Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Peter Salvesen

Predicting Interference-Free Performance with Linear Model Trees

Master's thesis in Computer Science
Supervisor: Magnus Jahre
June 2020

NTNU
Norwegian University of Science and Technology

Peter Salvesen

Predicting Interference-Free Performance with Linear Model Trees

Master's thesis in Computer Science
Supervisor: Magnus Jahre
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Project Description

Accurate performance accounting is a key component to effectively manage multi-core resources. It aims to estimate interference-free application performance in multi-core memory systems. Recently, GDP (Graph-based Dynamic Performance) was introduced as a new method of estimating interference-free performance at runtime with high accuracy. A disadvantage of GDP is that it is quite complex which makes it unattractive for use in commercial multi-core implementations. In this master thesis, the student should propose and evaluate simplifications to GDP that retain sufficient accuracy while reducing implementation complexity. Particular emphasis should be placed on explaining why the proposed techniques perform well.

Abstract

Modern multicore processors improve hardware utilization and throughput with resource sharing between cores. However, resource sharing also leads to an unpredictable application performance because of inter-application interference in the shared resources. Quantifying the performance without this interference, called the interference-free performance is a key component to effectively manage shared system resources. Several performance accounting systems have been proposed, predicting interference-free performance. They are able to predict quite accurately, but have significant storage overhead making them less attractive to be implemented by commercial vendors.

This master thesis proposes a novel way of predicting interference-free performance in multicore processors, with a lower storage overhead. Instead of modeling some key performance aspect of the shared memory system to predict performance, the behavior is learned by a regression model. Specifically, *linear model trees* (LMTs) are used combining decision trees and linear regression. The LMTs classify observations in the memory system with similar behavior. Succeedingly, they exploit linearity within each classification. The LMT can either predict the interference-free performance directly, or provide low-cost predictions to be used in other performance accounting models. The main focus of the work is retaining prediction accuracy compared to state-of-the-art while reducing the storage overhead of predictions.

The LMTs can be configured numerous ways, however two specific configurations point out how they can reduce storage overhead for interference-free performance prediction. Both configurations predict IPC directly in a LMT with 10 leaf nodes. The only difference between them is that a set of costly input features are removed to reduce storage overhead in one of the configurations. First configuration improves the prediction error compared to state-of-the-art by 1%, while reducing the storage overhead by 24%. Secondly, the reduced feature set configuration increases the prediction error compared to state-of-the-art by 17%. However, the storage overhead is reduced by 85%.

Sammendrag

Moderne flerkjerne-prosessorer bruker ofte delte hardwarekomponenter for å forbedre utnyttelsen av hver enkelt komponent og en bedre ytelse fra systemet i sin helhet. Men, slike delte komponenter fører også til at en applikasjon kan tilføre interferens for en annen applikasjon. Slik blir ytelsen til hver enkelt applikasjon uforutsigbar fordi den avhenger av hvilke applikasjoner som kjører samtidig. Å beregne hva ytelsen per applikasjon ville ha vært uten denne interferensen er en svært nyttig metrikk med tanke på å tildele delte systemressurser effektivt. En rekke systemer for å predikere en slik interferensfri ytelse har blitt foreslått. De beste av dem klarer å predikere interferensfri ytelse forholdsvis nøyaktig, men har en kompleks modellering som gjør at hardware-kostnaden ved å implementere systemene øker. Dette gjør systemene mindre attraktive for kommersielle prosessorprodusenter.

Denne masteroppgaven foreslår en ny måte å predikere interferensfri ytelse på, med fokus på å redusere hvor mye lagringskapasitet som kreves for å implementere prediksjonssystemet. Istedet for å modellere enkeltverdier som skal representere nøkkelkomponenter i en ytelsesmodell for delte minneressurser, benyttes læring av regresjonsmodeller for å predikere den resulterende ytelsen. Mer konkret: lineære trær benyttes som regresjonsmodell. Et linærtre er et beslutningstre som inneholder lineær regresjon i hver løvnode. De klassifiserer observasjoner med lik minneoppførsel, og drar nytte av linearitet mellom observasjonene. Lineærtrærne benyttes enten til å predikere interferensfri ytelse direkte eller til å estimere nøkkelkomponenter i en ytelsesmodell. Hovedfokuset i denne oppgaven har vært å bevare så mye av nøyaktigheten i prediksjonene som mulig, mens lagringen som kreves for å implementere prediksjonssystemet reduseres.

Lineære trær kan konfigureres en rekke måter. For å vise hvilket kraftfullt verktøy lineære trær kan være, presenteres resultatene for to ulike lineærtre-konfigurasjoner. Begge konfigurasjonene benytter 10 løvnoder i treet, og predikerer antall instruksjoner per sykel. Forskjellen mellom konfigurasjonene er at ene ikke bruker en kostbar input-verdi som krever mye lagringskapasitet. Det første oppsettet reduserte feilen i prediksjonene med 1% sammenlignet med "state-of-the-art", mens lagringskapasiteten som kreves for systemet reduseres med 24%. Den andre konfigurasjonen med et redusert input-sett har en økt feil sammenlignet med "state-of-the-art" med 17%, men reduserer lagringskapasiteten som kreves med hele 85%.

Acknowledgements

First of all, I want to thank Associate Professor Magnus Jahre for the invaluable feedback and discussions contributing to this thesis. Steering me in the the right direction was critical, focusing on the "correct" aspects which later lead to improvements of the previously proposed work. The contributions from this thesis would clearly have been much more limited without your in-depth expertise of the state-of-the-art.

Secondly, both friends and family should be thanked for providing support and motivation throughout the work with this thesis. Countless technical discussions and ping-pong games have helped keeping me on track with the work progress. Although many have contributed with interesting discussions and academic expertise, I want to mention one in particular. My dear uncle, Associate Professor Øyvind Salvesen, has patiently provided feedback and input through his knowledge within statistics.

Lastly, I want to show gratitude to UNINETT Sigma2, the National Infrastructure for High Performance Computing and Data Storage in Norway, for providing extensive computing resources for the simulations done in this project.

Contents

Figures

Tables

Chapter 1

Introduction

Modern Chip Multi-Processors (CMPs) commonly share memory system resources. This usually gives a valuable improved resource utilization, but can also lead to destructive interference between the cores [1]. Hence, the performance of an application depends on the co-running applications in the CMP. This affects Operating System (OS) policies, where independent progress of each process often is assumed [2]. Destructive interference between cores in a CMP breaks this assumption and can lead to unpredictable interactive performance, missed deadlines, priority inversion and not complying with service-level agreements [3]. Several means in both software and hardware can be taken to reduce this destructive interference. However, many of these methods require accurate estimates on how much this destructive interference affects the performance [2]. So called *interference-free* performance estimates can also be useful to cloud service providers, enabling them to bill users properly without accounting destructive interference from co-running applications [4, 5].

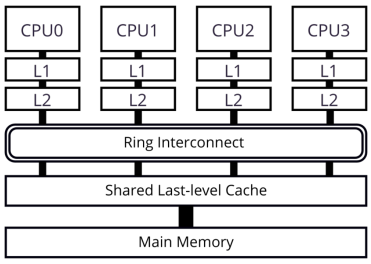


Figure 1.1: Processor setup in shared mode

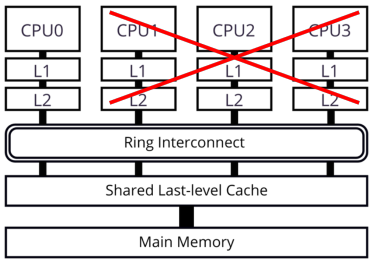


Figure 1.2: Processor setup in private mode

1.1 Predicting Interference-Free Performance

Evaluating performance prediction requires an establishment of precise terms and metrics of what is modeled. The processor setup in *shared mode* is showed in figure 1.1. In shared mode all cores are active and can thus cause destructive memory interference. To isolate the performance without this interference we define an opposing *private mode* for evaluation purposes. In private mode, one core has exclusive access to the memory system. This can be considered as having three idle cores (on a 4 core processor), illustrated in figure 1.2. Several performance accounting systems have been proposed trying to estimate private mode, or interference-free performance.

Private mode performance accounting systems can be used widely. Among others, they have been proposed used in shared memory resource management systems [2, 6–12] and interference-aware OS schedulers [3, 13–15]. However, predicting private mode performance is no straight-forward task. Modern processors have many latency-hiding mechanisms as non-blocking caches, out-of-order execution, and so on. This complicates private mode performance prediction, and thus causes area overhead and sometimes performance overhead from the performance accounting systems [2].

The current private mode performance prediction methods face some challenges, making them less attractive to be implemented by vendors. Some have a substantial complexity [2]. Others have a lower complexity but also much lower accuracy [3, 16]. Lastly, some also have a negative performance impact [17, 18]. Also, the performance accounting systems are not able to always predict accurately. Although some accounting systems are way more accurate than others [19], the current solutions typically have consistent errors for some problem types and have noteworthy errors overall. However, the accuracy of the private mode performance predictors is good enough to be useful for many use cases and give significant speedups in policies. This is specially the case for state-of-the-art [19] *Graph-based Dynamic Performance* (GDP) accounting [2]. Therefore, our main motivation for improving private mode performance accounting systems is reducing their overhead.

1.2 Assignment Interpretation

This section explicitly describes how the project description was perceived and which specific tasks that gave for the thesis. The project description is attached on page iii. The master thesis builds on a semester project from the fall 2019 [19]. In the semester project, performance accounting systems were analyzed with the current state-of-the-art challenges. Findings from the semester project was the primary source of focus areas for improvements in this master thesis. Thus, the main objective of the master project is interpreted to be: reducing implementation complexity compared to GDP, while retaining prediction accuracy. This is done to make private mode performance prediction methods more attractive to be imple-

mented in commercial multi-core implementations. The simplifications of GDP can be either substituting part of the GDP model or by using a completely new model. This resulted in two specific tasks, T1 and T2:

T1 Find and evaluate improvements to GDP.
T2 Explain the prediction accuracy of the found improvements. This includes explaining eventual strengths and weaknesses.

1.3 Project contributions

Private mode performance prediction is a trade-off between prediction accuracy and the resource cost of making the predictions. The complete memory system behavior cannot be modeled, so the predictors have to abstract some key behavior of the memory system to make predictions. This works well in many cases [19], but ideally private mode performance predictors are more lightweight than the current best solutions.

To increase the prediction accuracy, private mode performance accounting systems have modeled increasingly complex concepts. One such concept which is particularly difficult to model accurately is *Memory Level Parallelism* (MLP). MLP is when multiple cache misses are generated and serviced by the memory system in parallel [20]. This is illustrated in figure 1.3, where some compute cycles generate two memory requests which later stall the computation. The two memory requests are services in parallel, which exemplifies MLP. Instead of modeling complex concepts such as MLP, our approach to the problem is to see if the memory system behavior instead can be trained. Finding the right regression model, or combination of regression models, can reduce the area cost of private mode performance prediction while having sufficient accuracy to be useful. A well trained regression model can be seen as a wisely chosen heuristic to estimate the private mode performance.

The main contribution of this master thesis is a design space exploration and evaluation of using tree based methods for private mode performance prediction. Specifically, *linear model trees* (LMTs) are used standalone or as part of a performance model for private mode performance prediction. LMTs can be implemented

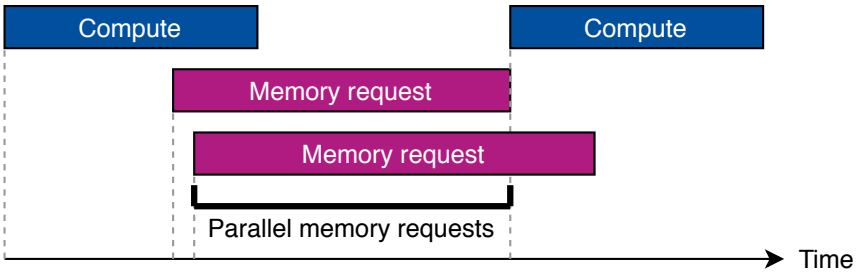


Figure 1.3: Illustration of Memory Level Parallelism

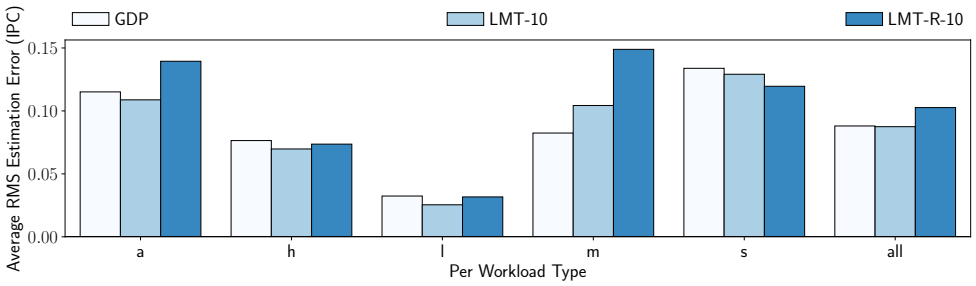


Figure 1.4: Average RMS workload error for IPC in regression evaluation

with a low area overhead. Also, LMTs have certain qualities making precise private mode performance predictions likely. Namely, classifying observations with similar memory system behavior and exploiting linearity in the observations. These qualities and how the LMTs are constructed is discussed more thoroughly in section 2.3.

To show the potential of LMTs and motivate further reading, figure 1.4 shows the average *root mean squared* (RMS) error for two LMTs compared to GDP within some defined workload types and overall. The LMTs have 10 leaf nodes where one of them has a reduced feature set, giving a lower storage overhead. The first LMT (LMT-10) reduces the error of GDP by 1%, while reducing the storage overhead by 24%. The LMT with a reduced feature set (LMT-R-10) has a 17% higher overall error compared to GDP. However, it reduces the storage overhead by 85%.

The motivation, implementation and evaluation of using LMTs for private mode performance prediction will of course be covered in more detail throughout this thesis. With the relating task in parenthesis, the contributions from this master project can be summarized as:

- (T1) Introducing linear model trees for private mode performance accounting. LMTs have not been used in private mode performance prediction before. How they work and the motivation of using them for private mode performance prediction, is presented.
- (T1) Showing how linear model trees can be implemented in hardware. Every component of the LMT is discussed to a corresponding hardware implementation. Latency and storage overhead estimates are included and compared to previous work.
- (T2) Explaining why linear model trees provide accurate performance accounting predictions. An extensive result set is generated, showing weaknesses and strengths of predictions using LMTs.
- (T2) Pointing out certain weaknesses of state-of-the-art, GDP. The critical path length (CPL) estimation mechanism in GDP does not work as intended in some scenarios. This leads to consequent overestimation of *instructions per cycle* (IPC).

Chapter 2

Background

This background chapter covers two main topics. First, the current private mode prediction methods are introduced. Knowing how they work give insight on the challenges of current performance accounting methods, and how those challenges have been proposed solved. This serves as the baseline to improve private mode performance prediction, which leads us to the second part. Using regression rather than modeling private mode performance itself is this master thesis approach to improve performance accounting. Following, the selected regression methods are introduced and why they have suitable qualities for private mode performance prediction.

2.1 Private mode performance prediction

Previously proposed performance accounting systems can be broadly partitioned into *invasive* and *transparent* systems. Where transparent systems can be further partitioned into *Architecture-centric* and *dataflow accounting*. This section will introduce acknowledged private mode performance systems. Figure 2.1 shows how those systems can be placed in the partitioning.

Invasive performance accounting systems [17, 18, 21] alter architectural policies. In periods of time, a single process is given higher priority in the memory system to minimize interference from other processes. The *Application Slowdown*

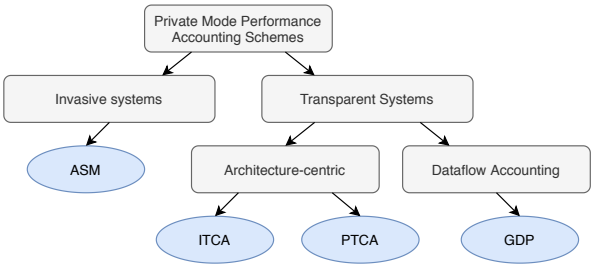


Figure 2.1: Classification of private mode performance accounting schemes

Model (ASM) [17] is an example of an invasive performance accounting system. Such systems can give accurate performance estimates but altering the policies can also give a negative performance impact, specially for latency-sensitive processes [2].

Transparent and architecture-centric performance accounting systems such as *Inter-Task Conflict-Aware CPU Accounting for CMPs* (ITCA) [16] and *Per-Thread Cycle Accounting* (PTCA) [3] do not alter architectural policies, and thus do not have any direct performance overhead. In general, such systems monitor certain conditions in the processor to determine if a cycle of memory latency also will be there in private mode, or not. An example of such a condition is whether the *Re-Order Buffer* (ROB) is full. A weakness of architecture-centric accounting mechanisms is that they only account behavior matching their pre-defined monitored conditions properly. Selecting the appropriate conditions is not trivial as the performance bottlenecks of applications vary a lot.

Dataflow Accounting is transparent performance accounting systems utilizing how dataflow dependencies between memory loads and commit periods are similar in both shared and private mode. *Graph-based Dynamic Performance accounting* (GDP) [2] is a private mode performance accounting system based on dataflow accounting. GDP is the current state-of-the-art of private mode performance accounting [19]. A known weakness of transparent architecture-centric policies is their inaccurate modeling of private mode MLP. This is a major improvement of dataflow accounting versus previous transparent accounting systems.

