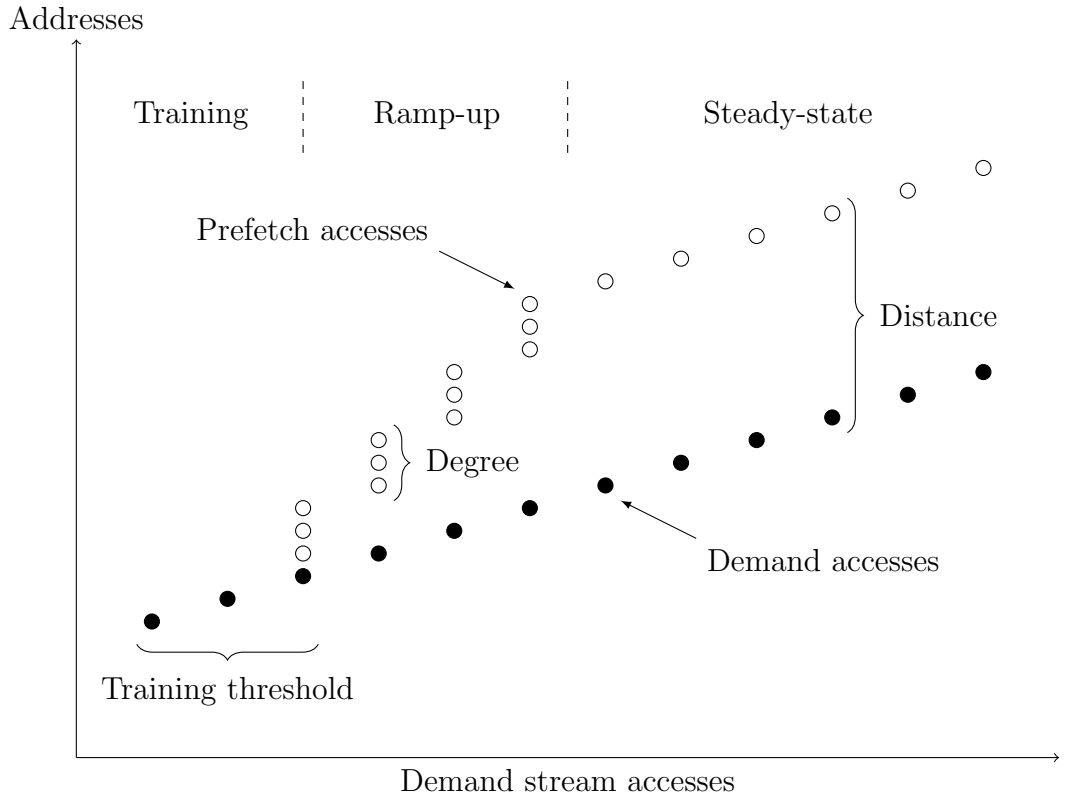


Figure 2.3: Stream prefetcher operation

requests that are early enough are called *timely*. The stream prefetcher distance is usually set (either at design time or dynamically [79]) to be large enough to make all prefetch requests issued in steady-state mode timely.

Chapter 3

Uniprocessor

Big things have small beginnings, sir.

Mr. Dryden
Lawrence of Arabia

In this chapter[1] we develop a DVFS performance predictor for the simplest processor configuration: the uniprocessor shown in Figure 3.1. We first describe the previously proposed predictors, *leading loads* and *stall time*, both based on a simple *linear* analytic model of performance. We then show experimentally that these predictors still have some room for improvement. Specifically, we show that

- the hardware mechanism used by the leading loads predictor to measure a key linear model parameter assumes a constant access latency memory system—an unrealistic assumption given the significant variation of DRAM request latencies in real DRAM systems, and
- the linear analytic model used by both predictors fails in the presence of prefetching because the model does not consider the performance impact of DRAM bandwidth saturation caused by prefetching.

[1] An earlier version of this chapter was previously published in [60].

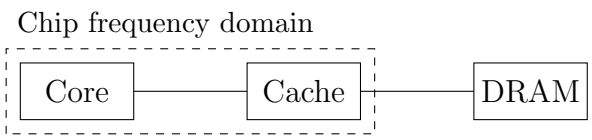


Figure 3.1: Uniprocessor

We address both of these shortcomings by

- designing a new hardware mechanism that accounts for variable DRAM request latencies when measuring the aforementioned linear model parameter, and
- developing a new *limited bandwidth* analytic model of performance under frequency scaling that does consider bandwidth saturation caused by prefetching.

Taken together, these improvements comprise our DVFS performance predictor for uniprocessors. According to this structure, we call this predictor *CRIT+BW*, since it is a sum of two parts: *CRIT*, the hardware mechanism for parameter measurement (which measures the length of a critical path through DRAM requests, hence the name "CRIT"), and *BW*, the limited bandwidth analytic model. We conclude the chapter by showing experimentally that CRIT+BW can make the processor more energy-efficient by guiding chip-level DVFS almost as well as an oracle predictor.

3.1 Background

We first describe the basic linear model of processor performance under frequency scaling; we then describe *leading loads* and *stall time*, the two previously proposed DVFS performance predictors based on the linear model.

3.1.1 Linear Model

The linear analytic model of performance under frequency scaling (*linear DVFS performance model* for short) arises from the observation that processor execution consists of two phases:

1. *compute*, that is on-chip computation, which continues until a burst of demand (instruction fetch or data load) DRAM accesses is generated, the processor runs out of ready instructions in the out-of-order instruction window, and stalls, and
2. *demand*, that is stalling while waiting for the generated demand DRAM accesses to complete.

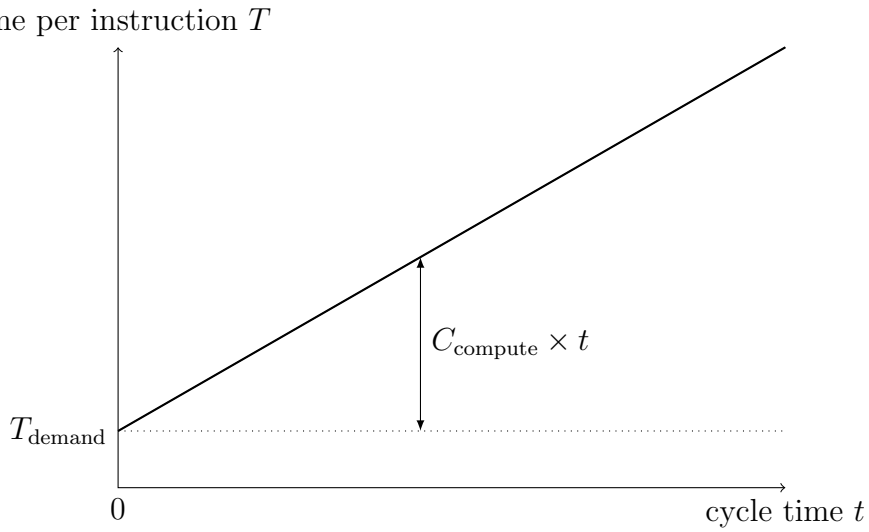


Figure 3.2: Linear DVFS performance model

This two-phase view of execution predicts a linear relationship between the average execution time per instruction T and chip cycle time t. To show this, we let

$$T = T_{\text{compute}} + T_{\text{demand}},$$

where T_{compute} denotes the average compute phase length per instruction and T_{demand} denotes the average demand phase length per instruction. As chip cycle time t changes due to DVFS, the average number of cycles C_{compute} the chip spends in compute phase per instruction stays constant; hence

$$T_{\text{compute}}(t) = C_{\text{compute}} \times t.$$

Meanwhile, T_{demand} remains constant for every frequency. Thus, given measurements of C_{compute} and T_{demand} at any cycle time, we can predict the average execution time per instruction at any other cycle time:

$$T(t) = C_{\text{compute}} \times t + T_{\text{demand}}. \tag{3.1}$$

This equation, illustrated in Figure 3.2, completely describes the linear DVFS performance model.

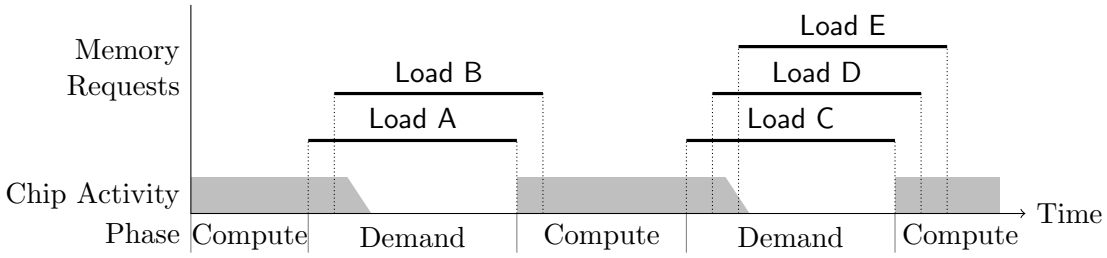


Figure 3.3: Abstract view of out-of-order execution with a constant access latency memory system assumed by leading loads

3.1.2 Leading Loads

Leading loads [22, 37, 71][2] is a previously proposed DVFS performance predictor based on the linear DVFS performance model and the assumption that the off-chip memory system has a constant access latency. Figure 3.3 shows the abstract view of execution implied by this assumption.

The major contribution of leading loads is its mechanism for measuring the two parameters of the linear model (T_{demand}, the average demand time per instruction, and C_{compute}, the average number of compute cycles per instruction) based on the constant access latency assumption.

We first start with measurement of T_{demand}. To measure T_{demand}, the leading loads predictor keeps a counter of total demand time and a counter of total instructions retired; T_{demand} is computed by dividing the former by the latter. Figure 3.3 shows that each demand request burst contributes the latency of a single demand request to the total demand time. The leading loads predictor measures that latency by measuring the length of the first (or "leading") demand request in the burst; since demand requests are usually data loads, this approach was named "leading loads" by Rountree et. al [71].

Once T_{demand} is measured, the other linear model parameter, C_{compute}, can be computed from T_{demand} and the easily measured execution time per instruction T.

[2]These three works propose very similar techniques. We use the name "leading loads" from Rountree et al. [71] for all three proposals.

Specifically, from Equation 3.1:

$$C_{\text{compute}} = \frac{T - T_{\text{demand}}}{t}. \tag{3.2}$$

The parameter measurement mechanism we just described and the the linear DVFS performance model comprise the leading loads predictor. This predictor can be used to control chip-level frequency as described in Section 2.2.

3.1.3 Stall Time

Like leading loads, the *stall time* [22, 37] DVFS predictor is the combination of the linear DVFS performance model and hardware mechanisms to measure its parameters. The key idea is simple: the time the processor spends unable to retire instructions due to an outstanding off-chip memory access should stay roughly constant as chip frequency is scaled (since this time depends largely on memory latency, which does not change with chip frequency). The stall time predictor uses this retirement stall time as a proxy for total demand time and computes T_{demand} by dividing the retirement stall time by the number of instructions retired. The C_{compute} parameter is computed exactly as in the leading loads predictor just described.

Unlike leading loads, the stall time predictor is not based on an abstract view of execution. Rather, the use of retirement stall time as a proxy for demand time is rooted in intuition.

3.2 CRIT: Accounting for Variable Access Latency Memory

Both leading loads and stall time DVFS performance predictors leave room for improvement; thus, we design an improved DVFS performance predictor we call *CRIT* (the reason for the name will soon become clear). Specifically, we note that

- leading loads is derived from an unrealistic abstract view of execution under a constant access latency memory system that fails to describe a more realistic variable access latency memory system, and

- stall time is rooted in intuition and is not based on an abstract view of execution, failing to provide a precise rationale for and hence confidence in its parameter measurement mechanism.

To overcome both of these shortcomings, we design our performance predictor from a more realistic abstract view of execution under a variable access latency memory system.

3.2.1 Experimental Observations

We first show experimentally that the abstract view of execution used by leading loads breaks in the presence of a real DRAM system.

Table 3.1 shows results of a simulation experiment on SPEC 2006 benchmarks.[3] In this experiment, we compare the length of an average "leading load" DRAM request to the length of an average demand DRAM request. Recall that the leading loads predictor is based on a view of execution where the leading loads have the same latency as the other demand DRAM requests. Table 3.1 shows that this view is incorrect for a modern DRAM system. In fact, the average leading load latency is generally less than the average demand DRAM request latency; the ratio is as low as 63% for `cactusADM`. This discrepancy makes sense in a realistic DRAM system where, unlike a constant access latency memory system, requests actually contend for service. Specifically, a "leading load" DRAM request is less likely to suffer from contention since, as the oldest demand request, it is prioritized in DRAM scheduling over other requests, including demand requests from the same burst. Note also that this discrepancy shows up in benchmarks like `bwaves`, `leslie3d`, and `milc` which spend a large fraction of execution time waiting on memory (as seen in the last table column). For these benchmarks, inaccurate measurement of demand time per instruction T_{demand} is most problematic, since the fraction of the error in T_{demand}

[3]Section 3.4 details the experimental methodology and simulated processor configuration.

Benchmark	Average leading load latency, cycles	Average DRAM request latency, cycles	Leading load latency relative to average demand request latency, %	Demand fraction of execution time as measured by leading loads, %
astar	151	164	92	19
bwaves	138	188	73	37
bzip2	146	171	86	14
cactusADM	182	290	63	25
calculix	135	135	100	9
dealII	106	108	98	6
gamess	99	112	89	0
gcc	121	126	96	9
GemsFDTD	181	215	84	45
gobmk	156	157	100	3
gromacs	101	109	92	4
h264ref	109	113	96	7
hmmer	141	145	97	0
lbm	241	252	96	48
leslie3d	118	161	73	44
libquantum	104	109	95	65
mcf	190	211	90	62
milc	137	173	79	64
namd	93	97	96	1
omnetpp	161	173	93	58
perlbench	153	162	95	3
povray	103	113	91	0
sjeng	175	188	93	5
soplex	112	132	85	61
sphinx3	105	118	89	50
tonto	92	104	89	1
wrf	123	173	71	19
xalancbmk	161	173	93	15
zeusmp	168	179	94	44

Table 3.1: Applicability of leading loads

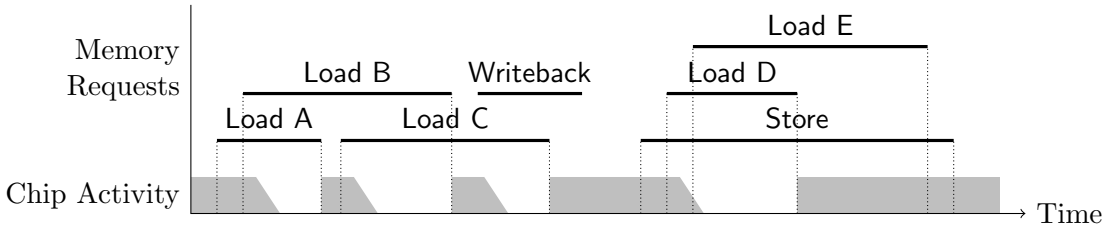


Figure 3.4: Abstract view of out-of-order processor execution with a variable latency memory system

measurement that propagates into the total predicted execution time per instruction $T = T_{\text{demand}} + T_{\text{compute}}$ is proportional to T_{demand}.

Thus we conclude that the abstract view of processor execution used by the leading loads predictor does not apply to a processor with a realistic DRAM system.

3.2.2 Variable Access Latency View of Processor Execution

This conclusion motivates the parameter measurement mechanism of our CRIT performance predictors; specifically, we design CRIT from a more realistic variable access latency view of processor execution.

Figure 3.4 illustrates the abstract view of processor execution when memory latency is allowed to vary. Note that the processor still eventually stalls under demand (instruction fetch and data load) memory requests, but the lengths of these requests are different.

The introduction of variable memory access latencies complicates the task of measuring the demand time per instruction T_{demand}. We must now calculate how execution time per instruction is affected by multiple demand requests with very different behaviors. Some of these requests are dependent and thus serialized (the first must return its data to the chip before the second one can be issued). Specifically, there are two kinds of dependence between requests:

1. *program dependence*, that is, when the address of the second request is computed from the data brought in by the first request, and

2. *resource dependence*, such as when the out-of-order instruction window is too small to simultaneously contain both instructions corresponding to the two memory requests.

Other requests, however, overlap freely.

To estimate T_{demand} in this case, we recognize that in the linear DVFS performance model, T_{demand} is the limit of execution time per instruction as chip frequency approaches infinity (or, equivalently, as chip cycle time approaches zero). In that hypothetical scenario, the execution time equals the length of the longest chain of dependent demand requests. We refer to this chain as the *critical path* through the demand requests.

To calculate the critical path, we must know which demand DRAM requests are dependent (and remain serialized at all frequencies) and which are not. We observe that independent demand requests almost never serialize; the processor generates independent requests as early as possible to overlap their latencies. Hence, we make the following assumption:

> If two demand DRAM requests are serialized (the first one completes before the second one starts), the second one depends on the first one.

3.2.3 Hardware Mechanism

We now describe CRIT, the hardware mechanism that uses the above assumption to estimate the critical path through load and fetch memory requests. CRIT maintains one global critical path counter P_{global} and, for each outstanding DRAM request i, a critical path timestamp P_i. Initially, the counter and timestamps are set to zero. When a request i enters the memory controller, the mechanism copies P_{global} into P_i. After some time ΔT the request completes its data transfer over the DRAM bus. At that time, if the request was generated by an instruction fetch or a data load, CRIT sets $P_{\text{global}} = \max(P_{\text{global}}, P_i + \Delta T)$. As such, after each fetch or load request i,

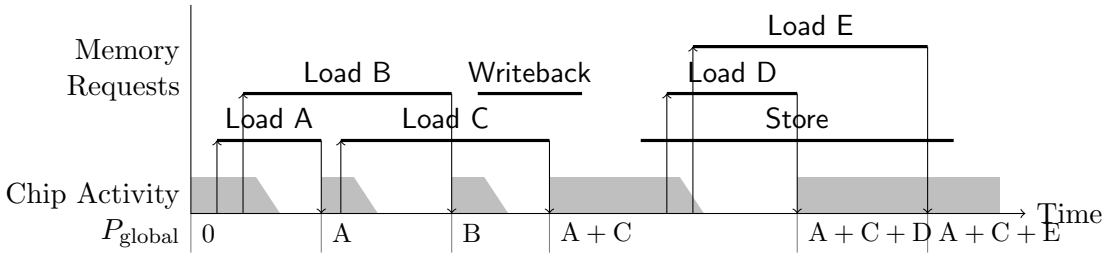


Figure 3.5: Critical path calculation example

CRIT updates P_{global} if request i is at the end of the new longest path through the memory requests.

Figure 3.5 illustrates how the mechanism works. We explain the example step by step:

1. At the beginning of the example, P_{global} is zero and the chip is in a compute phase.

2. Eventually, the chip incurs two load misses in the last level cache and generates two memory requests, labeled Load A and Load B. These misses make copies of P_{global}, which is still zero at that time.

3. Load A completes and returns data to the chip. Our mechanism adds the request's latency, denoted as A, to the request's copy of P_{global}. The sum represents the length of the critical path through Load A. Since the sum is greater than P_{global}, which is still zero at that time, the mechanism sets P_{global} to A.

4. Load A's data triggers more instructions in the chip, which generate the Load C request. Load C makes a copy of P_{global}, which now has the value A (the latency of Load A). Initializing the critical path timestamp of Load C with the value A captures the dependence between Load A and Load C: the latency of Load C will eventually be added to that of Load A.

5. Load B completes and ends up with B as its version of the critical path length. Since B is greater than A, B replaces A as the length of the global critical path.

6. Load C completes and computes its version of the critical path length as A + C. Again, since A + C > B, CRIT sets P_{global} to A + C. Note that A + C is indeed the length of the critical path through Load A, Load B, and Load C.

7. We ignore the writeback and the store because they do not cause a processor stall.[4]

8. Finally, the chip generates requests Load D and Load E, which add their latencies to A + C and eventually result in $P_{\text{global}} = \text{A} + \text{C} + \text{E}$.

We can easily verify the example by tracing the longest path between dependent loads, which indeed turns out to be the path through Load A, Load C, and Load E. Note that, in this example, leading loads would incorrectly estimate T_{demand} as A + C + D.

3.2.4 Summary

We have just described our hardware mechanism for measuring demand time per instruction T_{demand}, which is a workload characteristic and a parameter of the linear analytic model. The other parameter of the model, C_{compute} can be computed using Equation 3.2 as done by both leading loads and stall time predictors. Our mechanisms to measure/compute these parameters together with the linear analytic model comprise our CRIT performance predictor. We defer its evaluation until Section 3.5.1.

3.3 BW: Accounting for DRAM Bandwidth Saturation

Having proposed CRIT, a new parameter measurement mechanism for the linear analytic model, we now turn our attention to the linear model itself. In this section, we show experimentally that the linear model does not account for the performance impact of DRAM bandwidth saturation caused by prefetching. We then fix

[4]Stores may actually cause processor stalls due to insufficient store buffer capacity or memory consistency constraints [88]. CRIT can be easily modified to account for this behavior by treating such stalling stores as demands.

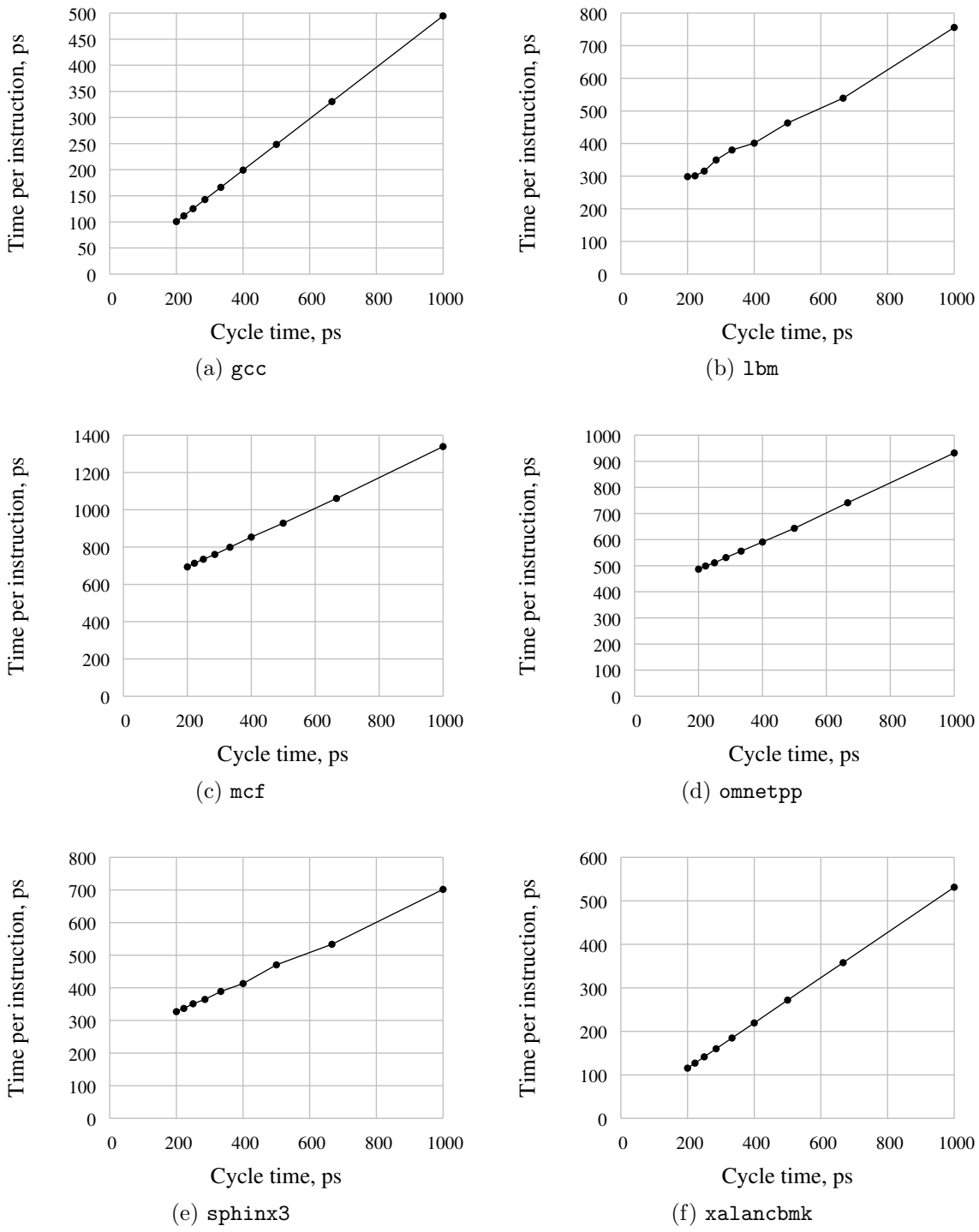
this problem and develop a new limited bandwidth analytic model on top of the linear model by taking into account bandwidth saturation. We also augment CRIT to work with this new analytic model and design hardware to measure an extra parameter needed by the new model. All together, these hardware mechanisms and the new limited bandwidth model comprise CRIT+BW, our DVFS performance predictor for uniprocessors.

3.3.1 Experimental Observations

Figure 3.6 shows how performance of six SPEC 2006 benchmarks scales under frequency scaling when prefetching is off. The six plots were obtained by simulating a 100K instruction interval of each benchmark at a range of frequencies from 1 GHz to 5 GHz. Note that most plots, except the one for `lbm`, match the linear analytic model. Comparing these plots to Figure 3.2, we see that some benchmarks (like `gcc` and `xalancbmk`) exhibit low demand time per instruction T_{demand}, spending most execution time in on-chip computation even at 5 GHz. Others (like `mcf` and `omnetpp`) exhibit high T_{demand} and spend most of their execution time at 5 GHz stalled for demand DRAM requests. Both kinds, however, are well described by the linear analytic model. The apparent exception `lbm` follows the linear model for most of the frequency range, but seems to taper off at 5 GHz. We shall soon see the reason for this anomaly.

Figure 3.6 shows how performance of six SPEC 2006 benchmarks scales under frequency scaling in the presence of a stream prefetcher (Section 2.5) over 100K instruction intervals where these benchmarks exhibit streaming data access patterns. Note that none of the plots match the linear model: instead of decreasing linearly with chip cycle time, the time per instruction saturates at some point.

