UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

Low-power architectures for automatic speech recognition

Hamid Tabani

Low-Power Architectures for Automatic Speech Recognition

Hamid Tabani

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Doctor of Philosophy

Department of Computer Architecture
Universitat Politècnica de Catalunya

Advisors: Jordi Tubella, Antonio González

January, 2018
Barcelona, Spain

Abstract

Automatic Speech Recognition (ASR) is one of the most important applications in the area of cognitive computing. Fast and accurate ASR is emerging as a key application for mobile and wearable devices. These devices, such as smartphones, have incorporated speech recognition as one of the main interfaces for user interaction. This trend towards voice-based user interfaces is likely to continue in the next years which is changing the way of human-machine interaction.

Effective speech recognition systems require real-time recognition, which is challenging for mobile devices due to the compute-intensive nature of the problem and the power constraints of such systems and involves a huge effort for CPU architectures to reach it. GPU architectures offer parallelization capabilities which can be exploited to increase the performance of speech recognition systems. However, efficiently utilizing the GPU resources for speech recognition is also challenging, as the software implementations exhibit irregular and unpredictable memory accesses and poor temporal locality. Furthermore, GPUs operate at a high power which make heat dissipation and battery life a primary concern for these devices.

The purpose of this thesis is to study the characteristics of ASR systems running on low-power mobile devices in order to propose different techniques to improve performance and energy consumption. Firstly, we provide a performance and energy characterization of Pocketsphinx, a popular toolset for ASR that targets mobile devices. We identify the computation of the Gaussian Mixture Model (GMM) as the main bottleneck, consuming more than 80% of the execution time. The CPI stack analysis shows that branches and main memory accesses are the main performance limiting factors for GMM computation. We propose several software-level optimizations driven by the power/performance analysis. Unlike previous proposals that trade accuracy for performance by reducing the number of Gaussians evaluated, we maintain accuracy and improve performance by effectively using the underlying CPU microarchitecture.

We use a refactored implementation of the innermost loop of the GMM evaluation code to ameliorate the impact of branches. Then, we exploit the vector unit available on most modern CPUs to boost GMM computation, introducing a novel memory layout for storing the means and variances of the Gaussians in order to maximize the effectiveness of vectorization. In addition, we compute the Gaussians for multiple frames in parallel, significantly reducing memory bandwidth usage. Our experimental results show that the proposed optimizations provide 2.68x speedup over the baseline Pocketsphinx decoder on a high-end Intel Skylake CPU, while achieving 61% energy savings. On a modern ARM Cortex-A57 mobile processor our techniques improve performance by 1.85x, while providing 59% energy savings without any loss in the accuracy of the ASR system.

Secondly, after optimizing the ASR application at software level, we focus on improving the performance of these applications by identifying bottlenecks in current processors at microarchitecture level. We faced many stalls in renaming stage due to lack of physical registers for optimized and vectorized GMM code. To alleviate the pressure on the register file, we exploit the observation that for a significant percentage of instructions that have a destination register, the produced value has only a single consumer. In this case, the RAW dependence guarantees that the producer-consumer instructions pair will be executed in program order and, hence, the same physical register can be used to store the value produced by both instructions. We propose a register renaming technique

that exploits this property to reduce the pressure on the register file. Our technique leverages physical register sharing by introducing minor changes in the register map table and the issue queue. We evaluated our renaming technique on top of a modern out-of-order processor. The proposed scheme supports precise exceptions and we show that it results in 9.5% performance improvements for GMM evaluation. Our experimental results show that the proposed register renaming scheme provides 6% speedup on average for the SPEC2006 benchmarks. Alternatively, our renaming scheme achieves the same performance while reducing the number of physical registers by 10.5%.

Finally, we propose a hardware accelerator for GMM evaluation that reduces the energy consumption by three orders of magnitude compared to solutions based on CPUs and GPUs. The proposed accelerator implements a lazy evaluation scheme where Gaussians are computed on demand, avoiding 50% of the computations. Furthermore, it employs a novel clustering scheme to reduce the size of the GMM parameters, which results in 8x memory bandwidth savings with a negligible impact on accuracy. Finally, it includes a novel memoization scheme that avoids 74.88% of floating-point operations. The end design provides a 164x speedup and 3532x energy reduction when compared with a highly-tuned implementation running on a modern mobile CPU. Compared to a state-of-the-art mobile GPU, the GMM accelerator achieves 5.89x speedup over a highly optimized CUDA implementation, while reducing energy by 241x.