All the presented private mode performance schemes use a hardware unit called *Auxiliary Tag Directories* (ATDs), therefore ATDs are introduced before the schemes themselves. ATDs are separate tag directories private to each core used to determine if cache accesses would have been hits or misses in private mode. Figure 2.2 shows how data is retrieved from a regular *Last-Level Cache* (LLC). Physical addresses are partitioned into tag, set, and offset bits. The tag and set bits are used to determine if it is a cache hit and the data can be outputted from the cache. An ATD is shown in figure 2.3. It represent the private mode state of the LLC for a selected core, but do not store any of the actual data. For each core the tag and replacement bits are stored to detect whether LLC accesses would have been hits

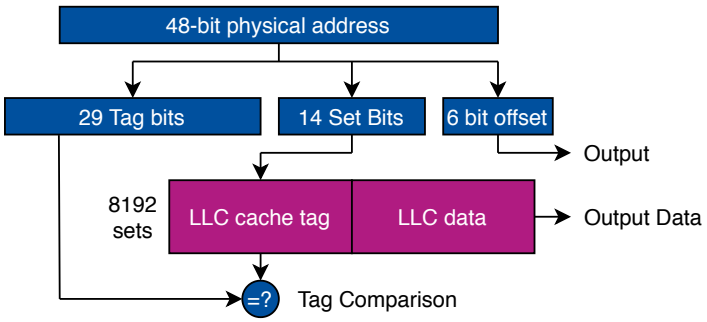


Figure 2.2: Regular LLC overview

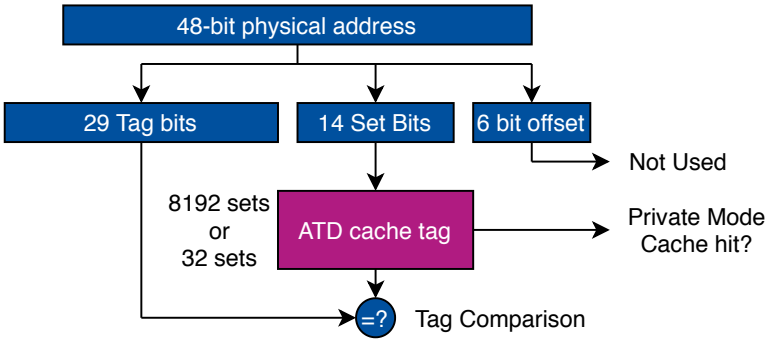


Figure 2.3: LLC ATD overview

or misses in private mode. Although ATDs do not store the cache data, they do have substantial storage overhead. To reduce this overhead, set sampling [22] is commonly used. With set sampling only a small number of sets are sampled in the ATD, assuming the behavior in those sets is representative for the whole cache. This usually gives sufficient accuracy while it removes much of the area overhead. For example, sampling 32 instead of 8192 sets reduces the storage overhead of the ATD by 99.6%.

Before introducing the individual performance accounting methods, some common terms for estimation are established. In general, T denote the total amount of cycles, C is compute cycles and S are stall cycles. The stall cycles are often divided in to different stall types to suit the performance model of each system. Whenever a variable is estimated through some mechanism, a hat ($\hat{}$) on the variable denotes an estimated value.

2.1.1 Invasive accounting: ASM

ASM [17] is an invasive performance accounting method. The main idea of ASM is that the performance of a program is strongly correlated with it's *Cache Access Rate* (CAR). The slowdown estimate is the ratio of CAR in private and shared mode.

$$Slowdown = \frac{\text{performance}_{private}}{\text{performance}_{shared}} \approx \frac{CAR_{private}}{CAR_{shared}} \tag{2.1}$$

In shared mode, CAR can easily be calculated. Where T is the the total number of cycles the shared cache accesses is counted, CAR_{shared} would be:

$$CAR_{shared} = \frac{\#Shared\ Cache\ Accesses}{T} \tag{2.2}$$

ASM avoids modeling MLP explicitly to estimate private mode performance and $CAR_{private}$. Instead, ASM introduces *epochs* where certain processes are given a higher priority in the memory system. This eliminates most of the interference of the other processes, and is the baseline of the $CAR_{private}$ estimate. Additionally,

ASM tries to quantify misses in the shared cache that would have been hits if running in private mode, so called *interference-induced misses*. This is sampled using ATDs. The stall cycles relating to interference-induced misses, S_{IIM}, is estimated using the difference of average miss and hit times within an epoch. When # IIM is the number of interference-induced misses, $\hat{S}_{IIM}$ will be estimated as:

$$\hat{S}_{IIM} = \#\ IIM \times (avg\text{-}miss\text{-}lat\ -\ avg\text{-}hit\text{-}lat) \tag{2.3}$$

When the highest priority application has no requests, the other applications can access the shared memory resources. This is done to limit the performance implications. However, it can also cause queuing delays for the highest priority application if it has requests arriving after another requests request is scheduled. Those stall cycles, S_{queue}, are estimated by multiplying the estimated number of private LLC misses with the average queuing delay.

$$\hat{S}_{queue} = \#\ Pr\text{-}LLC_{misses} \times \text{avg-queuing-delay} \tag{2.4}$$

$\hat{CAR}_{private}$ is estimated over epochs, similar to CAR_{shared}. Such as, when T represents the total cycles the system prioritized requests from a specific application, the stall cycles related to interference-induced misses ($\hat{S}_{IIM}$) and the queuing delay of private LLC misses ($\hat{S}_{queue}$), are subtracted. That gives the following equation for calculating $\hat{CAR}_{private}$ over one epoch ($\hat{CAR}_{private}$ estimates can also be aggregated over several epochs):

$$\hat{CAR}_{private} = \frac{\#\text{Application requests during epoch}}{T\ -\ \hat{S}_{IIM}\ -\ \hat{S}_{queue}} \tag{2.5}$$

Predictions with ASM are in many cases more accurate than architecture-centric transparent approaches [19]. However, limiting interference does not recreate private mode performance behavior accurately [2]. With congestion in the memory system, a core can accumulate a significant backlog of outstanding memory requests while running at low priority. Using the high priority epoch to get rid of this backlog does not match private mode execution where those backlogs do not occur. Also, the invasive technique comes at a performance cost with the changes to the architectural policies. In specific use cases this performance cost is significant, at a 57% performance reduction [2]. To summarize, ASM can often make accurate predictions but have some qualities making it less attractive to be implemented by vendors, mainly the performance cost of altering the policies.

2.1.2 Architecture-centric Accounting

ITCA

ITCA [13, 16] is a transparent architecture-centric performance accounting method. During execution it accounts cycles to separate quotas. One quota is stall cycles due to interference-induced shared cache misses. The other quota is CPU cycles while progressing or stalls that are not interference-induced. The idea is that if the

interference-induced stalls can be filtered effectively, the other quota will consist of the cycles the application uses in private mode. Therefore, the prediction accuracy of ITCA relies on how effectively the interference-induced stall cycles can be filtered. Denoting interference-induced stall cycles as $\hat{S}_{\text{interference-induced}}$, this can be viewed as:

$$\hat{T}_{Private} = T_{Shared} - \hat{S}_{\text{interference-induced}} \tag{2.6}$$

ITCA names a few specific conditions when it stops accounting progress to a task, and thus accounts it towards the interference-induced stall cycle estimate, $\hat{S}_{\text{interference-induced}}$. ATDs are used to find out whether misses are interference-induced or not. The monitored conditions are listed as:

1. Interference-induced instruction miss causing an empty ROB.
2. A full ROB while the oldest instruction in the ROB is caused by an interference-induced miss at the head of the ROB.
3. All *Miss Status Holding Register* (MSHR) entries are due to interference-induced misses

Stall cycles when either of these three conditions are met will be counted to the interference-induced stall cycles. However, ITCA fails to account all interference-induced cycles in the prediction. Only catching a small part of them, this results in conservative private mode estimates [2]. With little interference in the memory system ITCA can predict accurately, but the errors increase significantly with more congestion in the memory system [19].

PTCA

Similar to ITCA, PTCA [3] is a transparent architecture-centric private mode performance predictor, which originally was suggested for use in simultaneous multithreading processors [23]. Although, the techniques can be adjusted to other processor types too and a new paper targeted CMPs [3]. PTCA has more sophisticated conditions than ITCA on how the cycles are accounted, using *interval analysis* [24]. In interval analysis, the performance is estimated for intervals which are later aggregated for the complete program behavior. PTCA is known to be the most accurate architecture-centric performance prediction method [2]. The PTCA paper propose two variants of PTCA, through interpolation and extrapolation. The extrapolation variant will be covered here, as it has the best accuracy [3]. The PTCA counter architecture defines two distinct sources of inter-thread interference. First, interference-induced misses in the shared cache. Secondly, contention in the memory subsystems causing misses to take longer, called *waiting* stalls.

$$\hat{T}_{Private} = T_{Shared} - \hat{S}_{\text{interference-induced-misses}} - \hat{S}_{waiting} \tag{2.7}$$

The number of interference-induces misses in the shared cache is estimated using ATDs with set sampling. For the sampled sets, the interference cycles of interference-induced misses are counted when the an interference-induced miss

blocks the head of the ROB. These cycles are scaled to account for the sets that are not sampled in the ATD, and combined provide the $\hat{S}_{\text{interference-induced-misses}}$ estimate.

Estimating the waiting stall cycles coming from resource and bandwidth contention in the memory subsystems, $\hat{S}_{waiting}$, is slightly more comprehensive. It quantifies how misses that are not interference-induced have a larger penalty because of the memory subsystem congestion, where the additional stall cycles are called waiting cycles. First, the total amount of waiting cycles is estimated. This is done counting cycles lost to the following events:

- *Bus contention*, when a memory operation must wait to access the bus because it is occupied by another core.
- *Bank contention*, when a bank is occupied by another core and thus causes waiting cycles.
- *Interference-induced row buffer misses*, when a row buffer hit in private mode becomes a row buffer miss in shared mode. Servicing a row buffer miss is considerably longer than servicing a row buffer hit. A hardware unit with similar functionality as the ATDs check whether a row buffer miss would have been a row buffer hit in private mode. If so, the difference is accounted as waiting cycles. This is only necessary if an open-page policy is used.
- *Hardware prefetching*, when a load miss also appearing in the hardware prefetch queue blocks a commit at the head of the ROB it is accounted as waiting cycles. Assuming the prefetch would be timely in private mode, this is not accurate. However, they claim it accounts for a major fraction of interference due to prefetching.

The waiting cycles are kept track of with counters in the MSHRs. Note that only waiting cycles for misses that are not interference-induced are tracked, as the waiting cycles for interference-induced misses are covered in $\hat{S}_{\text{interference-induced-misses}}$. Also, waiting cycles are only accounted for when a long-latency load miss makes it to the ROB head and the ROB becomes full. This is done with insight of the interval analysis proposal [24]. As counting waiting cycles requires the knowledge if a miss is interference-induced or not, they are only counted for the sampled sets in the ATD. Therefore the results for those sets must be scaled to provide the overall $\hat{S}_{waiting}$ estimate.

In general, PTCA has a better prediction accuracy than ITCA. It is the most accurate architecture-centric performance predictor, yet less accurate than ASM and GDP [19]. However, some of the simplifications in the model can lead to mispredictions. The MLP estimate tends to be overestimated as certain limitations to MLP is not accounted for. Among those are long-latency instruction cache misses and branch mispredictions that depend on long-latency loads [23]. Weaknesses of PTCA can be exposed which can lead to severely over and underestimation in the performance predictions [2, 19].

2.1.3 Dataflow Accounting: GDP

GDP [2] is a transparent performance accounting system based on dataflow accounting. The backbone of GDP is Karkhanis' and Smith's analytical performance model [25], also based on interval analysis [24]. Assuming a perfect branch predictor and memory system, the performance model quantifies a steady-state system where imperfect components are estimated and the following performance loss subtracted. The basis of performance estimation using the model is that the combined compute and stall cycles divided on the number of committed instructions equals the performance, P, over a time period. For a process p, this is modeled:

$$CPI_p = P_p = (C_p + S_p^{Ind} + S_p^{Loads} + S_p^{Other})/Inst_p \tag{2.8}$$

To use the model for prediction, the stall cycles staying similar in shared and private mode are isolated. Memory independent stall cycles, S_p^{Ind}, and the private memory system load stall (PMS) stall cycles, S_p^{PMS}, are assumed to stay the same. S_p^{PMS} is not directly affected by shared mode interference as PMS by definition do not access shared units. This leaves two stall measures to be estimated. Shared memory system (SMS) load stall cycles, S_p^{SMS}, are estimated using the insight of dataflow accounting. The average private mode memory latency is estimated using DIEF and multiplied with the CPL to predict S_p^{SMS}. The CPL is estimated using an approximation of Kahn's algorithm [26] implemented in hardware off the critical path. Where $\hat{\lambda_p}$ denotes the DIEF latency prediction of the average private mode memory latency of a process, the SMS-load stall estimate, $\hat{S}_p^{SMS}$, will be:

$$\hat{S}_p^{SMS} = CPL_p \times \hat{\lambda_p} \tag{2.9}$$

The other stall cycles, $\hat{S}_p^{Other}$, is considered to have a smaller impact on the overall performance prediction. They are predicted in a simpler manner. $\hat{S}_p^{Other}$ denotes three quite rare stall events. First, stalls due to a full store buffer while the head of the ROB is a store instruction. Secondly, the L1 data cache can become blocked due to too many in-flight memory requests. This might lead to a stall when the load needing access to that blocked L1 data cache reach the head of the ROB. Lastly, the ROB might only contain wrong-path instructions due to a branch misprediction, which leads to a stall. The average shared memory latency can be calculated. For each of these events, it is assumed that the stall cycles scale proportionally with the ratio between the average shared mode memory latency and the estimated private mode memory latency.

Knowing how $\hat{S}_p^{Other}$ and $\hat{S}_p^{SMS}$ is predicted, the complete GDP prediction model is summarized as:

$$\hat{CPI}_p = (C_p + S_p^{Ind} + S_p^{PMS} + \hat{S}_p^{SMS} + \hat{S}_p^{Other})/Inst_p \tag{2.10}$$

GDP is more accurate than ITCA, PTCA and ASM. It is currently the state-of-the-art within private mode performance prediction [19]. The storage overhead

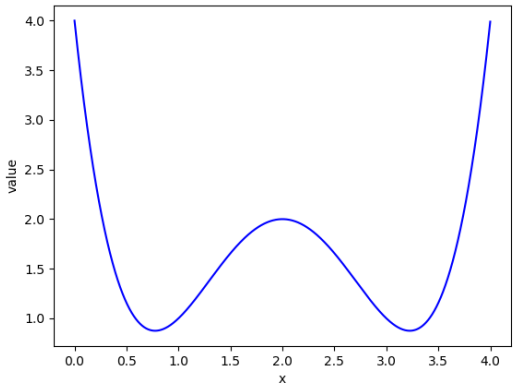


Figure 2.4: Example function to estimate

of GDP is 11.9KB [2]. Where the main contributors to this overhead is the ATDs (7.25KB), DIEFs interference and latency estimation buffers (2,80 KB) and the CPL estimator (1,85KB). This is substantial, but not more than previously proposed performance accounting systems [2].

2.2 Linear Model Trees

A Linear Model Tree is a regression method combining two other regression methods. It consists of both a tree structure and linear regression. This section introduces our use of both regression methods and how they are combined. The weights of the regression models are trained using a training set, minimizing a defined error for the training data. To visualize the main idea of the different regression models, they are applied to a one dimensional example function showed in figure 2.4

2.2.1 Decision Trees

Decision trees can be used for both classification and regression. A trained decision tree is a binary tree with classes or regression values in the leaf nodes. Every internal node splits the tree recursively comparing a single feature to a constant.

A decision tree used for regression can be seen as a piece-wise constant function. At each node, the "best guess" of the tree is the average output value of all observations in the training data ending up in that node in the tree. The splits of a decision tree can be determined in numerous ways where the best splitting method depends on how the tree is supposed to be used. The chosen metric to minimize should reflect the wanted behavior of the trained tree, and is often known as the *cost function*. For our regression usage, each split minimizes *mean squared error* (MSE) of the tree's predictions and the actual values after the split. The most common alternative cost function to MSE is *mean average error* (MAE). However,

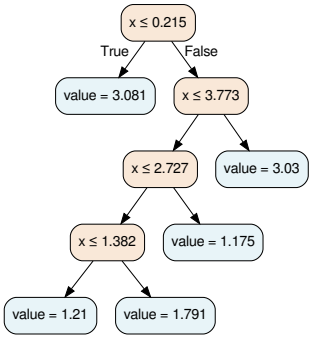


Figure 2.5: The Decision tree of the example function

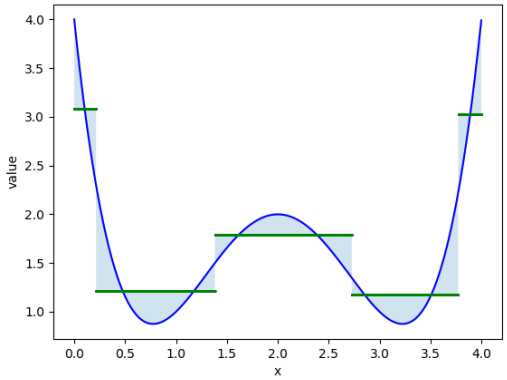


Figure 2.6: Example function estimated by a Decision Tree

MSE is often preferred penalizing large errors more. Step by step, the variance and overall errors in the training data are reduced recursively in the tree. If the usage of the tree reflects the training data, the predictions from the tree should also reduce data variance and errors. The resulting tree is a white box model. The output value from an observation in the tree can be explained following its path in the tree. The observable evaluation differs from black box models where an observed behavior cannot easily be explained. Although, the underlying mechanisms creating those exact weights in the decision tree are not evident by a single observation.

Overall, a decision tree can perform quite well for some problems and a trained tree evaluates data with quite simple comparisons. Typically, if the data is clustered with similar output values for subpopulations, the decision tree can effectively sort these subpopulations if some specific feature divide them. However, the training data for the tree can make it biased and without restrictions on the decision tree it is prone to *overfitting*, creating an over-complex regression model that do not generalize the data. In theory, a decision tree can be arbitrarily large with a single observation from the training set in each leaf node. This can be prevented several ways. A solution is using cross-validation on the model. Cross-validation can be done numerous ways, but all with the purpose of estimating when to stop training a regression model to prevent overfitting [27]. Another solution is putting restrictions on the size of the tree, limiting it to a certain number of leaf nodes or a limited height in the tree.

Figure 2.5 shows the resulting decision tree trained on the example function in figure 2.4, with a limitation of 5 leaf nodes in the tree. The resulting predictions and errors are displayed in figure 2.6, with an average RMS value of 0,31.

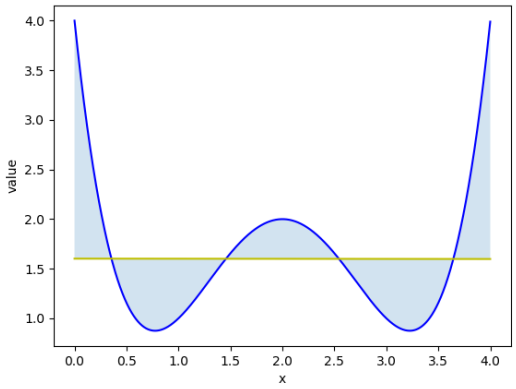


Figure 2.7: Example function estimated by linear regression

2.2.2 Linear Regression

Linear regression tries to approximate a function to an n-dimensional plane. It minimizes some metric between the linear approximation and observed targets. We use Ordinary least squared (OLS) Linear Regression, minimizing MSE of the linear approximation. There are many alternatives to OLS, often with a slightly higher complexity. There are three common ways to alter the least squares fit [27]. First, by using a way of *subset selection* to only use some features that are believed to be related to the response. Secondly, *regularization* can be used, shrinking coefficients towards zero to reduce variance. Lastly, *dimension reduction* is ways to project observations to a subspace of lower dimension. The projection is then used for OLS. Our baseline for linear regression is using OLS with a subset selection. Section 3.3.3 shows how this is done. The approach was selected due to its low hardware overhead, where initial testing of other linear regression methods had negligible accuracy implications.