The linear model fails to describe processor performance in these examples due to the special nature of prefetching. Unlike demand DRAM requests, a prefetch DRAM request is issued in advance of the instruction that consumes the request's data. Recall from Section 2.5 that a prefetch request is *timely* if it fills the cache before



(a) `gcc` (b) `lbm`

(c) `mcf` (d) `omnetpp`

(e) `sphinx3` (f) `xalancbmk`

Figure 3.6: Linear model applicability with prefetching off: performance impact of frequency scaling on 100K instruction intervals of SPEC 2006 benchmarks

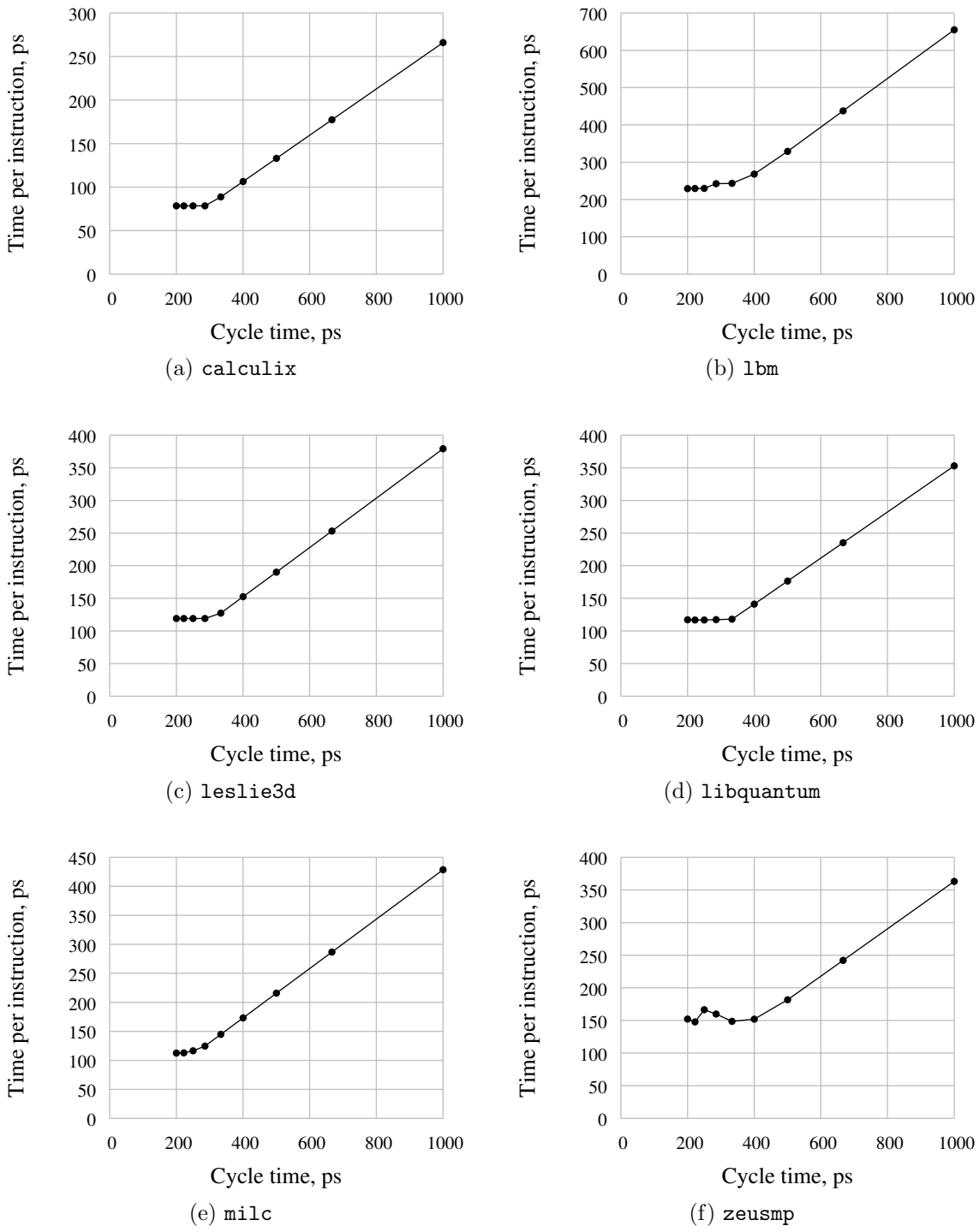


Figure 3.7: Linear model failure with prefetching on: performance impact of frequency scaling on 100K instruction intervals of prefetcher-friendly SPEC 2006 benchmarks

the consumer instruction accesses the cache. Timely prefetches do not cause processor stalls; hence, their latencies do not affect execution time. Without stalls, however, the processor may generate prefetch requests at a high rate, exposing another performance limiter: the rate at which the DRAM system can satisfy DRAM requests—the DRAM bandwidth.

Table 3.2 provides more insight into DRAM bandwidth saturation. Recall from Section 2.4 that modern DRAM has three potential bandwidth bottlenecks: bus, banks, and row open rate (determined by the "four activate window", or FAW, timing constraint). For each SPEC 2006 benchmark, the table lists the fraction of time each of these potential bandwidth bottlenecks is more than 90% utilized. The left side of the table shows this data for simulation experiments with prefetching off; the right side shows results with a stream prefetcher enabled. Note that bandwidth saturation of any potential bottleneck is rare when prefetching is off (with the exception of `lbm` which explains its anomalous behavior seen earlier). On the other hand, in the presence of a stream prefetcher, the DRAM bus is not only often saturated but is also the only significant DRAM bandwidth bottleneck.

The simulation results in Table 3.2 clearly show that the DRAM bus is the major DRAM bandwidth bottleneck; we now explain why. To become bandwidth-bound, a workload must generate DRAM requests at a high rate. DRAM requests can be generated at a high rate if they are independent; that is, the data brought in by one is not needed to generate the addresses of the others. In contrast, dependent requests (for example, those generated during linked data structure traversals) cannot be generated at a high rate, since the processor must wait for a long latency DRAM request to complete before generating the dependent request. The independence of DRAM requests needed to saturate DRAM bandwidth is a hallmark of streaming workloads, that is workloads that access data by iterating over arrays (either consecutively or using a short stride). An extreme example is `lbm`, which simultaneously streams over ten arrays (which are actually subarrays within two three-dimensional arrays). The memory layout of such arrays (e.g., column-major or row-major) is usually cho-

	Fraction of time each potential DRAM bandwidth bottleneck is more than 90% utilized, in %					
	Prefetching off			Prefetching on		
Benchmark	Bus	Banks	FAW	Bus	Banks	FAW
astar	2.3	0.0	0.0	2.7	0.0	0.0
bwaves	0.3	4.5	0.0	88.9	0.3	0.0
bzip2	6.5	0.1	0.0	12.0	0.0	0.0
cactusADM	8.0	0.7	0.0	7.9	0.6	0.0
calculix	7.1	0.0	0.0	7.9	0.0	0.0
dealII	0.2	0.0	0.0	0.4	0.0	0.0
gamess	0.0	0.0	0.0	0.0	0.0	0.0
gcc	0.3	0.0	0.0	0.8	0.0	0.0
GemsFDTD	6.1	0.2	0.0	48.4	0.2	0.0
gobmk	0.1	0.0	0.0	0.2	0.0	0.0
gromacs	0.2	0.0	0.0	0.2	0.0	0.0
h264ref	0.8	0.0	0.0	2.1	0.0	0.0
hmmer	0.0	0.0	0.0	0.1	0.0	0.0
lbm	94.9	0.1	0.0	99.3	0.0	0.0
leslie3d	1.8	5.4	0.0	43.4	0.1	0.0
libquantum	17.4	0.7	0.0	80.9	0.0	0.0
mcf	0.1	0.0	0.0	0.2	0.0	0.0
milc	18.0	2.1	0.0	39.7	0.0	0.0
namd	0.1	0.0	0.0	0.1	0.0	0.0
omnetpp	0.3	0.0	0.0	0.9	0.0	0.0
perlbench	0.5	0.0	0.0	0.6	0.0	0.0
povray	0.0	0.0	0.0	0.0	0.0	0.0
sjeng	0.0	0.0	0.0	0.0	0.0	0.0
soplex	2.7	1.7	0.0	71.7	1.4	0.0
sphinx3	1.0	0.3	0.0	36.7	0.2	0.0
tonto	0.2	0.0	0.0	0.0	0.0	0.0
wrf	1.3	1.4	0.0	7.4	0.3	0.0
xalancbmk	0.2	0.0	0.0	1.4	0.0	0.0
zeusmp	0.4	0.1	0.0	1.2	0.0	0.0

Table 3.2: Bandwidth bottlenecks in the uniprocessor

sen with these streaming DRAM access patterns in mind in order to exploit row locality, making the DRAM bus the most likely bandwidth bottleneck even without prefetching. Once on, the stream prefetcher further speeds up streaming workloads and uncovers even more row locality; therefore, streaming workloads become even more likely to saturate the DRAM bus.[5]

Now that we uncovered the performance limiting effect of DRAM bandwidth saturation and observed the DRAM bus to be the dominant DRAM bandwidth bottleneck, we are ready to develop a new DVFS performance predictor that takes these observations into account. We start with a new analytic model and then develop hardware mechanisms to measure its parameters.

3.3.2 Limited Bandwidth Analytic Model

We now describe the limited bandwidth analytic model of uniprocessor performance under frequency scaling, illustrated in Figure 3.8, that takes into account the performance limiting effect of finite memory bandwidth exposed by prefetching. This model splits the chip frequency range into two parts:

1. the low frequency range where the DRAM system can service memory requests at a higher rate than the chip generates them, and

2. the high frequency range where the DRAM system cannot service memory requests at the rate they are generated.

In the low frequency range, shown to the right of $t_{\text{crossover}}$ in Figure 3.8, the prefetcher runs ahead of the demand stream because the DRAM system can satisfy prefetch requests at the rate the prefetcher generates them. Therefore, DRAM bandwidth is not a performance bottleneck in this case. In fact, in this case the time per

[5]Irregular access prefetchers [12, 32, 35, 89] (yet to be used in commercial processors) may also saturate DRAM bandwidth; however, unlike stream prefetchers, these prefetchers generate requests mapped to different DRAM rows. In this high bandwidth yet low row locality scenario, the DRAM bus may no longer be the dominant bandwidth bottleneck.

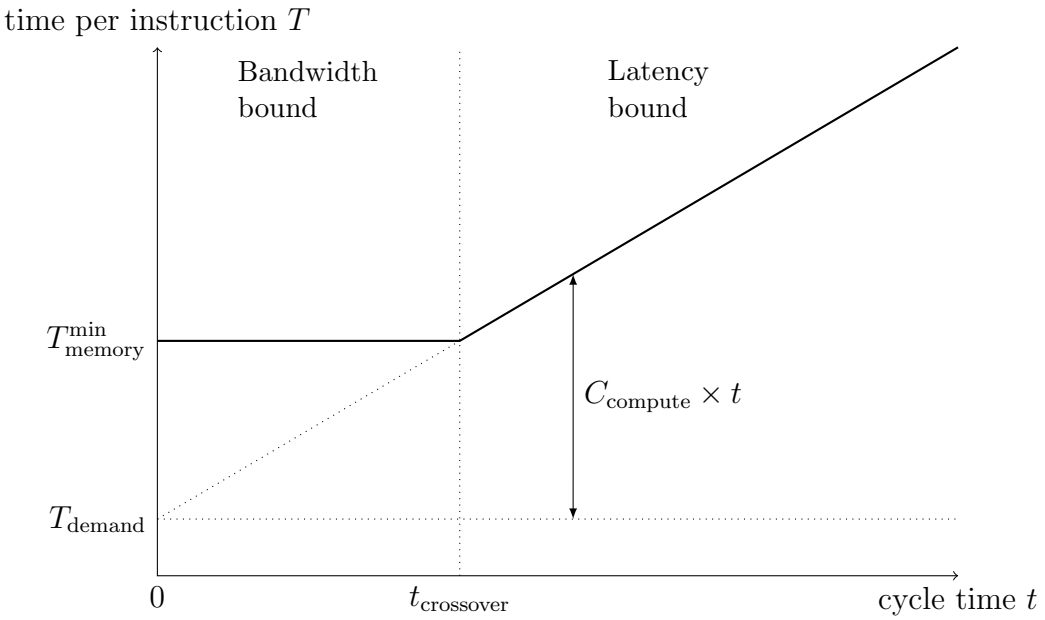


Figure 3.8: Limited bandwidth DVFS performance model

instruction T is modeled by the original linear model, with only the non-prefetchable demand memory requests contributing to the demand time per instruction T_{demand}.

In the high frequency range, shown to the left of $t_{\text{crossover}}$ in Figure 3.8, the prefetcher fails to run ahead of the demand stream due to insufficient DRAM bandwidth. As the demand stream catches up to the prefetches, some demand requests stall the processor as they demand data that the prefetch requests have not yet brought into the cache. In this high frequency range the execution time per instruction is determined solely by $T^{\min}_{\text{memory}}$: the minimum average time per instruction that the DRAM system needs to satisfy all of the memory requests. Therefore, time per instruction T does not depend on chip frequency in this case.

The limited bandwidth DVFS performance model shown in Figure 3.8 has three parameters:

1. the demand time per instruction T_{demand},

2. the number of compute cycles per instruction C_{compute}, and

3. the minimum memory time per instruction $T^{\min}_{\text{memory}}$.

Given the values of these parameters, we can estimate the execution time per instruction T at any other cycle time t as follows:

$$T(t) = \max\left(T^{\min}_{\text{memory}},\ C_{\text{compute}} \times t + T_{\text{demand}}\right). \tag{3.3}$$

3.3.2.1 Approximations Behind $T^{\min}_{\text{memory}}$

In this section, we address the implicit approximations related to the $T^{\min}_{\text{memory}}$ workload characteristic used by the limited bandwidth model. Specifically, our notion that the minimum memory time per instruction $T^{\min}_{\text{memory}}$ is a workload characteristic that stays constant across chip frequencies relies on two approximations:

1. The average number of DRAM requests per instruction (both demands and prefetches) remains constant across chip frequencies. This approximation is generally accurate in the absence of prefetching; however, with prefetching on this approximation is less accurate. Specifically, modern prefetchers may adapt their aggressiveness based on bandwidth consumption, which may change with chip frequency, causing the average number of DRAM requests per instruction to also vary with chip frequency.

2. The DRAM scheduler efficiency remains the same across chip frequencies. In particular, we approximate that the average overhead of switching the DRAM bus direction per DRAM request remains the same across chip frequencies.

3.3.3 Parameter Measurement

We now develop the hardware mechanisms for measuring the three parameters of the limited bandwidth analytic model. These mechanisms are complicated by the fact that the analytic model allows for two modes of processor operation: latency-bound and bandwidth-bound. Therefore we have to ensure our mechanisms work in both modes.

3.3.3.1 Demand Time per Instruction T_{demand}

In Section 3.2 we have already developed the CRIT hardware mechanism to measure demand time per instruction T_{demand} for the linear analytic model; however, it requires a slight alteration to work with our new limited bandwidth analytic model.

In fact, CRIT does not measure T_{demand} correctly in the bandwidth-bound mode of operation. Like leading loads and stall time, CRIT is based on the assumption that the number of demand DRAM requests per instruction and their latencies remain (on average) the same across chip frequencies. Prefetching in bandwidth-bound mode violates this assumption.

This effect concerns prefetch DRAM requests that are timely when the core is latency-bound (recall from Section 2.5 that *timely* prefetch requests bring their data into the cache before a demand request for that data is generated). Specifically, DRAM bandwidth saturation at higher chip frequencies increases DRAM request service time, causing some of these timely prefetch requests to become untimely. In fact, limited bandwidth may force the prefetcher to drop some prefetch requests altogether due to the prefetch queue and MSHRs being full, causing extra demand requests for the would be prefetched data. As a result, when bandwidth-bound, a core appears to exhibit more demand time per instruction than when latency-bound (due to a higher number of demand requests and untimely prefetch requests than in latency-bound mode). This effect is undesirable because for a DVFS performance predictor to be accurate all workload characteristics must stay the same across core frequencies.

To solve this problem we add extra functionality to the existing stream prefetcher to classify demand requests and untimely prefetch requests as "would be timely" if the core were latency-bound. This classification is based on the two modes the stream prefetcher uses to prefetch a stream (see Section 2.5 for details): a) the *ramp up* mode, started right after the stream is detected, in which the prefetcher continually increases the distance between the prefetch stream and the demand stream, and b) the *steady state* mode, which the prefetcher enters after the distance between the prefetch stream

and the demand stream becomes large enough to make the prefetch requests timely. As demand accesses to the last level cache are reported to the stream prefetcher for training, some may match an existing prefetch stream, mapping to a cache line for which, according to prefetcher state, a prefetch request has already been issued. In this case we assume that the previously issued prefetch request was actually dropped due to prefetch queue or MSHR saturation typical of bandwidth-bound execution. Such a prefetch request would not have been dropped had the core been latency-bound; in fact, this prefetch request would have been timely had the prefetch stream been in steady state. Thus, we mark each demand last level cache access that matches a prefetch stream in steady state as a "would be timely" request. Prefetch requests generated for streams in the steady state mode are also marked "would be timely."

This classification enables CRIT to measure demand time per instruction T_{demand} in both modes of processor operation. Specifically, when computing the critical path through the demand DRAM requests (Section 3.2.3), CRIT now ignores all demand DRAM requests marked "would be timely," because these demand requests are predicted to be timely prefetch requests in the latency-bound mode.

3.3.3.2 DRAM Bandwidth Utilization

While DRAM bandwidth utilization is not a parameter of the limited bandwidth analytic model, we do measure it to detect the current processor operation mode (latency-bound or bandwidth-bound) and to measure minimum memory time per instruction $T^{\text{min}}_{\text{memory}}$.

Recall from earlier experimental observations in Section 3.3.1 that the DRAM bus is the dominant DRAM bandwidth bottleneck of a uniprocessor with a stream prefetcher; hence, in this case, DRAM bandwidth utilization is simply DRAM bus utilization.

To measure current DRAM bus utilization U_{bus} we first compute the maximum rate at which the bus can satisfy DRAM requests. This maximum rate depends on the average time each DRAM request occupies the bus. To measure the average time

each DRAM request occupies the bus we consider both the actual data transfer time and the overhead of switching the bus direction. Specifically, in the DDR3 [58] system we use, a DRAM request requires four DRAM bus cycles to transfer its data (a 64 byte cache line) over the data bus. In addition, each bus direction switch prevents the bus from being used for 9.5 cycles on average (2 cycles for a read to write switch and 17 cycles for a write to read switch). Hence, the average time each DRAM request occupies the 800 MHz bus (cycle time 1.25 ns) is

$$T_{\text{request}} = \left(4 + 9.5 \times \frac{\text{Direction switches}}{\text{Requests}}\right) \times 1.25 \times 10^{-9}.$$

Note that T_{request} is easily computed from DRAM system design parameters and performance counters. Once T_{request} is known, the maximum DRAM request rate is simply $1/T_{\text{request}}$. Finally, DRAM bus utilization U_{bus} is computed using this maximum DRAM request rate and the measured DRAM request rate:

$$U_{\text{bus}} = \frac{\text{Measured DRAM request rate}}{\text{Maximum DRAM request rate}} = \text{Measured DRAM request rate} \times T_{\text{request}}.$$

Note that we include both data transfer time and direction switch overhead time in our definition of DRAM bus utilization.

3.3.3.3 Detecting Current Operation Mode

In order to compute the remaining parameters of the limited bandwidth model, our performance predictor must determine the current operating mode of the processor: latency-bound or bandwidth-bound. To develop the relevant mechanism, we make two observations:

1. DRAM bus utilization is close to 100% in the bandwidth-bound mode, but not in the latency-bound mode.

2. The fraction of time the processor spends either computing or stalling on non-"would be timely" demand DRAM requests is close to 100% in the latency-bound mode, but not in the bandwidth-bound mode (in which the processor

spends a non-trivial fraction of time stalling on "would be timely" DRAM requests and due to MSHRs being full).

Therefore, to determine the operating mode of the processor, the performance predictor can simply compare DRAM bus utilization to the fraction of time the processor spends either computing or stalling on a non-"would be timely" demand DRAM request. To make this comparison, the processor measures the needed quantities as follows:

- *DRAM bus utilization* is measured as described in Section 3.3.3.2.
- *Fraction of time spent computing* is measured as fraction of time instruction retirement is not stalled due to a demand DRAM request or MSHRs being full.
- *Fraction of time spent stalled on non-"would be timely" demand DRAM requests* is computed as the demand time per instruction T_{demand} (measured as described in Section 3.3.3.1) divided by the easily measured total execution time per instruction T.

Having computed these quantities, the performance predictor determines the operating mode of the processor to be:

- bandwidth-bound if DRAM bus utilization is greater than the fraction of time spent computing or stalling on non-"would be timely" demand DRAM requests, or
- latency-bound otherwise.

3.3.3.4 Compute Cycles per Instruction C_{compute}

The measurement technique for compute cycles per instruction C_{compute} depends on the operating mode of the processor. In the latency-bound mode the main assumption of the linear model still holds: the processor is always either computing

or stalling on (non-"would be timely") demand DRAM requests. Therefore, C_{compute} is easily computed using Equation 3.2, restated here:

$$C_{\text{compute}} = \frac{T - T_{\text{demand}}}{t}.$$

This equation, however, does not apply in the bandwidth-bound mode, where the processor, in addition to computing or stalling on non-"would be timely" demand DRAM requests, may also be in a third state: stalling on "would be timely" DRAM requests or due to MSHRs being full. Therefore, we devise a different way to measure C_{compute} in the bandwidth-bound mode: as the number of cycles the processor is not stalled due to a demand DRAM request or MSHRs being full divided by the number of instructions retired.

3.3.3.5 Minimum Memory Time per Instruction $T^{\text{min}}_{\text{memory}}$

The measurement technique for the minimum memory time per instruction $T^{\text{min}}_{\text{memory}}$ also depends on the operating mode of the processor.

In the latency-bound mode, $T^{\text{min}}_{\text{memory}}$ is computed from DRAM bus utilization U_{bus} and the measured execution time per instruction T. Since $T^{\text{min}}_{\text{memory}}$ is the minimum bound on T reached when DRAM bus bandwidth is utilized 100%, the relationship is simple:

$$T^{\text{min}}_{\text{memory}} = U_{\text{bus}} \times T.$$

In the bandwidth-bound mode, according to the limited bandwidth model, the execution time per instruction T is simply $T^{\text{min}}_{\text{memory}}$; therefore, $T^{\text{min}}_{\text{memory}} = T$.

3.3.4 Hardware Cost

Table 3.3 details the storage required by CRIT+BW. The additional storage is only 1088 bits. The mechanism does not add any structures or logic to the critical path of execution.

Storage Component	Quantity	Width	Bits
Global critical path counter P_{global}	1	32	32
Copy of P_{global} per memory request	32	32	1024
"Would be timely" bit per memory request	32	1	32
Other counters assumed to exist already	–	–	–
Total bits			1088

Table 3.3: Hardware storage cost of CRIT+BW

Frequency		Front end		OOO Core	
Min	1.5 GHz	Microinstructions/cycle	4	Microinstructions/cycle	4
Max	4.5 GHz	Branches/cycle	2	Pipeline depth	14
Step	100 MHz	BTB entries	4K	ROB size	128
		Predictor	hybrid[a]	RS size	48

All Caches			ICache	DCache	L2	Stream prefetcher [84]			
Line	64 B	Size	32 KB	32 KB	1 MB	Streams	16	Distance	64
MSHRs	32	Assoc.	4	4	8	Queue	128	Degree	4
Repl.	LRU	Cycles	1	2	12	Training threshold			4
		Ports	1R,1W	2R,1W	1	L2 insertion			mid-LRU

DRAM Controller		DDR3 SDRAM [58]				DRAM Bus	
Window	32 reqs	Chips	8×256 MB	Row	8 KB	Freq.	800 MHz
Priority scheduling[b]		Banks	8	CAS[c]	13.75 ns	Width	8 B

[a] 64K-entry gshare + 64K-entry PAs + 64K-entry selector.
[b] Priority order: row hit, demand (instruction fetch or data load), oldest.
[c] CAS = $t_{\text{RP}} = t_{\text{RCD}}$ = CL;
other modeled DDR3 constraints: BL, CWL, $t_{\{\text{RC, RAS, RTP, CCD, RRD, FAW, WTR, WR}\}}$.

Table 3.4: Simulated uniprocessor configuration

3.4 Methodology

We compare energy saved by CRIT+BW to that of the state-of-the-art (leading loads and stall time) and to three kinds of potential energy savings (computed using offline DVFS policies). Before presenting the results, we justify our choice of energy as the efficiency metric, describe our simulation methodology, explain how we compute potential energy savings, and discuss our choice of benchmarks.

3.4.1 Efficiency Metric

We choose energy (or, equivalently,[6] *performance per watt*) as the target efficiency metric because a) energy is a fundamental metric of interest in computer system design and b) of the four commonly used metrics, energy is best suited for DVFS performance prediction evaluation.

Specifically, energy is a fundamental metric of interest in computer system design because it is directly related to operation cost (in data center applications) and battery life (in mobile applications), which are, in turn, directly related to the data center operator's bottom line and the quality of user experience, respectively.

In addition, of the four commonly used metrics—energy, energy delay product (EDP), energy delay-squared product (ED^2P), and performance (execution time)—energy is best suited for DVFS performance predictor evaluation because it can be targeted by a simple DVFS performance controller (so that most of the benefit comes from DVFS performance prediction) and allows comparisons to optimal results.

Specifically, energy has the desirable property that the optimal (most energy-efficient) frequency for an execution interval does not depend on the behavior of the rest of the execution. Therefore, the DVFS controller need not keep track of past long-term application behavior and predict future long-term application behavior in order to reduce energy consumption using DVFS. In addition, this property means

[6] Energy and performance per watt are equivalent in the sense that in any execution interval, the same operating point is optimal for both metrics.

that a locally optimal oracle predictor (one that correctly predicts the best chip frequency for every execution interval) is also globally optimal (that is, optimal over the entire execution of the workload). Since simulation of such locally optimal oracle predictors is, though slower than nonoracle simulation, still feasible, this property enables comparisons to globally optimal results.

In contrast, EDP and ED^2P do not have this property, making it hard to isolate the benefits of DVFS performance prediction in the results and precluding comparisons to optimal results. Sazeides et al. [73] discuss these issues in greater detail.

The remaining metric, performance, is not applicable to chip-level DVFS. In the uniprocessor case, optimizing performance does not require a performance prediction: the optimal frequency is simply the highest frequency.[7]

3.4.2 Timing Model

We use an in-house cycle-level simulator of an x86 superscalar out-of-order processor driven by the x86 functional model from Multi2Sim [85]. The simulator models port contention, queuing effects, and bank conflicts throughout the cache hierarchy and includes a detailed DDR3 SDRAM model. Table 3.4 lists the baseline processor configuration.

3.4.3 Power Model

We model three major system power components: chip power, DRAM power, and other power (fan, disk, etc.).

We model chip power using McPAT 0.8 [50] extended to support DVFS. Specifically, to generate power results for a specific chip frequency f, we:

1. run McPAT with a reference voltage V_0 and frequency f_0,

[7] We shall target performance (within a power budget) when we consider performance prediction for per-core DVFS in Chapters 4 and 5.