If a problem is known to be linear, linear regression can give accurate results. Otherwise, it can be too simple to fully represent the actual data. Figure 2.7 displays the example function approximated by linear regression. The average RMS error of using linear regression on the example function is 0,66.

2.2.3 Combining decision trees and linear regression

A *linear model tree* (LMT) is a combined model with a decision tree having linear regression in the leaf nodes. This makes the predictions piece-wise linear instead of piece-wise constant using a decision tree alone. Combining two quite simple models, the resulting combined model is still not very complex and can be remarkably accurate for suitable problems. A linear model tree is approximating the example function in figure 2.8, and is considerably more accurate than both a decision tree and linear regression alone. For the example function, OLS linear

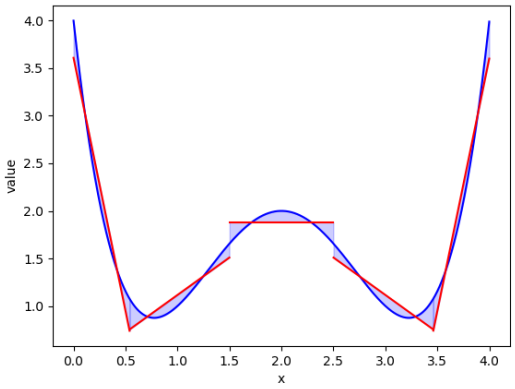


Figure 2.8: Example function estimated by a linear model tree

regression had an average RMS error of 0,66 while the decision tree alone had an average RMS error of 0,31. Combining the two regression methods to a LMT reduces the average RMS error to 0,12. Hence, for suitable problems LMTs combine relatively simple regression methods to significantly reduce the prediction errors, compared to using any of the regression methods alone. Ideally, the decision trees divide the observations into locally linear subpopulations. This cannot be guaranteed but will provide a low cost, accurate model if that is the case.

2.3 LMTs in private mode performance prediction

Linear Model Trees have certain qualities making them attractive for private mode prediction. We call the key memory system metrics tracked during program execution the memory system *footprint* over a period of time. The primary notion of using LMTs is that applications with a similar footprint also will have a similar prediction target value. Assuming this footprint lead to an effective classification in a LMT tree structure, exploiting the linear nature of the memory system can improve accuracy over using constant values for the subpopulations. Therefore, three aspects of why LMTs are well suited for predicting private mode performance are presented.

Classifying subpopulations: The tree structure of LMTs classifies subpopulations with similar behavior and similar performance. With wisely chosen features, this serves as a coarse classification of memory system behavior for private mode performance prediction. Key features are tracked and are the basis for which path to take in the tree structure. A compute bound application[1] requires little cache access and little BW, leading to a high proportion compute cycles over a period. The high amount of compute cycles is an important metric classifying a subpopulation with little interference-induced stall cycles. Another example, an application

[1]The behavior of gamess, in SPEC2006

requiring much cache[2] can be classified by the tree with how many shared cache accesses the program has where how many of those cache accesses were hits determine the expected performance. Similarly, every application has a characteristic footprint over an interval in the processor. This footprint is assumed to be similar for applications with similar performance. The regression model is trained to reduce the variance of the performance in the training data following classification. If the general patterns of memory system behavior can be contained in the training data, the classifications provided by the tree structure should also be viable for prediction.

Linearity: The memory system of a computer is linear by nature. Assuming no cache levels, the performance of a memory system is determined by BW alone. For GPUs this insight is used to classify performance as being either BW bound or compute bound [28]. Although, the memory system for CPUs is more latency-sensitive and multiple cache levels breaks the linearity and corresponding assumptions for GPUs. There are fewer studies examining linearity for CPU memory systems. However, MISE [18] discusses *memory bound* applications, applications that spends a significant fraction of its execution time outside of the compute phase. It shows that for memory bound applications, the performance is roughly proportional to the rate at which memory requests are served. And thus, the performance is linear to the request service rate for memory bound applications. The request service rate is not a feature in our LMTs. However, various key components for the request service rate are used. Hence, if the LMT is trained properly it should be able to capture some of this linearity. This motivates the use of linear regression in the leaf nodes and why this should give significantly more accurate predictions than using constant values as in a decision tree.

Scalability: An important quality of LMTs is how their area overhead can be scaled. Adding nodes to the tree improves the accuracy, although with diminishing returns. However, not all use cases of private mode performance prediction requires the same accuracy. Using LMTs, the area overhead can be reduced to the point where the accuracy is sufficient. In this work, tree sizes of 10, 40 and 80 leaf nodes are used to show how increasing the area used for regression can improve the accuracy.

[2]The behavior of twolf0 from SPEC200

Chapter 3

Implementing Linear Model Trees

Having established the motivation of using LMTs for private mode performance prediction, this section explains how they can be implemented. LMTs are suitable to predict various key aspects of private mode performance prediction. In light of the GDP performance model, several prediction models will be introduced replacing parts or the whole performance model by a LMT. All models uses interval analysis, where LMTs assists is the prediction of every interval. Later, how LMTs can be implemented in hardware and the corresponding storage overhead is presented. As *area* overhead estimates typically require synthesis, storage overhead estimates are consequently used to compare the cost of implementing in hardware. Implementing the performance accounting system in hardware is necessary to make it transparent and thus not causing any performance overhead.

3.1 Defining LMT performance models

In theory, LMTs can be used to predict any output value. This work presents three main ways to use LMTs for private mode performance prediction. First, by predicting IPC directly. Secondly, through using the performance model of GDP by providing both stall and latency estimates with LMTs.

Predicting IPC directly: The simplest way of using LMTs for private mode performance prediction is predicting IPC directly in the LMT. Without any connection to other performance models, the area overhead of this prediction method solely comes from the LMT itself. This configuration will be mentioned as *LMT-IPC*.

Predicting Stall cycles: A major strength of the GDP performance model is isolating the components staying the same in shared and private mode. The majority of the storage overhead in GDP comes from the latency predictions of DIEF. Those are used to predict $\hat{S}_p^{SMS}$ and $\hat{S}_p^{Other}$. Replacing costly parts of the prediction requiring DIEF with LMTs can reduce area overhead while still providing accurate performance predictions, and benefiting of the strengths of the GDP performance

model.

The most important estimated value of the GDP performance model is the SMS-load stalls, $\hat{S}_p^{SMS}$. This value can be estimated with a LMT. The goal of such a prediction is removing the area overhead of DIEF. However, the other estimated stall cycles, $\hat{S}_p^{Other}$, still rely on the DIEF latency estimates for prediction. Three ways of handling $\hat{S}_p^{Other}$ are implemented. They are called *LMT-Stall*, with number 1, 2 or 3. Having this in mind, the implemented models using LMT for stall cycle prediction are:

LMT-Stall-1: First, using GDP exactly the same way to model $\hat{S}_p^{Other}$. This do not remove the need for the DIEF latency estimates, and thus do not remove the main cause of storage overhead. However, this configuration is included for evaluation purposes to see how the SMS-load stall estimates using a LMT changes the GDP accuracy.

LMT-Stall-2: Secondly, $\hat{S}_p^{Other}$ is estimated using OLS linear regression. This provides a lightweight regression method replacing the costly DIEF scaling of other stall cycles in the GDP performance model.

LMT-Stall-3: The third approach to remove the need of DIEF for estimating $\hat{S}_p^{Other}$, is to avoid modeling it at all. Instead, the combined stall cycles of $\hat{S}_p^{SMS}$ and $\hat{S}_p^{Other}$ is the target of the LMT predictions. This simplifies the GDP performance model somewhat, as the combined stall prediction is the only value that must be estimated.

Predicting latency: The DIEF latency estimates are solely an input to calculate $\hat{S}_p^{SMS}$ and $\hat{S}_p^{Other}$ in the GDP performance model. Predicting the same latency values by a LMT will leave the rest of the model exactly the same. A moderately sized LMT has lower area overhead than DIEF. Hence, this configuration can reduce the area overhead of the performance prediction. This configuration is called *LMT-Lat*

3.2 Hardware Implementation

A hardware implementation of a linear model tree consist of three main parts. First, the features need to be gathered. Secondly, the binary tree must be traversed, classifying the input data. Lastly, linear regression calculates the output value using the weights corresponding with the classification by the tree. If a decision tree is used for regression instead of a linear model tree, the output of the tree points to a constant instead of a pointer to linear regression weights.

Feature retrieval: Retrieving the features mainly depends on strategically placed counters. The exception from this is the private LLC estimates. Those are provided using ATDs (Auxiliary Tag Directories) with set sampling [22]. The set sampling gives sufficient accuracy while reducing the area overhead of ATDs significantly. All the features are stored available to be used for evaluation in the decision tree and linear regression.

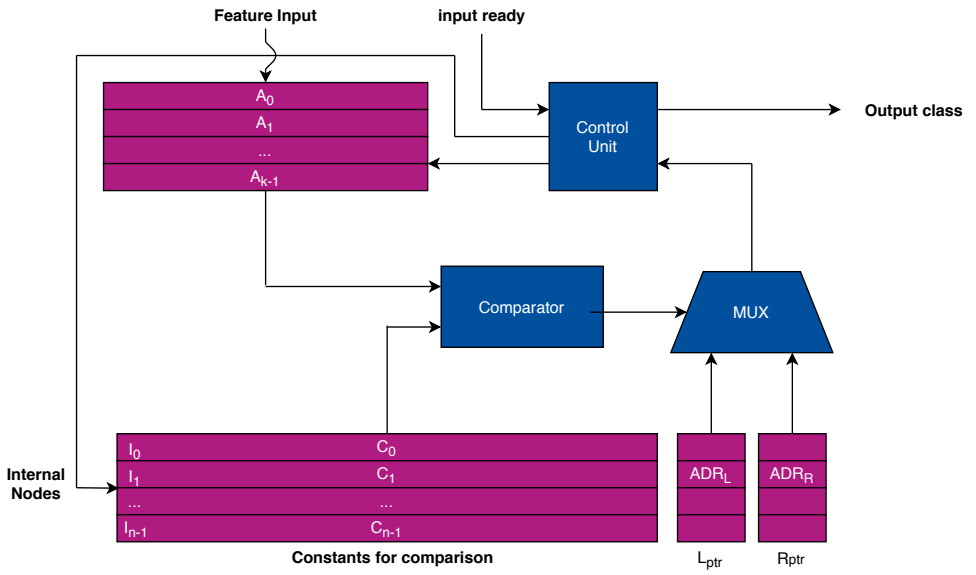


Figure 3.1: Hardware model of the tree structure

Tree structure: The decision tree is implemented with focus on area efficiency. Other proposed hardware techniques typically aim to accelerate decision tree evaluation using parallelization and pipelining [29, 30]. Our chosen design is much simpler, although containing some similar elements. Figure 3.1 shows the architecture of the area efficient tree evaluation. Each internal node (every split in the tree) is stored as an entry in a table. The entry consists of two parts. First, the constant used for determining which way to go in the tree through the comparator. Secondly, addresses for the nodes to the left and right in the tree. These addresses contain the address itself and which feature should be compared to the constant at the address. A specific feature number is reserved to denote that the child is a leaf node. The tree is traversed iteratively until reaching a leaf node. The address of this node is used as an input to the linear regression, pointing to the entry with the correct weights to use. If a decision tree is used for regression without linear regression in the leaf nodes, the output here would point to a constant instead of weights for linear regression.

Linear Regression: Each leaf node represent a linear regressor. The weights for the regression is stored in a table with one entry per leaf node. The evaluation of a linear regressor can be seen as the dot product of the weights with a predefined array of features, plus some constant. In the same iterative manner as for the tree structure, the output of the linear regression is calculated. The hardware structure will only need one adder and one multiplicator used iteratively to multiply a feature value with the linear regression weight and add the result to the aggregated output value.

3.3 Performance and Area Overhead

The most important qualities of private mode prediction methods is their accuracy compared to the performance and area overhead of implementing the system. This section describes the performance and area overhead of LMTs. First, the LMT is examined standalone before the complete models are evaluated. Lastly, several means that can reduce storage overhead are discussed.

3.3.1 Linear Model Tree

Implementing LMTs for private mode performance prediction in hardware does not give a direct performance overhead, because it is off the critical path in the processor. Although, it can cause an indirect delay if a delay from the private mode performance prediction causes an untimely quota allocation in some policy. For example, hardware management policies repartitioning shared resources can make use of the estimates. However, such events are very rare calculated every one to five million cycles [6, 7, 31]. For this usage, a low area cost of the hardware implemented LMTs is more important than fast calculations of the results. Of course the calculations can be done in the processor itself not causing any additional storage overhead, but that comes at a performance cost demanding the processor resources at regular intervals. Nevertheless, the hardware calculations will not take more than a couple of hundred cycles to compute at maximum. If the policies are recalculated at regular intervals, the results from the private mode prediction can be scheduled to be available at the time it is needed. The latency of the calculations is slightly larger than prior work [2, 3, 13, 17]. However, if this is problematic for some use cases the latency of the evaluations can be much reduced at an area cost [29]. The following paragraphs break down the latency of hardware calculations and the storage overhead of the components in LMTs.

Latency calculation

The latency of the hardware calculations depends on the tree size, which features are used and how many features are used for linear regression. Assuming addition and subtraction takes one cycle, multiplication 3 cycles and 25 cycles for division, the components of the LMT will have the following latencies:

- **Retrieving features:** Most features do not require any preprocessing. However, some features we have used are aggregated values or average values. If the average values are calculated using division before the tree evaluation, this will lead to a 25 cycle latency to make the features available. If all use the same division unit to save area, this will lead to 75 cycles of latency as we have used three average feature values.
- **Tree traversal:** The latency from tree traversal depends on the height of the tree, and will naturally increase for larger trees. The comparator uses one cycle, where the Control Unit should be able to check if it is a leaf node

and make the next comparison ready in one cycle. Therefore, every internal node in the tree takes two cycles to traverse. For the LMT-IPC tree with 10 leaf nodes the minimum heigh before reaching a leaf node is 2, giving a latency of 4 cycles. The maximum height is 4, giving a latency of 8 cycles. The most nodes necessary to traverse in the LMT-IPC tree with 80 leaf nodes is 10, giving a latency of 20 cycles.

- **Linear regression:** Using k features for linear regression, the latency will be $k \times 3 + 1$ cycles using a single multiplication unit. It mostly comes from multiplying the weights in the linear regression. The extra cycle comes from adding the results, where all but the last addition can be hidden with pipelining the multiplication and addition units. Using 12 features, this gives a latency of 37 cycles.

With these latencies, the maximum latency for our 80 leaf node LMT-IPC is a total of 135 cycles, while the maximum latency of the 10 leaf node LMT-IPC is 125 cycles. This is 64 and 54 cycles more than the GDP latency of 71 cycles [2]. However, area efficiency has been the main focus of our work. If low latency is important for a specific use, several means can be taken reducing the latency. For example: only using features not requiring preprocessing, a balanced tree of height 4 and 6 features for linear regression will give a latency of 27 cycles. This is likely to have an accuracy impact, but can be worth considering if a low latency is critical. Also, at an area cost the preprocessing and linear regression can have a lower latency by parallelizing the calculations with several hardware units.

Storage overhead

The typical use of private mode performance predictions value area-efficiency over a low latency of the calculations. Therefore, the main focus of this work have been reducing area overhead. The following paragraphs break down the area overhead of the subcomponents in LMTs. An overview of the storage overhead can be found in table 3.1.

Retrieving features: Storing the feature values causes area overhead. All the features have to be stored with a certain precision. For simplicity, all features and constants have been stored with the same precision. Storing features with different precision can reduce storage overhead but come at an area cost due to logic handling the different bits used for storage. We analyzed the training data and

Leaf Nodes	Tree structure (Byte)	Linear regression weights (Byte)	Total storage overhead (Byte)
10	45	390	471
40	224	1560	1859
80	474	3120	3669

Table 3.1: Per core LMT storage overhead without ATDs

chose the lowest amount of bits able to represent the highest feature value in the training set, which lead to using 24 bits representing numbers. In our setup a maximum of 25 features are tracked, resulting in a total storage overhead for representing those features of 75 bytes per core. For the smallest tree sizes not all features are used, which results in slightly less storage overhead of the features.

Most features are based on counting certain events in the processor. These do not contribute to additional storage overhead besides storing the feature value. However, ATDs are used to estimate private mode hits, misses and accesses to the LLC. The ATDs contribute significant storage overhead. Assuming 48 bits are used for physical addresses there are 29 tag bits, 13 set bits and 6 bits for offset. Using set sampling [22] with 32 sets for the 16-way cache, the ATD storage overhead per core will be $29 \times 16 \times 32 = 14848$ bits. Hence, the total storage overhead of ATDs for the 4 core processor is 7.25 KB. However, ATDs can also be used for other purposes than private mode performance prediction. For example, in cache partitioning policies [22], [6]. Since ATDs are much used and can be used across policies, they do not cause any additional storage overhead if they are already implemented for other usage.

Tree structure: In the tree structure, the storage overhead is in the table representing the internal nodes of the tree. Each entry in the table uses 24 bits to store the constant for comparison in that internal node. Additionally, necessary information of the two children nodes must be stored. Each children node need information of the child node address, and which feature is compared in that node. For n leaf nodes, $\lceil \sqrt{n} \rceil$ bits are needed to address every node. Note that if a tree has n leaf nodes, it has n - *1* internal nodes. Leaf nodes and internal nodes are stored in separate tables, so the same address numbers are used twice (once for internal nodes and once for leaf nodes) reducing the number of needed bits for addressing. The leaf node addresses are used for linear regression, not tree traversal. For k features, $\lceil \sqrt{k} \rceil$ bits are needed to address which feature should be used. In our tree, a specific feature number is reserved to denote a leaf node. This saves 2 bits per internal node in the tree, compared to having flags denoting leaf node or not.

Linear regression: The area overhead for the linear regressors in the leaf nodes is simply the weights of the linear regression. One weight is needed per input feature, and one additional constant. The storage overhead of these weights are stored in a table with one entry per linear regressor. Therefore, each entry uses $24 * (n + 1)$ bits for n features used in linear regression. The baseline LMT implementation used in this project selects 12 features for linear regression. See section 4.5.2 for details on this.

3.3.2 Total model storage overhead

For the LMT-IPC, and LMT-Stall-3 models, there are no storage overhead besides the LMT and ATDs. The LMT-Stall-2 model only adds the weights for one linear regressor to this. For other performance models using part of the GDP calculations,

Model	**Leaf Nodes**	**Total storage overhead (KB)**	**Percent of GDP overhead (%)**
LMT-IPC	10	9.09	76
LMT-IPC	40	14.5	122
LMT-IPC	80	21.6	181
LMT-Stall-1	10	11.9	100
LMT-Stall-1	40	17.3	145
LMT-Stall-1	80	24.4	204
LMT-Stall-2	10	9.13	77
LMT-Stall-2	40	14.5	122
LMT-Stall-2	80	21.6	181
LMT-Stall-3	10	9.09	76
LMT-Stall-3	40	14.5	122
LMT-Stall-3	80	21.6	181
LMT-Lat	10	10.9	92
LMT-Lat	40	16.4	137
LMT-Lat	80	23.4	197

Table 3.2: Total storage overhead for combined models

additional storage overhead come from those components. Section 2.1.3 contains a breakdown of the storage overhead in GDP. LMT-Stall-1 contain all this storage overhead except the CPL estimator, while LMT-Lat requires only the CPL estimator (not overhead from DIEF buffers). The combined storage overhead for selected tree sizes are displayed in table 3.2. The total storage overhead of the models combines overhead from ATDs, the LMT representation and eventual parts from GDP.

Compared to GDP, most of the models reduce the storage overhead with small trees. Most notably, LMT-IPC and LMT-Stall-3 reduces the storage overhead compared to GDP by 24%. The storage overhead besides ATDs is reduced by 60%. In some scenarios this might be a more appropriate comparison as ATDs can be used across various policies and thus do not cause additional storage overhead, being used for private mode performance prediction.

3.3.3 Reducing storage overhead

Depending on the tree size, different parts of the tree contributes the most storage overhead. In particular ATDs for small trees and linear regression weights for large trees. For a specific use case this is an optimization problem between accuracy and the area cost of the predictions. This section mentions how the storage overhead of the ATDs and linear regression weights can be reduced.

Reducing number of features in general

Reducing the number of features reduces the storage cost. Adding and removing available features usually have a limited cost, with counters keeping track of certain events. However, this may also affect the storage overhead in other parts of the LMT. The white-box model of the tree structure in LMTs makes the usage of each individual feature available. If a feature is neither used for linear regression or in the decision tree, it can be removed without any accuracy impact. If some feature is used very little (typically once in a large decision tree), the performance impact of removing the feature can often be negligible and easily tested. Using insight from the knowledge of the memory system, the tree can be built with a compact feature set for example by trying to remove strongly correlated features. Alternatively, a more extensive analysis of choosing features for the decision tree can be examined. A way to do this is using R^2 scores for a number of selected features iteratively as for the feature selection for the linear regression (section 4.5.1).

Changing number of features used for linear regression

In the linear model trees, the weights for linear regression is a main contributor to storage overhead. For the 10, 40 and 80 leaf node LMT-IPC, it contributes 17%, 42% and 56% of the storage overhead. The storage overhead for the linear regressors scale linearly with the number of features used for linear regression. At the cost of accuracy, the storage overhead can be much reduced by using fewer features for linear regression. Section 5.4.1 analyzes how varying the number of features used affects the accuracy of the predictions. Using 6 features instead of 12 reduces the storage overhead of the weights for linear regression by 46%. Adjusting the features used for linear regression versus the tree size may give more accurate predictions at the same storage cost.

A way to select the input for linear regression with fewer features is through dimension reduction methods such as PCA (Principal Component Analysis) and PLS (Partial Least Squares) [27]. These methods try to retain as much information of all the features as possible in a feature set of a lower dimension. For example, PCA could be implemented as preprocessing for the features used in linear regression to retain as much accuracy as possible while reducing the storage overhead of linear regression weights. PCA is making a projection of all input features onto a feature set of a lower dimension maximizing the variance of the data in the reduced feature set. This will yield good results if fewer dimensions can explain the majority of the data variance. The cost of PCA is the preprocessing, where each feature after PCA is a linear combination of the input features of PCA. Hence, the preprocessing cost will correspond to a matrix multiplication and be relatively small compared to the reduced storage overhead for linear regression weights if it yields similar accuracy. In some cases the input features of PCA need to be scaled to give good results, also causing some additional area overhead. This can be done numerous ways, but through *standardization* or *normalization* is the most

Leaf Nodes	**Total storage overhead (KB)**	**Percent of GDP overhead (%)**
10	1.84	15
40	7.26	61
80	14.3	120

Table 3.3: Total LMT storage overhead for various sizes without ATDs

common. Standardization is scaling the feature input to a standard normal distribution with the mean at zero and a standard deviation of one. Normalization scales the feature input to the range between zero and one.

Prediction without ATDs

For small trees the ATDs take up a substantial part of the total storage overhead. For the 10, 40 and 80 leafnode LMT-IPC it takes up 80%, 50% and 34% of the storage overhead. This could be reduced by using fewer sets for set sampling in the ATD, at the cost of accuracy. Alternatively, the features requiring ATDs can be removed. Assuming this makes ATDs redundant, LMT-IPC will have the area overhead shown in table 3.3. For a 10 leaf node LMT-IPC, the area overhead of GDP is reduced by 85%. Section 5.4.2 shows results without using ATDs as a part of the sensitivity analysis. If ATDs are also implemented for other purposes than private mode performance prediction, the improved accuracy of predicting with private LLC estimates is preferred.

Chapter 4

Methodology

The workflow of the project is briefly visualized in figure 4.1. First, an extensive set of simulations in the cycle accurate M5 simulator [32] was the basis of the semester project preceding this master thesis. Many data points were gathered from these simulations. These data points were analyzed using Scikit Learn [33] to see if some regression method could be used to predict private mode performance. The gathered data points were divided into a training and test set. The training set was used to train the regression models, where the test set was used to check their accuracy. Whenever results are mentioned as *regression evaluation*, it means that the results are generated using Scikit Learn with the regression model on the test set. Later, selected LMTs were exported to the M5 Simulator and tested with new workloads in the M5 Simulator. The generated results from these simulations are called *simulator evaluation*.

4.1 M5 Simulator

For experiment simulations and gathering of data, an extended version of the M5 simulator [32] is used. The derived version [34] is inspired by commercial CMP implementations [35]. It consists of two private (L1 and L2) cache levels and one shared cache (LLC). We have consistently used four cores, connected to the LLC through a shared ring interconnect. A high-level overview of the setup is visualized in figure 1.1, where detailed specifications of the processor setup in the simulator can be found in table 4.1.

4.2 Workload generation

The multi-programmed workloads used for simulations are generated from 51 benchmarks from SPEC2000 [36] and SPEC2006 [37]. The benchmarks are categorized as streaming benchmarks (S), highly memory sensitive benchmarks (H), medium memory sensitive benchmarks (M) and low memory sensitive benchmarks (L). The details of the classification can be found in appendix A. Using

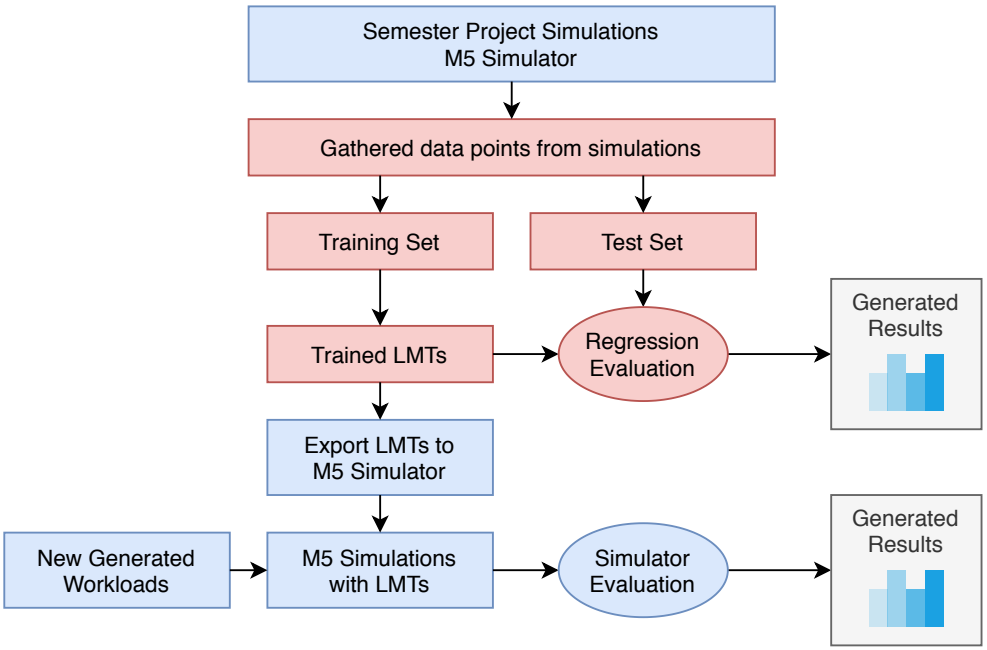


Figure 4.1: Workflow of simulation and regression analysis

Parameter	Value
Clock frequency	4 GHz
Processor Cores	128 entry reorder buffer, 32 entry load/store queue, 64 entry instruction queue, 4 instructions/cycle, 4 integer ALUs, 2 integer multiply/divide, 4 FP ALUs, 2 FP multiply/divide, 2048 entry hybrid branch predictor, 2048 entry 4-way BTB
L1 Data Cache	2-way, 64KB, 3 cycles latency, 16 MSHRs
L1 Instr. Cache	2-way, 64KB, 3 cycles latency, 16 MSHRs
L2 Private Cache	4-way, 1MB, 9 cycles latency, 16 MSHRs
L3 Shared Cache	16-way, 8MB, 16 cycles latency, 256 MSHRs, 4 banks
Ring Interconnect	4 cycles per hop transfer latency, 32 entry request queue, 1 request ring, 1 response ring
Main memory	DDR4-2666, 18-18-18-43 timing, 64 entry read queue, 64 entry write queue, 16 banks, 2 channels, FR-FCFS scheduling, open page policy

Table 4.1: Model Parameters

SimPoints [38, 39], a 100 million instruction sample is used and considered representative for the benchmark behavior. Four benchmarks are randomly selected for the workloads, where a single benchmark can at most appear twice in the same workload.

For the evaluation of different performance prediction methods of the semester project, a set of workloads were generated choosing benchmarks from the specific benchmark classifications. The set of workloads consisted of 25 S-workloads, 25, H-workloads, 10 M-workloads and 10 L-workloads. Additionally 10 A-workloads was generated, using benchmarks randomly from all classifications. The results from these simulations was the basis of the initial comparison of performance accounting methods in the semester project [19] and data used for regression. To verify the performance of the trained regression model in the simulator, a new set of workloads was generated. This has a new random seed, and has 5 workloads within each of the same categories as the initial workload set. The new workload set was used to test LMT-IPC in simulation evaluation where a new workload set was necessary to ensure evaluation not identical to the behavior in the test and training set. 5 workloads were chosen as an appropriate trade-off where there were enough workloads to show the general trends without demanding too much computing resources (ten-thousands of computing hours on a supercomputer have been used for the simulations of the semester project and master thesis).

The multi-programmed workloads are simulated until all benchmarks have committed 100 million instructions. If a benchmark reach 100 million instructions before all benchmarks are done, it is restarted. This is done to ensure a similar behavior from the other processes running in shared mode. To provide a fair comparison between private and shared mode, the instruction sample points in shared mode are stored. The number of cumulative committed instructions in these sample points are used as an input to the private mode experiments. There, it is ensured that the private mode experiments measure the IPC over the exact same instructions in each interval. Having the exact same instructions in each interval is critical comparing the predicted private mode IPC in shared mode with the measured IPC in private mode.

4.3 Scikit-Learn

Scikit-Learn [33] is a much used, open-source machine learning library for python. It was used for all regression analysis of the data in this project. The generated models from Scikit-learn was exported to the M5 simulator for simulator evaluation through custom made scripts.

The tree structure of the LMTs are the decision tree used for regression, provided by Scikit-Learn. It is trained with MSE as the metric to reduce variance for. This results in all the splits of the LMT. The linear regression in the leaf nodes use the OLS linear regression provided by Scikit-Learn. The only adjustment of the output values from the linear regression is that eventual negative predictions

are replaced by zero. This is because negative predictions do not make sense for either IPC, stall or latency estimates.

4.4 Data and testset

All the data used for training and testing the regression models come from simulations run during the semester project preceeding this master thesis [19]. It consists of roughly 76000 data points with traced features and a calculated target value which is supposed to be predicted by the regression models. As explained for the workload generation, these datapoints come from 10 workloads within workload categories A, L and M, and 25 workloads within workload categories H and S.

Workload number 7, 8 and 9 within each category was selected as test set. Initially, workload number 1 through 6 within each category was selected to be the training set. This training set is denoted as the *balanced* training set. Initial testing of the regression models showed that the balanced training set did not contain enough data to properly show the strengths of LMTs. This was specially the case in the congested workload categories, and when the tree sizes grew. A larger tree would intuitively give more accurate predictions, but this is not the case if the training data contain too little information to properly train the tree. Therefore, a training data set with *unbalanced* training data have been used in general. The unbalanced training data uses all available data generated from the semester project [19]. Sampling new data points require running simulations, where there are no limit on how many data points that can be generated. However, running extensive workloads on the simulator often takes weeks. Therefore, generating additional training data for the regression was not viable within the scope of this thesis. In the semester project, H and S workloads were examined more closely than A, M and L workloads. 25 workload configurations were generated within S and H workloads. For the unbalanced training set used, all workloads except number 7, 8 and 9 (the test set) were used for training. This means that the training set has 22 workloads within H and S workloads, but only 7 workloads within A, L and M workloads. This is not ideal but is the better option to properly show the qualities of using LMTs for private mode performance prediction. To explicitly show how the data point are divided, table 4.2 shows which workload configurations

Set name	**A**	**H**	**L**	**M**	**S**
test set	7-9	7-9	7-9	7-9	7-9
balanced training set	1-6	1-6	1-6	1-6	1-6
unbalanced training set	1-6	1-6, 10-25	1-6	1-6	1-6, 10-25

Table 4.2: Workload configurations used in data sets, per workload type

Feature name
Aggregated Shared Mode LLC Misses and Writebacks for all cores
Average Shared Mode Latency
Average Shared Mode PMS Latency
Average Shared Mode Shared Store Latency
Compute Cycles
Memory Independent Stall Cycles
Number of Shared Mode Hidden Loads
Number of Shared Mode Shared Stores
Number of Shared Mode Write Stalls
Private Mode LLC Access Estimate
Private Mode LLC Hit Estimate
Private Mode LLC Writeback Estimate
Shared Mode Empty ROB Stall Cycles
Shared Mode IPC
Shared Mode LLC Accesses
Shared Mode LLC Hits
Shared Mode LLC Writebacks
Shared Mode PMS Stall Cycles
Shared Mode Private Blocked Stall Cycles
Shared Mode Stall Cycles
Shared Mode Total Latency
Shared Mode Write Stall Cycles
Summarized Shared Mode LLC Misses and Writebacks
Total Number of Shared Mode Memory Requests
Total Shared Mode PMS and SMS Stall Cycles

Table 4.3: Traced features available for regression

are used for testing and training, within the workload categories. The unbalanced training set is used consequently except for section 5.1.2, which shows results using the balanced training set. For a project at a larger scale this will not be problematic as new data points can be sampled by simulations. The data points are based on counting discrete events, and thus have no data impurity. Training with more data points from sampled workloads will make the LMT model more robust.

4.5 Feature Selection

Which features are available is a key aspect of succeeding with any regression model. The available features for training the LMTs are displayed in table 4.3. The traced features are centered around SMS-load stall prediction and were originally

traced to evaluate the performance of predictions made by GDP during simulations. However, the traced features are seemingly suitable for private mode performance prediction with regression too. Although, it is not unlikely that other features we have not traced might provide even better data points to the LMTs. Adding additional features representing other events in the processor can only increase the contained information and prediction accuracy. The following sections describe how only some of the features were chosen and used for linear regression. All features were on the other hand available to make splits in LMTs.

4.5.1 Coefficient of Determination

The coefficient of determination or R^2 is a statistic that gives information on how good data predicted of a model fits the real data. To properly define R^2, we need to define two other metrics. Where y is the real data values and p is predictions, SS_{res} (residual sum of squares) is defined in equation 4.1. It measures deviations between predicted and actual data.

$$SS_{res} = \sum_i (y_i - p_i)^2 \tag{4.1}$$

The SS_{tot} (total sum of squares) is a quantity defined as the squared differences between actual data and the mean of all the data. It is defined in equation 4.2, where y is the real data and $\bar{y}$ is the mean of y observations ($\bar{y} = 1/n \times \sum_i^n (y_i)$).

$$SS_{tot} = \sum_i (y_i - \bar{y})^2 \tag{4.2}$$

R^2 is defined in equation 4.3, as 1 minus the ratio between SS_{res} and SS_{tot}. A perfect model will have a R^2 of 1. There is no lower bound of R^2 values. However, a model simply predicting the average value of a set with training data will have a R^2 value of 0 for the training set. Therefore, the regression should have a R^2 value significantly larger than 0 to contain the general trends of the data.

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}} \tag{4.3}$$

4.5.2 Linear Regression

The features used for linear regression in the leaf nodes of the linear model tree was selected iteratively using the coefficient of determination. The first chosen feature was the feature giving the highest R^2 value predicting the chosen target using linear regression. The second chosen feature was the feature giving the highest R^2 value in pair with the first one. This was continued until all features had been sorted from the one feature contributing the most in linear regression to the least given that the preceding features have already been chosen.

target (no. features)	$\mathbf{R^2}$	% of max
latency (1)	0.39856	51.88
latency (3)	0.64262	83.64
latency (6)	0.69204	90.08
latency (9)	0.75249	99.25
latency (12)	0.76487	99.55
latency (25)	0.76829	100.00
SMS-load stall cycles (1)	0.16059	20.80
SMS-load stall cycles (3)	0.38938	50.44
SMS-load stall cycles (6)	0.55030	71.28
SMS-load stall cycles (9)	0.57310	74.24
SMS-load stall cycles (12)	0.62421	80.86
SMS-load stall cycles (15)	0.76665	99.31
SMS-load stall cycles (25)	0.77197	100.00
IPC (1)	0.58773	74.29
IPC (3)	0.65545	82.86
IPC (6)	0.68030	86.00
IPC (9)	0.75875	95.91
IPC (12)	0.78171	98.81
IPC (25)	0.79108	100.00

Table 4.4: R^2 values for selected features

The features were selected on the complete training set, so there may be variations in which features are important for linear regression in the different subpopulations classified by the decision tree. Table 4.4 shows the R^2 values of a set of iteratively chosen features to predict IPC, SMS-load stall cycles or latency. At twelve features the linear regression seemed to contain most of the information possible by linear regression (The R^2 scored for a set of features compared to using all features). Therefore, 12 features was used for linear regression in the leaf nodes. Using more features requires more training data to make the regression stable, and increases the area overhead. Therefore 12 features seemed like a good initial compromise. Appendix B shows the R^2 scores for every iteratively added feature for each target value, and thus which specific features are used for linear regression in each case.