Component	Parameter	Value	
		@1.5 GHz	@4.5 GHz
Chip	Static power (W)	12	35
	Peak dynamic power (W)	2	51
DRAM	Static power (W)	1	
	Precharge energy (pJ)	79	
	Activate energy (pJ)	46	
	Read energy (pJ)	1063	
	Write energy (pJ)	1071	
Other	Static power (W)	40	

Table 3.5: Uniprocessor power parameters

2. scale voltage using $V = \max(V_{\min}, \frac{f}{f_0} V_0)$,

3. scale reported dynamic power using $P = \frac{1}{2} C V^2 f$, and

4. scale reported static power linearly with voltage [7].

We model DRAM power using CACTI 6.5 [63] and use a constant static power as a proxy for the rest of system power.

Table 3.5 details the power parameters of the system.

3.4.4 DVFS Controller

Every 100K retired instructions, the DVFS controller chooses a chip frequency for the next 100K instructions.[8] Specifically, the controller chooses the frequency estimated to cause the least system energy consumption. To estimate energy consumption at a candidate frequency f while running at f_0, the controller:

1. obtains measurements of

[8] We chose 100K instructions because it is the smallest quantum for which the time to change chip voltage (as low as tens of nanoseconds [39, 40], translating to less than 1K instructions or less than 1% of the 100K instruction interval) can be neglected.

- execution time $T(f_0)$,
- chip static power $P_{\text{chip static}}(f_0)$,
- chip dynamic power $P_{\text{chip dynamic}}(f_0)$,
- DRAM static power $P_{\text{DRAM static}}(f_0)$,
- DRAM dynamic power $P_{\text{DRAM dynamic}}(f_0)$, and
- other system power $P_{\text{other}}(f_0)$

for the previous 100K instructions from hardware performance counters and power sensors,

2. obtains a prediction of execution time $T(f)$ for the next 100K instructions from the performance predictor (either leading loads, stall time, or CRIT+BW),
3. calculates chip dynamic energy $E_{\text{chip dynamic}}(f_0)$ and DRAM dynamic energy $E_{\text{DRAM dynamic}}(f_0)$ for the previous interval using $E = PT$,
4. calculates $E_{\text{chip dynamic}}(f)$ by scaling $E_{\text{chip dynamic}}(f_0)$ using $E = \frac{1}{2}CV^2$,
5. calculates $P_{\text{chip static}}(f) = \frac{V}{V_0}P_{\text{chip static}}(f_0)$ as in [7],
6. and finally calculates total estimated system energy

$$\begin{aligned} E(f) &= E_{\text{chip}}(f) + E_{\text{DRAM}}(f) + E_{\text{other}}(f) \\ &= P_{\text{chip static}}(f) \times T(f) + E_{\text{chip dynamic}}(f) + \\ &\quad P_{\text{DRAM static}}(f_0) \times T(f) + E_{\text{DRAM dynamic}}(f) + \\ &\quad P_{\text{other}}(f_0) \times T(f). \end{aligned}$$

To isolate the effect of DVFS performance predictor accuracy on energy savings, we do not simulate delays associated with switching between frequencies. Accounting for these delays requires an additional prediction of whether the benefits of switching outweigh the cost. If the accuracy of that prediction is low, it could hide the benefits of high performance prediction accuracy, and vice versa.

3.4.5 Offline Policies

We model three offline DVFS controller policies: *dynamic optimal*, *static optimal*, and *perfect memoryless*.

The *dynamic optimal* policy places a lower bound on energy consumption. We compute this bound as follows:

1. run the benchmark under study at each chip frequency,
2. for each interval, find the minimum consumed energy across all frequencies,
3. total the per-interval minimum energies.

The *static optimal* policy chooses the chip frequency that minimizes energy consumed by the benchmark under study, subject to the constraint that frequency must remain the same throughout the run. The difference between dynamic and static optimal results yields potential energy savings due to benchmark phase behavior.

The *perfect memoryless* policy simulates a perfect *memoryless* performance predictor. We call a predictor *memoryless* if it assumes that for each chip frequency, performance during the next interval equals performance during the last interval. This assumption makes sense for predictors that do not "remember" any state (other than the measurements from the last interval); hence the name "memoryless." Note that all predictors discussed in this dissertation are memoryless. For each execution interval, the perfect memoryless policy chooses the chip frequency that would minimize energy consumption during the *previous* interval.

The perfect memoryless policy provides a quasi-optimal[9] bound on energy saved by memoryless predictors. A large difference between dynamic optimal and perfect memoryless results indicates that a memoryless predictor cannot handle the frequency of phase changes in the benchmark under study. Getting the most energy

[9]We call this bound *quasi-optimal* because an imperfect memoryless predictor may actually save more energy than the perfect memoryless predictor if the optimal frequency for the previous interval does not remain optimal in the next interval.

savings out of such benchmarks may require "memoryful" predictors that can detect and predict application phases. We leave such predictors to future work.

3.4.6 Benchmarks

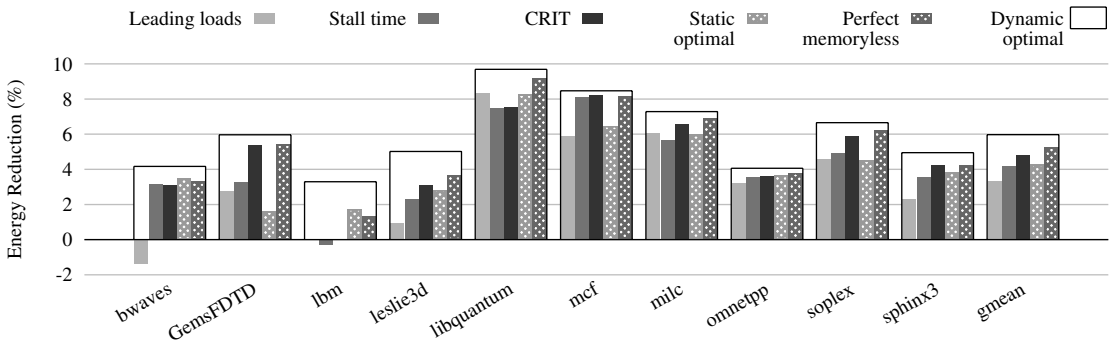
We simulate SPEC 2006 benchmarks compiled using the GNU Compiler Collection version 4.3.6 with the `-O3` option. We run each benchmark with the reference input set for 200M retired instructions starting from checkpoints taken using Pincpt [59] at the beginning of a representative region selected using Pinpoints [68].

To simplify the analysis of the results, we classify the benchmarks based on their memory intensity. We define a benchmark as *memory-intensive* if it generates 10 or more DRAM requests per thousand instructions at the baseline 3.7 GHz frequency with no prefetching.

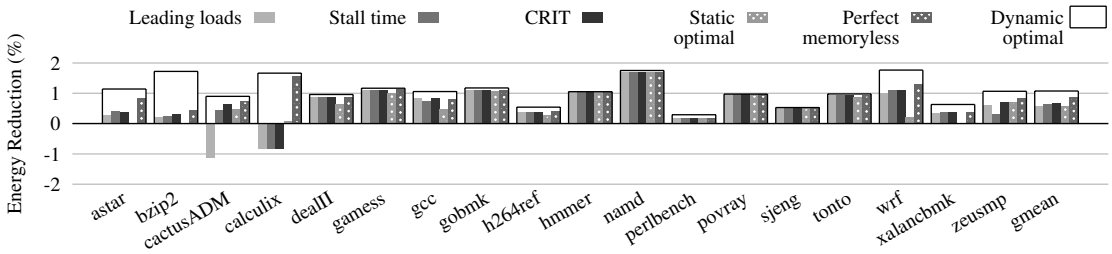
3.5 Results

We show CRIT and CRIT+BW results for two configurations: with prefetching turned off (for CRIT) and with a stream prefetcher (for CRIT+BW). In both cases, we show normalized energy reduction relative to the energy consumed at 3.7 GHz, the most energy-efficient static frequency across SPEC 2006 (which happens to be the same for both cases).

Before analyzing the results, we first explain their presentation using Figure 3.9a as an example. Note that, for each benchmark, the figure shows five bars within a wide box. The height of the box represents dynamic optimal energy reduction. Since no other DVFS policy can save more energy than dynamic optimal, we can use this box to bound the other five bars. The five bars inside the box represent energy reduction due to 1) leading loads, 2) stall time, 3) CRIT or CRIT+BW, 4) optimal static DVFS policy, and 5) perfect memoryless DVFS policy. This plot design allows for easy comparisons of realized and potential gains for each benchmark and simplifies comparison of potential gains across benchmarks at the same time.



(a) Memory-intensive benchmarks



(b) Non-memory-intensive benchmarks

Figure 3.9: Energy savings with prefetching off

3.5.1 CRIT (Prefetching Off)

Figure 3.9a shows realized and potential energy savings across the ten memory-intensive workloads. On average, CRIT and stall time realize 4.8% and 4.2% out of potential 6.0% energy savings, whereas leading loads only realizes 3.3%. For completeness, Figure 3.9b shows energy savings for low memory intensity benchmarks (note the difference in scale).

The subpar energy savings by leading loads are due to its constant memory access latency approximation. As described in Section 3.1.2, leading loads measures the latency of the first load in each cluster of simultaneous memory requests to compute the demand component T_{demand} of total execution time per instruction T. As we have already discussed in Section 3.2.1, it turns out that in such clusters, the leading load latency is usually *less* than that of the other requests. This discrepancy is due to the fact that the first memory request in a cluster is unlikely to contend with another request for a DRAM bank, whereas the later requests in the cluster likely have to

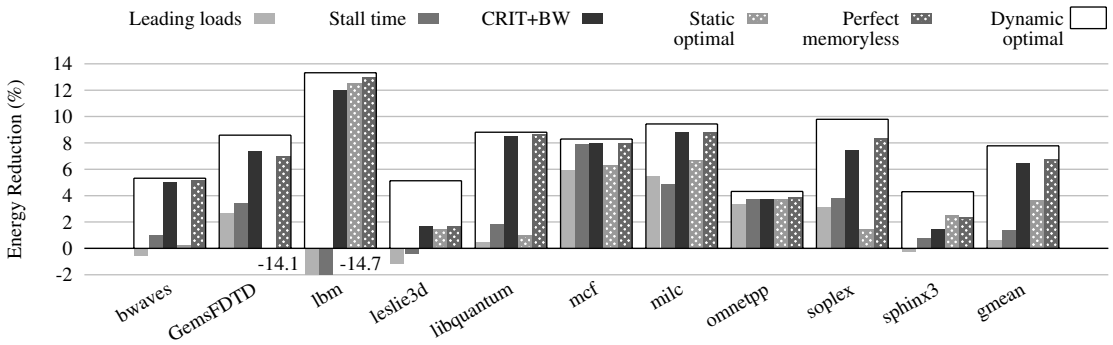
wait for the earlier ones to free up the DRAM banks. This underestimate of T_{demand} results in subpar energy savings, exemplified by `bwaves`, `leslie3d`, and `mcf`.

The fact that stall time beats leading loads validates our focus on realistic memory systems. Both our experiments and prior work [22, 37] show that when evaluated with a constant access latency memory, leading loads saves more energy than stall time. Evaluation of the two predictors with a realistic DRAM system, however, actually reverses this conclusion. Thus we conclude that DVFS performance predictors must be evaluated with realistic memory systems.

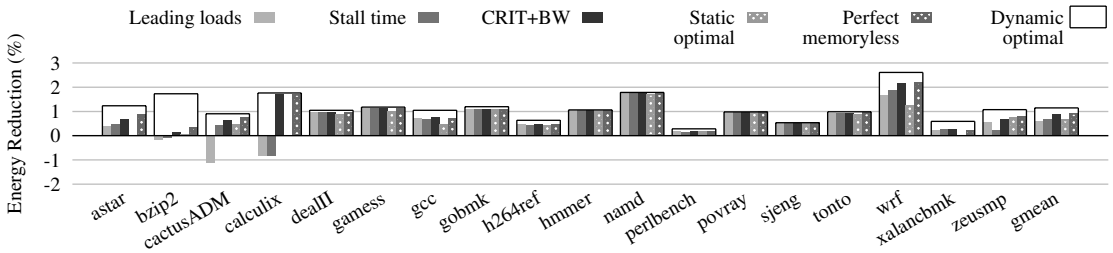
While CRIT generally outperforms stall time and leading loads, all three performance predictors fail to save much energy on `lbm` and `calculix`. These two benchmarks generate enough DRAM requests to saturate DRAM bandwidth even though prefetching is off (for `calculix` this behavior is limited to 8M out of simulated 200M instructions). As discussed in Section 3.3, the linear DVFS performance model assumed by all three predictors does not account for this DRAM bandwidth saturation; thus, all three predictors fail to accurately predict performance of these benchmarks resulting in suboptimal energy savings.

3.5.2 CRIT+BW (Prefetching On)

Figure 3.10a shows realized and potential energy reduction across the ten memory-intensive benchmarks with a stream prefetcher enabled. On average, CRIT+BW realizes 6.4% out of potential 7.8% energy savings, whereas stall time and leading loads only realize 1.4% and 0.6% respectively. CRIT+BW saves significantly more energy than stall time and leading loads because the limited bandwidth model used by CRIT+BW takes into account DRAM bandwidth saturation, whereas the linear model used by stall time and leading loads does not. For completeness, Figure 3.10b shows energy savings for non-memory-intensive benchmarks (note the difference in scale).



(a) Memory-intensive benchmarks



(b) Non-memory-intensive benchmarks

Figure 3.10: Energy savings with prefetching on

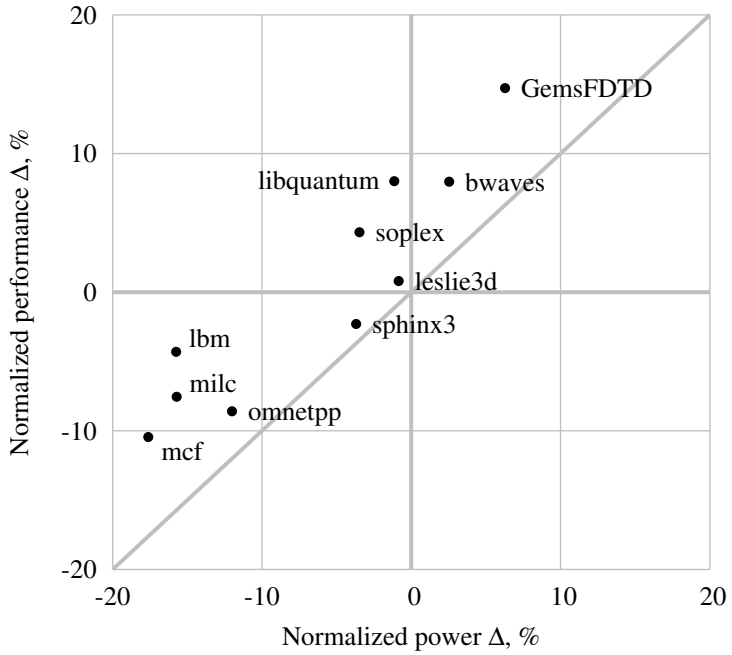


Figure 3.11: Performance delta versus power delta under DVFS with CRIT+BW for memory-intensive benchmarks

Note that CRIT+BW realizes close to oracle energy savings on both `lbm` and `calculix`, the two workloads on which CRIT fails due to ignoring the performance impact of DRAM bandwidth saturation.

Note also that CRIT+BW adapts to dynamic phase behavior of `bwaves`, `GemsFDTD`, `libquantum`, and `soplex`. This effect is evident from the large difference between energy savings realized by CRIT+BW and the energy savings achieved by the static optimal DVFS policy on these benchmarks.

Finally, note that CRIT+BW realizes less than half of dynamic optimal energy savings on `leslie3d` and `sphinx3`. This suboptimal showing is due to the frequent phase changes in both benchmarks, evident from the large difference between dynamic optimal energy savings and the energy savings achieved by the perfect memoryless DVFS policy. The 100K instruction intervals used by CRIT+BW are too long to allow CRIT+BW to adapt to the fast-changing phase behavior of these benchmarks.

3.5.2.1 Power and Performance Tradeoff

Figure 3.11 details how CRIT+BW trades off power and performance to reduce energy. The figure plots performance delta versus power delta (normalized to performance and power achieved at the baseline 3.7 GHz frequency). The diagonal line consists of points where performance and power deltas are equal, resulting in the same energy as the baseline.

CRIT+BW trades off power and performance differently across workloads. On `GemsFDTD` and `bwaves`, CRIT+BW spends extra power for even more performance, while on `lbm`, `mcf`, `milc`, `omnetpp`, and `sphinx3` CRIT+BW allows performance to dip to save more power.

Note that CRIT+BW improves performance *and* saves power on `leslie3d`, `libquantum`, and `soplex`. CRIT+BW does so by exploiting phase behavior of these benchmarks. In some phases, CRIT+BW spends extra power for more performance; in others, it makes the opposite choice. On average, both performance and power consumption improve.

3.6 Conclusions

In this chapter we have developed CRIT+BW, a DVFS performance predictor for uniprocessors that enables DVFS to realize close to optimal energy savings. We have also shown that taking into account the details of the memory system (such as variable DRAM access latencies and stream prefetching) is key to accurate DVFS performance prediction.

We note that most of the extra energy savings realized by CRIT+BW on top of the savings realized by prior work comes not from better model parameter measurement (the CRIT part of CRIT+BW), but rather from the new limited bandwidth DVFS performance model (the BW part of CRIT+BW). This discrepancy makes sense: no matter how accurate a parameter measurement mechanism is, it cannot help the performance predictor if the performance model used is inaccurate itself.

This observation motivates our approach to DVFS performance prediction for chip multiprocessors in the chapters ahead. Specifically, we shall focus most of our attention on the DVFS performance model, keeping parameter measurement mechanisms as simple as possible.

Chapter 4

Private Cache Chip Multiprocessor

> A brave little theory, and actually quite coherent for a system of five or seven dimensions—if only we lived in one.
>
> Academician Prokhor Zakharov on "Superstring Theory"
> *Sid Meyer's Alpha Centauri*

In this chapter we develop a DVFS performance predictor for the private cache chip multiprocessor shown in Figure 4.1. We start by showing experimentally that, just like in the uniprocessor, the DRAM bus is the major DRAM bandwidth bottleneck in the private cache chip multiprocessor equipped with a stream prefetcher. We then take a short detour from performance prediction and propose *scarce row hit prioritization*, a DRAM scheduling technique that improves performance when the DRAM bus is saturated and, more importantly for this dissertation, makes performance more predictable. We then get back to performance prediction and propose the *independent latency shared bandwidth* (ILSB) model of performance under frequency scaling as well as the hardware mechanisms that measure parameters of this model. Finally, we show that the DVFS performance predictor comprised of our ILSB model

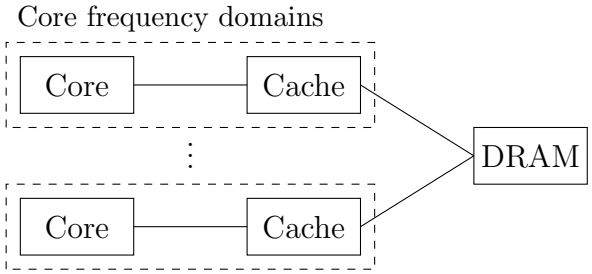


Figure 4.1: Private cache CMP

and our parameter measurement mechanisms can be used to improve performance within the peak power budget, realizing most of the oracle performance gains.

4.1 Experimental Observations

We first show experimentally that of the three possible DRAM bandwidth bottlenecks (bus, banks, and FAW—see Section 2.4 for details) the DRAM bus is the major bandwidth bottleneck in a private cache chip multiprocessor with a stream prefetcher. For this experiment, we simulated a four-core private cache chip multiprocessor[1] at 5 GHz (an unrealistically high frequency chosen on purpose to induce more bandwidth saturation) and measured the utilization of all three bottlenecks. Table 4.1 lists the fraction of time each potential bandwidth bottleneck was more than 90% utilized. The table shows that the bus is the most common bandwidth bottleneck with and without stream prefetching; however, with stream prefetching on, the bus becomes the only major bandwidth bottleneck.

The observation that the DRAM bus is the major bandwidth bottleneck will come in useful in two places later in this chapter. First, we shall use this observation to explain part of the performance benefit of the *scarce row hit prioritization* DRAM scheduling technique (Section 4.2). Second, we shall use this observation to simplify how our *independent latency shared bandwidth* model (Section 4.3) accounts for the performance constraint imposed by finite DRAM bandwidth.

4.2 Scarce Row Hit Prioritization

In this section, we take a slight detour from the main topic of this dissertation—performance prediction—and develop *scarce row hit prioritization*, a DRAM scheduling technique. We do so not because this technique improves performance (which it does), but because it makes performance more predictable.

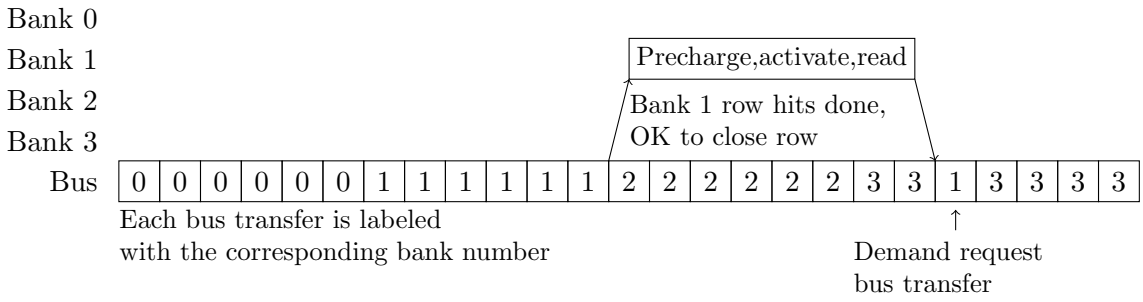
[1]Section 4.5.3 details the simulation methodology and the simulated processor configuration; the workloads used come from the ALL set of four-core workloads described in Section 4.5.4.

Workload	Fraction of time each potential DRAM bandwidth bottleneck is more than 90% utilized, in %					
	Prefetching off			Prefetching on		
	Bus	Banks	FAW	Bus	Banks	FAW
astar, gamess, sphinx3, milc	12.3	6.3	0.0	88.3	0.0	0.0
bwaves, libquantum, povray, gcc	3.0	20.8	0.0	95.4	0.2	0.0
bzip2, soplex, namd, libquantum	51.9	7.8	0.0	99.6	0.0	0.0
cactusADM, h264ref, astar, sphinx3	22.1	6.5	0.0	75.8	0.1	0.0
calculix, sphinx3, GemsFDTD, lbm	96.2	0.4	0.0	100.0	0.0	0.0
dealII, bzip2, soplex, tonto	29.3	1.9	0.0	88.0	0.1	0.0
gamess, milc, zeusmp, mcf	20.4	0.1	0.0	53.2	0.0	0.0
gcc, cactusADM, tonto, soplex	35.1	1.1	0.0	89.9	0.2	0.0
GemsFDTD, wrf, libquantum, calculix	50.4	4.1	0.0	99.8	0.0	0.0
gobmk, gcc, mcf, dealII	1.7	0.1	0.0	7.9	0.0	0.0
gromacs, mcf, hmmer, namd	1.7	0.0	0.0	2.7	0.0	0.0
h264ref, bwaves, cactusADM, omnetpp	35.8	7.2	0.0	95.9	0.3	0.0
hmmer, tonto, calculix, sjeng	6.9	0.0	0.0	8.7	0.0	0.0
lbm, namd, milc, bzip2	99.5	0.0	0.0	99.9	0.0	0.0
leslie3d, zeusmp, gobmk, xalancbmk	9.5	5.5	0.0	79.6	0.1	0.0
libquantum, hmmer, omnetpp, GemsFDTD	40.5	2.0	0.0	99.7	0.0	0.0
mcf, lbm, gromacs, perlbench	98.5	0.0	0.0	99.2	0.0	0.0
milc, omnetpp, xalancbmk, gromacs	28.2	0.0	0.0	81.0	0.0	0.0
namd, sjeng, perlbench, povray	0.7	0.0	0.0	0.7	0.0	0.0
omnetpp, gobmk, bzip2, leslie3d	30.5	4.1	0.0	88.8	0.1	0.0
perlbench, xalancbmk, bwaves, h264ref	6.0	2.9	0.0	89.8	0.3	0.0
povray, astar, sjeng, hmmer	4.4	0.0	0.0	6.7	0.0	0.0
sjeng, gromacs, gamess, zeusmp	0.5	0.2	0.0	2.2	0.0	0.0
soplex, dealII, h264ref, cactusADM	39.5	1.7	0.0	91.8	0.1	0.0
sphinx3, calculix, leslie3d, astar	34.1	16.2	0.0	98.3	0.0	0.0
tonto, povray, lbm, wrf	98.3	0.1	0.0	99.7	0.0	0.0
wrf, GemsFDTD, gcc, gobmk	13.3	0.6	0.0	56.9	0.1	0.0
xalancbmk, leslie3d, wrf, bwaves	39.3	16.7	0.0	99.1	0.1	0.0
zeusmp, perlbench, dealII, gamess	1.0	0.0	0.0	3.4	0.0	0.0

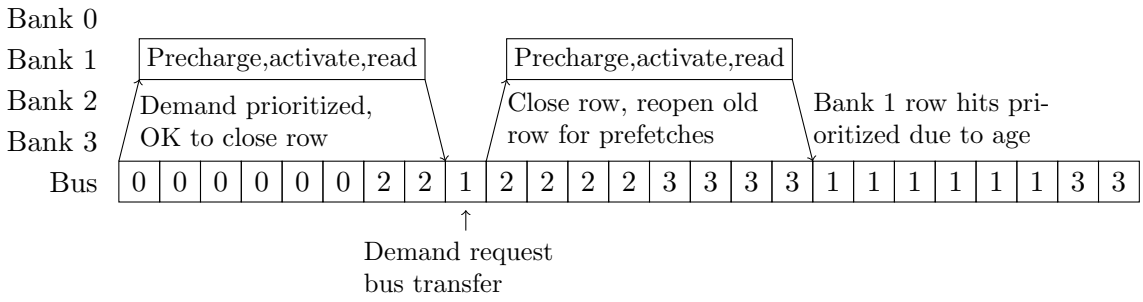
Table 4.1: Bandwidth bottlenecks in the private cache CMP

Scenario at the start:

- each bank has 6 prefetch row hit requests,
- bank 1 also has a single demand row conflict request,
- the requests are oldest in bank 0, then bank 1, and so on.



(a) Priority order: row hit, demand, age



(b) Priority order: demand, age

Figure 4.2: Simplified timing diagrams illustrating DRAM scheduling under two different priority orders when row hits are abundant

4.2.1 Problem

The DRAM scheduling policy we have assumed so far suffers from a problem of priority inversion in the presence of abundant row hit requests. We first illustrate the problem with an example and then generalize.