4.6 Metrics

Establishing proper metrics is important in quantifying errors between predicted and actual data values. Also, a such metric is suitable to compare the accuracy of different private mode performance predictors. Shared mode estimates are noted $\hat{\alpha}$, while the actual values are noted α. Absolute errors are used, where an error $E = \hat{\alpha} - \alpha$. Absolute errors are preferred over relative errors, as IPC is the primary prediction target. IPC values can be close to 0 for some data points. This can give unreasonably high relative errors while the absolute values are small. Therefore, it seems like a more fair comparison of the data points in the IPC range to use absolute values. The chosen metric for comparing aggregated error values is *Root Mean Squared* (RMS) errors. For n error estimates E, the corresponding RMS error would be:

$$RMS = \sqrt{1/n * \sum_{i=1}^{n} E_i^2} \tag{4.4}$$

Both under- and overestimated values are captured to an aggregated RMS value. Thus, both variability and bias is measured in the RMS score. The aggregated errors for a job is measured in RMS. When job errors are aggregated for all jobs or jobs within a workload category, the arithmetic mean of the RMS errors of those jobs is used.

Chapter 5

Results

This results chapter evaluates the prediction accuracy of LMT-based prediction compared to GDP. First, LMTs predicting IPC directly are evaluated. These accuracies are compared to GDP at a high level but also in more specific cases. LMTs predicting IPC has results for both regression evaluation and simulator evaluation. This is the only usage of simulator evaluation, as the LMTs in the other performance models have only regression evaluation. Secondly, the results of using LMTs for stall predictions in the GDP performance model is presented. Thirdly, results with LMTs for latency predictions as an input to GDP is shown. Lastly, there is a sensitivity analysis looking at how the results of LMTs change with different LMT setups. To briefly summarize the different models using LMTs, the setups are listed:

LMT-IPC: Predicts IPC Directly using LMTs.

LMT-Stall-1: Predicts shared memory system load (SMS-load) stall cycles with LMTs, but is otherwise identical to GDP

LMT-Stall-2: Predicts SMS-load stall cycles with LMTs, and estimates $\hat{S}_p^{other}$ from the GDP performance model by linear regression.

LMT-Stall-3: Predicts the stall cycles from $\hat{S}_p^{SMS}$ and $\hat{S}_p^{other}$ by a LMT, combined. Otherwise uses the performance model of GDP.

LMT-Lat: Uses LMTs to predict latency which is used by GDP.

5.1 IPC prediction

The final output of private mode performance accounting is IPC estimates. This section shows results using LMTs predicting IPC directly. First, LMT-IPC and GDP are compared using regression evaluation with both the unbalanced and the balanced training set for the LMT. Later, LMT-IPC and GDP are compared using the unbalanced training set and simulator evaluation.

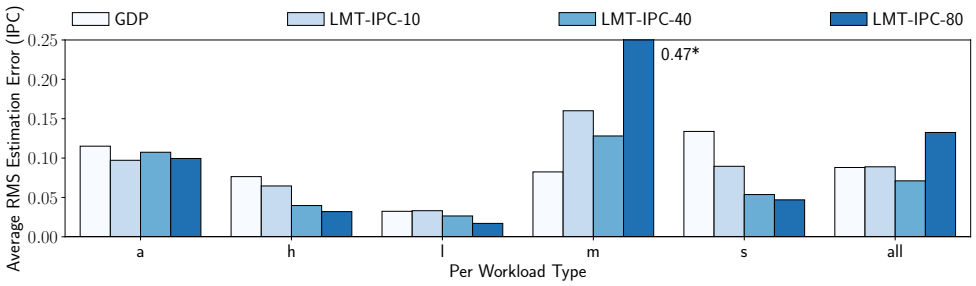


Figure 5.1: Average RMS workload error for IPC in regression evaluation

5.1.1 Regression evaluation

Using linear model trees to predict IPC directly gives accurate private mode performance estimates. Figure 5.1 shows the estimation errors per workload compared to GDP. For all the workloads combined, the 10 leaf node LMT-IPC has 1.0% larger errors compared to GDP. The 40 leaf node LMT-IPC reduces this error by 19%, while the 80 leaf node tree increases the error by 51%. This can also be analyzed per workload. For A workloads, the LMT-IPC regression reduces the prediction errors compared to GDP by 16%, 7% and 14% for the 10, 40 and 80 leaf node configurations. The corresponding numbers are 16%, 48% and 58% for H workloads, and 33%, 60% and 65% for S workloads. For L workloads the predictions are 2% worse than GDP for the 10 leaf node tree, 18% better than GDP for the 40 leaf node tree and 47% better than GDP for the 80 leaf node tree. However, it is worth pointing out that the GDP errors for L workloads are much smaller than for other workloads, so the LMT prediction errors are not that huge in absolute values. For M workloads, LMT-IPC with 10, 40 and 80 leaf nodes perform 94%, 55% and 466% worse than GDP.

The results within each category matches the expectations of the regression, except for the M and A workloads. Within H, L and S workloads the trees gives a higher accuracy for larger trees and have significantly better predictions than GDP, at least for large tree sizes. However, within the M workload the errors surprisingly increase for larger tree sizes and the regression has worse IPC predictions than GDP. Within A workloads, a single benchmark causes the majority of the error. This is facerec, a benchmark also within the M workload category. To understand why this happens, the M workloads are examined with a closer look. Figure 5.2 shows the average RMS error per benchmark for the M workloads of the test set. It becomes clear that some jobs have very large errors, with increasing errors for larger trees. This indicates that the model is *undertrained* for some of the classifications appearing in M workloads. In other words, the training set do not contain enough information or have enough data points to properly train the LMT.

Figure 5.3 shows IPC predictions and the measured private IPC per cumulative committed instructions for the 80 leaf node LMT with the parser benchmark in M workload number 8. It also shows how the linear model tree have fairly

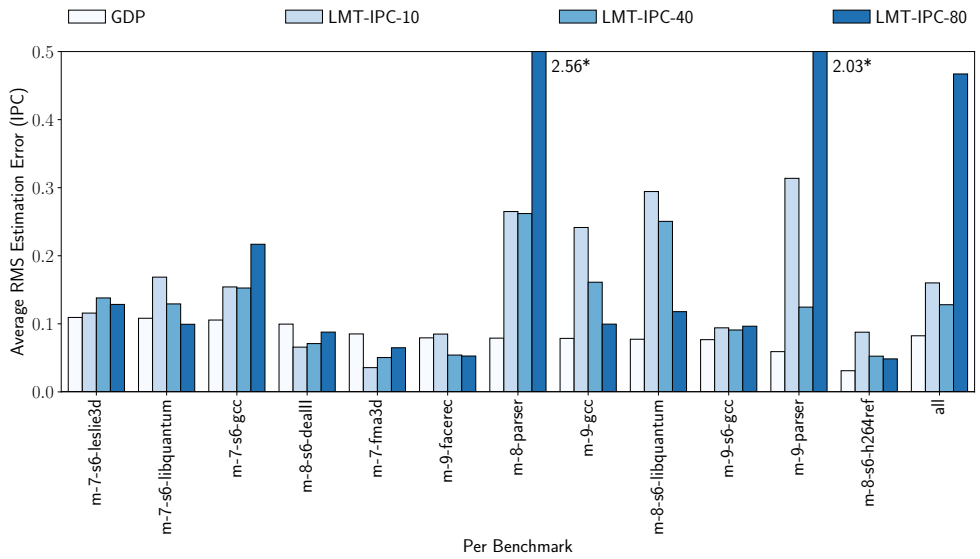


Figure 5.2: Per benchmark errors for M workloads

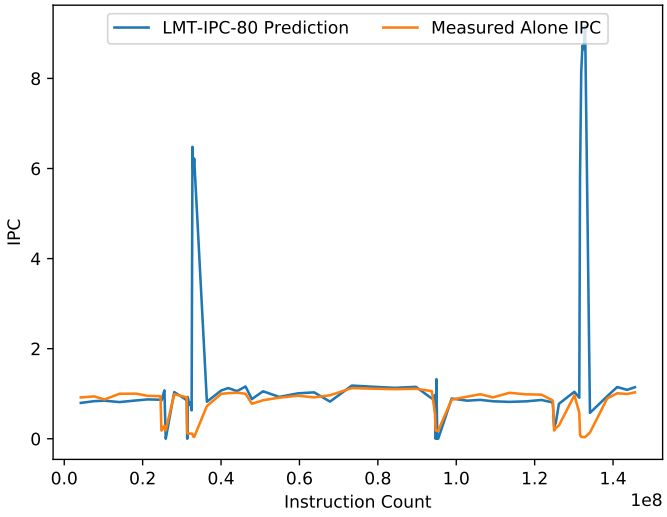


Figure 5.3: IPC prediction and measured IPC for m-8-parser

good predictions besides huge spikes in the IPC predictions, predicting an IPC of around 6 and 9 for short periods of time. This confirms that the linear model tree is undertrained for the classification in that period. If the linear regression in some leaf node classifications are undertrained, the planes generated by the linear regression may be unstable and thus give huge IPC mispredictions in some cases. The predictions for M workloads would most likely yield better results with more training data within the workload. However, the LMT-IPC is able to capture the general trends. Having spikes with huge misprediction is better than consistent mispredictions over time. If used by a policy, the "wrong" behavior would have been corrected quickly. Most likely the next time the policy did rescheduling.

The found outliers of IPC prediction in the M workloads points to an interesting aspect. It seems like when the LMT works, having more nodes in the tree gives better results. However, when the training data is poor for some behavior, adding nodes to the LMT makes the linear regression increasingly unstable and thus the prediction errors get worse.

5.1.2 Balanced training set

With more data points for H and S workloads in the training set, the imbalanced data could have a negative impact of the performance within L and M workloads, making those workloads in the training set less significant. An imbalanced training set indirectly reduces the misprediction penalties in the parts of the training set with little data, and increases the misprediction penalties in the parts with more data. Good training data is vital for getting accurate results with a trained model. To ensure balanced training data with enough samples for regression in every leaf node, huge training sets are required. Too few samples in the leaf nodes give unstable linear regression, and is the cause of increasing errors for larger trees in some workloads. A way to ensure enough samples for training is using cross validation. This can also prevent overfitting. Although, with the massive constraints on tree sizes put on the LMTs for our purposes, overfitting seems very unlikely.

Hence, balancing the training data can reduce errors in L and M workloads. Also, in many systems processing with little congestion in the memory system is often very common. Therefore it is interesting to look at the corresponding results

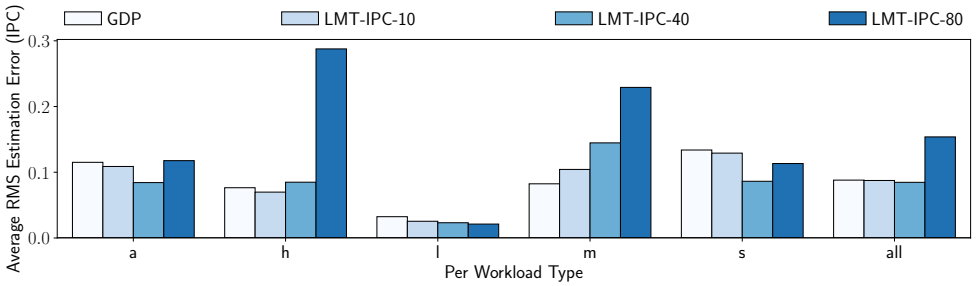


Figure 5.4: Average RMS workload error with balanced training data

using a balanced training set although the overall errors are expected to increase, specially within H and S workloads. Figure 5.4 shows the average RMS errors for the balanced training set. Several interesting points can be read of the results with the balanced training set.

First, the results within L and M workloads are slightly improved, specially in the small tree sizes. This is because the L and M workloads are relatively more important in the balanced training set. With LMT-IPC of 10, 40 and 80 leaf nodes, the L workload accuracy is 21%, 28% and 35% better than GDP. This shows that LMTs perform well compared to GDP also if the expected behavior consist of mostly little congestion in the memory system. LMT-IPC is still struggling in M workloads, but the prediction accuracy is better with fewer outliers.

Secondly, as the training data has a lower quality, the benefit of adding nodes in the tree diminishes. The accuracy benefits of adding nodes in the tree for A, H and S workloads become very limited, and sometimes makes linear regression unstable and increases the errors. This shows the remarkable accuracy benefits of adding training data within H and S workloads. For all LMT-IPC combined, the prediction errors are 69% lower on H workloads in the unbalanced training set and 42% better combined for S workloads. Looking at how the prediction errors are lowered for H and S workloads with extra training data in the unbalanced training set, a similar accuracy improvement is likely with more training data for M workloads. This strengthens the claim that M workload errors would have been lowered with more and better training data. The M workload errors would most likely have improved similar to the H and S workloads in the unbalanced training set. Hence, the overall accuracy of the prediction can be made better with a larger balanced training set without increasing the area cost in hardware.

5.1.3 GDP and streaming benchmarks

As the LMT-IPC have a higher predictior error than GDP in M workloads, GDP have significantly higher errors than the LMT-IPC in S workloads. Figure 5.5 shows the

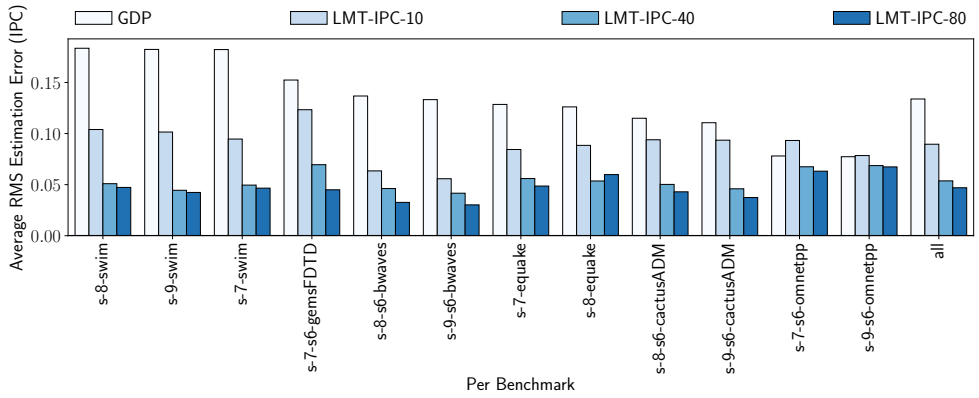


Figure 5.5: Per benchmark errors for S workloads

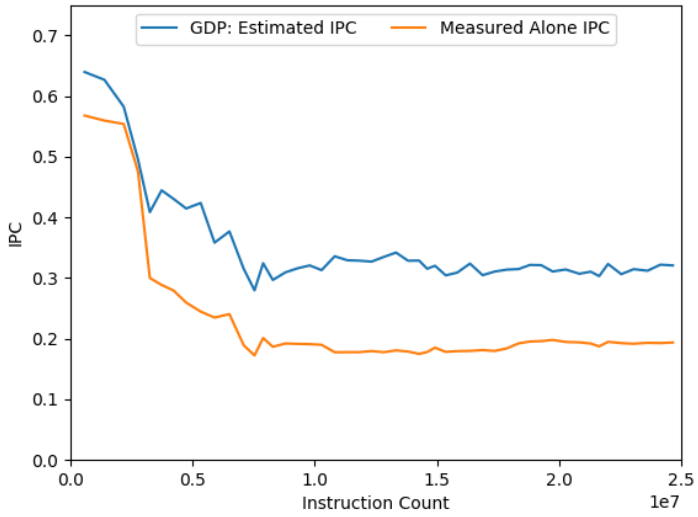


Figure 5.6: IPC prediction and measured IPC for s-0-lucas0

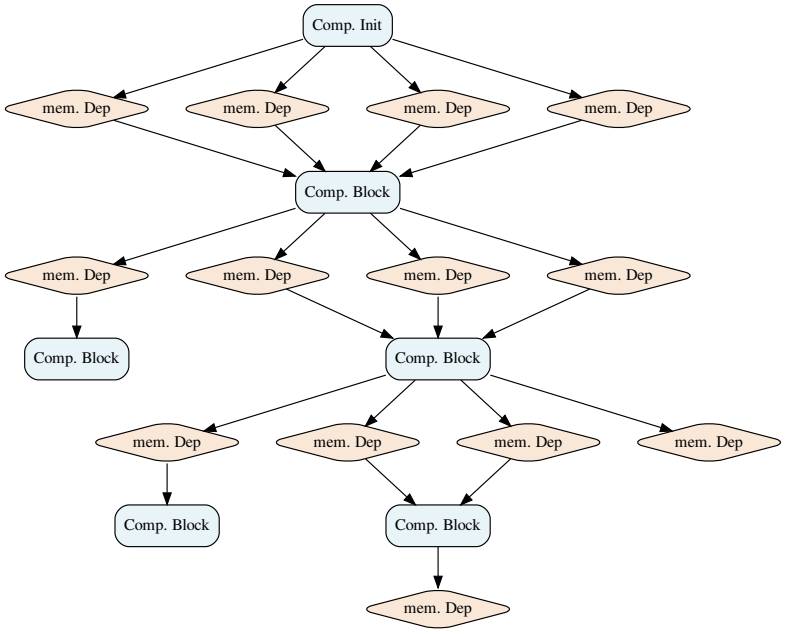
per benchmark error within S workloads using the unbalanced training set for LMT-IPC. It becomes clear that for some benchmarks GDP consistently have huge mispredictions.

Examining the GDP accuracy on the training set as well, it becomes clear that GDP systematically overestimates IPC for some S workload benchmarks. One of these is lucas0. Figure 5.6 shows how GDP consistently overestimates IPC for lucas0 over most of the execution.

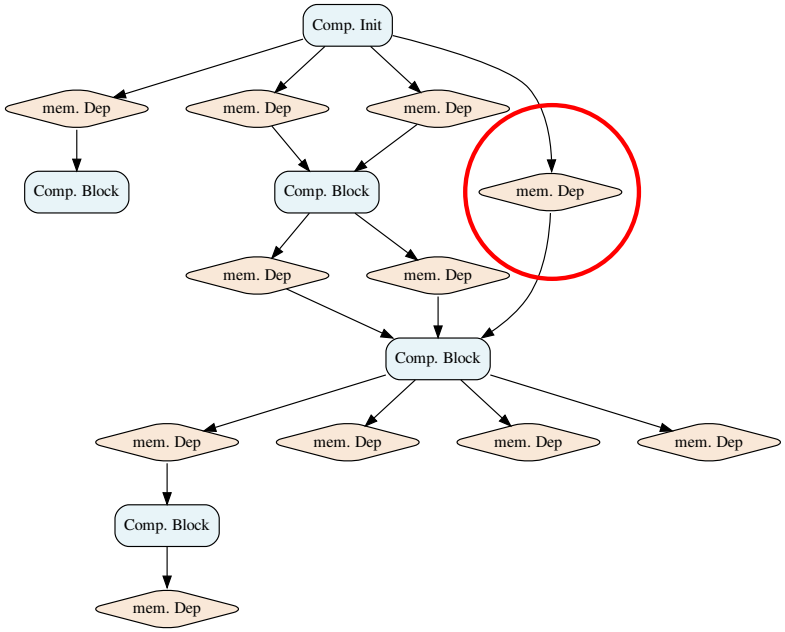
The overestimating behavior of GDP was examined more thoroughly. GDP has a CPL estimator, where CPL and average memory latency estimates are the key components of how GDP predicts SMS-stall cycles. Figure 5.7a visualizes part of the private mode dataflow graph of compute blocks and memory dependencies. The complete dataflow graph shows a similar ordered pattern between dependencies and compute cycles, in which the CPL is estimated. However, in shared mode the dataflow graph is much messier. Figure 5.7b shows a snippet of the shared mode dataflow graph. Note how a memory dependency makes a shortcut past a compute block in the dependency graph. The congestion in the memory system in shared mode leads non-overlapping memory dependencies in private mode to overlap in shared mode. Hence, the calculated CPL in shared mode will be shorter than the CPL in private mode. This consistent underestimation of CPL causes the IPC estimates to consistently be too high.

5.1.4 Simulator evaluation

To verify the promising results from the regression evaluation, the LMT-IPC models trained on the unbalanced training set were exported to the cycle accurate



(a) private mode



(b) shared mode

Figure 5.7: Snippet of dataflow graph of lucas0 in private and shared mode

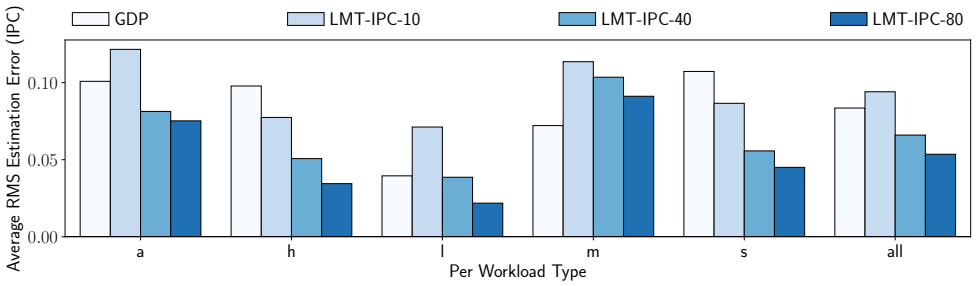


Figure 5.8: Average RMS workload errors for IPC in simulator evaluation

M5 simulator for a simulator evaluation[1]. The hardware structure proposed in section 3.2 was modeled in the M5 simulator, to represent the LMTs. The 10, 40 and 80 leaf node LMT-IPCs were tested together with GDP on a completely new set of workloads. Figure 5.8 shows the results from the simulator evaluation per workload, and the combined performance for all workloads. Overall, the results correspond very well with the findings from the regression evaluation. The small differences is well inside expected behavior variation of different benchmarks within a workload category.

The pattern where the LMTs are unstable for M workloads is the same, although in the simulator evaluation the predictions become slightly better with increasing tree sizes. Removing cumulative committed instructions made regression evaluation worse for M workloads, but better in most other cases. This somewhat explains the relatively better M workload predictions and worse L workload predictions in the simulator evaluation. The S workloads are still the category where GDP struggles the most, but the predictions are slightly better than in the regression evaluation.

[1]The simulator evaluations contains Cumulative Committed Instructions as a feature. This was later removed to prevent the regression model of knowing how long into a benchmark the running application is. The simulator evaluation was not re-run, without the feature. Running the workloads can take weeks, and the general pattern in the results of regression evaluation stayed the same

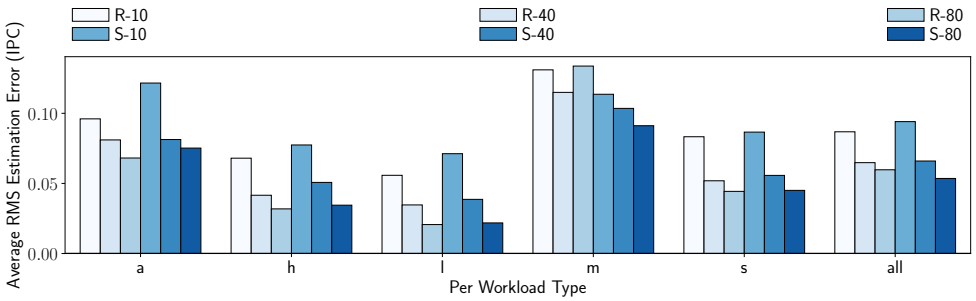


Figure 5.9: Average RMS workload errors for LMT-IPC in regression and simulator evaluation

Considering the overall performance of all the categories, the results correspond very well with the regression evaluation. Figure 5.9 shows results for LMT-IPC in simulator evaluation (S) and regression evaluation (R) when cumulative committed instructions was included in regression evaluation. There are some small variances between them, but note how the accuracies are almost identical overall. The key takeaway from simulator evaluation is that LMTs in private mode performance prediction is absolutely viable also implemented in a processor. The results from the simulator evaluation confirm the results from regression evaluation. The correspondence between the results in regression and simulator evaluation also shows how regression evaluation is suitable to find promising LMT configurations in a more lightweight manner.

In any way, the results from the simulator evaluation verifies the results from the regression evaluation. This again confirms that LMTs are suitable for private mode performance prediction, and that the predictions can be calculated at runtime in a processor. The results for the LMT-IPC prediction also points to some flaws of the training data for the LMT-IPC, where an improvement of the training data inevitably will better the accuracy of the LMT-IPC.

5.2 Stall prediction

A major strength of the GDP performance model is isolating parameters that are the same in shared and private mode, and then only estimating some key stall metrics. The most important of those is shared memory system-load stalls (SMS-load stalls). Thus, estimating SMS-load stalls with LMTs can be a useful input to some performance model. Figure 5.10 shows the SMS-load stall prediction error of GDP and LMTs. Compared to GDP, the relative accuracy of the stall predictions are much better than IPC predictions in some categories, and worse in other categories. Namely, for 10, 40 and 80 LMTs the GDP error is reduced by 44%, 58% and 73% in S workloads, 12%, 36% and 43% in H workloads and 5%, 24% and 27% in A workloads. In M workloads the 10 and 40 leaf node LMT perform 40% and 7% worse than GDP where the 80 leaf node LMT performs 15% better. However, for L workloads the 10, 40 and 80 LMTs perform 245%, 213% and 166% worse than

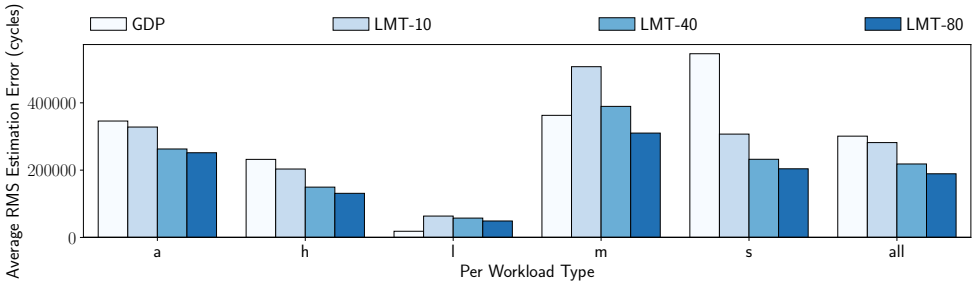


Figure 5.10: Average RMS workload errors for stall predictions

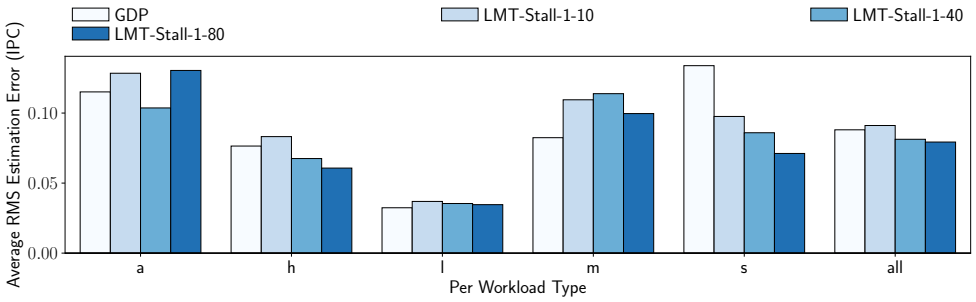


Figure 5.11: Average RMS workload error for IPC predictions, using LMT-Stall-1

GDP. Although, the absolute errors for L workloads are still quite small. This section shows results where these SMS-load stall estimates are used in a performance model to examine the resulting IPC estimation accuracy.

LMT-Stall-1: One way of using SMS-load stall estimates from LMTs is straight into the GDP predictions. Figure 5.11 shows the average IPC error per workload for GDP and GDP using a LMT for its SMS-load stall predictions. Several interesting points can be read from these results. First, using the exactly same predictions for $\hat{S}_p^{other}$, the relative accuracy of the IPC prediction and SMS-load stall predictions does not correspond. The L workload errors are mostly the same for the IPC prediction, while they are considerable larger in the SMS-load stall predictions. This tells us that much of the L workload error of GDP comes from the $\hat{S}_p^{other}$ estimates, or that some assumption for the performance model is inaccurate. For A, H and S workloads the IPC accuracy is somewhat improved but the relative change is much smaller than the improved accuracy for SMS-load stall prediction. Within the M workload, the combined model performs worse than GDP alone.

Using GDP with SMS-load stall estimates from LMTs can give better predictions than GDP alone. Specially if the expected usage of a processor contains much S workload behavior. However, this combination will not reduce the area overhead of GDP. Most of the area overhead in GDP is in the latency estimates provided by DIEF. Those are needed predicting $\hat{S}_p^{other}$, also used for these predictions. Although, removing the CPL estimator can give minor area reductions using LMTs instead. However, the accuracy of GDP with LMT SMS-load stall estimates is not as accurate as predicting IPC directly with linear model trees, with a higher area cost. This is because some errors can be magnified when propagated in the performance model, making the RMS metric worse due to having larger penalties on larger errors. Also, GDPs predictions of $\hat{S}_p^{other}$ also contains errors which are included. Therefore, this configuration is less attractive than predicting IPC directly with LMTs.

LMT-Stall-2: To make a combined performance model attractive, the area overhead compared to GDP has to be reduced. A way to do this is estimating $\hat{S}_p^{other}$ through linear regression. Then, the performance model does not need the DIEF latency estimates and can thus reduce the area overhead much more. Figure

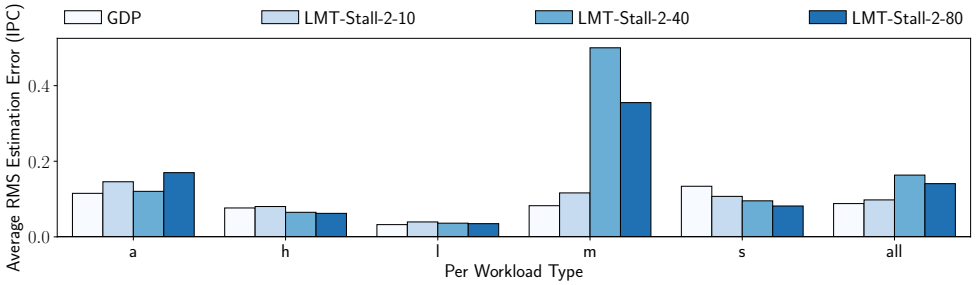


Figure 5.12: Average RMS workload error for IPC predictions, using LMT-Stall-2

5.12 shows the average RMS error for IPC estimates using the GDP performance model with LMTs for SMS-load stalls and linear regression for $\hat{S}_p^{other}$ estimates. This configuration would have approximately the same area overhead as predicting IPC directly by LMTs. However, the accuracy of the model is not as competitive as predicting IPC directly. It struggles in the L and M workloads, but has better results than GDP in the H and S workloads. Modeling $\hat{S}_p^{other}$ with linear regression alone seems to be too simple to effectively estimate the value properly. Specially within M workloads this become clear, giving overall results worse than LMT-Stall-1. The overall performance of LMT-Stall-2 is in general just under the accuracy of GDP, where there are only small benefits of adding nodes to the LMT.

LMT-Stall-3: Our third approach to using LMTs in a hybrid performance model based on the GDP performance model, is combining $\hat{S}_p^{SMS}$ and $\hat{S}_p^{Other}$ to a single stall estimate which is predicted by a LMT. The performance model isolates everything that stays the same in shared and private mode, where LMTs are used to predict the combined changing stall cycles. Figure 5.13 shows the results of this configuration compared to GDP. Overall, the accuracy is very similar to LMTs predicting IPC directly. To see the prediction accuracy compared to LMTs predicting IPC directly, figure 5.14 shows the average RMS errors for the workloads compared to LMT-IPC predictions. Interestingly, the M workload predictions are better with 10 leaf nodes than for predicting IPC directly. Otherwise, they follow similar patterns although using LMTs for stall predicates give higher errors in A and S workloads. Overall, predicting IPC directly gives slightly more accurate predic-

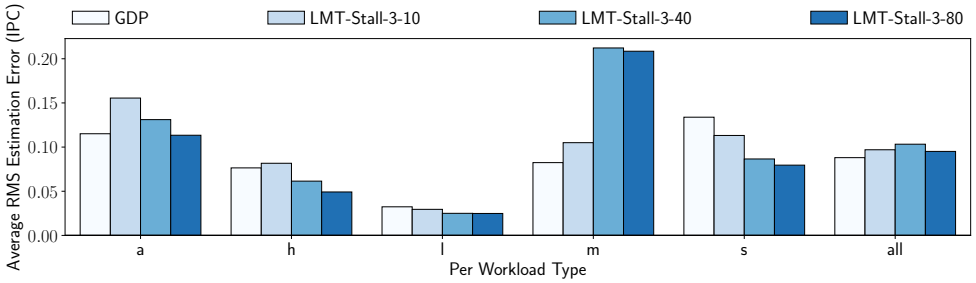


Figure 5.13: Average RMS workload error for IPC predictions, using LMT-Stall-3

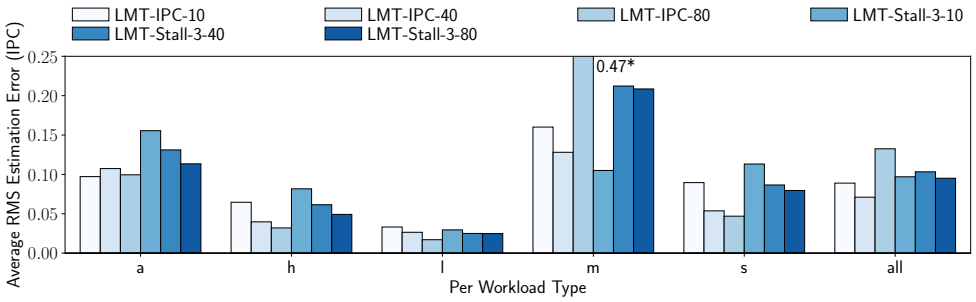


Figure 5.14: Average RMS errors of LMT-IPC and LMT-Stall-3

tions for the same tree sizes. LMT-Stall-3 seems to work effectively in many cases but outliers in the predicted data may be magnified in the performance model also here, giving relatively worse RMS scores.

5.3 Latency Prediction

DIEF is the main contributor to area overhead in GDP. Replacing it with a more lightweight latency predictor could reduce the GDP area cost significantly. Figure 5.15 shows the accuracy errors per workload category for DIEF and LMT latency predictions. The overall results of LMTs are similar to DIEF. However, LMTs perform better within H and S workloads. Similarly, DIEF is better for L and M workloads. The drawbacks of this depends somewhat on the expected use of a processor. Section 5.1.2 discusses how better training data and a balanced training set can improve the accuracy within L and M workloads. This is applicable also for latency predictions.

Figure 5.16 shows IPC estimation errors using both DIEF and LMTs for the latency estimates in GDP. The general trends from the latency predictions also occur for the IPC predictions. However, for L workloads the relative difference is much less than for latency predictions alone. With little congestion in the memory system, a greater part of the error can come from the CPL estimation and the model

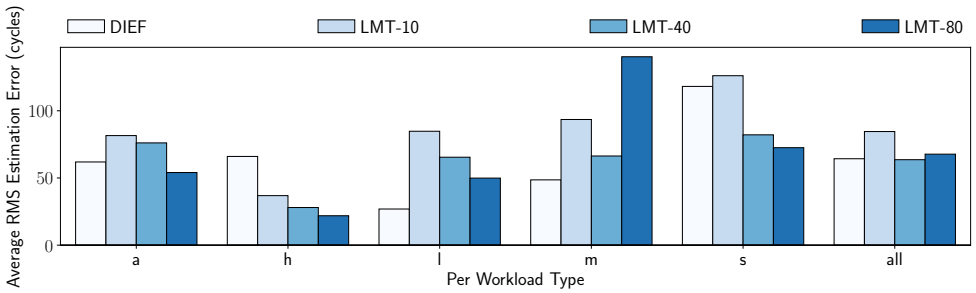


Figure 5.15: Average RMS workload latency estimation errors

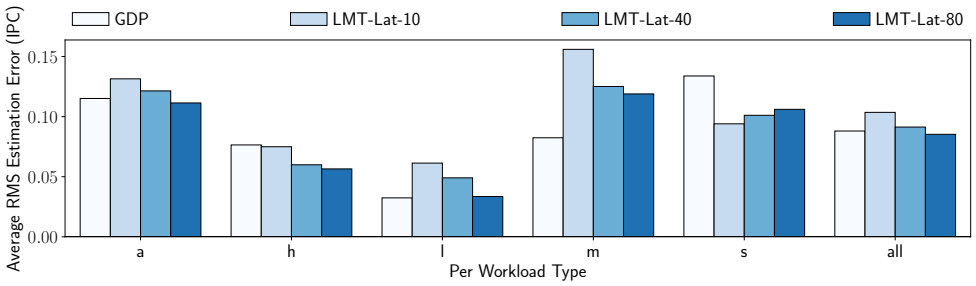


Figure 5.16: Average RMS workload errors for GDP with DIEF and LMT-lat

not able to abstract the complete system behavior. For A, H, M and S workloads, GDP with LMTs tend to give slightly worse IPC predictions than expected from the latency predictions. A reason for this might be outliers in the latency predictions magnified in the GDP performance model. Although LMTs can have good predictions in general, they are more prone to have outliers due to misclassifications in the tree or unstable planes in the linear regression.