Figure 4.2a provides an example to illustrate the problem. At the start of the example, each bank has 6 prefetch row hit requests outstanding; in addition, bank 1 has a single demand row conflict request. Recall from Section 2.4 that the DRAM scheduler prioritizes a) row hit requests, b) demand (instruction fetch and data load) requests, and c) oldest requests, in that order. Under this regular priority order, the row hits from banks 0 and 1 are scheduled first (due to row hit and oldest

request prioritization). Once bank 1's row hits are done, the bank's open row may be closed and another row opened for the remaining demand request. In contrast, Figure 4.2b shows the same example, except the DRAM scheduler no longer prioritizes row hits; instead, demand requests are given first priority. Note that a) the demand request is satisfied much earlier under this priority order, and b) the bus is completely utilized under both priority orders. Clearly, for this specific example, "row hit first" prioritization is harmful because it delays the only demand request (on which its originating core is likely stalling) without any increase in bus utilization.

In general, the traditional "row hit first" priority order makes sense for DRAM scheduling decisions made when row hits are scarce but becomes harmful when row hits are abundant. The benefit of prioritizing non-demand row hit requests over demand row conflict requests comes from the resulting increase in bus utilization; thus, if bus utilization is likely to be high anyway due to the abundance of outstanding row hit requests, non-demand row hit requests should not be prioritized over demand row conflicts.

4.2.2 Mechanism

These observations lead us to an improved DRAM scheduling mechanism that adjusts its priority order based on the scarcity or abundance of outstanding row hit requests. Specifically, when considering DRAM commands for a specific bank, the scheduler calculates how many row hits are outstanding to the other banks in the channel. If that number is greater than a threshold, the row hits are deemed *abundant* and the scheduler uses the "demand, age" priority order. Otherwise, the row hits are deemed *scarce* and the scheduler uses the regular "row hit, demand, age" priority order—hence the name "scarce row hit prioritization."

In our experiments, we have found the threshold value of 7 outstanding row hits to work best. Note that, in the DDR3 DRAM system we model, 7 row hits are not enough to completely utilize the DRAM bus during the extra bank latency of a single row conflict ($7 \times 4 = 28$ DRAM cycles vs. 39 DRAM cycles, respectively).

This apparent discrepancy is explained by the fact that the threshold value does not include row hit requests likely to be generated in the future.

4.2.3 Results

Figure 4.3 shows the performance gains achieved by scarce row hit prioritization on a four-core private cache CMP.[2] The figure shows both per-core and overall performance gains.[3] Note that performance of most workloads improves; only a couple suffer negligible performance degradation (less than 0.3%). On average across these workloads, scarce row hit prioritization improves performance by 2.1%.

Scarce row hit prioritization attains these gains because, as we observe in Section 4.1, the DRAM bus is often saturated. This observation implies that row hit requests are often abundant, and thus the DRAM scheduler is often faced with the priority inversion problem addressed by scarce row hit prioritization. This argument is supported by the fact that all workloads on which scarce row hit prioritization achieves significant performance gains (say, more than 2%) often exhibit DRAM saturation (more than 50% of the time as shown in Table 4.1).

4.2.4 Impact on Performance Predictability

More important to this dissertation is how scarce row hit prioritization improves predictability of performance under frequency scaling. Specifically, scarce row hit prioritization improves performance predictability in two ways:

1. It makes the latency of demand requests of a core approximately constant with respect to the changes in frequencies of the other cores. Specifically, scarce row hit prioritization bounds the time demand requests of one core have to wait for the non-demand row hit requests of other cores, thereby making the latency of demand requests of one core roughly independent of the number of outstanding

[2]Section 4.5 describes our experimental methodology in detail.

[3]We use the geometric mean of individual cores' performances as the multicore performance metric. We justify this choice of metric in Section 4.5.1.

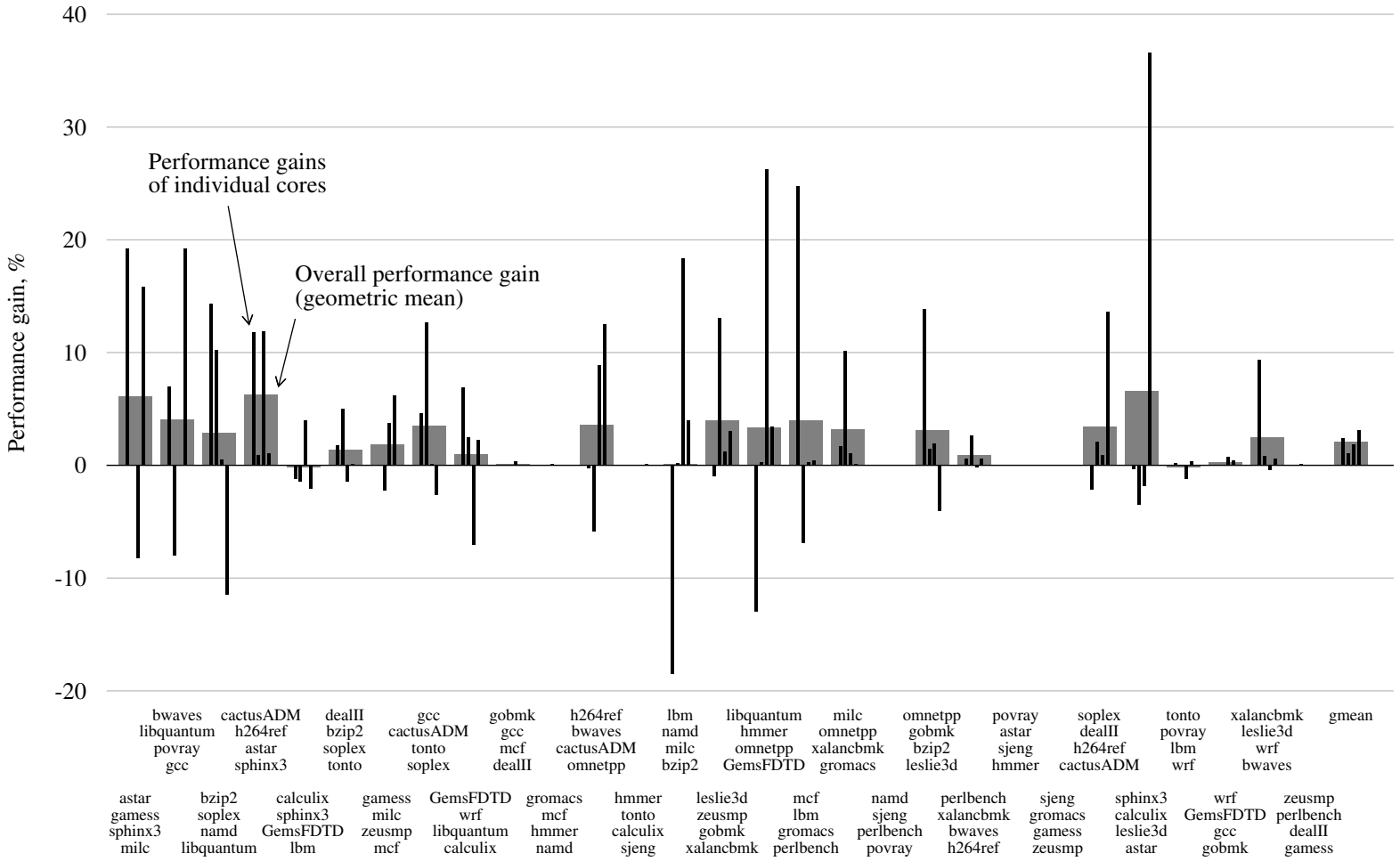


Figure 4.3: Performance benefit of scarce row hit prioritization over indiscriminate row hit prioritization

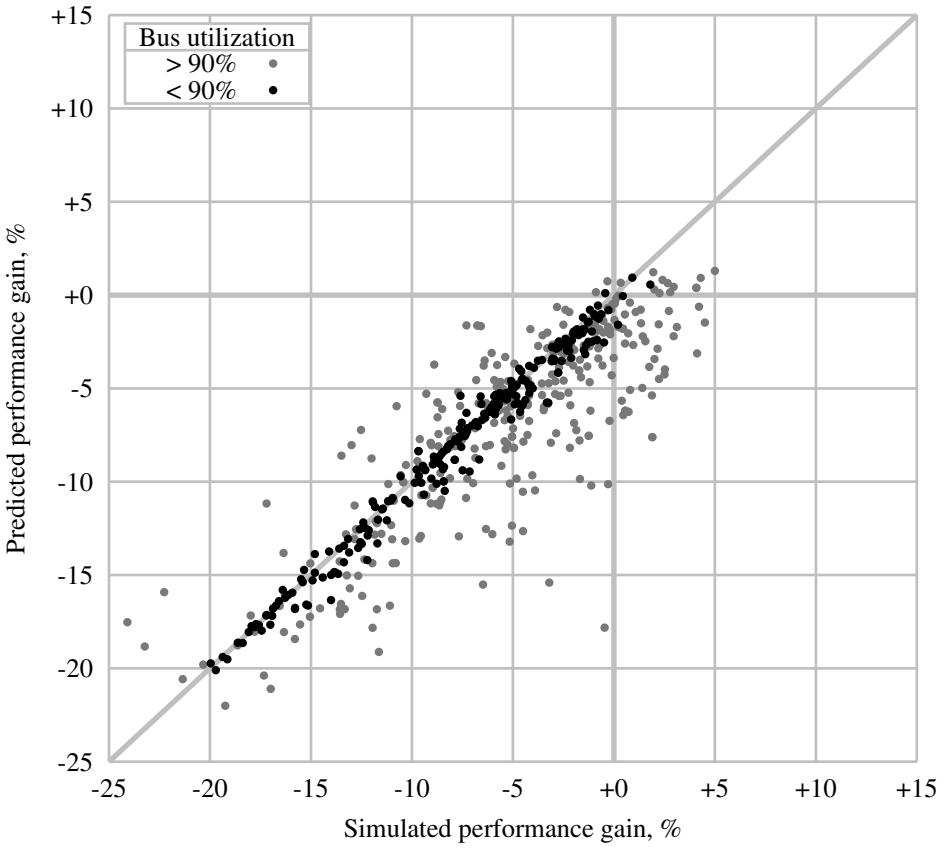
non-demand requests from other cores (which may change with the other cores' frequencies).

2. It makes the queuing latency of non-demand requests of different cores roughly equal. Specifically, scarce row hit prioritization mitigates priority inversion of not only demand requests, but also old requests, including old non-demand requests. Therefore, scarce row hit prioritization makes a core that continuously generates prefetch row hit requests less likely to deny DRAM service to the older prefetch row conflict requests from the other cores. That is, scarce row hit prioritization makes the DRAM scheduler treat non-demand requests in a more first-come-first-serve fashion, keeping the average non-demand request queuing latencies of different cores roughly the same.

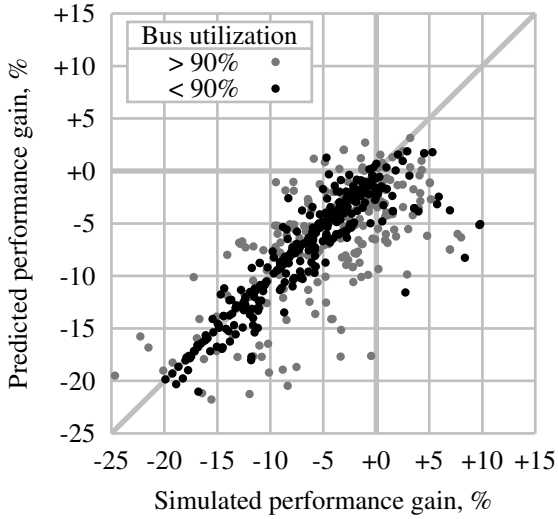
We shall rely on both of these effects when designing our analytic model of CMP performance under frequency scaling in Section 4.3.

4.3 Independent Latency Shared Bandwidth Model

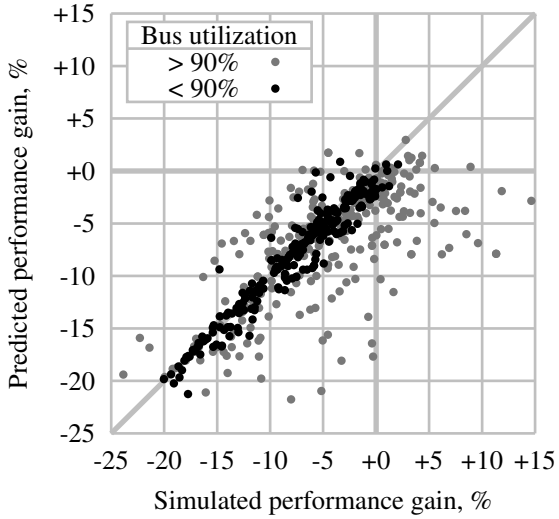
After our short dabble in DRAM scheduling, we now return back to performance prediction—the main topic of this dissertation—and develop the analytic model of performance under frequency scaling for private cache chip multiprocessors. We develop this model for private cache CMPs in two steps. We first show that the linear model previously proposed for uniprocessors remains accurate when applied to each core of a private cache CMP independently of the other cores as long as DRAM bandwidth is not saturated. We then augment the linear model to handle performance effects of DRAM bandwidth sharing which are exposed by DRAM bandwidth saturation. The result is our *independent latency shared bandwidth* (ILSB) analytic model.



(a) Scarce row hit, demand, oldest



(b) Row hit, oldest (FR-FCFS)



(c) Row hit, demand, oldest

Figure 4.4: Accuracy of the linear model applied to a four-core private cache CMP with three different DRAM scheduling priority orders

4.3.1 Applicability of Linear Model

Our experimental results suggest that, when the DRAM system is not saturated, the linear model, applied to each latency-bound core independently of other cores, still accurately describes latency-bound core performance. We first present these results and then explain their justification and consequences.

4.3.1.1 Experimental Results

Figure 4.4a presents the experimental data. The figure shows the prediction accuracy (predicted performance gain vs. simulated performance gain) of a performance predictor based on the linear model applied independently to each core. To generate the figure, we simulate 29 random four-core combinations of SPEC 2006 benchmarks for 1M cycles in the baseline frequency configuration (each core at 2.8GHz) and in 20 random *iso-power* (that is, with the same peak power as the baseline) frequency configurations in which core frequencies range from 1.6GHz to 4.0GHz. We plot the performance delta measured in simulation on the X axis and the performance delta predicted by the linear analytic model on the Y axis. The color of each dot depends on the DRAM bus utilization[4] of the corresponding workload in the baseline frequency configuration: dark if DRAM bus utilization is less than 90% and light otherwise. The figure clearly shows that the predicted delta roughly equals the simulated delta as long as bus utilization is under 90%; thus we conclude that the linear model remains accurate for CMPs as long as the DRAM bus is not saturated.

4.3.1.2 Justification

To justify the applicability of the linear analytic model to latency-bound CMPs demonstrated in Figure 4.4a, we consider how DRAM interference affects a latency-bound core in a multicore workload. The performance of a latency-bound core depends on the average latency of its demand DRAM requests, which depends on the

[4]DRAM bus utilization is defined in Section 3.3.3.2.

amount of DRAM interference presented by other cores. Since the average number of outstanding demand requests in the DRAM system is usually small (at most 6 in these experiments), the bulk of the interference is due to non-demand requests. However, since a) demand requests are prioritized over non-demand requests and b) the priority inversion problem is mitigated by scarce row hit prioritization (Section 4.2), the impact of this interference is limited; at most, demand requests only have wait for a short sequence of another core's non-demand row hits to complete before getting serviced. Thus, while the number of outstanding non-demand requests does change under core frequency scaling, their impact on demand request latency stays relatively constant. This observation suggests that the performance of each latency-bound core could be predicted independently of the others, as confirmed by our experimental results.

The importance of demand prioritization and scarce row hit prioritization is highlighted in Figures 4.4b and 4.4c which show the linear model applied independently to individual cores to be less accurate without these mechanisms.

4.3.1.3 Consequences

Figure 4.4a also shows that bandwidth-bound workloads stand to gain the most performance within the power budget from per-core DVFS. Specifically, note that most of the workloads with positive simulated performance gains (in the right half of the plot) are bandwidth-bound and that the linear model fails to predict these gains. A more accurate analytic model that takes bandwidth saturation into account could predict these gains correctly, allowing these gains to be realized with per-core DVFS. This observation further underscores the importance of taking finite bandwidth into account when designing performance predictors—a major point of focus of this dissertation.

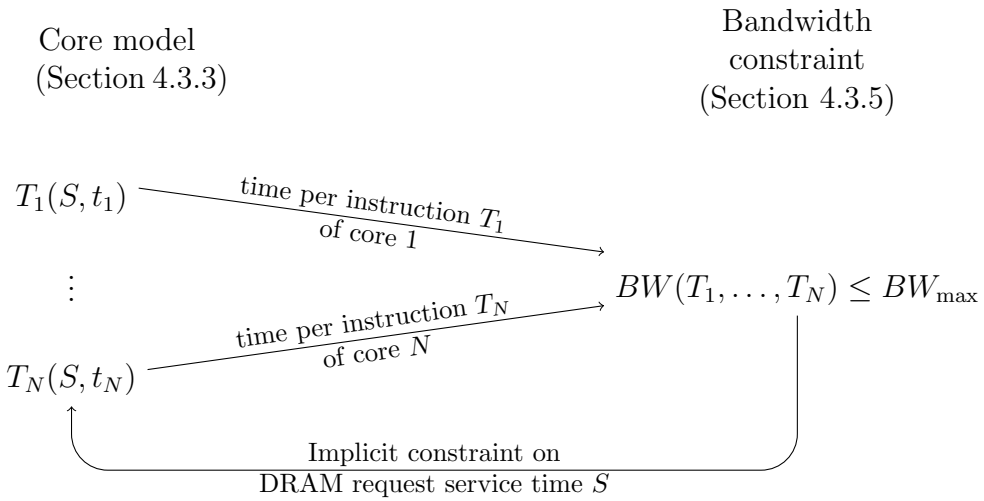


Figure 4.5: High level structure of the ILSB analytic model

4.3.2 Overview of Analytic Model

To accurately predict performance when DRAM bandwidth is saturated we need a new analytic model. Our previous limited bandwidth model for uniprocessors (Section 3.3.2) does not apply to CMPs because it does not account for DRAM bandwidth sharing. Specifically, the limited bandwidth model treats the core performance limit imposed by achievable DRAM bandwidth as a measurable parameter $T_{\text{memory}}^{\min}$, the minimum memory time per instruction. In a chip multiprocessor, however, the DRAM bandwidth achievable by a core is not a measurable parameter; instead the achievable bandwidth is a variable affected by the bandwidth consumption of the other cores, which may change drastically under frequency scaling. Thus, the limited bandwidth model is insufficient; a new analytic model is needed.

We construct this analytic model from an intuition of how limited DRAM bandwidth affects CMP performance. The intuition is simple: once the total DRAM bandwidth demanded by the cores exceeds available bandwidth, DRAM requests start spending more time waiting to be served, thereby reducing core performance until bandwidth demand equals available bandwidth. Note the feedback loop: core performance affects bandwidth demand, which affects request service time, which

affects core performance. The structure of our analytic model, shown at a high level in Figure 4.5, reflects this intuition.

We develop the model in three steps. First we develop the core model that expresses the time per instruction T_i of core i as a function of the core's average DRAM request service time S_i and cycle time t_i. Second, we make and justify a key approximation: that the average DRAM request service time is the same across cores; that is $S_1 = \cdots = S_N = S$. Third, we establish the bandwidth constraint: the total DRAM bandwidth $BW(T_1, \ldots, T_N)$ must be no greater than the maximum achievable DRAM bandwidth $BW_{\max}$. Together, these parts form our *independent latency shared bandwidth* (ILSB) analytic model of chip multiprocessor performance under frequency scaling.

4.3.3 Core Model

To develop the analytic model of core performance, we start with our previous limited bandwidth model for uniprocessors (Equation 3.3, Figure 3.8):

$$T(t) = \max\left(T^{\min}_{\text{memory}},\ C_{\text{compute}} \times t + T_{\text{demand}}\right).$$

Our goal is to augment this model to express how DRAM sharing affects bandwidth-bound core performance via average DRAM request service time S.

The average minimum memory time per instruction $T^{\min}_{\text{memory}}$ can be expressed as a function of average DRAM service latency using Little's law [52]. In queuing theory notation, Little's law states that the average queue length L is the product of the average arrival rate λ and the average time in queue W, that is $L = \lambda \times W$. Therefore, the maximum arrival rate supported by a finite queue of size Q is $\lambda_{\max} = Q/W$. Analogously, the maximum rate of DRAM requests supported by a core is the ratio of the number of DRAM request buffers (or MSHRs) and the average DRAM service latency S; that is, $\lambda_{\max} = N_{\text{MSHRs}}/S$. Assuming that the core generates R requests per instruction, we can express the maximum number of instructions per unit time $IPT_{\max} = \lambda_{\max}/R$. The minimum memory time per instruction $T^{\min}_{\text{memory}}$ is

the reciprocal of $IPT_{\max}$; therefore

$$T^{\min}_{\text{memory}}(S) = \frac{1}{IPT_{\max}} = \frac{R}{\lambda_{\max}} = \frac{R}{N_{\text{MSHRs}}} \times S.$$

We substitute this expression for $T^{\min}_{\text{memory}}(S)$ into the linear bandwidth model to generate our core model (illustrated in Figure 4.6):

$$T(S,t) = \max\left(\frac{R}{N_{\text{MSHRs}}} \times S,\ C_{\text{compute}} \times t + T_{\text{demand}}\right). \tag{4.1}$$

Note that for a given core frequency t there is a special DRAM service time $S_{\text{crossover}}$ for which the two sides of the max() are equal; that is, the core is on the edge between being latency-bound or bandwidth-bound. Setting the two sides of the max() equal,

$$\frac{R}{N_{\text{MSHRs}}} \times S_{\text{crossover}} = C_{\text{compute}} \times t + T_{\text{demand}},$$

we derive the expression for this value:

$$S_{\text{crossover}} = (C_{\text{compute}} \times t + T_{\text{demand}}) \times \frac{N_{\text{MSHRs}}}{R}. \tag{4.2}$$

We shall use the concept of this special boundary DRAM request service time (illustrated in Figure 4.8a) to explain how the entire analytic model fits together in Section 4.3.6.

4.3.4 Equal DRAM Request Service Time Approximation

To simplify our thinking about how DRAM bandwidth is shared among the cores, we make an approximation: the average service time (which includes both queuing and access latency) of a DRAM request is the same for each core. We justify this approximation by a) explaining why the average non-demand (store, prefetch, and writeback) DRAM request queuing latency is roughly the same across cores and b) showing that the average service time of all (demand and non-demand) DRAM requests from bandwidth-bound cores is dominated by non-demand DRAM request queuing latency.

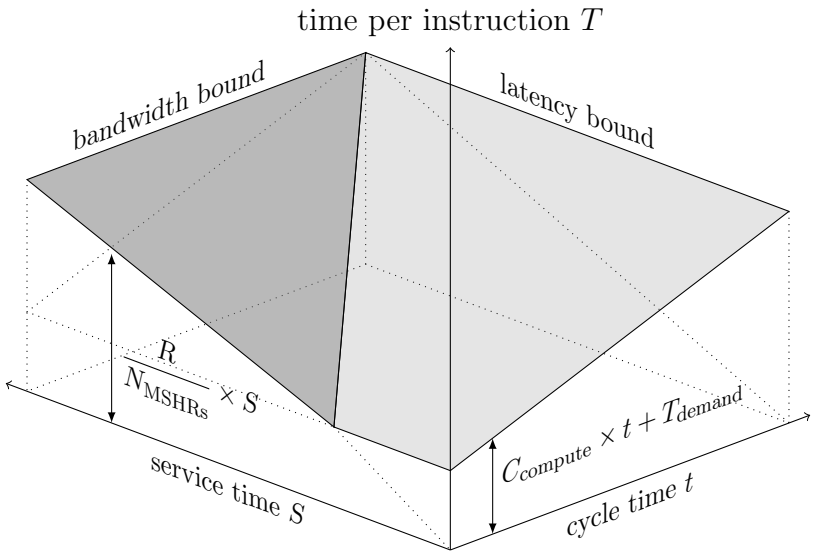


Figure 4.6: Core analytic model

To explain why average non-demand DRAM request queuing latency is roughly the same across cores, we recall our scarce row hit prioritization technique (Section 4.2). Specifically, scarce row hit prioritization mitigates priority inversion in DRAM scheduling, making DRAM scheduling of non-demand requests roughly first-come-first-serve (FCFS). Intuitively, since FCFS scheduling does not prioritize among cores, the average queuing time of a non-demand request does not depend on which core generated it. This intuition is supported by the PASTA ("Poisson arrivals see time averages") principle [26] from queuing theory.[5] Therefore, the average queuing latency of a non-demand DRAM request should be roughly the same for all cores.

We further approximate that the whole average DRAM request service time (including queuing and access latencies of both demand and non-demand requests) remains the same across cores. This approximation relies on two observations. First, as seen in the core analytic model (Section 4.3.3), the average DRAM request latency determines core performance only if the core is bandwidth-bound; therefore, our approximation need only be accurate for bandwidth-bound cores. Second, most DRAM

[5]The PASTA principle is proven true for an FCFS queue, Poisson inter-arrival time distribution, and independent inter-arrival and service times.

requests from bandwidth-bound cores are non-demand requests that spend most of their latency enqueued behind other requests. Therefore, the average service time of DRAM requests from bandwidth-bound cores is dominated by non-demand DRAM request queuing latency which we have just reasoned to be roughly the same across cores.

4.3.5 Bandwidth Constraint

The bandwidth constraint model is a mathematical expression of the fact that DRAM bandwidth BW is constrained by the maximum achievable DRAM bandwidth $BW_{\max}$.