Using LMTs for latency prediction can reduce the are overhead of GDP significantly while preserving most of the accuracy. However, predicting IPC directly or stall estimates in a combined performance model seems like a better option considering the best accuracy using the same area.

5.4 Sensitivity analysis

The accuracy of the LMTs rely on many aspects. The most notable are probably training data, available features, the tree structure and the linear regression. Besides the training data, these parts of the system can be altered using less or more area to make the predictions less or more accurate. If implemented in a processor, the subparts of the LMTs should be optimized so they together give the best accuracy for the area used predicting. In the end there is a tradeoff between accuracy and area, where the area is best spent on the components giving the highest accuracy improvements. Adding nodes to the tree adds accuracy, but with diminishing returns. Adding features used for linear regression adds accuracy but also with diminishing returns. This is the case for all adjustable parts of the LMT where the area resources spent in any part of the tree should be tuned to overall giving the best predictions.

This section gives a sensitivity analysis altering parts of the LMT subsystems to see how it will change the IPC prediction accuracy. As the LMT predicting IPC directly has the best results, it is used as the baseline for the sensitivity analysis.

5.4.1 Number of features for linear regression

The accuracy of the linear regression depends on the number of features available for the regression. Adding features will add precision if the information is not yet

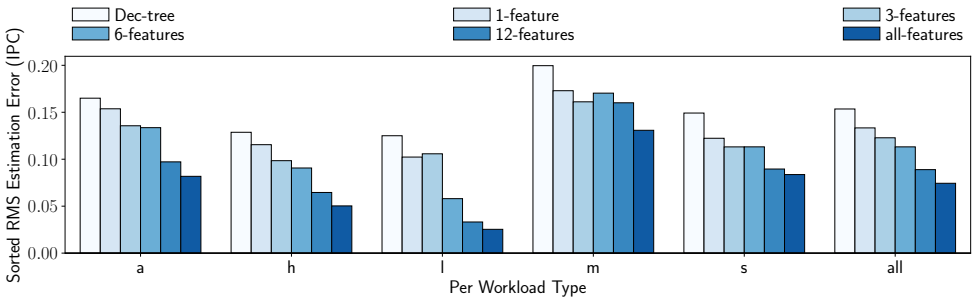


Figure 5.17: Average RMS workload errors for varying feature sizes in a 10 leaf node LMT-IPC

contained in other features or linear combinations of features. However, it is not clear exactly how much of this accuracy relies on having many features for the linear regression. Therefore, the number of features used for linear regression was varied to check how much this altered the results in the test set. Reducing the number of features used for linear regression will reduce the storage overhead for the linear regression weights in the leaf nodes of a linear model tree. The test set was rerun with 1, 3, 6, 12 and all features for linear regression in the linear model tree with 10 leaf nodes. These results are displayed in figure 5.17, including a decision tree (A LMT with a constant in the leaf nodes instead of a linear regressor) for comparison.

As expected, using more features for linear regression yields better results. Using all features takes away more than half of the error in a decision tree. In general, increasing the number of features used also increases the accuracy, with only a few minor exceptions. Although the twelve best features were selected iteratively using R^2 scores for linear regression on all training data, it becomes clear that the accuracy can be improved further. This is caused by different features being important for linear regression in different subpopulations of the trees. However, using all 25 features only makes the predictions a bit more accurate than using 12 features. As the storage overhead for linear regression weights is doubled, the improved accuracy is 16% better than using 12 features. A way to take advantage of different features being important in different subpopulations is using different features for linear regression in the different leaf nodes. The iterative R^2 analysis could be repeated for every leaf node. However, this would also make evaluation more complex and add area.

Depending on the tree size, adjusting the amount of features used might yield better results for the same area. We have examined the 10 leaf node tree. Using 3 features for linear regression here instead of 12 features increases the prediction error of 44%. However, it reduces the storage overhead of weights for linear regression by 60%. This area could be spent making the tree larger, and perhaps resulting in more precise predictions at the same area cost. Such aspects should be considered choosing the LMT to be used in a specific implementation.

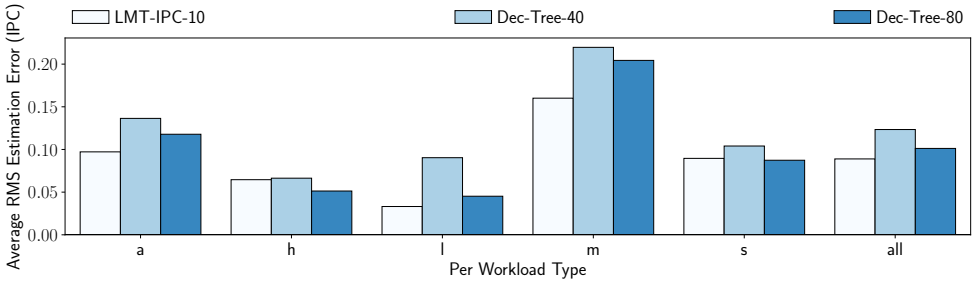


Figure 5.18: Average RMS errors for LMT and regular decision trees

Decision tree vs Linear Model Tree

The linear regression weights are one of the major contributors to area overhead in LMTs. Decision trees can be used without linear regression in the leaf nodes at a much lower area cost. Figure 5.18 shows the average RMS prediction error for chosen tree sizes of LMT and decision trees. They are chosen to examine the prediction accuracy of LMTs and decision trees with similar area overhead. Focusing only on the tree, the storage overhead of the 10 leaf node LMT is 471 Bytes per core. The storage overhead of the decision trees are 419 Bytes per core for the 40 leaf node tree, and 789 Bytes of the 80 leaf node tree. Although the decision trees perform slightly better in H workloads, the overall performance is worse than the 10 leaf node LMT. Even the 80 leaf node decision tree is less accurate than the 10 leaf node LMT, even though it has a higher area cost. Hence, LMT will always the better option for area efficient private mode performance prediction unless the available area is less than the minimal viable LMT. The exception is worth mentioning, although very accurate predictions cannot be expected at such a low area cost.

5.4.2 Auxiliary Tag Directories

ATDs are needed providing private LLC estimates as input features in the LMTs. However, ATDs cause much area overhead. In small trees, they contribute more

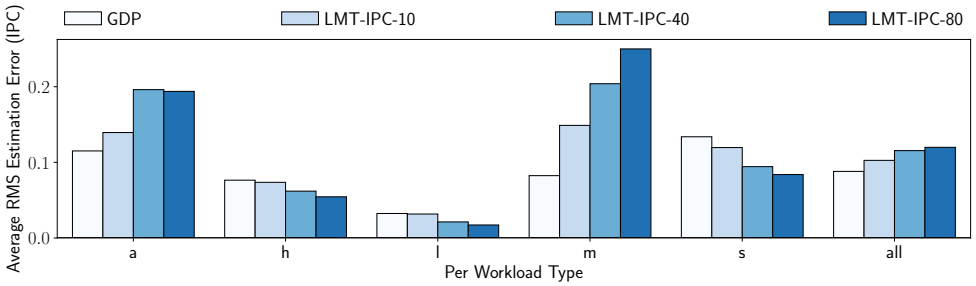


Figure 5.19: Average RMS errors for linear model trees without ATDs

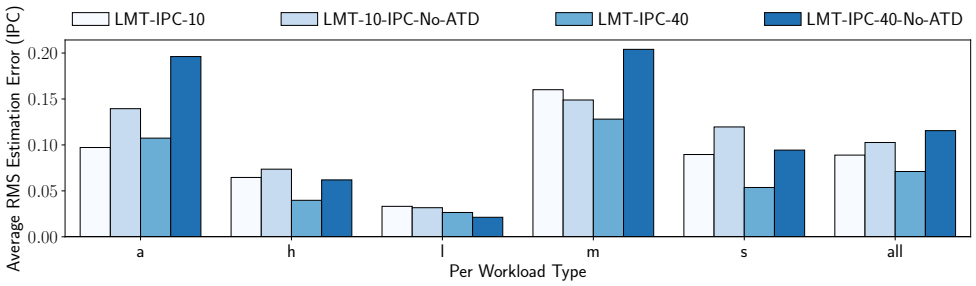


Figure 5.20: Average RMS errors for linear model trees with and without ATDs

than half the area overhead alone. If ATDs are already implemented in processors for other purposes this is not a problem. Otherwise, using LMTs without ATDs can be a way to significantly reduce the area overhead even further. In regression evaluation we have run our test set without the features requiring ATDs. Figure 5.19 shows the average RMS prediction errors of LMTs without ATDs. The general patterns are similar those predicting with ATDs. Although, the errors are slightly larger and adding nodes to the LMT seem to give less accuracy improvements. Notably, the diminishing returns of adding nodes in the tree converges to a higher average RMS estimation error. However, if this error is acceptable, removing ATDs can be beneficial to reduce the storage overhead. As for the LMT-IPC with ATDs, the accuracy in M workloads is expected to be better with more training data.

For larger trees, the area overhead of adding ATDs is less compared to the total overhead. Therefore, removing ATDs is most likely to be beneficial to reduce area for small trees. Figure 5.20 shows the prediction performance of the 10 and 40 leaf node tree both with and without ATDs. The prediction errors without ATDs are in general higher than predicting with ATDs. Quite surprisingly, for L workloads predicting without ATDs is more accurate than predicting with ATDs. The explanation for this is the low congestion in the memory system in L workloads. This brings the private mode behavior of the memory system closer to the shared mode behavior. Hence, where it chooses to split the tree based on private LLC estimates this is a somewhat redundant information in the L workloads. Thus, splitting on other features reduces the variance more for L workloads and then increases the L workload accuracy. Overall, the 10 leaf node LMT without ATDs has an increased prediction error of 17% compared to GDP, while reducing the area overhead by 85%. The 40 leaf node LMT without ATD increased prediction error with 31% compared to GDP, while reducing the area overhead by 39%. These area reductions assume ATDs become redundant if not used for private mode performance prediction.

5.4.3 Upper bound on prediction values

Specially within the undertrained M workloads, the LMT-IPC predict unrealistically high IPC values. However, even in well trained LMTs there are no guarantee

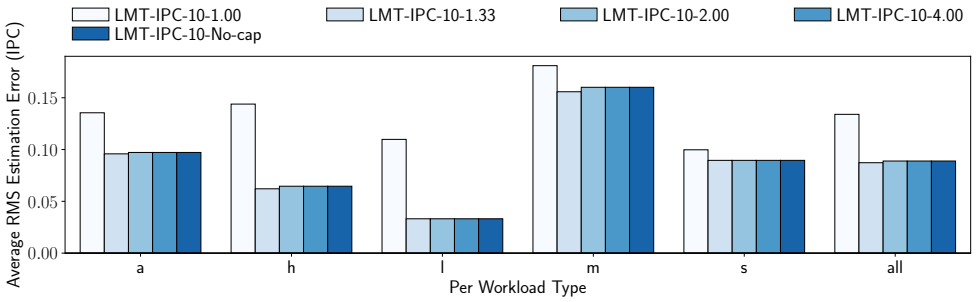


Figure 5.21: Average RMS errors of LMT-IPC-10 with varying upper bound

that misclassifications do not occur leading to outliers and spikes in predicted values. This can be some edge case where some particularly high value make the IPC prediction much higher than the private mode IPC. One way to prevent spikes in the predicted IPC values is through providing upper bounds or caps on how large prediction values can be. Assuming this will improve accuracy, a critical design choice is how large this cap should be. The theoretical maximum IPC for our implemented simulator is 4 (table 4.1). Hence, setting an upper bound of 4 for predictions will never make predicted values too low. However, over five million cycles the IPC is never close to being 4 in our training set. Therefore, having a lower upper bound can improve accuracy even though the theoretical maximum IPC is higher than the upper bound. The LMT-IPC have been tested on the test set, with four different configurations of upper bounds. The upper bound is set to be 4, 2, 1.33 and 1. Additionally, the baseline LMT-IPC with no upper bound is included. The upper bound of 1.33 was chosen because that was the highest measured IPC over an interval in the training set. The other upper bounds are supposed to show how the estimation errors change besides an upper bound of the theoretical maximum and the maximum measured in the training set. The upper bounds are tested on LMT-IPC with 10, 40 and 80 leaf nodes.

10 leaf nodes: The 10 leaf node LMT-IPC with upper bounds are displayed in figure 5.21. First, it becomes clear that the upper bound of 1 is too low and increases the prediction errors. Otherwise, the prediction errors are almost identical. As there are many target values in the training data with an IPC over 1, increased errors with a too low cap is expected. Having upper bounds at 2 and 4 gives the exact same prediction errors as having no cap at all. Therefore it seems like having stable planes in the linear regression make significant error reductions by upper bounds unlikely. This also points to the LMTs having few outliers for well trained trees.

40 leaf nodes: The results for the 40 leaf node LMT-IPC with upper bounds in figure 5.22 shows more interesting results than for the 10 leaf node LMT-IPC, in the A workloads. The upper bound of 1 still predicts much worse than all other configurations, while the upper bound of 1.33 is the best. However, in the A workload category having an upper bound of 4 and 2 gives increasing accuracy com-

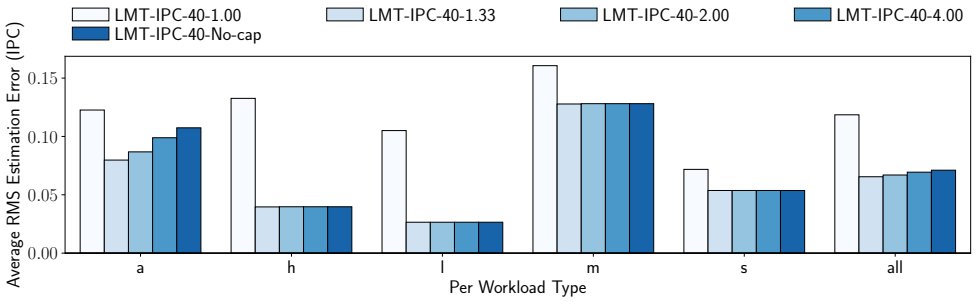


Figure 5.22: Average RMS errors of LMT-IPC-40 with varying upper bound

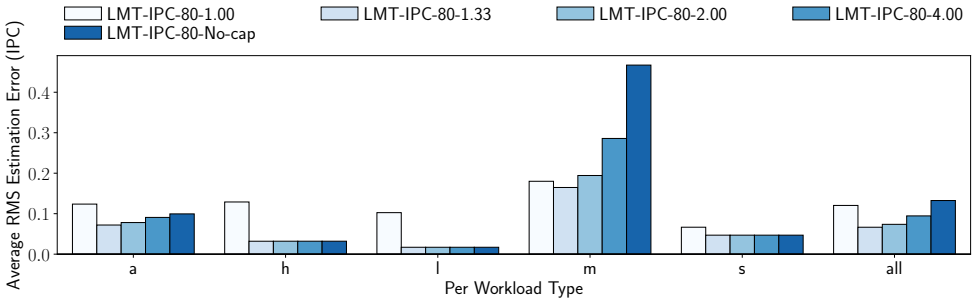


Figure 5.23: Average RMS errors of LMT-IPC-80 with varying upper bound

pared to having no upper bound. Hence, some case in a few data points where huge IPC values are predicted seem to be avoided with the cap. For all other workload categories there are only marginal accuracy benefits with the upper bound of 1.33, compared to having larger upper bounds.

80 leaf nodes: Having upper bounds on IPC predictions give the most interesting results in the 80 leaf node LMT-IPC, displayed in figure 5.23. Clearly, having upper bounds on the prediction values helps correcting some of the errors because of unstable linear regression with many leaf nodes in the LMT-IPC. Specially within M workloads, the upper bounds reduces the errors. With the upper bound of 1.33, the "spikes" are gone and the errors are reduced by 65% in the M workloads compared to having no upper bound. Although, with proper training data there might not be a need for these upper bounds on predictions.

The final notion of comparing LMT-IPC predictions with upper bounds is that they mostly improve misprediction due to undertrained and unstable linear regression. However, the decision tree cannot guarantee that some observation has linear regression weights giving too high predictions. To correct this, having upper bounds on predictions can be a way to gain some small extra percentages of accuracy. Also, if the trees are made very small making outliers in prediction more likely, upper bounds can be a way to prevent large mispredictions. Otherwise, regular spikes in the IPC predictions should indicate that more training data is needed.

Chapter 6

Conclusion and future work

6.1 Conclusion

The main goal of this project was reducing the area overhead of accurate private performance predictions. First, the LMT structure was proposed for private mode performance prediction, explaining why the regression model had qualities making accurate performance accounting likely. In short, the results show that the proposed way of using LMTs work very well by effectively classifying subpopulations with a similar memory behavior, and exploiting linearity in the memory system. Hence, the assumptions made motivating the use of LMTs for performance accounting are viable. The trained LMT-IPC model on the balanced training set reduces prediction error compared to state-of-the-art GDP with 1%, while reducing the storage overhead by 24%. Assuming ATDs are implemented to be used across policies, the additional storage overhead for private mode performance prediction is reduced by 60%. If ATDs become redundant if not used for private mode performance prediction, another LMT-IPC setup is proposed having an error increase of 17% compared to GDP where the storage overhead is reduced by 85%. If accuracy is important, the results also show how the number nodes in the LMT can be extended to reduce prediction errors. For the LMT-IPC with 40 leaf nodes, the prediction errors compared to state-of-the-art GDP are reduced by 19%.

6.2 Future Work

Although the proposed use of LMTs for private mode performance prediction improves the challenges mentioned in the project description, some weak points of the current setup have been exploited as a part of the design space exploration. Within the scope of this project, there was not time enough to examine all possible ways of using LMTs for private mode performance prediction, where simulation evaluation and gathering data often take weeks. Being the first to propose LMTs for that use, several possible ways to improve the model further have come to mind. It is likely that more work and a closer examination of LMTs for private

mode performance prediction can lead to an even better accuracy at the same area cost. Following, possible ways to improve the LMT predictions are discussed.

Better training data: Section 5.1.2 revealed that the accuracy can be significantly improved with more and better training data. Specially within M workloads, the LMT struggle with our test and training set. However, the accuracy improvements were huge for H workloads with additional training data. Hence, this is likely to also be the case for M workloads. Seeing how precise the overall accuracy can be with more and better training data for the LMTs is an obvious place to start improving the method.

Linear Regression MSE as decision tree cost function: A way which most likely better the splits in the tree for our usage is by having the linear regression MSE error as the cost function determining splits in the tree. This is a better option as the LMT error that should be minimized is the MSE after linear regression. This would require making a linear regression model of the data for every optional split in the tree. This would heavily increase the computational intensity of making the tree, but do not alter the LMT evaluation. For the training data, the MSE splits (not linear regression MSE) are an upper bound of accuracy compared to linear regression MSE. Hence, using linear regression MSE is likely to improve prediction accuracy but how much the improvement will be is not clear.

PCA preprocessing to reduce storage overhead: Section 3.3.3 points out how fewer features used for linear regression can much reduce storage overhead. Besides ATDs, the majority of the storage overhead in the trees come from weights for linear regression. A way to retain information used for linear while reducing the number of weights and storage overhead is through dimension reduction. PCA preprocessing maximizes the input variance for a reduced set of dimensions. Examining setups using PCA can be very interesting to see if that can make the accuracy even better for the same storage cost.

GDP performance model: An important quality of the GDP performance model is how it isolates components staying the same in both shared and private mode. Intuitively, estimating only the changing components should give more accurate predictions than estimating all components combined. However, for our setup this is not the case. Although providing stall estimates gives quite good results, predicting IPC directly has better overall results. While the relative accuracy of predicting stall cycles is slightly better than IPC, some error gets propagated in the performance model making the IPC predictions through stall cycles slightly larger. Therefore, it would be interesting to see if the model can be tuned to make IPC predictions via stall cycles more accurate than IPC directly.

IPC predictions without ATD: Although section 5.4.2 shows preliminary results for predictions without ATDs. However, a more thorough examination if predictions without ATDs give sufficient accuracy is needed. The general pattern is that much of the information of the memory system behavior is contained and sufficient to predict accurately. Although, the estimation errors are slightly larger. However, the massive reduction in storage overhead should be considered if ATDs become redundant if not used for private mode performance prediction. It is also

possible that adding features can include some of the information lost by not using ATDs.

Per leaf node feature selection: Section 5.4.1 shows how different features are important in different leaf nodes. The R^2 analysis is currently done on all training data. However, the prediction errors are likely to be reduced by having an individual analysis of which features to use in each leaf node. For the 10 leaf node LMT-IPC, using all 25 features reduces the prediction errors compared to using 12 features by 16%. Using all features will add much storage overhead for the linear regression weights. A way to gain some accuracy while not having to double the storage cost of the weights is by selecting features on a per leaf node basis. However, this will add some extra logic to handle which features are used in each leaf node.

Bibliography

[1] K. J. Nesbit, M. Moreto, F. J. Cazorla, A. Ramirez, M. Valero, and J. E. Smith, "Multicore resource management," *IEEE micro*, vol. 28, no. 3, pp. 6–16, 2008.

[2] M. Jahre and L. Eeckhout, "GDP: Using dataflow properties to accurately estimate interference-free performance at runtime," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2018, pp. 296–309.

[3] K. Du Bois, S. Eyerman, and L. Eeckhout, "Per-thread cycle accounting in multicore processors," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, pp. 1–22, 2013.

[4] *Amazon EC2 pricing*, `https://aws.amazon.com/ec2/pricing/`.

[5] *Microsoft azure, linux virtual machines pricing*, `https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/`.

[6] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, 2006, pp. 423–432.

[7] R. Wang and L. Chen, "Futility scaling: High-associativity cache partitioning," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE, 2014, pp. 356–367.

[8] X. Zhou, W. Chen, and W. Zheng, "Cache sharing management for performance fairness in chip multiprocessors," in *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, IEEE, 2009, pp. 384–393.

[9] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, IEEE, 2007, pp. 146–160.

[10] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems," in *2008 International Symposium on Computer Architecture*, IEEE, 2008, pp. 63–74.

[11] M. Xie, D. Tong, K. Huang, and X. Cheng, "Improving system throughput and fairness simultaneously in shared memory CMP systems via dynamic bank partitioning," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2014, pp. 344–355.

[12] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE, 2010, pp. 65–76.

[13] C. Luque, M. Moreto, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, and M. Valero, "CPU accounting for multicore processors," *IEEE Transactions on Computers*, vol. 61, no. 2, pp. 251–264, 2011.

[14] A. Jaleel, H. H. Najaf-Abadi, S. Subramaniam, S. C. Steely, and J. Emer, "CRUISE: Cache replacement and utility-aware scheduling," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, 2012, pp. 249–260.

[15] R. Das, R. Ausavarungnirun, O. Mutlu, A. Kumar, and M. Azimi, "Application-to-core mapping policies to reduce memory system interference in multi-core systems," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2013, pp. 107–118.

[16] C. Luque, M. Moreto, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, and M. Valero, "Itca: Inter-task conflict-aware cpu accounting for CMPs," in *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, IEEE, 2009, pp. 203–213.

[17] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu, "The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2015, pp. 62–75.

[18] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu, "MISE: Providing performance predictability and improving fairness in shared main memory systems," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2013, pp. 639–650.

[19] P. Salvesen, "Towards low-overhead private mode performance prediction in multicores," NTNU, 2019.

[20] A. Glew, "MLP yes! ILP no," *ASPLOS Wild and Crazy Idea Session*, vol. 98, 1998.

[21] F. J. Cazorla, P. M. Knijnenburg, R. Sakellariou, E. Fernández, A. Ramirez, and M. Valero, "Predictable performance in SMT processors," in *Proceedings of the 1st Conference on Computing Frontiers*, 2004, pp. 433–443.

[22] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A case for MLP-aware cache replacement," in *33rd International Symposium on Computer Architecture (ISCA'06)*, 2006, pp. 167–178.

[23] S. Eyerman and L. Eeckhout, "Per-thread cycle accounting in SMT processors," *ACM Sigplan Notices*, vol. 44, no. 3, pp. 133–144, 2009.

[24] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A mechanistic performance model for superscalar out-of-order processors," *ACM Trans. Comput. Syst.*, vol. 27, no. 2, May 2009, ISSN: 0734-2071. DOI: 10.1145/1534909.1534910. [Online]. Available: https://doi.org/10.1145/1534909.1534910.

[25] T. S. Karkhanis and J. E. Smith, "A first-order superscalar processor model," in *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.*, IEEE, 2004, pp. 338–349.

[26] A. B. Kahn, "Topological sorting of large networks," *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, 1962.

[27] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An introduction to statistical learning*. Springer, 2013, vol. 112.

[28] X. Zhao, M. Jahre, and L. Eeckhout, "HSM: A hybrid slowdown model for multitasking GPUs," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1371–1385.

[29] R. J. Struharik and L. A. Novak, "Hardware implementation of decision tree ensembles," *Journal of Circuits, Systems and Computers*, vol. 22, no. 05, p. 1 350 032, 2013.

[30] S. Lopez-Estrada and R. Cumplido, "Decision tree based FPGA-architecture for texture sea state classification," in *2006 IEEE International Conference on Reconfigurable Computing and FPGA's (ReConFig 2006)*, IEEE, 2006, pp. 1–7.

[31] Y. Xie and G. H. Loh, "PIPP: Promotion/insertion pseudo-partitioning of multi-core shared caches," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 174–183, 2009.

[32] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The m5 simulator: Modeling networked systems," *IEEE Micro*, vol. 26, no. 4, pp. 52–60, Jul. 2006, ISSN: 1937-4143. DOI: 10.1109/MM.2006.82.

[33] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[34] A. P. Magnus Jahre, *Magnus' m5 simulator*, https://github.com/magnusjahre/MM5.

[35] J. Casazza, "*intel core i7-800 processor series and the intel core i5-700 processor series based on intel microarchitecture (nehalem)*", 2009, Published: White paper, Intel Corp.

[36] S. C. w. p. 2007, *Spec*, `http://www.spec.org/cpu2000/`..

[37] J. L. Henning, "2006, SPEC CPU2006 benchmark descriptions," in *SIGARCH Computer Architecture News, vol. 34, no. 4*, pp. 1–17.

[38] G. H. E. Perelman and B. Calder, "Picking statistically valid and early simulation points," in *PACT'03: Proc. of the 12th Int. Conf. on Parallel Architectures and Compilation Techniques, 2003*, p. 244.

[39] J. L. G. Hamerly E. Perelman and B. Calder, "Simpoint 3.0: Faster and more flexible program analysis," in *Journal of Instruction Level Parallelism, 2005*.

Appendix A

Workload Generation

We use 51 SPEC2000 [36] and SPEC2006 [37] benchmarks to generate multi-programmed workloads that contain one benchmark per physical core. Our starting point is to use SimPoints [38], [39] to select a representative 100 million instruction sample for each benchmark. Then, we functionally simulate the benchmark until the start of this sample and take a checkpoint. This checkpoint contains private cache state. Finally, we profile the representative sample of each benchmark while varying the available LLC-ways and off-chip bandwidth. More specifically, we vary LLC capacity by making 1, 2, 4, 8, or all (16) ways available to the benchmark in an 8MB LLC and letting it access between 1/16, 1/8, 1/4, 1/2, or all of the bandwidth of a single-channel DDR4 system.

Based on these profiles, we classify the benchmarks into four categories. We first sort the benchmarks based on their speed-up from the configuration with the most resources (i.e., 16 ways and all of the DRAM bandwidth) compared to the configuration with least resources (i.e., 1 way and 1/16 of the DRAM bandwidth). Then, we classify the 10 benchmarks with the highest overall speed-up and speed-up of more than 1.985 between the 16-way and 1-way configuration as highly memory sensitive (H). After removing these benchmarks, we classify the 10 benchmarks with the highest speed-ups as streaming (S); and we manually verify that they are exhibiting streaming behavior. Of the remaining benchmarks, the 20 with the highest speed-ups are classified as having medium sensitivity (M), and the remaining 11 benchmarks are classified as having low sensitivity (L).

Using this classification, we randomly generate 25 workloads with the H-benchmarks, 25 S- benchmark workloads, 10 M-benchmark workloads, and 10 L-benchmark workloads. To account for heterogeneous workload mixes, we also randomly generate 10 workloads where benchmarks are drawn from all categories (i.e., the A-workloads). We require that a single benchmark appears at most twice in a workload.

Appendix B

Iterative Feature Selection

This appendix is an addition to the selection of features for linear regression 4.5.2. It consists of tables showing every added feature and the corresponding R^2 value for itself and the preceding features combined.

B.1 latency

Table B.1 shows the iterative selection of features to predict latency using linear regression.

Feature	**R^2**
Shared Mode Total Latency	0.39856
Private Mode LLC Writeback Estimate	0.57341
Total Number of Shared Mode Memory Requests	0.64262
Average Shared Mode Latency	0.6643
Aggregated Shared Mode LLC Misses and Writebacks for all cores	0.68296
Average Shared Mode PMS Latency	0.69204
Private Mode LLC Hit Estimate	0.70314
Shared Mode LLC Hits	0.72966
Private Mode LLC Access Estimate	0.75249
Shared Mode LLC Accesses	0.75777
Number of Shared Mode Shared Stores	0.76296
Shared Mode Write Stall Cycles	0.76487
Number of Shared Mode Hidden Loads	0.76542
Shared Mode LLC Writebacks	0.7658
Shared Mode IPC	0.76611
Compute Cycles	0.76801
Number of Shared Mode Write Stalls	0.76814
Shared Mode PMS Stall Cycles	0.76823
Shared Mode Private Blocked Stall Cycles	0.76827
Average Shared Mode Shared Store Latency	0.76828
Shared Mode Empty ROB Stall Cycles	0.76829
Shared Mode Stall Cycles	0.76829
Memory Independent Stall Cycles	0.76829
Summarized Shared Mode LLC Misses and Writebacks	0.76829
Total Shared Mode PMS and SMS Stall Cycles	0.76829

Table B.1: Iterative feature selection using R^2 for latency

Feature	**R^2**
Private Mode LLC Writeback Estimate	0.16059
Shared Mode PMS Stall Cycles	0.26265
Shared Mode Total Latency	0.38938
Average Shared Mode Latency	0.49273
Shared Mode Stall Cycles	0.53165
Aggregated Shared Mode LLC Misses and Writebacks for all cores	0.5503
Average Shared Mode PMS Latency	0.56205
Shared Mode IPC	0.56768
Compute Cycles	0.5731
Summarized Shared Mode LLC Misses and Writebacks	0.57872
Shared Mode LLC Writebacks	0.61006
Total Number of Shared Mode Memory Requests	0.62421
Private Mode LLC Hit Estimate	0.66981
Private Mode LLC Access Estimate	0.76084
Number of Shared Mode Hidden Loads	0.76665
Number of Shared Mode Write Stalls	0.76802
Shared Mode LLC Accesses	0.76914
Memory Independent Stall Cycles	0.77006
Shared Mode Write Stall Cycles	0.77084
Average Shared Mode Shared Store Latency	0.77119
Number of Shared Mode Shared Stores	0.77141
Shared Mode Private Blocked Stall Cycles	0.77145
Shared Mode Empty ROB Stall Cycles	0.77197
Total Shared Mode PMS and SMS Stall Cycles	0.77197
Shared Mode LLC Hits	0.77197

Table B.2: Iterative feature selection using R^2 for SMS-load stall cycles

B.2 SMS-load stalls

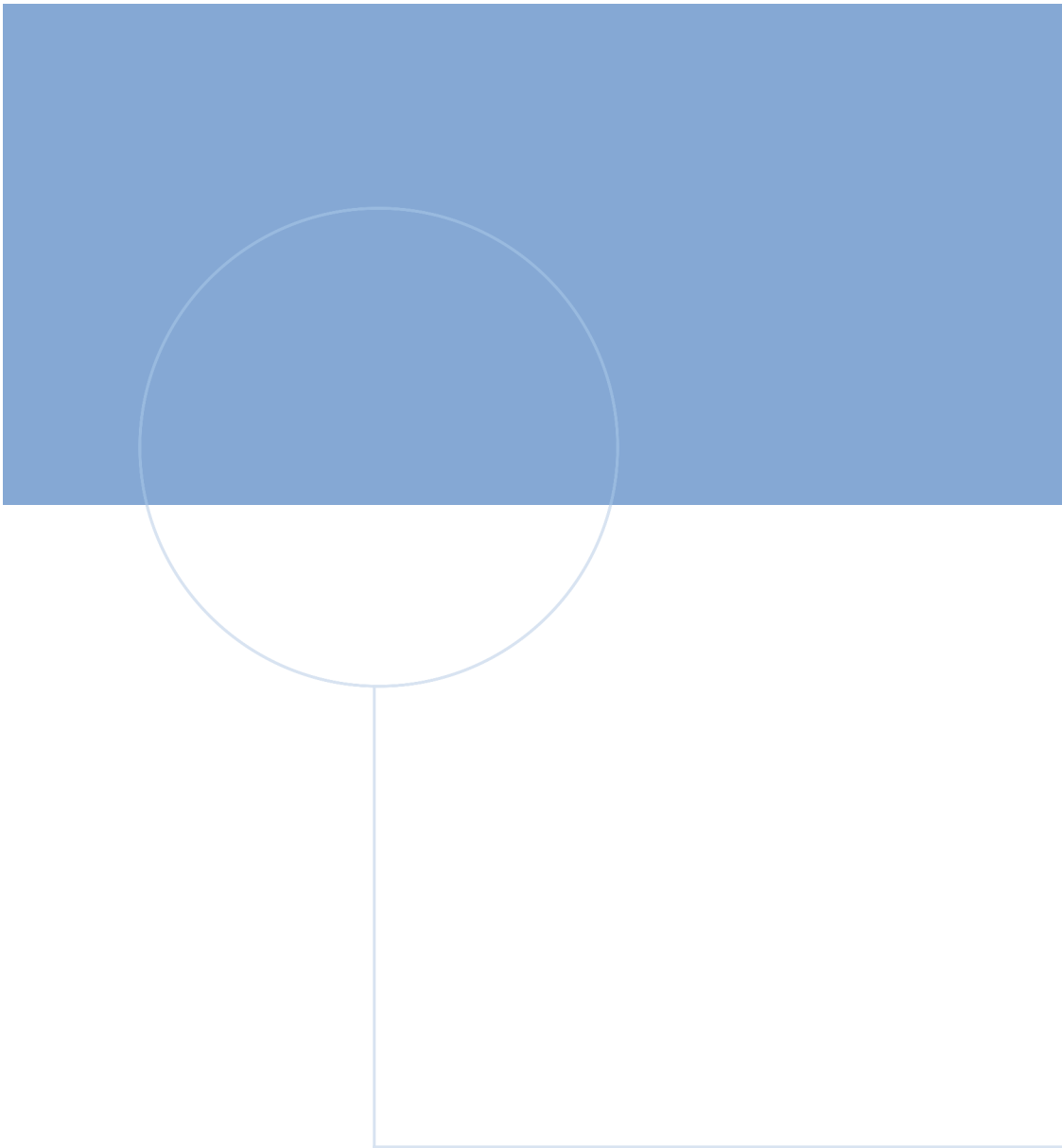
Table B.2 shows the iterative selection of features to predict SMS-load stall cycles using linear regression.

Feature	R^2
Shared Mode IPC	0.58773
Private Mode LLC Writeback Estimate	0.63798
Shared Mode Total Latency	0.65545
Shared Mode PMS Stall Cycles	0.66992
Aggregated Shared Mode LLC Misses and Writebacks for all cores	0.6763
Compute Cycles	0.6803
Memory Independent Stall Cycles	0.68991
Shared Mode LLC Accesses	0.70016
Private Mode LLC Hit Estimate	0.75875
Shared Mode LLC Hits	0.76731
Private Mode LLC Access Estimate	0.77499
Average Shared Mode PMS Latency	0.78171
Average Shared Mode Shared Store Latency	0.7845
Total Number of Shared Mode Memory Requests	0.78698
Shared Mode Stall Cycles	0.78898
Number of Shared Mode Shared Stores	0.78988
Shared Mode LLC Writebacks	0.79041
Shared Mode Write Stall Cycles	0.79069
Number of Shared Mode Hidden Loads	0.79081
Shared Mode Private Blocked Stall Cycles	0.79091
Average Shared Mode Latency	0.79097
Number of Shared Mode Write Stalls	0.79103
Shared Mode Empty ROB Stall Cycles	0.79108
Total Shared Mode PMS and SMS Stall Cycles	0.79108
Summarized Shared Mode LLC Misses and Writebacks	0.79108

Table B.3: Iterative feature selection using R^2 for IPC

B.3 IPC

Table B.3 shows the iterative selection of features to predict IPC using linear regression.



NTNU

Norwegian University of
Science and Technology