To express this constraint, we first derive the total DRAM bandwidth BW. We start with the DRAM bandwidth BW_i of a single core i. Since a) the stream prefetcher makes the DRAM bus the main DRAM bottleneck in our system and b) each DRAM request occupies the same amount of time on the DRAM bus (the time needed to transfer a cache line), the bandwidth consumption of each request is the same; hence we express DRAM bandwidth as the DRAM request rate (the number of requests completed per unit time). Thus, if the time per instruction of core i is T_i and the number of DRAM requests per instruction of core i is R_i, the bandwidth of core i is $BW_i = R_i/T_i$. Hence, the total DRAM bandwidth of all N cores is

$$BW(T_1, \ldots, T_N) = \sum_{i=1}^{N} \frac{R_i}{T_i}. \tag{4.3}$$

To constrain BW by maximum achievable DRAM bandwidth $BW_{\max}$, we consider two cases: a) the DRAM bandwidth is not saturated and b) the DRAM bandwidth is saturated. The DRAM bandwidth is not saturated if the total bandwidth demanded by all cores would be less than maximum bandwidth even if all of the cores were latency-bound. In this case, according to the core analytic model (Section 4.3.3), the average DRAM request service time S has no effect on core performance; therefore we assume it to be zero for simplicity. The DRAM bandwidth is saturated if the total bandwidth demanded by all cores would be equal or greater

Variables

t_i	cycle time of core i
T_i	time per instruction of core i
S	average DRAM request service time
BW	total bandwidth (requests per unit time)

Parameters

$C_{\text{compute } i}$	compute cycles per instruction of core i
$T_{\text{demand } i}$	demand stall time per instruction of core i
R_i	DRAM requests per instruction of core i
$BW_{\max}$	maximum bandwidth (requests per unit time)
N_{MSHRs}	number of MSHRs in each core

For each core i:

$$T_i = \max\left(\frac{R_i}{N_{\text{MSHRs}}} \times S,\ C_{\text{compute } i} \times t_i + T_{\text{demand } i}\right)$$

For entire system:

$$BW = \sum_i \frac{R_i}{T_i}$$

$$\begin{cases} BW < BW_{\max} & \text{if } S = 0 \\ BW = BW_{\max} & \text{otherwise} \end{cases}$$

Figure 4.7: Complete mathematical description of the independent latency shared bandwidth (ILSB) model of chip multiprocessor performance under frequency scaling

than the maximum DRAM bandwidth if all of the cores were latency-bound. In this case, the average DRAM request service time S matters; in fact, S must be just long enough to make some cores bandwidth-bound, making these cores slower and reducing bandwidth demand to equal the maximum achievable bandwidth $BW_{\max}$. The bandwidth constraint for these two cases can be expressed mathematically as

$$\begin{cases} BW < BW_{\max} & \text{if } S = 0 \\ BW = BW_{\max} & \text{otherwise} \end{cases} \tag{4.4}$$

4.3.6 Combined Model

Having described the individual parts of our analytic model (the core model, the equal DRAM service time approximation, and the bandwidth constraint model) we can now examine the analytic model as a whole. For ease of reference, Figure 4.7 provides the complete mathematical description of the model.

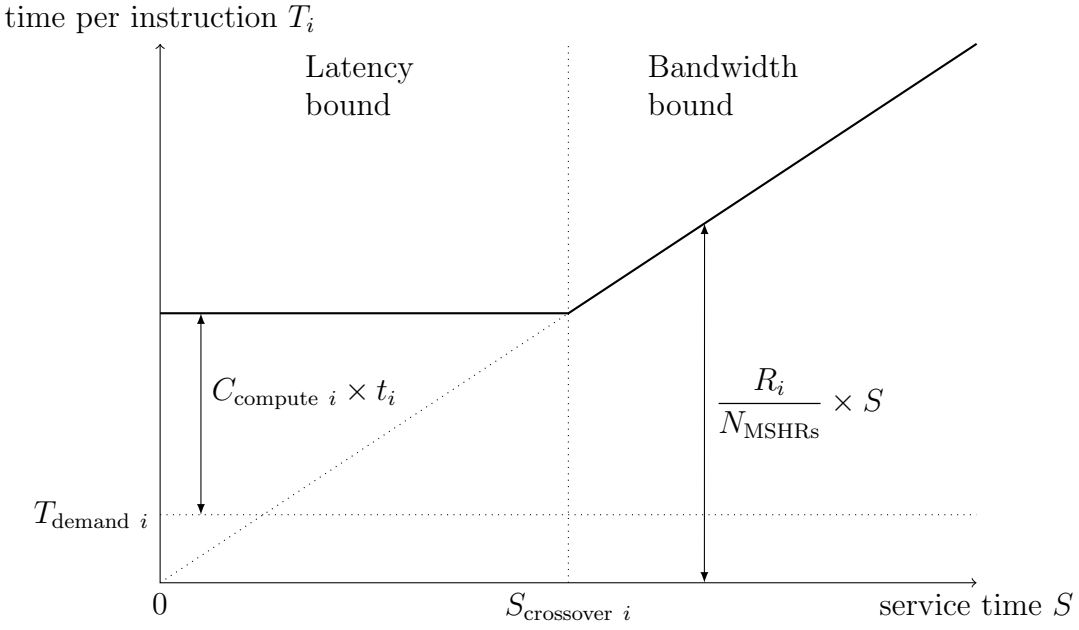
To examine how the entire model fits together, we consider the simple case of a two core CMP. The analytic model for this special case is graphically illustrated in Figure 4.8.

There are three possible cases: both cores are latency bound, one is bandwidth bound and another is latency bound, and both cores are bandwidth bound.

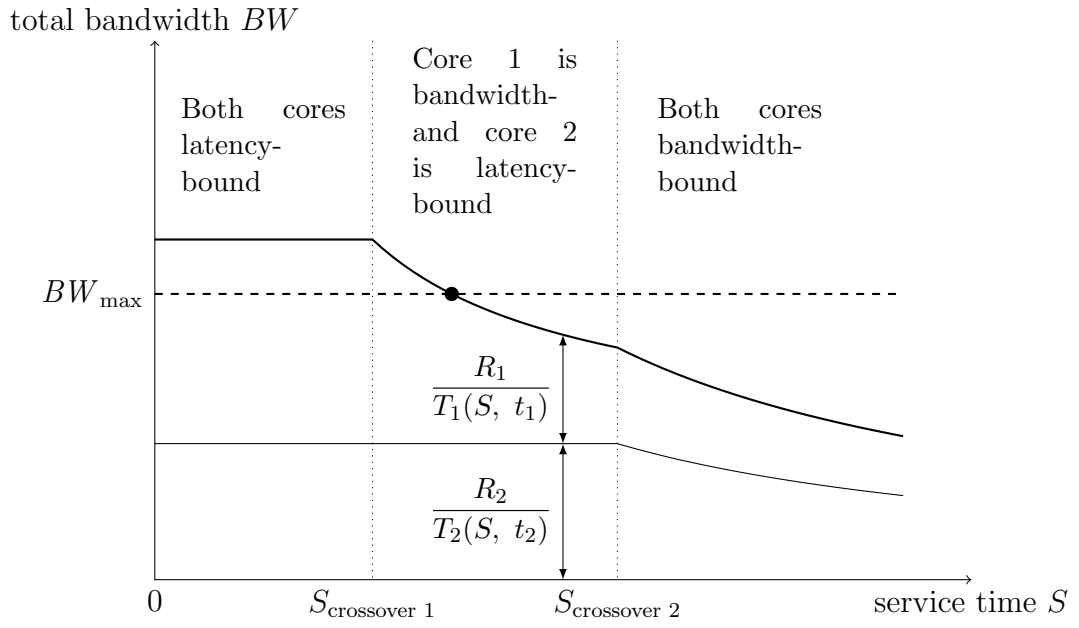
Suppose both cores (core 1 and core 2) are latency-bound. Then, as evident from Figure 4.8a, the average DRAM service time S (assumed to be the same for both cores as discussed in Section 4.3.4) is less than both $S_{\text{crossover 1}}$ and $S_{\text{crossover 2}}$. In this case, according to the core model, performance of each core is independent of S. If core performance does not change with S, then neither does the bandwidth consumption. Therefore, as shown in Figure 4.8b, the total bandwidth is constant for that range of DRAM service time S. Note that, in the figure, this bandwidth happens to be greater than the maximum achievable DRAM bandwidth BW_{max}; therefore, this case (both cores latency bound) violates the bandwidth constraint.

Suppose now that core 1 is bandwidth-bound and core 2 is latency-bound; that is, DRAM service time S is between $S_{\text{crossover 1}}$ and $S_{\text{crossover 2}}$. Then, as seen on the right side of Figure 4.8a, the time per instruction of core 1 is proportional to S. Therefore, the performance (instructions per unit time) and hence the bandwidth consumed by core 1 are proportional to $1/S$. Meanwhile, the bandwidth consumed by core 2 remains the same since core 2 is still latency-bound. Therefore, as shown in the middle of Figure 4.8b, as we increase S the total bandwidth decreases. In fact, total bandwidth becomes low enough to satisfy the bandwidth constraint.

The last case, both cores bandwidth-bound, should now be straightforward. As shown in Figure 4.8b, in this case the bandwidth consumption of both cores decreases



(a) Performance of core i as a function of service time S



(b) Total DRAM bandwidth as a function of service time S

Figure 4.8: Graphical illustration of the independent latency shared bandwidth model applied to two cores

with DRAM service time S. Note that this case, like the first, violates the bandwidth constraint. Specifically, while the bandwidth constraint is an inequality if $S = 0$ (all cores latency-bound), the constraint is an equation otherwise to reflect the notion that DRAM service time S is just long enough to keep DRAM bandwidth equal to $BW_{\max}$. In this case, however, DRAM bandwidth is actually less than $BW_{\max}$; hence, the bandwidth constraint is violated.

4.3.7 Solution

We now explain how the system of equations (and one inequality) comprising our ILSB model can be solved to obtain time per instruction T_i of each core i as a function of provided core cycle times t_i and the model parameters. Specifically, we first explain the key idea of the solution and then describe how the solution can be computed numerically or derived algebraically.

The key to solving the ILSB model lies in two observations:

1. The average DRAM request time S is the "root" variable of the system: if the value of S is known, all other variables can be computed from it (first the time per instruction T_i of each core i using Equation 4.1 and then total bandwidth BW using Equation 4.3 and the already computed T_i).

2. Total bandwidth BW is a monotonically non-increasing function of average DRAM request service time S. This fact makes intuitive sense: an increase in average DRAM request service time S cannot lead to performance improvement of any core and thus cannot lead to an increase of total bandwidth consumption.

Taken together, these observations imply that the ILSB model can be solved numerically using binary search over the range of the DRAM request service time S. Specifically, starting with some value of S, we can compute the total bandwidth BW, check whether BW is less than or greater than the bandwidth constraint $BW_{\max}$, adjust S in the appropriate direction and repeat. Once S is found, Equation 4.1 can be used to compute the predicted performance of each core.

The system may also be solved algebraically by considering $N + 1$ cases (where N is the number of cores). Without loss of generality, we number the cores $\{1, 2, \ldots, N\}$ in order of their respective values of $S_{\text{crossover}}$ such that $S_{\text{crossover 1}} \leq S_{\text{crossover 2}} \leq \cdots \leq S_{\text{crossover } N}$. The $N + 1$ cases to consider are:

$$\begin{array}{lrll}
\text{Case 1:} & 0 \leq S < S_{\text{crossover 1}} & \text{(all cores latency-bound)} \\
\text{Case 2:} & S_{\text{crossover 1}} \leq S < S_{\text{crossover 2}} & \begin{pmatrix}\text{core 1 bandwidth-bound,} \\ \text{others latency-bound}\end{pmatrix} \\
& \vdots & \\
\text{Case } N+1\text{:} & S_{\text{crossover } N} \leq S & \text{(all cores bandwidth-bound)}
\end{array}$$

Equation 4.3 states that total bandwidth $BW = \sum_{i=1}^{N} \frac{R_i}{T_i}$; we can now expand this expression. Specifically, for each case, we know which cores are latency-bound and which cores are bandwidth-bound; therefore, we know which side of the max() in Equation 4.1 for T_i applies for each core i. This knowledge allows us to express total bandwidth $BW_{\text{case } j}$ for each case j in which cores $[1, j-1]$ are bandwidth-bound and cores $[j, N]$ are latency-bound:

$$BW_{\text{case } j} = \underbrace{(j-1) \times \frac{N_{\text{MSHRs}}}{S}}_{\text{Bandwidth-bound cores}} + \underbrace{\sum_{i=j}^{N} \frac{R_i}{C_{\text{compute } i} \times t_i + T_{\text{demand } i}}}_{\text{Latency-bound cores}}. \tag{4.5}$$

For any case j except case 1, the bandwidth constraint (Section 4.3.5) posits $BW_{\text{case } j} = BW_{\max}$, an equation we can solve for average DRAM request service time S; as discussed in Section 4.3.5, we set S to be zero in case 1 (all cores latency-bound) for simplicity. Thus,

$$\begin{aligned}
S_{\text{case 1}} &= 0 \\
S_{\text{case } j \neq 1} &= \frac{(j-1) \times N_{\text{MSHRs}}}{BW_{\max} - \sum_{i=j}^{N} \frac{R_i}{C_{\text{compute } i} \times t_i + T_{\text{demand } i}}}.
\end{aligned} \tag{4.6}$$

Of these possible values of S, only one[6] corresponds to the case that contains the solution of the system:

$$S = \begin{cases} S_{\text{case 1}} & BW_{\text{case 1}} < BW_{\max} \\ S_{\text{case 2}} & S_{\text{crossover 1}} \leq S_{\text{case 2}} < S_{\text{crossover 2}} \\ & \vdots \\ S_{\text{case } j} & S_{\text{crossover } j-1} \leq S_{\text{case } j} < S_{\text{crossover } j} \\ & \vdots \\ S_{\text{case } N+1} & S_{\text{crossover } N} \leq S_{\text{case } N+1} \end{cases} \tag{4.7}$$

Given the expression for the average DRAM request service time S, we can plug it into the Equation 4.1 to find time per instruction T_i for each core i as a function of core cycle time t_i.

4.3.8 Approximations Behind ILSB

In this section, we address the implicit approximations behind the ILSB model. While these approximations are the same as those behind the limited bandwidth model of uniprocessor performance (explained in Section 3.3.2.1), ILSB assumes them to hold true for a chip multiprocessor. Specifically, we approximate that:

1. The average number of DRAM requests per instruction (both demands and prefetches) remains constant across core frequency combinations. As described in Section 3.3.2.1, this approximation may be problematic if prefetcher aggressiveness is allowed to vary.

2. The DRAM scheduler efficiency remains the same across core frequency combinations; in particular, the average overhead of switching the DRAM bus direction per DRAM request remains the same.

[6]The uniqueness of the solution is evident from the mostly monotonically decreasing behavior of total bandwidth BW as a function of S, illustrated in Figure 4.8b. The only exception to this behavior is the case of all cores being latency-bound, in which BW is constant with respect to S; however, in this case we have forced the solution to be unique by defining it as $S = 0$.

4.4 Parameter Measurement

Having developed the ILSB model, we now design the hardware mechanisms that measure its parameters. As described in Section 2.2, the ILSB model and these hardware mechanisms together comprise our performance predictor. We first describe measurement of core model parameters and then measurement of the maximum DRAM bandwidth $BW_{\max}$.

4.4.1 Core Model Parameters

The core model parameters are easy to obtain; they are either design parameters (N_{MSHRs}, the number of MSHRs per core), easily measured using simple performance counters (R, the number of DRAM requests per instruction), or measured using mechanisms we proposed previously for the uniprocessor (demand time per instruction T_{demand} and compute cycles per instruction C_{compute}, measured using uniprocessor mechanisms described in Sections 3.3.3.1 and 3.3.3.4 respectively).

Note that in accordance with our decision to use the simplest parameter measurement mechanisms possible (Section 3.6), we use stall time rather than CRIT to measure T_{demand}. Specifically, we employ the "would be timely" request classification from Section 3.3.3.1 and apply it to stall time as follows: we measure T_{demand} as the time a core stalls on non-"would be timely" demand DRAM requests divided by the number of instructions retired.

4.4.2 Maximum DRAM Bandwidth

To measure maximum DRAM bandwidth $BW_{\max}$, we reuse the relevant uniprocessor mechanism; we also extend that mechanism to work with multiple DRAM channels (a common feature of chip multiprocessors).

Specifically, for a single channel, we measure the average time T_{request} each DRAM request occupies the DRAM bus as described in Section 3.3.3.2. In our experiments, we observe that a bandwidth saturated DRAM system shared by multiple

cores of a chip multiprocessor achieves roughly 97% bus utilization; hence, we compute the the maximum bandwidth $BW_{\max}$ for a single channel as follows:

$$BW_{\max} \text{ of a single DRAM channel} = 0.97 \times \frac{1}{T_{\text{request}}}. \tag{4.8}$$

The addition of multiple DRAM channels introduces a new wrinkle in maximum DRAM bandwidth measurement. Specifically, since DRAM requests map to channels non-uniformly in the short term, the average DRAM bus utilization drops significantly. To deal with this problem we extend the DRAM controller to measure channel level parallelism *CLP*. For every interval of 128 DRAM requests (the number of requests in the DRAM controller queue), the DRAM controller determines which channel satisfied the most requests. We call this channel *critical*. The *CLP* is simply 128 divided by the number of requests satisfied by the critical channel. We also ensure that T_{request} is computed using only critical channel measurements because the other, less stressed channels usually allow more bus direction switch overhead.

With these changes, the processor can compute the maximum DRAM bandwidth $BW_{\max}$ for the general case of multiple DRAM channels:

$$BW_{\max} = 0.97 \times CLP \times \frac{1}{T_{\text{request}}}. \tag{4.9}$$

4.5 Methodology

The independent latency shared bandwidth (ILSB) analytic model and the mechanisms for measuring its parameters comprise our DVFS performance predictor for private cache chip multiprocessors; we evaluate the performance improvement within the power budget due to this predictor using the methodology detailed below.

4.5.1 Metric

We choose performance within a power budget as the target metric because performance and power budget are, respectively, a fundamental metric of interest and a fundamental constraint in computer system design. In addition, switching from

energy (targeted in Chapter 3) to performance as the target metric demonstrates that DVFS performance prediction is beneficial in diverse usage scenarios.

We define multicore performance as the geometric mean of instructions per unit time for each core; we choose this multicore performance metric over the more commonly used weighted speedup [76] for two reasons:

1. We wish to evaluate our DVFS performance predictor in isolation. Specifically, since weighted speedup is computed using each core's performance running alone on the system, a performance predictor targeting weighted speedup would have to include a mechanism to estimate each core's alone performance—a mechanism that would introduce prediction errors and complicate evaluation of the analytic model itself.

2. We wish to exploit the potential performance benefit of per-core DVFS only; we do not wish to exploit the imbalance in how the target metric weighs performance of each core. Specifically, we consider the potential performance benefit of per-core DVFS to come from the imbalance in the marginal utility of extra power consumption among cores. However, weighted speedup introduces another source of potential performance improvement: the imbalance in how weighted speedup weighs performance of each core. For example, a per-core DVFS controller could cause a 2x performance loss on one core and a 10% performance gain on another and still deliver a net gain in weighted speedup. We attribute such performance gains to the imbalance in how weighted speedup weighs performance of each core and not to per-core DVFS. Indeed, any other shared resource management technique (such as DRAM bandwidth partitioning) could exploit this imbalance in weighted speedup. In contrast, the geometric mean weighs performance of each core equally: a 2x performance loss must be offset by a more than 2x performance gain in order to yield a net gain in geometric mean. Thus, a DVFS controller that improves the geometric mean of each core's performance exploits only the imbalance in marginal utility of

General		Frontend (Medium/Big)		OOO Core (Medium/Big)	
Cores	4	Microinstr./cycle	4/8	Microinstr./cycle	4/8
Base freq.	2.8 GHz	Branches/cycle	2	Pipeline depth	14
Min freq.	1.5 GHz	BTB entries	4K	ROB size	128/256
Max freq.	4.5 GHz	Predictor	hybrid[a]	RS size	48/96

All Caches			ICache	DCache	L2	Stream prefetcher [84], per core			
Line	64 B	Size	32 KB	32 KB	1 MB	Streams	16	Distance	64
MSHRs	32	Assoc.	4	4	8	Queue	128	Degree	4
Repl.	LRU	Cycles	1	2	12	Training threshold			4
		Ports	1R,1W	2R,1W	1	L2 insertion			mid-LRU

DRAM Controller		DDR3 [58] Channel				DRAM Bus	
Window	128 reqs	Chips	8×256 MB	Row	8 KB	Freq.	800 MHz
Priority scheduling[b]		Banks	8	CAS[c]	13.75 ns	Width	8 B

[a] 64K-entry gshare + 64K-entry PAs + 64K-entry selector.
[b] Priority order: scarce row hit (Section 4.2), demand (instr. fetch or data load), oldest.
[c] CAS = $t_{\mathrm{RP}} = t_{\mathrm{RCD}} = \mathrm{CL}$;
other modeled DDR3 constraints: BL, CWL, $t_{\{\mathrm{RC,\ RAS,\ RTP,\ CCD,\ RRD,\ FAW,\ WTR,\ WR}\}}$.

Table 4.2: Simulated private cache CMP configuration

extra power consumption among cores—an imbalance that cannot be exploited by other shared resource management techniques.

4.5.2 DVFS Controller

We evaluate our performance predictor by modeling a DVFS controller that operates as described in Section 2.2. In all experiments except the relevant sensitivity study the DVFS controller considers a change of core frequencies every 1M baseline 2.8 GHz cycles (roughly 357 µs).

4.5.3 Simulation

We use an in-house cycle-level simulator driven by the x86 functional model from Multi2Sim [85]. We evaluate both private cache and shared cache CMPs in three four-core configurations: a) 4-wide medium cores with one DRAM channel, b) 8-wide

big cores with one DRAM channel, and c) 8-wide big cores with 2 DRAM channels. The 8-wide core represents modern SIMD-capable cores (our simulator currently does not support SIMD). Table 4.2 details the simulation parameters.

Our simulation proceeds in three stages:

1. Instruction level warmup, in which each application in the four-core workload runs for 50M instructions and warms up the branch predictors and the cache hierarchy. This stage allows even low IPC benchmarks like `mcf` to properly warm up these structures.

2. Cycle accurate warmup, in which the processor is simulated in the baseline 2.8 GHz configuration for 10M cycles, after which all statistic counts are reset. This stage warms up the entire microarchitecture and ends in the same microarchitectural state for both the baseline and the simulated predictors.

3. Cycle accurate simulation proper. For baseline 2.8 GHz experiments, this stage proceeds for 100M cycles; for predictor experiments, this stage proceeds until all cores retire the same number of instructions as in the baseline. The core performance is computed at the time the core reaches its baseline instruction count; however, the core continues to run to provide interference for other cores.

4.5.4 Workloads

We use two sets of workloads: ALL and BW. The ALL workloads are 29 random combinations of SPEC2006 benchmarks chosen so that each benchmark appears in exactly four workloads and no workload has two of the same benchmark. The BW workloads were generated as follows. We ran 200 random combinations of SPEC2006 in the baseline medium core configuration. Of the 200 workloads, a third (67) were bandwidth bound (more than 90% bus utilization). The 25 BW workloads were randomly chosen from these 67 workloads.

The SPEC 2006 benchmarks were compiled using the GNU Compiler Collection version 4.3.6 with the `-O3` option. The benchmarks were run from checkpoints

689 frequency combinations for full performance study, GHz (44 ordered combinations listed)				
2.8 2.8 2.8 2.8	2.6 2.6 2.8 3.0	2.6 2.6 2.6 3.2	2.4 2.8 2.8 3.0	2.4 2.6 3.0 3.0
2.4 2.6 2.8 3.2	2.4 2.4 3.0 3.2	2.4 2.4 2.6 3.4	2.2 2.8 3.0 3.0	2.2 2.8 2.8 3.2
2.2 2.6 2.6 3.4	2.2 2.4 2.8 3.4	2.2 2.4 2.4 3.6	2.2 2.2 3.2 3.2	2.2 2.2 3.0 3.4
2.2 2.2 2.6 3.6	2.2 2.2 2.2 3.8	2.0 2.6 3.0 3.2	2.0 2.6 2.8 3.4	2.0 2.4 3.2 3.2
2.0 2.4 2.6 3.6	2.0 2.2 2.8 3.6	2.0 2.2 2.4 3.8	1.8 3.0 3.0 3.0	1.8 2.8 3.0 3.2
1.8 2.6 2.6 3.6	1.8 2.4 3.0 3.4	1.8 2.0 3.2 3.4	1.8 2.0 3.0 3.6	1.8 2.0 2.6 3.8
1.8 2.0 2.0 4.0	1.8 1.8 2.2 4.0	1.6 2.8 2.8 3.4	1.6 2.6 3.2 3.2	1.6 2.4 2.8 3.6
1.6 2.4 2.4 3.8	1.6 2.2 3.2 3.4	1.6 2.2 2.6 3.8	1.6 2.0 2.2 4.0	1.6 1.8 2.8 3.8
1.6 1.8 2.4 4.0	1.6 1.6 3.4 3.4	1.6 1.6 3.2 3.6	1.6 1.6 1.6 4.2	
121 frequency combinations for relevant sensitivity study, GHz (10 ordered combinations listed)				
2.8 2.8 2.8 2.8	2.4 2.4 2.8 3.2	2.0 2.8 2.8 3.2	2.0 2.4 3.2 3.2	2.0 2.4 2.4 3.6
2.0 2.0 2.8 3.6	1.6 2.4 2.8 3.6	1.6 2.0 2.0 4.0	1.6 1.6 3.2 3.6	1.6 1.6 2.4 4.0
43 frequency combinations for oracle and relevant sensitivity study, GHz (5 ordered combinations listed)				
2.8 2.8 2.8 2.8	2.2 2.2 2.8 3.4	1.6 2.8 2.8 3.4	1.6 1.6 3.4 3.4	1.6 1.6 2.2 4.0

Table 4.3: Core frequency combinations. Only ordered combinations are listed; the rest are permutations of the listed combinations.

taken using Pincpt [59] at the beginning of their 200M instruction long representative regions determined using Pinpoints [68].

4.5.5 Frequency Combinations

All experiments except for oracle studies and the relevant sensitivity study were run with 689 available four-core frequency combinations. We chose the combinations by a) enumerating all combinations of core frequencies between 1.6 GHz and 4.2 GHz (with a step of 200 MHz), b) discarding all combinations that were over the peak dynamic power budget of the baseline 2.8 GHz configuration (assuming that peak dynamic power budget is proportional to the cube of frequency), and c) discarding all non-Pareto optimal combinations (a combination is Pareto optimal if in all other frequency combinations at least one core is assigned a lower frequency than

the same core in the considered combination).

Two smaller sets of frequency combinations (for oracle and sensitivity studies) were generated in the same way using steps of 400 MHz (yielding 121 combinations) and 600 MHz (43 combinations).

All three sets of frequency combinations are specified in Table 4.3. Note that the table does not list every single combination. Instead, the table lists only ordered combinations (those in which the frequencies are sorted in ascending order); the rest can be generated by permuting the listed ordered combinations. For example, an ordered combination $\{2.6, 2.6, 2.6, 3.2\}$ can be permuted into three other frequency combinations: $\{3.2, 2.6, 2.6, 2.6\}$, $\{2.6, 3.2, 2.6, 2.6\}$, and $\{2.6, 2.6, 3.2, 2.6\}$.

4.5.6 Oracle Policies

We measure potential gains by simulating two oracle DVFS control policies: myopic oracle and perfect memoryless.

The *myopic oracle* policy uses oracle knowledge of the next DVFS interval to choose the frequency combination for that interval. Specifically, the myopic oracle predictor simulates all frequency combinations for the next interval and chooses the one with the highest performance. We call this oracle "myopic" because it has no knowledge beyond the next DVFS interval. Due to this limitation (necessary to keep simulation time reasonable), the oracle may choose short term optimal frequency combinations that are suboptimal in the long run.

We carry over the *perfect memoryless* policy from our uniprocessor methodology (Section 3.4.5). Like the myopic oracle policy, the perfect memoryless policy relies on simulating all frequency combinations for each DVFS interval; unlike the myopic oracle policy, the perfect memoryless policy switches to the frequency combination that results in the highest performance during the previous (rather than next) DVFS interval. This policy mimics a performance predictor that correctly predicts the best frequency combination for every interval after the interval is complete and assumes the same frequency combination is the best for the next interval.

To reduce simulation time, we run oracle studies (including the baseline, real predictors, and both oracle policies) for 10M cycles instead of 100M cycles after warmup and with only 43 frequency combinations.

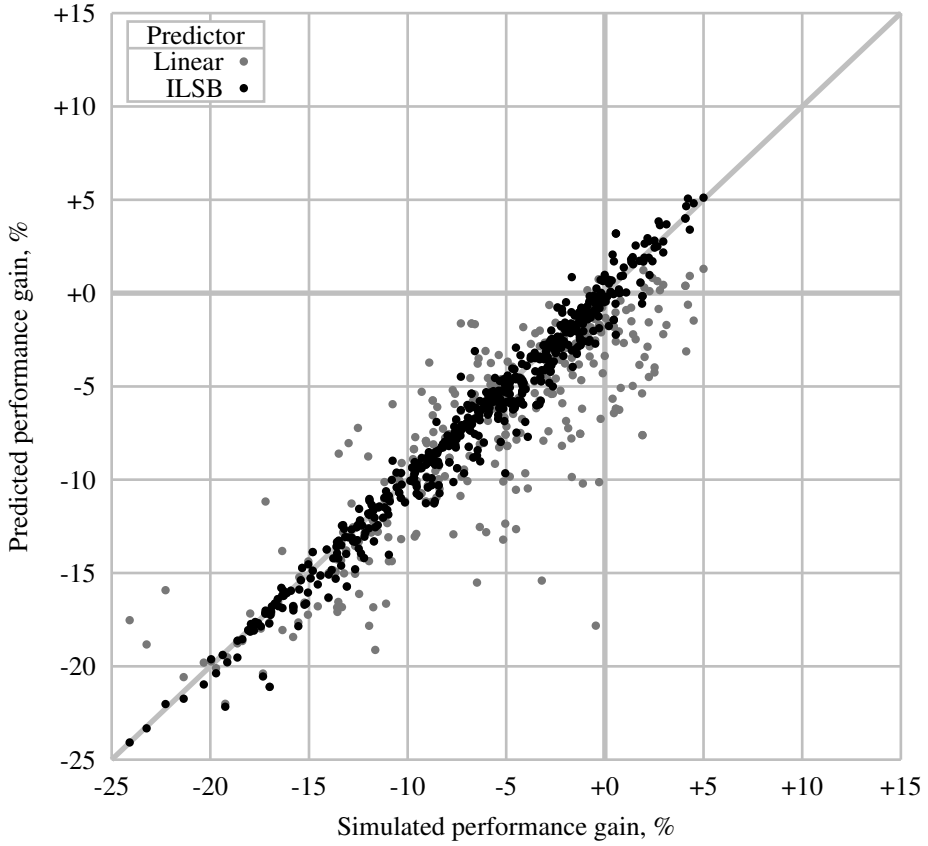
4.6 Results

Before delving into the performance results, we first revisit the prediction accuracy study from Section 4.3.1.1. Figure 4.9a compares the prediction accuracy of our performance predictor based on our ILSB model to that of a predictor based on the prior linear model. This figure together with Figure 4.4a show that the ILSB model leads to significant improvement in prediction accuracy, particularly among bandwidth-bound workloads. Specifically, ILSB identifies 79% of points representing a performance gain, whereas the linear model identifies only 25% (both ILSB and linear model identify 99% of performance loss points).

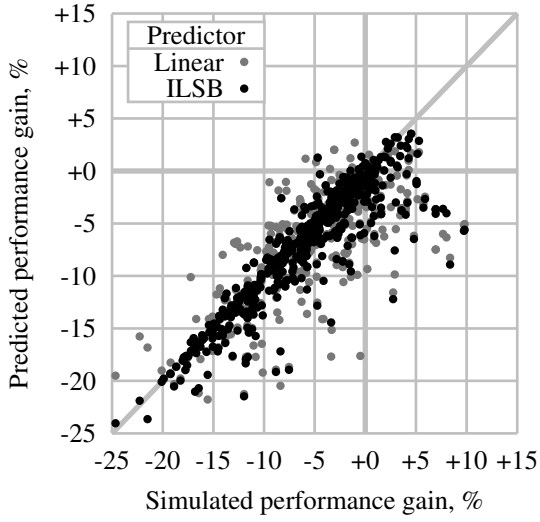
We also show the prediction accuracy of the ILSB model applied to a private cache CMP without demand prioritization and scarce row hit prioritization in Figures 4.9b and 4.9c. Note the degradation in prediction accuracy demonstrated by these figures. As discussed in Sections 4.2.4, 4.3.1, and 4.3.4, demand prioritization and scarce row hit prioritization are major factors behind the ILSB model assumptions; the significant improvement in prediction accuracy we see when these features are turned on supports these explanations.

Figure 4.10 shows the detailed results of an oracle study for a private cache CMP (medium cores, 1 DRAM channel) with ALL workloads. Note that oracle performance gains vary from 0% to 7% depending on the workload; for most workloads, our performance predictor realizes at least 80% of oracle performance.

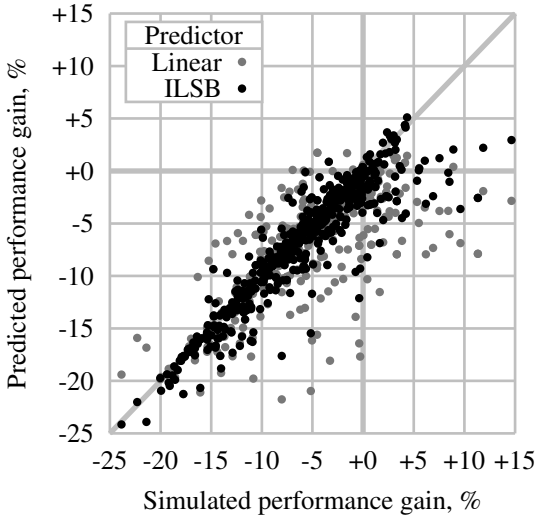
Since bandwidth-bound workloads promise the largest performance gains, we also show the detailed results of a full performance study for a private cache CMP (medium cores, 1 DRAM channel) with BW workloads in Figure 4.11. Note that the performance predictor based on the prior linear model delivers less than half of the



(a) Scarce row hit, demand, oldest



(b) Row hit, oldest (FR-FCFS)



(c) Row hit, demand, oldest

Figure 4.9: Accuracy of the ILSB model and the linear model applied to a four-core private cache CMP with three different DRAM scheduling priority orders

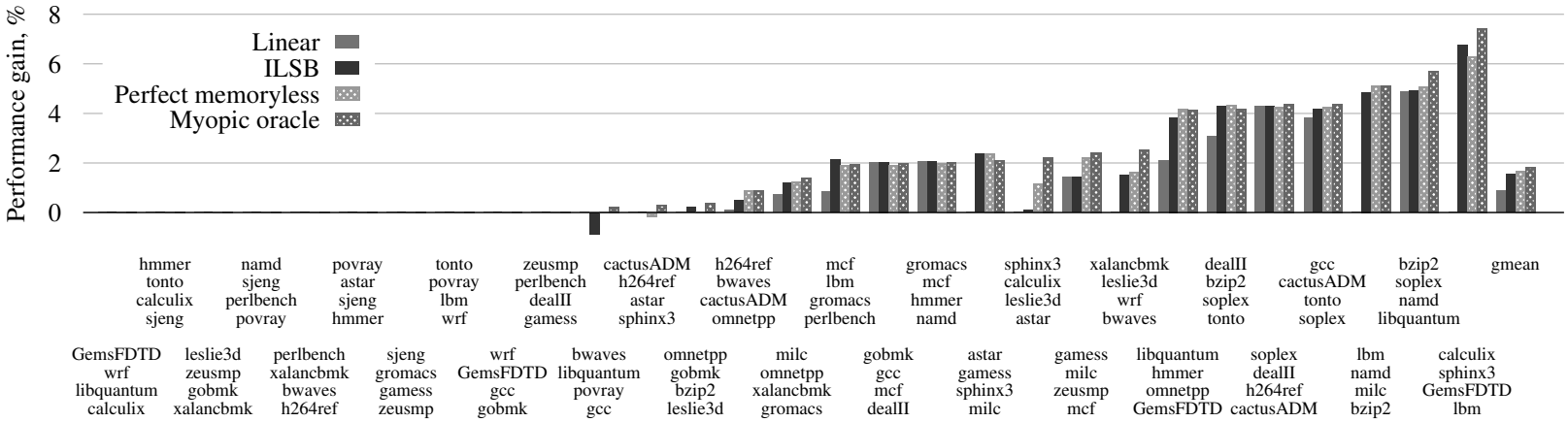


Figure 4.10: Oracle performance study on private cache CMP (medium cores, 1 DRAM channel) with ALL workloads

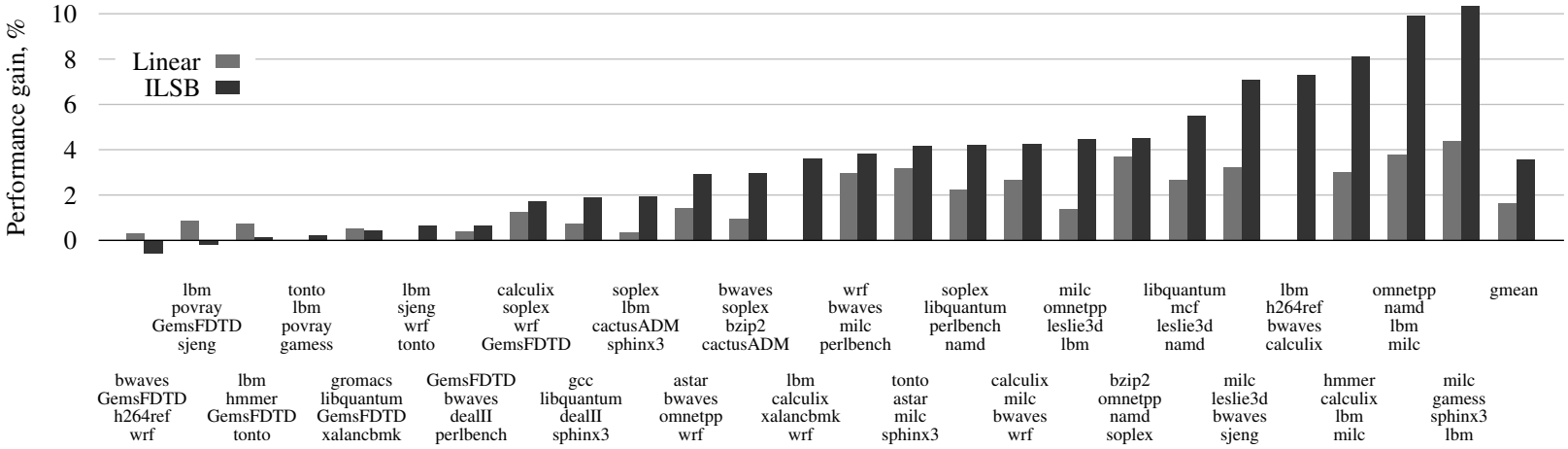


Figure 4.11: Full performance study on private cache CMP (medium cores, 1 DRAM channel) with BW workloads

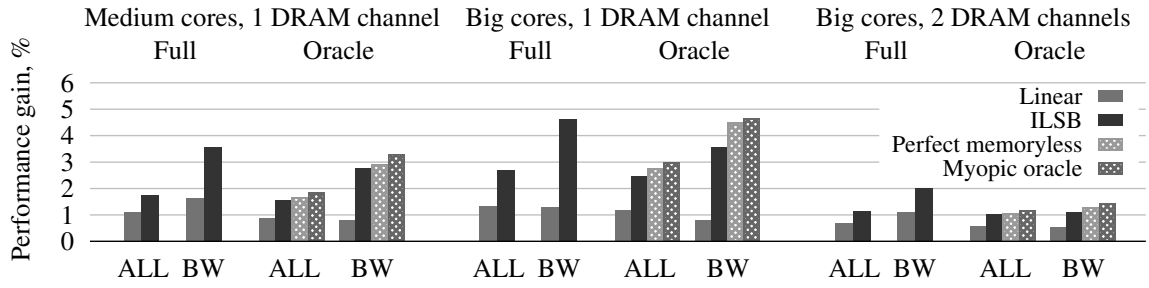


Figure 4.12: Summary of experimental results for the private cache CMP

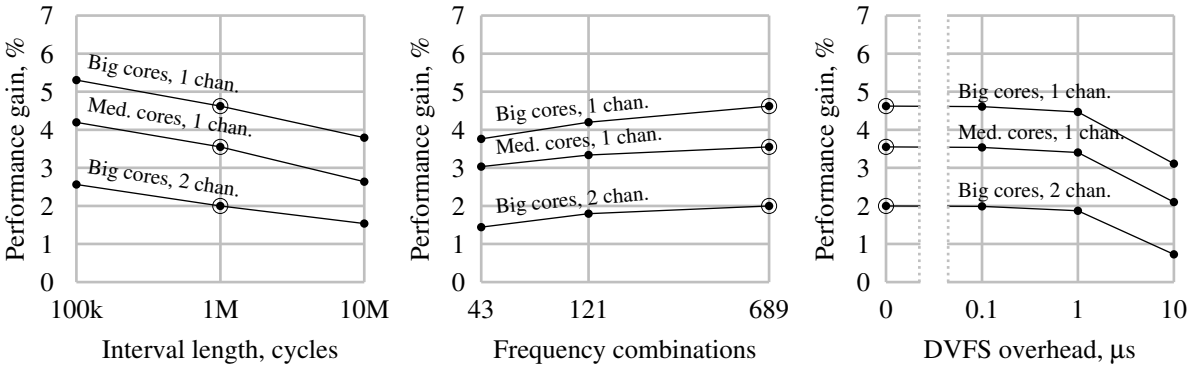


Figure 4.13: Sensitivity studies for the private cache CMP with BW workloads (circled points represent default settings)

benefit of our performance predictor, particularly on those workloads for which our predictor provides the most benefit (more than 5%).

We summarize the rest of our experimental results in Figure 4.12, showing average performance gains in the three simulated configurations, including full and oracle studies, and using the ALL and BW workload sets. Note that our predictor delivers 5% performance improvement on BW workloads in the big cores, 1 DRAM channel configuration.

Figure 4.13 shows that our performance predictor works for a variety of DVFS interval sizes, numbers of frequency combinations, and DVFS overheads. Note in particular that voltage and frequency switch overheads of up to 1 µs do not degrade

performance significantly. Higher overheads can still be tolerated by increasing the length of the DVFS interval.

4.7 Conclusions

In this chapter, we have shown that accurate DVFS performance prediction for private cache chip multiprocessors is feasible. Specifically, we have designed a DVFS performance predictor for private cache CMPs that helps the DVFS controller realize most of the gains realized by an oracle predictor. As in the uniprocessor case (Chapter 3), our DVFS performance predictor for private cache CMPs derives most of its benefit from taking into account the finite off-chip bandwidth.

Chapter 5

Shared Cache Chip Multiprocessor

> Exactly! It is absurd—improbable—it cannot be.
> So I myself have said. And yet, my friend, *there it is!*
> One cannot escape from the facts.
>
> Hercule Poirot
> *Murder on the Orient Express*

In this chapter we extend the DVFS performance predictor we designed for the private cache CMP to work with the shared cache CMP shown in Figure 5.1. We first explain the problems posed by the introduction of shared last level cache, then show that our predictor for the private cache CMP still works with the shared cache CMP, explain why, and propose two improvements to make the predictor more robust.

5.1 Problems Posed by Shared Cache

The introduction of shared last level cache complicates DVFS performance prediction in two ways.

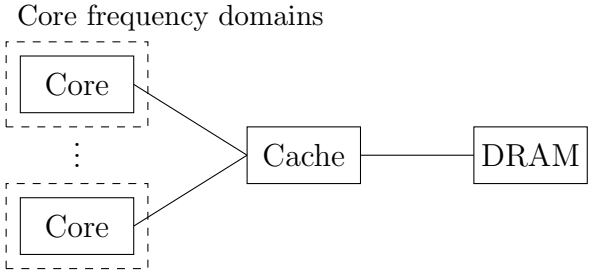


Figure 5.1: Shared cache CMP

First, unlike private caches, the shared cache lies outside of core frequency domains. Therefore, as core frequencies change, the time needed to access the last level cache remains the same. This fact contradicts our prior view of latency-bound core performance, which assumes that last level cache access time scales with core frequency and only DRAM latencies remain the same under core frequency scaling.

Second, the natural[1] partition of shared cache capacity among the cores may change under core frequency scaling, violating an important but so far implicit assumption of our independent latency shared bandwidth (ILSB) model. This assumption states that the number of DRAM requests per instruction is a workload characteristic that stays constant under frequency scaling. Of course, if the shared cache capacity allocated to a core changes under frequency scaling, the shared cache miss rate of the core would also change, thereby changing the number of DRAM requests per instruction for that core and thus violating the assumption.

5.2 Experimental Observations

Despite these potential concerns, our experiments show that the DVFS performance predictor we designed for the private cache CMP in Chapter 4 performs well in the shared cache CMP configuration. The methodology of these experiments mirrors the methodology of our private cache CMP experiments (Section 4.5) with the obvious replacement of private caches with a shared last level cache. Table 5.1 lists the simulation parameters.

Figures 5.3, 5.4, and 5.2 present these experimental results. As in the private cache CMP evaluation, we show an oracle and a full performance study for the medium core, 1 DRAM channel configuration (Figures 5.3, 5.4) and then present a summary of the rest of the experiments (Figure 5.2).

[1]The *natural* shared cache partition is the one that results from the regular LRU replacement policy without any explicit shared cache partitioning mechanisms.

General		Frontend (Medium/Big)		OOO Core (Medium/Big)	
Cores	4	Microinstr./cycle	4/8	Microinstr./cycle	4/8
Base freq.	2.8 Ghz	Branches/cycle	2	Pipeline depth	14
Min freq.	1.5 GHz	BTB entries	4K	ROB size	128/256
Max freq.	4.5 GHz	Predictor	hybrid[a]	RS size	48/96

All Caches			ICache	DCache	L2	Stream prefetcher [84], per core			
Line	64 B	Size	32 KB	32 KB	4 MB	Streams	16	Distance	64
MSHRs	32	Assoc.	4	4	8[c]	Queue	128	Degree	4
Repl.	LRU	Cycles	1	2	16	Training threshold			4
		Ports	1R,1W	2R,1W	1	L2 insertion			mid-LRU

DRAM Controller		DDR3 [58] Channel				DRAM Bus	
Window	128 reqs	Chips	8×256 MB	Row	8 KB	Freq.	800 MHz
Priority scheduling[b]		Banks	8	CAS[d]	13.75 ns	Width	8 B

[a] 64K-entry gshare + 64K-entry PAs + 64K-entry selector.
[b] Priority order: scarce row hit (Section 4.2), demand (instr. fetch or data load), oldest.
[c] L2 associativity is 16 in experiments with shared cache partitioning on.
[d] CAS = $t_{\mathrm{RP}} = t_{\mathrm{RCD}} = \mathrm{CL}$;
other modeled DDR3 constraints: BL, CWL, $t_{\{\mathrm{RC,\ RAS,\ RTP,\ CCD,\ RRD,\ FAW,\ WTR,\ WR}\}}$.

Table 5.1: Simulated shared cache CMP configuration

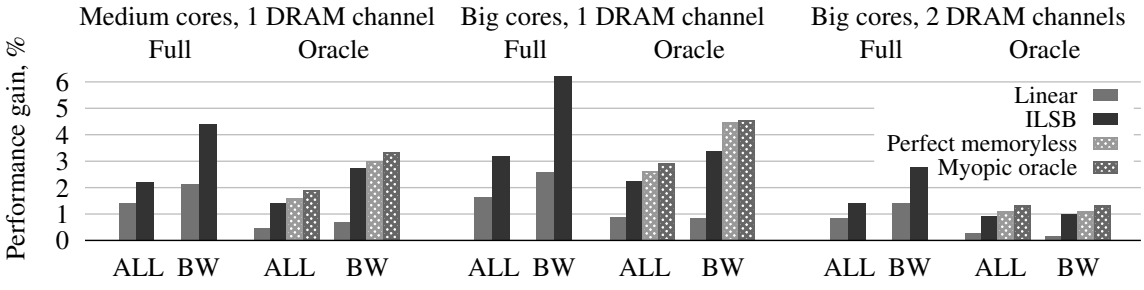


Figure 5.2: Summary of experimental results for our DVFS performance predictor for the private cache CMP applied to the shared cache CMP

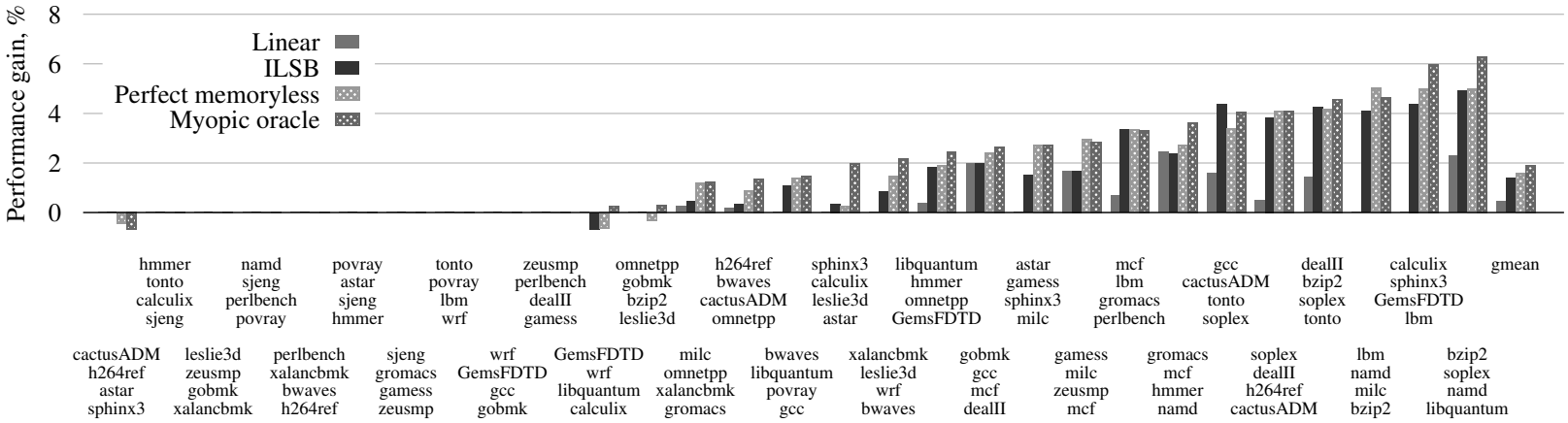


Figure 5.3: Oracle performance study of our DVFS performance predictor for the private cache CMP applied to the shared cache CMP (medium cores, 1 DRAM channel) with ALL workloads

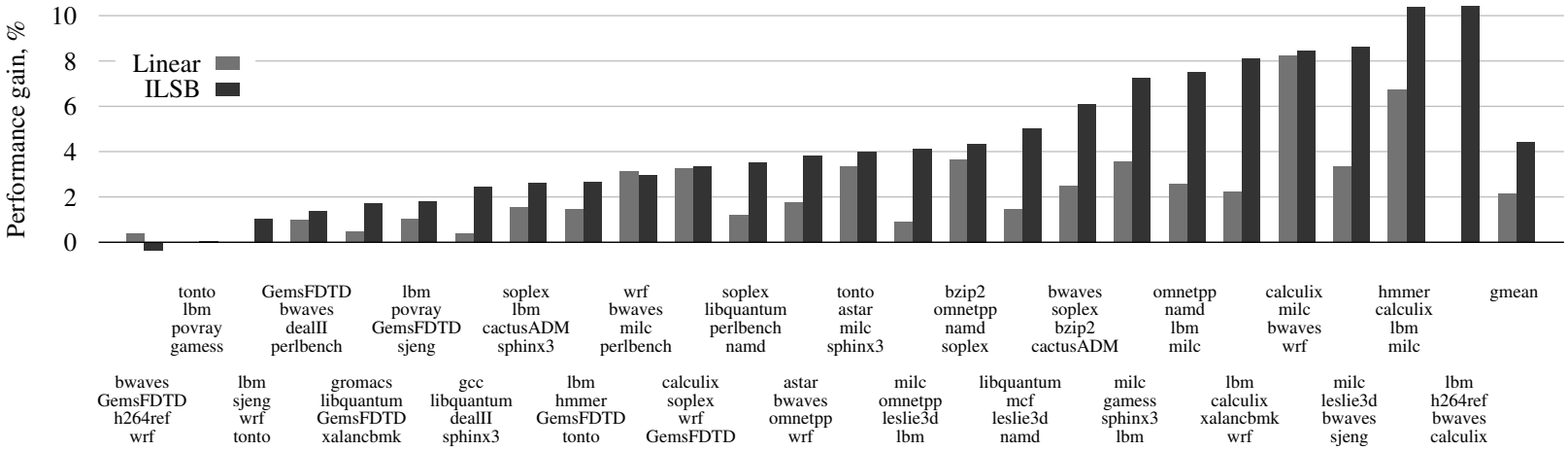


Figure 5.4: Full performance study of our DVFS performance predictor for the private cache CMP applied to the shared cache CMP (medium cores, 1 DRAM channel) with BW workloads

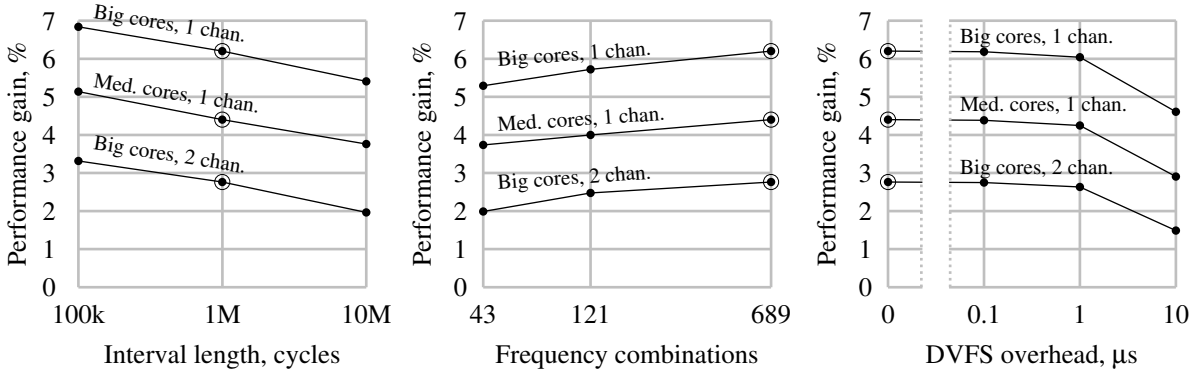


Figure 5.5: Sensitivity studies of our DVFS performance predictor for the private cache CMP applied to the shared cache CMP with BW workloads (circled points represent default settings)

As evident from these figures, our DVFS performance predictor for private cache CMPs maintains its high performance in the presence of a shared cache. Specifically, in the medium core, 1 DRAM channel configuration, the predictor realizes 74% of average oracle performance gains (Figure 5.3) and delivers 4.4% average performance gains on bandwidth-bound workloads (Figure 5.4). The summary of experimental results shown in Figure 5.2 demonstrates similar trends for the rest of the simulated configurations. In particular, our DVFS performance predictor for the private cache CMP improves performance of bandwidth-bound workloads by 6.2% in the big core, 1 DRAM channel configuration.

The sensitivity studies shown in Figure 5.5 confirm that our DVFS predictor for the private cache CMP still works well with the shared cache CMP for a variety of key parameter settings.

5.3 Analysis

We now explain why the DVFS performance predictor we designed for the private cache CMP in Chapter 4 still works well with the shared cache CMP. There are two major reasons: the ability of the out-of-order processor to hide shared LLC

access latencies and the relative constancy of the natural shared cache partition under frequency scaling. We discuss each reason in turn.

The ability of the out-of-order processor to hide shared LLC access latencies mitigates the first problem posed by shared cache: the fact that the shared cache lies outside of core frequency domains making shared cache access latency independent of core frequencies. As discussed in Section 5.1, this fact is a potential problem because we designed the evaluated DVFS performance predictor assuming the opposite: that LLC latencies scale with core frequencies (as they do in the private cache CMP). However, due to the ability of the out-of-order processor to hide these latencies, they do not have much impact on performance. In fact, in our baseline studies across the three simulated configurations, the fraction of time a core stalls on an LLC access does not exceed 8% and is less than 4% in 91% of the cases. Since LLC access latencies stall the cores for only a small fraction of execution time, the error due to inaccurately predicting that fraction is small relative to total execution time and thus has negligible effect on performance prediction accuracy.

The relative constancy of the natural shared cache partition under frequency scaling mitigates the second potential problem posed by shared cache: the potential failure of the assumption that the number of DRAM requests per instruction of a core remains the same under frequency scaling.

To show that the natural shared cache partition does not change significantly under frequency scaling, we first define the distance between two shared cache partitions:

> The *distance* between two shared cache partitions is the fraction of shared cache capacity not allocated to the same core in both partitions.

For example, given two shared cache partitions {10%, 20%, 30%, 40%} and {15%, 20%, 40%, 25%}, the distance between them is 15%—the sum of the 5% gained by core 1 and the 10% gained by core 3 at the expense of core 4. More formally, treating a

shared cache partition x among N cores as a vector $\{x_1, x_2, \ldots, x_N\}$ of cache fractions allocated to each core, we define:

$$\text{distance between two partitions } A \text{ and } B = \frac{1}{2}\sum_{i=1}^{N} |A_i - B_i|. \tag{5.1}$$

So defined, the distance between two shared cache partitions is equivalent to Manhattan distance normalized to make 1 the greatest possible distance; however, we choose this definition for its intuitive power rather than formalism.

Using this definition, the distance between the time-averaged shared cache partition in the baseline and in the DVFS experiments using the ILSB model is at most 5%. Therefore, core frequency scaling in our experiments does not significantly change the natural shared cache partition, making the shared cache act like private caches (of different sizes) and hence maintaining the assumption that the number of DRAM requests per instruction remains the same under frequency scaling.

5.4 Robust Mechanism

Despite this evidence that the DVFS performance predictor we designed for the private cache CMP works with shared cache CMP, we propose two improvements to address the two potential problems posed by the shared cache and thus make our DVFS performance predictor more robust.

The first improvement addresses the first problem posed by shared cache (Section 5.1): the fact that shared LLC access latencies no longer scale with core frequencies. To account for this fact, we redefine the demand DRAM request time per instruction T_{demand} to be the demand LLC or DRAM request time per instruction. We augment the T_{demand} measurement mechanism (described in Section 4.4.1 based on earlier Section 3.3.3.1) to account for the extra demand time due to LLC requests. Specifically, the mechanism adds the time retirement stalls on LLC accesses per instruction to T_{demand} measured as in the previous mechanism.[2]

[2]The resulting T_{demand} measurement mechanism would be especially useful in CMPs comprised of in-order cores which are not able to hide LLC access latencies as well as out-of-order cores can.

The second improvement addresses the second problem posed by shared cache: the potential change in the natural shared cache partition under frequency scaling. To deal with this problem, we propose using an existing shared cache partitioning mechanism such as Utility-Based Cache Partitioning (UCP) by Qureshi et al.[69]. An explicit shared cache partitioning mechanism, in addition to improving performance, keeps the shared cache partition constant and makes the shared cache act as private caches, improving performance predictability as well.

5.5 Results

We now present the experimental results for our robust DVFS performance predictor for the shared cache CMP. Note that in these results, the methodology is identical to that of experimental observations in Section 5.2 except that all of the experiments including the baseline employ Utility-Based Cache Partitioning (UCP) invoked every 10M baseline cycles. Therefore, these results are not comparable to the experimental observations in Section 5.2 because the baselines are different. For example, the new baseline with UCP performs 2.6% better on average than the old baseline in the medium cores, 1 DRAM channel configuration.

Figures 5.6 and 5.7 show the results of oracle and full performance studies with our robust DVFS performance predictor. The results of the oracle study show that the predictor delivers 80% of oracle performance, whilst the full study shows that the predictor delivers 3.9% average performance improvement on bandwidth-bound workloads in the medium cores, 1 DRAM channel configuration.

Figure 5.8 provides the summary of experimental results for the robust DVFS performance predictor for the shared cache CMP. Note that the predictor realizes most of the oracle gains and delivers an average 5% improvement on bandwidth-bound workloads in the big cores, 1 DRAM channel configuration. For completeness, Figure 5.9 presents the sensitivity study results.

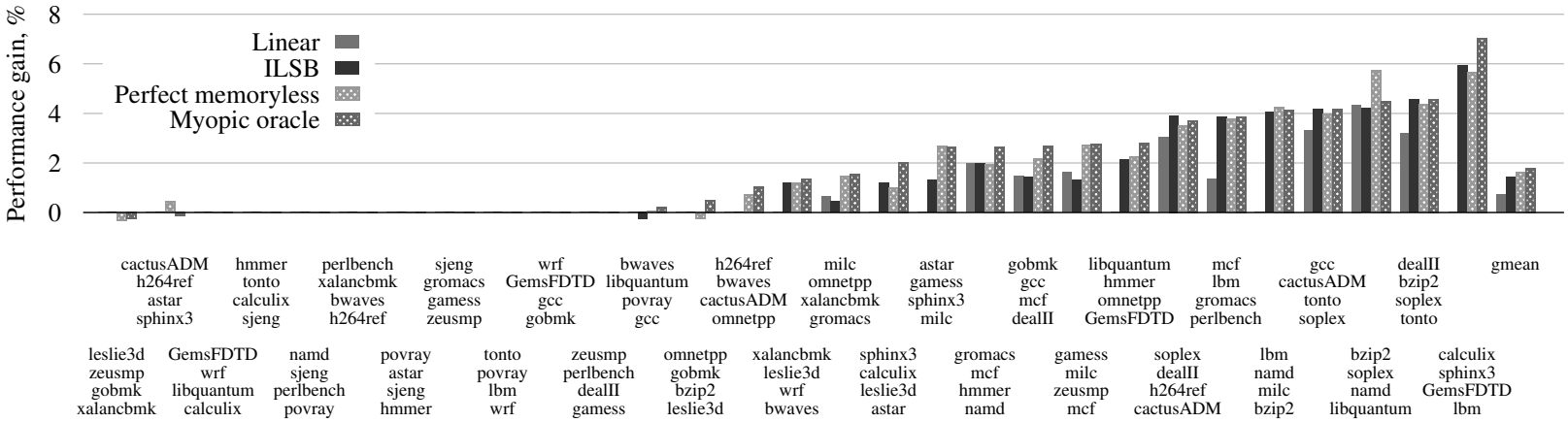


Figure 5.6: Oracle performance study of our robust DVFS performance predictor for the shared cache CMP (medium cores, 1 DRAM channel) with ALL workloads

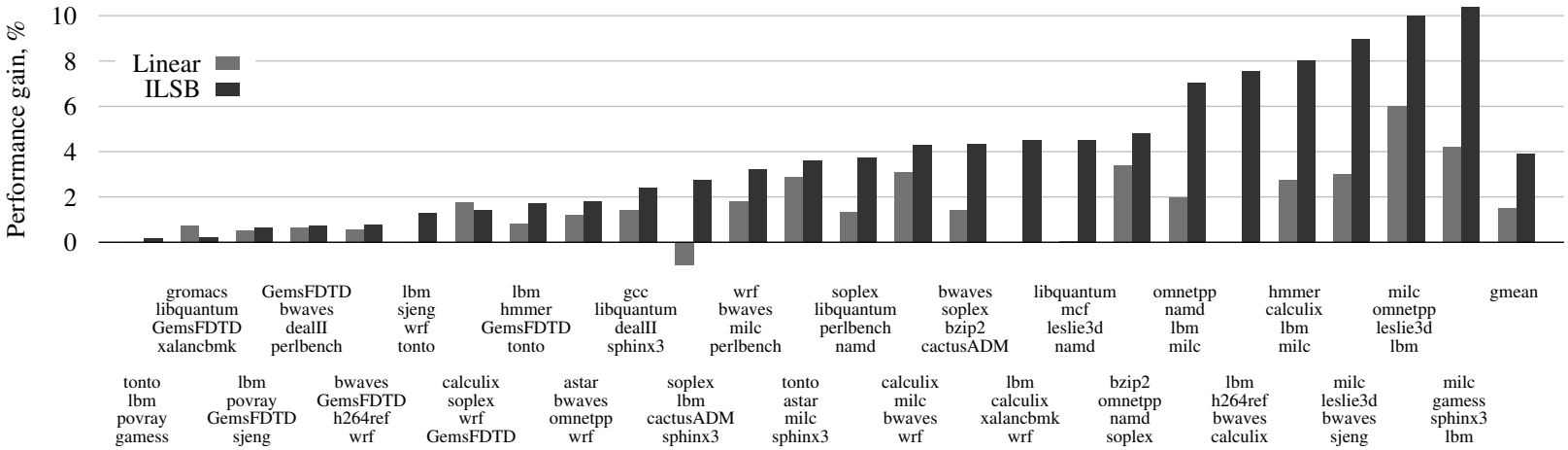


Figure 5.7: Full performance study of our robust DVFS performance predictor for the shared cache CMP (medium cores, 1 DRAM channel) with BW workloads

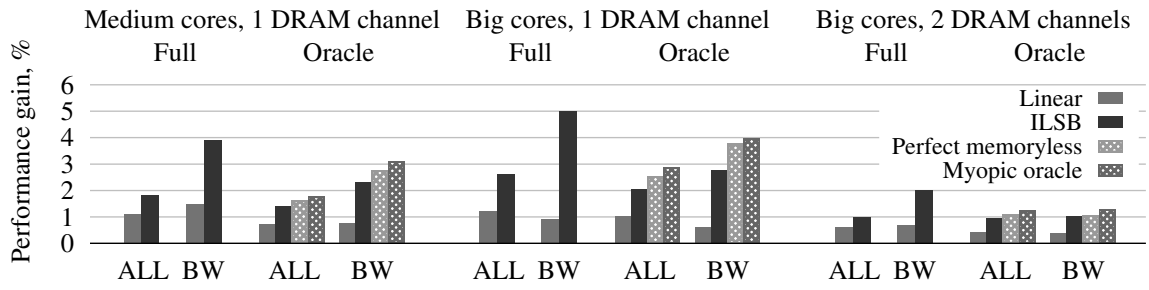


Figure 5.8: Summary of experimental results for our robust DVFS performance predictor for the shared cache CMP

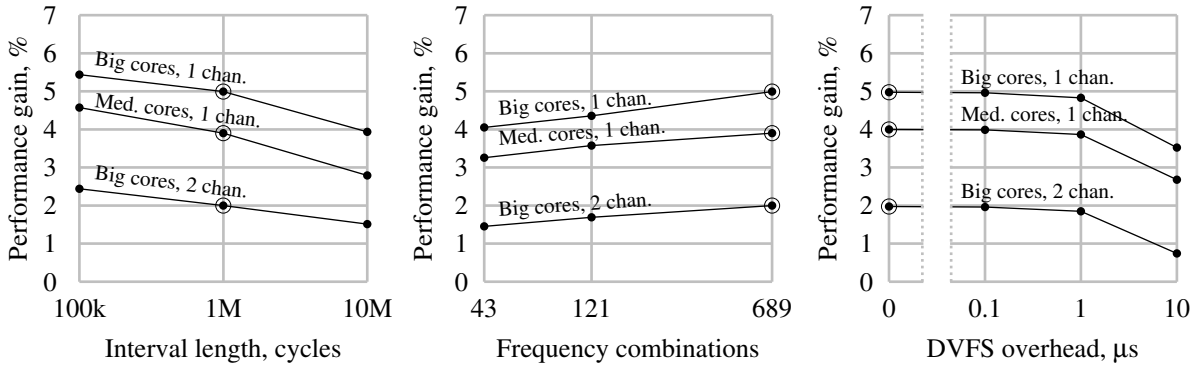
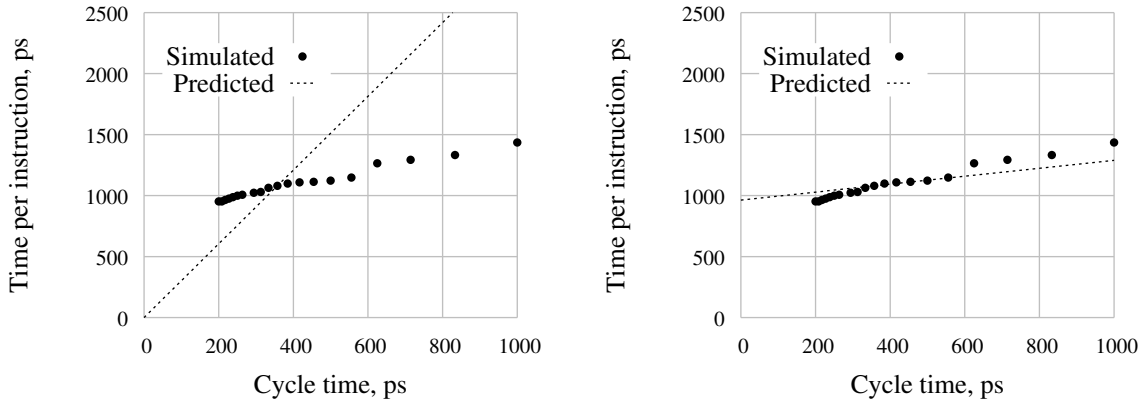


Figure 5.9: Sensitivity studies of our robust DVFS performance predictor for the shared cache CMP with BW workloads (circled points represent default settings)

5.6 Case Study

To ensure that our improved T_{demand} measurement mechanism which takes LLC stall time into account works as intended, we examine a case where this improvement actually matters—a case in which LLC access time has a major performance impact. Specifically, we a) design a microbenchmark whose performance, unlike SPEC2006 benchmarks, is dominated by LLC access time and b) evaluate whether our robust DVFS performance predictor still works well when confronted with this microbenchmark.

The microbenchmark, `dep_chain` (short for "dependency chain"), traverses a linked list contained in 256KB of memory. Thus, the linked list is too big to fit in the level 1 data cache but fits with plenty of room to spare into the shared LLC.



(a) Demand DRAM access stall time

(b) Demand DRAM and LLC access stall time

Figure 5.10: Simulated and predicted performance of `dep_chain` versus core cycle time for two different T_{demand} measurement mechanisms. Predicted performance is based on workload measurements at the baseline 2.8 GHz frequency.

Therefore, the performance of `dep_chain` is determined mostly by the latencies of serialized LLC accesses generated during the linked list traversal. In fact, when run on a uniprocessor, `dep_chain` causes the core to stall on demand LLC accesses 89% of the execution time (at the baseline 2.8GHz core frequency).

Figure 5.10 demonstrates that our improvement to T_{demand} measurement mechanism significantly improves prediction accuracy on `dep_chain`. Specifically, the figure shows simulated versus predicted performance of `dep_chain` on a uniprocessor (configured as described in Section 3.4) as a function of core cycle time. Note that the old T_{demand} measurement mechanism based on DRAM stall time incorrectly measures T_{demand} to be zero, resulting in poor prediction accuracy. In contrast, our improved T_{demand} measurement mechanism leads to significantly more accurate performance prediction.

An additional experiment proves that this improvement in prediction accuracy translates into tangible performance gains. We set up this experiment as follows. First, we introduce the compute-bound microbenchmark `accum` which continuously increments a counter. Second, we evaluate our DVFS performance predictor based on the ILSB model on the following four core workload: {`dep_chain`, `dep_chain`,

`dep_chain`, `accum`}. When this DVFS performance predictor uses the improved T_{demand} measurement mechanism, it leads the DVFS controller to 2.3% overall performance improvement (+14.3% on `accum` and −1.4% on each `dep_chain`). In contrast, when the DVFS performance predictor uses the old T_{demand} measurement mechanism, it merely maintains the baseline performance level by not switching core frequencies at all. Therefore, our improved T_{demand} measurement mechanism based on demand DRAM and LLC stall time does successfully address the problem of the shared LLC being outside of core frequency domains.

5.7 Conclusions

In this chapter we have shown experimentally that the DVFS predictor we designed for the private cache CMP in Chapter 4 applies to the shared cache CMP despite the potential problems caused by the shared cache. We explained these experimental results and proposed a new robust DVFS predictor designed specifically for the shared cache CMP.

Chapter 6

Related Work

Amy: How are we going to get there without a hovercar?
Fry: Wait. In my time we had a way of moving objects long distances without hovering.
Hermes: Impossible!
Fry: It was called... let me think... It was really famous. Ruth Gordon had one... The wheel.
Leela: Never heard of it.
Prof. Farnsworth: Show us this "the wheel."

Futurama, "Mother's Day"

We describe and compare with related work on several levels. We first place our work on DVFS performance predictors in the broader context of the prior work on adaptive processor control. We then compare our DVFS performance predictors with others in the literature. We also compare our limited bandwidth and independent latency shared bandwidth analytic models, which focus on memory aspects of performance, to previously proposed analytic models of memory system's impact on performance. Finally, we describe work related to scarce row hit prioritization.

6.1 Adaptive Processor Control

DVFS performance predictors explored in this dissertation are part of a broader research area of adaptive processor control. An *adaptive* processor is a processor that can dynamically adjust one or more architectural parameters to better fit the running workload characteristics. Clearly, DVFS-capable processors fall under this definition (chip and core frequencies being the adjustable architectural parameters).

We first present a taxonomy of adaptive processor control approaches (Section 6.1.1, summarized in Figure 6.1) and categorize previously proposed adaptive

Performance prediction	
mechanistic	based on mathematical models of microarchitecture details
statistical regression	based on black box mathematical models derived statistically from design-time training runs
machine learning	based on black box mathematical models "learned" at runtime
Other	
proxy metric prediction	based on a predictor of a proxy metric that correlates with performance (e.g. cache hit rate)
heuristics	based on a (typically empirically derived) set of rules
feedback-driven	based on trial and error or hill climbing algorithms

Figure 6.1: Taxonomy of adaptive processor control approaches.

processor controllers based on their place in the taxonomy and the adjustable parameters they control (Table 6.1). We then discuss the tradeoffs between the various approaches and explain why "mechanistic" performance prediction, our chosen approach, is preferable to the alternatives.

6.1.1 Taxonomy of Adaptive Processor Controllers

Adaptive processor controllers may be divided into those based on performance prediction and others. A performance predictor consists of two parts:

1. a mathematical model that expresses processor performance as a function of a) workload characteristics and b) adjustable parameter values, and
2. the hardware mechanisms that measure workload characteristics required by the mathematical model.

The adaptive processor uses a performance predictor as described in Section 6.1.2. As a quick overview, the adaptive processor controller spends some time measuring the necessary workload characteristics, plugs them into the model, and chooses the combination of adjustable parameter values that maximizes the performance function. This process is repeated to ensure the processor always operates in a near-optimal configuration.

The performance prediction approach has two main advantages over the others:

1. It enables the controller to switch to the predicted optimal configuration in one shot, resulting in quick reaction to workload phase changes.
2. It leaves open the possibility of composing multiple local controllers of separate adjustable parameters into one global controller, which can compare performance impacts of adjusting very different parameters. It's unclear how such composability could be achieved with other approaches which do not estimate the performance impact of parameter adjustment.

Further classification of performance prediction and other approaches follows. Table 6.1 lists citations for prior work according to their place in this classification and the adjustable parameters they control.

6.1.2 Performance Prediction

We classify performance predictors into a) "mechanistic," b) statistical regression based, and c) machine learning based.

Mechanistic[1] [3] performance predictors are designed from an understanding of relevant microarchitectural details. The designers of such predictors analyze the low level microarchitectural behavior and construct mathematical models that express at a high level the aggregate performance impact of this low level behavior.

In contrast, statistical regression and machine learning are *black box* approaches that require no understanding of microarchitectural details. Statistical regression models are based on design time experiments whereas machine learning approaches aim to "learn" the performance function at runtime.

Mechanistic performance prediction is superior to these black box approaches because it is more robust. Since mechanistic predictors are based on the physical reality of what happens, they should always provide accurate predictions. In contrast,

[1] "Of or relating to theories which explain phenomena in purely physical or deterministic terms" [75]. In computer architecture, this term was first used by Eyerman et al. [23] to characterize performance models.

Parameter type	Performance prediction			Other		
	Mechanistic	**Statistical regression**	**Machine learning**	**Proxy metric**	**Heuristic**	**Feedback-driven**
Chip/core frequency	22, 37, 71	10, 13–15, 49	18, 61	1	8	–
Core clock domain frequencies	–	–	–	–	–	11
Core structure sizes	–	19	–	43	–	–
Shared cache partition	66, 67	–	2	33, 62, 69	20, 34, 91	77, 78
DRAM bandwidth partition	53–55, 66, 67	–	2	–	20, 41, 42, 64, 65, 80	–
Prefetcher aggressiveness	–	–	51	24	21, 79	–
Symmetric core schedule	82	–	–	–	–	77, 81
Asymmetric core schedule	86	56	–	44, 72	46	–

Table 6.1: Citations for prior adaptive processor controllers categorized by approach and adjusted parameters.

such confidence is unwarranted in case of black box predictors. It is possible that an application not tested at design time may stress some part of the processor for which no input to the black box model has been provisioned by the designer. In this case, the black box approach may result in an inaccurate prediction.

From a research point view, mechanistic performance predictors also have the advantage of providing insights into why performance changes the way it does—insights that black box approaches cannot reveal.

The downside of mechanistic performance predictors is the nontrivial design effort they require. Specifically, designing a mechanistic performance predictor requires an understanding of the major factors driving processor performance—an understanding that takes time to develop.

6.1.3 Other Approaches

Other approaches found in the literature are a) proxy metric prediction, b) heuristics, and c) feedback-driven.

The proxy metric prediction approach resembles performance prediction, except the predicted metric is not a performance metric (such as instructions per cycle or energy per instruction) but a proxy metric that correlates with performance. A common example is a shared cache partitioning mechanism that optimizes for global hit rate [33, 62, 69].

Another approach is based on heuristics, ad hoc rules that guide parameter adjustment in accordance with the designer's intuitive understanding of the configuration space. Heuristics are particularly common in shared cache partitioning [20, 34, 91] and prefetcher throttling [21, 79].

The final subcategory covers all feedback-driven approaches such as trial and error and hill climbing. Feedback-driven controllers adjust processor parameters and measure the resulting performance in order to explore the configuration space so that the optimal configuration can be found. Such controllers may spend a lot of time in

suboptimal configurations and are slow to settle on the optimal configuration after a workload phase change. These problems get worse as the the size of the configuration space grows exponentially with the number of adjustable parameters.

6.2 DVFS Performance Prediction

Most prior work on DVFS performance prediction [10, 13–15, 18, 49, 61] addresses the problem above the microarchitectural level and does not explore hardware modification. Hence, this work can only use already existing hardware performance counters as inputs to their performance and power models. These counters were not designed to predict the performance impact of DVFS and thus do not work well for that purpose. Hence, these papers resort to statistical [10, 13–15, 49] and machine learning [18, 61] techniques.

In contrast, in this dissertation we tackle DVFS control at the microarchitecture level, designing new hardware counters with the explicit goal of measuring workload characteristics needed for accurate DVFS performance prediction. This approach was introduced by leading loads [22, 37, 71] and stall time [22, 37] proposals already discussed in Sections 3.1.2 and 3.1.3.

6.3 Analytic Models of Memory System Performance

Some prior work presents analytic models that, though not targeting frequency scaling, also focus on memory performance. An early example is the negative feedback model of Bucher and Calahan [4] which, like our ILSB model, has a feedback loop structure. Bucher and Calahan, however, do not consider multiple applications or bandwidth saturation caused by prefetching. Prior analytic models of CMP bandwidth partitioning [53–55, 66, 67] are also related to our ILSB model; however, they do not take into account demand request prioritization in DRAM scheduling. In contrast, our ILSB model does; in fact, as described in Section 4.3.1.2, demand pri-

oritization is what allows us to model latency-bound core performance independently of the other cores.

The Roofline performance model [90] is particularly relevant to our limited bandwidth model because it is also based on the basic insight that processor performance is driven by the slowest of multiple bottlenecks; in case of the Roofline model, these bottlenecks are compute bandwidth and memory bandwidth. There are, however, three important differences between the two models.

1. The two models serve different purposes and link different quantities. Specifically, the Roofline model is meant to guide software optimization and uses the aforementioned basic insight to express performance as a function of *operational intensity* (a workload characteristic equal to the ratio of compute operations per byte transmitted). In contrast, the linear bandwidth model is meant to guide DVFS at runtime and expresses performance (time per instruction) as a function of chip cycle time.

2. Due to this difference in purpose, the accuracy requirements for the two models are also different. Specifically, the Roofline model need only provide a rough upper bound on performance. In contrast, the linear bandwidth model needs to predict performance accurately enough to guide runtime DVFS.

3. While both models consider memory bandwidth as one of two performance bottlenecks, the other bottleneck differs among the two models. Specifically, the second bottleneck of the Roofline model is compute bandwidth, whereas the second bottleneck of the limited bandwidth model is a combination of compute bandwidth and demand DRAM request latency.

Most other prior analytic models of memory system performance [9, 17, 25, 83, 92] also serve a different purpose than our analytic models. Specifically, these models are designed for address design space exploration [9, 25, 83], predicting DRAM efficiency [92], and predicting the benefit of low power DRAM operation [17]. Unlike

our analytic models, these models do not take the latency-bound and bandwidth-bound modes of core operation into account; most [9, 17, 25, 92] assume that cores are always latency-bound while one [83] assumes that processing is always bandwidth-bound.

6.4 Prioritization in DRAM Scheduling

Finally, we compare to prior work related to scarce row hit prioritization (Section 4.2). Multiple prior works on DRAM scheduling [36, 41, 42, 48, 64, 65] identify priority inversion due to row hit prioritization as a performance problem. We divide these works into those that, like scarce row hit prioritization, deal with the problem at the level of short term scheduling and those that deal with it as part of a long term scheduling policy.

The FR-FCFS-Cap policy of Mutlu and Moscibroda [64], Prefetch-Aware DRAM Controller (PADC) of Lee et al. [48] and the Minimalist Open-Page policy of Kaseridis et al. [36] are the short term scheduling mechanisms. The FR-FCFS-Cap policy is the "row hit, oldest" priority order with a cap on the number of consecutive row hits prioritized. Unlike scarce row hit prioritization, this policy may delay scheduling a higher priority row conflict until a number of consecutive low priority row hits to the same bank is satisfied even if the number of outstanding row hits to other banks is enough to maintain high bus utilization. The Prefetch-Aware DRAM Controller (PADC) dynamically chooses between two priority orders, "row hit, oldest" and "demand, row hit, oldest." However, PADC does not consider the "row hit, demand, oldest" priority order (our baseline) which we find to perform better. The Minimalist Open-Page policy changes address-to-bank mapping to increase bank level parallelism of streaming workloads and reduce the length of bursts of consecutive row hits. In general, any short term scheduling technique that tackles priority inversion, such as these techniques and scarce row hit prioritization, helps DVFS performance prediction by making chip multiprocessor performance more predictable as described in Section 4.2.4.

Long term DRAM scheduling mechanisms [41, 42, 64, 65] pose a problem for DVFS performance prediction. Unlike short term scheduling mechanisms, the long term prioritization of some cores over others results in unequal DRAM service time of bandwidth-bound cores, violating the equal DRAM request service time approximation (Section 4.3.4) of our ILSB model. We leave the problem of DVFS performance prediction in the presence of these long term DRAM scheduling mechanisms to future work.

Chapter 7

Conclusions

> In the instant that you love someone,
> In the second that the hammer hits,
> Reality runs up your spine,
> And the pieces finally fit.
>
> Elton John
> *The One*

Looking back at the major contributions of this dissertation—the mathematical models of performance under frequency scaling, the mechanisms to measure parameters of these models, and the evaluation results—we reach three conclusions:

1. Performance predictors must be designed for and evaluated with realistic memory systems.

2. Finite off-chip bandwidth must be considered in performance predictor design.

3. Accurate DVFS performance prediction at runtime is feasible.

We elaborate on these conclusions below.

7.1 Importance of Realistic Memory Systems

This dissertation shows that performance predictors in general, even those not related to DVFS, must be designed and evaluated with realistic memory systems in mind. Specifically, we have seen in Chapter 3 that prior DVFS performance predictors (leading loads and stall time) fail to accurately predict performance under frequency scaling because they ignore major qualitative characteristics of real memory systems

(such as variable access latency in the case of leading loads and stream prefetching in the case of both). These problems, however, are not specific to DVFS. In fact, frequency scaling is just one of many mechanisms the processor can use to change core performance. Whatever the mechanism (e.g., switching between in-order and out-of-order [38] execution modes or adjusting prefetcher aggressiveness [79]), predicting its performance impact accurately requires considering the major characteristics of realistic memory systems.

As we recommend taking realistic memory systems into account, we emphasize their *qualitative* rather than *quantitative* characteristics. Specifically, this dissertation considers memory systems with the following qualitative characteristics: variable access latency, stream prefetching, and demand access prioritization. We fully expect that our DVFS performance predictors will work with real (not simulated) memory systems with these qualitative characteristics even if their quantitative characteristics (such as exact DRAM timings or number of DRAM channels) differ from the ones we used in simulation experiments.

7.2 Performance Impact of Finite Off-Chip Bandwidth

This dissertation also shows that finite off-chip bandwidth is a major performance factor that must be considered in performance predictor design. Specifically, all DVFS performance predictors we propose—targeting the uniprocessor (Chapter 3), the private cache chip multiprocessor (Chapter 4), and the shared cache chip multiprocessor (Chapter 5)—deliver most performance or energy efficiency gains on bandwidth-bound workloads. The DVFS performance predictors that don't take bandwidth into account—those based on the linear DVFS performance model—fail to realize these gains. Again, the causes of these effects are not specific to DVFS and apply to any kind of performance predictor.

Note that our models of performance under frequency scaling (limited bandwidth and the independent latency shared bandwidth) can be readily adapted to help performance predictors unrelated to DVFS take finite bandwidth into account.

In fact, the parts of these models that deal with bandwidth apply directly to those applications of performance prediction where the number of DRAM requests per instruction (for each core) remains the same at any adjustable parameter setting.

7.3 Feasibility of Accurate DVFS Performance Prediction

Finally and most importantly, this dissertation proves the feasibility of accurate DVFS performance prediction using mechanistic models of performance under frequency scaling. Specifically, the DVFS performance predictors we designed in this dissertation are accurate enough to help the DVFS controller realize, depending on the processor configuration, 72–85% of average oracle gains in performance or energy efficiency (81–96% of average perfect memoryless gains) across SPEC 2006 benchmarks or their randomly chosen combinations. We therefore conclude that DVFS performance predictors based on mechanistic performance models can in fact be accurate enough to realize most of the benefit of an oracle predictor—the very thesis we set out to prove.

Bibliography

[1] Abhishek Bhattacharjee and Margaret Martonosi. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *Proc. 36th Int. Symp. Comput. Arch. (ISCA 2009)*, pages 290–301, June 2009.

[2] Ramazan Bitirgen, Engin Ipek, and Jose F. Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *Proc. 41st ACM/IEEE Int. Symp. Microarch. (MICRO-41)*, pages 318–329, November 2008.

[3] George E. P. Box and Norman R. Draper. *Empirical Model-Building and Response Surfaces*. John Wiley & Sons, 1987. Section 1.4.

[4] Ingrid Y. Bucher and Donald A. Calahan. Models of access delays in multiprocessor memories. *IEEE Trans. Parallel Distrib. Syst. (TPDS)*, 3(3):270–280, 1992.

[5] Thomas D. Burd and Robert W. Brodersen. Energy efficient CMOS microprocessor design. In *Proc. 28th Hawaii Int. Conf. Syst. Sci. (HICCS-28)*, volume 1, pages 288–297, January 1995.

[6] Michael Butler, Leslie Barnes, Debjit Das Sarma, and Bob Gelinas. Bulldozer: An approach to multithreaded compute performance. *IEEE Micro*, 31(2):6–15, March 2011.

[7] J. Adam Butts and Gurindar S. Sohi. A static power model for architects. In *Proc. 33rd ACM/IEEE Int. Symp. Microarch. (MICRO-33)*, pages 191–201, December 2000.

[8] Qiong Cai, José González, Ryan Rakvic, Grigorios Magklis, Pedro Chaparro, and Antonio González. Meeting points: Using thread criticality to adapt multicore hardware to parallel regions. In *Proc. 17th Int. Conf. Parallel Arch. and Compilation Techniques (PACT'08)*, pages 240–249, October 2008.

[9] Hyojin Choi, Jongbok Lee, and Wonyong Sung. Memory access pattern-aware DRAM performance model for multi-core systems. In *Proc. 2011 IEEE Int. Symp. Perf. Anal. of Syst. and Soft. (ISPASS-2011)*, pages 66–75, April 2011.

[10] Kihwan Choi, Ramakrishna Soma, and Massoud Pedram. Fine-grained dynamic voltage and frequency scaling for precise energy and performance trade-off based on the ratio of off-chip access to on-chip computation times. In *Proc. Conf. Design, Automation, and Test in Europe (DATE 2004)*, pages 4–9, February 2004.

[11] P. Choudhary and D. Marculescu. Power management of voltage/frequency island-based systems using hardware-based methods. *IEEE Trans. Very Large Scale Integration (VLSI) Syst.*, 17(3):427–438, March 2009.

[12] Jamison Collins, Suleyman Sair, Brad Calder, and Dean M. Tullsen. Pointer cache assisted prefetching. In *Proc. 35th ACM/IEEE Int. Symp. Microarch. (MICRO-35)*, pages 62–73, June 2002.

[13] Gilberto Contreras and Margaret Martonosi. Power prediction for Intel XScale processors using performance monitoring unit events. In *Proc. 2005 Int. Symp. Low Power Electron. and Design (ISLPED'05)*, pages 221–226, August 2005.

[14] Matthew Curtis-Maury, James Dzierwa, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *Proc. 20th Int. Conf. Supercomputing (ICS'06)*, pages 157–166, June 2006.

[15] Matthew Curtis-Maury, Ankur Shah, Filip Blagojevic, Dimitrios S. Nikolopoulos, Bronis R. de Supinski, and Martin Schulz. Prediction models for multi-dimensional power-performance optimization on many cores. In *Proc. 17th Int. Conf. Parallel Arch. and Compilation Techniques (PACT'08)*, pages 250–259, October 2008.

[16] Howard David, Chris Fallin, Eugene Gorbatov, Ulf R. Hanebutte, and Onur Mutlu. Memory power management via dynamic voltage/frequency scaling. In *Proc. 8th ACM Int. Conf. Autonomic Computing (ICAC 2011)*, pages 31–40, June 2011.

[17] Qingyuan Deng, David Meisner, Luiz Ramos, Thomas F. Wenisch, and Ricardo Bianchini. MemScale: Active low-power modes for main memory. In *Proc. 16th Int. Conf. Arch. Support for Programming Languages and Operating Syst. (ASPLOS XVI)*, pages 225–238, March 2011.

[18] Gaurav Dhiman and Tajana Simunic Rosing. Dynamic voltage frequency scaling for multi-tasking systems using online learning. In *Proc. 2007 Int. Symp. Low Power Electron. and Design (ISLPED'07)*, pages 207–212, August 2007.

[19] Christophe Dubach, Timothy M. Jones, Edwin V. Bonilla, and Michael F. P. O'Boyle. A predictive model for dynamic microarchitectural adaptivity control. In *Proc. 43rd ACM/IEEE Int. Symp. Microarch. (MICRO-43)*, pages 485–496, December 2010.

[20] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. Fairness via source throttling: A configurable and high-performance fairness substrate for multicore memory systems. *ACM Trans. Comput. Syst. (TOCS)*, 30(2):7:1–7:35, April 2012.

[21] Eiman Ebrahimi, Onur Mutlu, Chang Joo Lee, and Yale N. Patt. Coordinated control of multiple prefetchers in multi-core systems. In *Proc. 42nd ACM/IEEE Int. Symp. Microarch. (MICRO-42)*, pages 316–326, December 2009.

[22] Stijn Eyerman and Lieven Eeckhout. A counter architecture for online DVFS profitability estimation. *IEEE Trans. Comput. (TOC)*, 59:1576–1583, November 2010.

[23] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. A mechanistic performance model for superscalar out-of-order processors. *ACM Trans. Comput. Syst. (TOCS)*, 27:3:1–3:37, May 2009.

[24] Marius Grannaes and Lasse Natvig. Dynamic parameter tuning for hardware prefetching using shadow tagging. In *Proc. 2nd Workshop Chip Multiprocessor Memory Syst. and Interconnects (CMP-MSI)*, June 2008.

[25] Nagendra Gulur, Mahesh Mehendale, Raman Manikantan, and Ramaswamy Govindarajan. ANATOMY: An analytical model of memory system performance. In *Proc. 2014 ACM Int. Conf. on Measurement and Modeling of Comput. Syst. (SIGMETRICS '14)*, pages 505–517, June 2014.

[26] Mor Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*, chapter 13, pages 242–244. Cambridge University Press, 2013.

[27] Mark Horowitz, Thomas Indermaur, and Ricardo Gonzalez. Low-power digital design. In *IEEE Symp. Low Power Electron. (ISLPE'94) Digest of Tech. Papers*, pages 8–11, October 1994.

[28] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual Version 029*, March 2014.

[29] Canturk Isci, Gilberto Contreras, and Margaret Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *Proc. 39th ACM/IEEE Int. Symp. Microarch. (MICRO-39)*, pages 359–370, December 2006.

[30] Canturk. Isci and Margaret Martonosi. Phase characterization for power: Evaluating control-flow-based and event-counter-based techniques. In *Proc. 12th IEEE Int. Symp. High Perf. Comput. Arch. (HPCA-12)*, pages 121–132, February 2006.

[31] Bruce Jacob, Spencer W. Ng, and David T. Wang. *Memory Systems: Cache, DRAM, Disk*, chapter 7, pages 315–351. Elsevier Science, 2010.

[32] Akanksha Jain and Calvin Lin. Linearizing irregular memory accesses for improved correlated prefetching. In *Proc. 46th ACM/IEEE Int. Symp. Microarch. (MICRO-46)*, pages 247–259, December 2013.

[33] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely, Jr., and Joel Emer. Adaptive insertion policies for managing shared caches. In *Proc. 17th Int. Conf. Parallel Arch. and Compilation Techniques (PACT'08)*, pages 208–219, October 2008.

[34] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High performance cache replacement using re-reference interval prediction (RRIP). *SIGARCH Comput. Archit. News*, 38(3):60–71, June 2010.

[35] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *Proc. 24th Int. Symp. Comput. Arch. (ISCA 1997)*, pages 252–263, June 1997.

[36] Dimitris Kaseridis, Jeffrey Stuecheli, and Lizy Kurian John. Minimalist open-page: A DRAM page-mode scheduling policy for the many-core era. In *Proc. 44th ACM/IEEE Int. Symp. Microarch. (MICRO-44)*, pages 24–35, December 2011.

[37] Georgios Keramidas, Vasileios Spiliopoulos, and Stefanos Kaxiras. Interval-based models for run-time DVFS orchestration in superscalar processors. In *Proc. ACM Int. Conf. Computing Frontiers (CF'10)*, pages 287–296, May 2010.

[38] Khubaib, M. Aater Suleman, Milad Hashemi, Chris Wilkerson, and Yale N. Patt. MorphCore: An energy-efficient microarchitecture for high performance

ILP and high throughput TLP. In *Proc. 45th ACM/IEEE Int. Symp. Microarch. (MICRO-45)*, pages 305–316, December 2012.

[39] Wonyoung Kim, David Brooks, and Gu-Yeon Wei. A fully-integrated 3-level DC-DC converter for nanosecond-scale DVFS. *IEEE J. Solid-State Circuits (JSSC)*, 47(1):206–219, January 2012.

[40] Wonyoung Kim, Meeta S. Gupta, Gu-Yeon Wei, and David Brooks. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *Proc. 14th IEEE Int. Symp. High Perf. Comput. Arch. (HPCA-14)*, pages 123–134, February 2008.

[41] Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *Proc. 16th IEEE Int. Symp. High Perf. Comput. Arch. (HPCA-16)*, February 2010.

[42] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Proc. 43rd ACM/IEEE Int. Symp. Microarch. (MICRO-43)*, pages 65–76, December 2010.

[43] Vasileios Kontorinis, Amirali Shayan, Dean M. Tullsen, and Rakesh Kumar. Reducing peak power with a table-driven adaptive processor core. In *Proc. 42nd ACM/IEEE Int. Symp. Microarch. (MICRO-42)*, pages 189–200, December 2009.

[44] David Koufaty, Dheeraj Reddy, and Scott Hahn. Bias scheduling in heterogeneous multi-core architectures. In *Proc. Euro. Conf. Comp. Syst. (EuroSys 2010)*, pages 125–138, April 2010.

[45] Rajesh Kumar and Glenn Hinton. A family of 45nm IA processors. In *2009 IEEE Int. Solid-State Circuits Conf. (ISSCC 2009) Digest Tech. Papers*, pages 58–59, February 2009.

[46] Nagesh B. Lakshminarayana, Jaekyu Lee, and Hyesoon Kim. Age based scheduling for asymmetric multiprocessors. In *Proc. Conf. High Perf. Computing Networking, Storage, and Anal. (SC09)*, November 2009.

[47] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. IBM POWER6 microarchitecture. *IBM J. of Research and Develop.*, 51(6):639–662, November 2007.

[48] Chang Joo Lee, Onur Mutlu, Veynu Narasiman, and Yale N. Patt. Prefetch-aware DRAM controllers. In *Proc. 41st ACM/IEEE Int. Symp. Microarch. (MICRO-41)*, pages 200–209, November 2008.

[49] Sang Jeong Lee, Hae-Kag Lee, and Pen-Chung Yew. Runtime performance projection model for dynamic power management. In *Advances in Comput. Syst. Arch. 12th Asia-Pacific Conf. (ACSAC 2007) Proc.*, volume 4697 of *Lecture Notes in Computer Science*, pages 186–197. Springer-Verlag, August 2007.

[50] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proc. 42nd ACM/IEEE Int. Symp. Microarch. (MICRO-42)*, pages 469–480, December 2009.

[51] Shih-wei Liao, Tzu-Han Hung, Donald Nguyen, Chinyen Chou, Chiaheng Tu, and Hucheng Zhou. Machine learning-based prefetch optimization for data center applications. In *Proc. Conf. High Perf. Computing Networking, Storage, and Anal. (SC09)*, November 2009.

[52] John D. C. Little. A proof for the queuing formula: $L = \lambda W$. *Operations Research*, 9(3):383–387, June 1961.

[53] Fang Liu. *Analytically Modeling the Memory Hierarchy Performance of Modern Processor Systems*. PhD thesis, North Carolina State University, 2011.

[54] Fang Liu, Xiaowei Jiang, and Yan Solihin. Understanding how off-chip memory bandwidth partitioning in chip multiprocessors affects system performance. In *Proc. 16th IEEE Int. Symp. High Perf. Comput. Arch. (HPCA-16)*, February 2010.

[55] Fang Liu and Yan Solihin. Studying the impact of hardware prefetching and bandwidth partitioning in chip-multiprocessors. In *Proc. ACM SIGMETRICS Joint Int. Conf. on Measurement and Modeling of Comput. Syst. (SIGMETRICS'11)*, pages 37–48, June 2011.

[56] Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Faissal M. Sleiman, Ronald Dreslinski, Thomas F. Wenisch, and Scott Mahlke. Composite cores: Pushing heterogeneity into a core. In *Proc. 45th ACM/IEEE Int. Symp. Microarch. (MICRO-45)*, pages 317–328, December 2012.

[57] Steven M. Martin, Krisztian Flautner, Trevor Mudge, and David Blaauw. Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads. In *Proc. 2002 IEEE/ACM Int. Conf. Comp.-Aided Design (ICCAD'02)*, pages 721–725, November 2002.

[58] Micron Technology, Inc. *MT41J512M4 DDR3 SDRAM Datasheet Rev. K*, April 2010. `http://download.micron.com/pdf/datasheets/dram/ddr3/2Gb_DDR3_SDRAM.pdf`.

[59] Rustam Miftakhutdinov. Pincpt: A tool for checkpointing architectural state. http://pincpt.sourceforge.net.

[60] Rustam Miftakhutdinov, Eiman Ebrahimi, and Yale N. Patt. Predicting performance impact of DVFS for realistic memory systems. In *Proc. 45th ACM/IEEE Int. Symp. Microarch. (MICRO-45)*, pages 155–165, 2012.

[61] Michael Moeng and Rami Melhem. Applying statistical machine learning to multicore voltage and frequency scaling. In *Proc. ACM Int. Conf. Computing Frontiers (CF'10)*, pages 277–286, May 2010.

[62] Miquel Moreto, Francisco J. Cazorla, Alex Ramirez, and Mateo Valero. MLP-aware dynamic cache partitioning. In Per Stenström, Michel Dubois, Manolis Katevenis, Rajiv Gupta, and Theo Ungerer, editors, *High Performance Embedded Architectures and Compilers*, volume 4917 of *Lecture Notes in Computer Science*, pages 337–352. Springer Berlin Heidelberg, 2008.

[63] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. CACTI 6.0: A tool to model large caches. Technical Report HPL-2009-85, HP Laboratories, April 2009.

[64] Onur Mutlu and Thomas Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *Proc. 40th ACM/IEEE Int. Symp. Microarch. (MICRO-40)*, pages 146–160, December 2007.

[65] Onur Mutlu and Thomas Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *Proc. 35th Int. Symp. Comput. Arch. (ISCA 2008)*, pages 63–74, June 2008.

[66] Tae Cheol Oh. *Analytical Models for Chip Multiprocessor Memory Hierarchy Design and Management*. PhD thesis, University of Pittsburgh, 2010.

[67] Tae Cheol Oh, Kiyeon Lee, and Sangyeun Cho. An analytical performance model for co-management of last-level cache and bandwidth sharing. In *Proc. 19th IEEE Int. Symp. on Modeling, Anal., and Simulation of Comput. and Telecommun. Syst. (MASCOTS 2011)*, pages 150–158, July 2011.

[68] Harish Patil, Robert S. Cohn, Mark Charney, Rajiv Kapoor, Andrew Sun, and Anand Karunanidhi. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *Proc. 37th ACM/IEEE Int. Symp. Microarch. (MICRO-37)*, pages 81–92, December 2004.

[69] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches.

In *Proc. 39th ACM/IEEE Int. Symp. Microarch. (MICRO-39)*, pages 423–432, December 2006.

[70] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter R. Mattson, and John D. Owens. Memory access scheduling. In *Proc. 27th Int. Symp. Comput. Arch. (ISCA 2000)*, pages 128–138, June 2000.

[71] Barry Rountree, David K. Lowenthal, Martin Schulz, and Bronis R. de Supinski. Practical performance prediction under dynamic voltage frequency scaling. In *2011 Int. Green Computing Conf. and Workshops (IGCC'11)*, July 2011.

[72] Juan Carlos Saez, Manuel Prieto, Alexandra Fedorova, and Sergey Blagodurov. A comprehensive scheduler for asymmetric multicore systems. In *Proc. Euro. Conf. Comp. Syst. (EuroSys 2010)*, pages 139–152, April 2010.

[73] Yiannakis Sazeides, Rakesh Kumar, Dean M. Tullsen, and Theofanis Constantinou. The danger of interval-based power efficiency metrics: When worst is best. *Comp. Arch. Lett. (CAL)*, 4(1), January 2005.

[74] Timothy Sherwood, Suleyman Sair, and Brad Calder. Phase tracking and prediction. In *Proc. 30th Int. Symp. Comput. Arch. (ISCA 2003)*, pages 336–347, San Diego, California, June 2003.

[75] John Simpson, editor. *Oxford English Dictionary*. Oxford University Press, 3rd edition, June 2001. Entry: "mechanistic, *adj.*".

[76] Allan Snavely and Dean M. Tullsen. Symbiotic job scheduling for a simultaneous multithreaded processor. In *Proc. 9th Int. Conf. Arch. Support for Programming Languages and Operating Syst. (ASPLOS-IX)*, pages 234–244, November 2000.

[77] Shekhar Srikantaiah, Reetuparna Das, Asit Mishra, Chita Das, and Mahmut Kandemir. A case for integrated processor-cache partitioning in chip multiprocessors. In *Proc. Conf. High Perf. Computing Networking, Storage, and Anal. (SC09)*, November 2009.

[78] Shekhar Srikantaiah, Mahmut Kandemir, and Qian Wang. SHARP control: controlled shared cache management in chip multiprocessors. In *Proc. 42nd ACM/IEEE Int. Symp. Microarch. (MICRO-42)*, pages 517–528, 2009.

[79] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *Proc. 13th IEEE Int. Symp. High Perf. Comput. Arch. (HPCA-10)*, pages 63–74, February 2007.

[80] Lavanya Subramanian, Vivek Seshadri, Yoongu Kim, Ben Jaiyen, and Onur Mutlu. MISE: Providing performance predictability and improving fairness in shared main memory systems. In *Proc. 19th IEEE Int. Symp. High Perf. Comput. Arch. (HPCA-19)*, pages 639–650, February 2013.

[81] M. Aater Suleman, Moinuddin K. Qureshi, Khubaib, and Yale N. Patt. Feedback-directed pipeline parallelism. In *Proc. 19th Int. Conf. Parallel Arch. and Compilation Techniques (PACT'10)*, pages 147–156, September 2010.

[82] M. Aater Suleman, Moinuddin K. Qureshi, and Yale N. Patt. Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on CMPs. In *Proc. 13th Int. Conf. Arch. Support for Programming Languages and Operating Syst. (ASPLOS-XIII)*, pages 277–286, March 2008.

[83] Guangyu Sun, C. Hughes, Changkyu Kim, Jishen Zhao, Cong Xu, Yuan Xie, and Yen-Kuang Chen. Moguls: A model to explore the memory hierarchy for bandwidth improvements. In *Proc. 38th Int. Symp. Comput. Arch. (ISCA 2011)*, pages 377–388, June 2011.

[84] Joel Tendler, J. Steve Dodson, J. S. Fields Jr., Le Hung, and Balaram Sinharoy. POWER4 system microarchitecture. *IBM J. of Research and Develop.*, 46:5–25, October 2001.

[85] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2Sim: A simulation framework for CPU-GPU computing. In *Proc. 21st*

Int. Conf. Parallel Arch. and Compilation Techniques (PACT'12), pages 335–344, September 2012.

[86] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. Scheduling heterogeneous multi-cores through performance impact estimation (PIE). In *Proc. 39th Int. Symp. Comput. Arch. (ISCA 2012)*, pages 213–224, June 2012.

[87] Frederik Vandeputte, Lieven Eeckhout, and Koen De Bosschere. A detailed study on phase predictors. In *Proc. 11th Int. Euro-Par Conf. Parallel Process. (Euro-Par 2005)*, pages 571–581, August 2005.

[88] Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Mechanisms for store-wait-free multiprocessors. In *Proc. 34th Int. Symp. Comput. Arch. (ISCA 2007)*, pages 266–277, June 2007.

[89] Thomas F. Wenisch, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Making address-correlated prefetching practical. *IEEE Micro*, 30(1):50–59, January 2010.

[90] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Comm. ACM (CACM)*, 52(4):65–76, 2009.

[91] Yuejian Xie and Gabriel H. Loh. PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proc. 36th Int. Symp. Comput. Arch. (ISCA 2009)*, pages 174–183, June 2009.

[92] George L. Yuan and Tor M. Aamodt. A hybrid analytical DRAM performance model. In *Proc. 5th Workshop on Modeling, Benchmarking and Simulation (MoBS 2009)*, June 2009.