Keywords

Automatic Speech recognition, GMM, Hardware Accelerator, Microarchitecture, Register Renaming.

Acknowledgement

Firstly, I would like to express my sincere gratitude to my advisor Prof. Antonio González for offering me the opportunity to start a research career at UPC and providing me financial support during these years. I thank him for the continuous support of my Ph.D study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Ph.D study. It was, is and will be a great pleasure to work with him.

I am very grateful to have Dr. Jose-Maria Arnau as my advisor. I would like to show my greatest appreciation to him for all that he taught me, the daily meetings and helpful discussions that I had with him. Jose-Maria always gives me constructive comments and I will never forget his patience and warm encouragement during my Ph.D. Without his guidance and persistent help, finishing this thesis would not have been possible.

I would like to thank my pre-defense and defense committee members Professor Jose Duato, Professor Emilio Luque and Professor Joan Manuel Parcerisa for their valuable feedback and comments.

I would particularly like to thank my other advisor Prof. Jordi Tubella for all his support. I also thank the staff in the department of Computer Architecture, human resource and other units of UPC for their kind help and support during these years.

I owe a very important debt to my dear friend, Reza Yazdani during these years. Reza has been extraordinarily supportive and optimistic and I will never forget this.

I would like to thank all the members of ARCO. I was lucky enough to be part of a large, well-known and enriching research group. Special thanks to Sudhanshu Jha, Gem Dot, Enrique de Lucas, Marti Torrents, Franyell Silfa, Marti Anglada, Marc Riera, Albert Segura and Josue Quiroga for all the good moments that we had at UPC.

And finally, last but by no means least, I would like to thank my great family, my lovely parents and my dear sister for their endless love, continuous encouragement and support. I owe my deepest gratitude to my parents and I would like to dedicate this thesis to them. I have been extremely fortunate in my life to have parents who have shown me unconditional love and support. The relationships and bonds that I have with my parents hold an enormous amount of meaning to me. I admire them for all of their accomplishments in life, for their independence and for all of the knowledge and wisdom that they have passed on to me over the years. My parents have played a key role in the development of my identity and shaping the individual that I am today.

This last word of acknowledgment I have saved for my best friend, who has been with me all these years and has made them the best years of my life. Without doubt, her support, encouragement, patience and unwavering love were undeniably the bedrock upon which the past three years of my life have been built.

Declaration

I declare that this thesis was composed and written by myself and the proposals presented are my own except where explicitly stated in the text. This work has not been submitted for any other degree or professional qualification except as specified. Some of the proposed techniques and results presented in this thesis has been published in the following papers:

- Hamid Tabani, Jose-Maria Arnau, Jordi Tubella, Antonio González, " Performance Analysis and Optimization of Automatic Speech Recognition," Multi-Scale Computing Systems (**TMSCS)**, *IEEE Transactions on,* 2017, DOI: 10.1109/TMSCS.2017.2739158.
- Hamid Tabani, Jose-Maria Arnau, Jordi Tubella, Antonio González, " An Ultra Low-power Hardware Accelerator for Acoustic Scoring in Speech Recognition," Parallel Architecture and Compilation Techniques **(PACT)**, *26th International Conference on*, Sep. 2017, Portland, USA.
- Hamid Tabani, Jose-Maria Arnau, Jordi Tubella, Antonio González, " A Novel Register Renaming Technique for Out-of-Order Processors, " High Performance Computer Architecture **(HPCA)**, *24th International Conference on*, Feb. 2018, Vienna, Austria.

Glossary

Acoustic Model An acoustic model is used in Automatic Speech Recognition to represent the relationship between an audio signal and the phonemes or other linguistic units that make up speech. The model is learned from a set of audio recordings and their corresponding transcripts.

ASR. Automatic Speech Recognition, the conversion of a speech signal to a symbolic representation by computational means.

Beam. In a heuristic search algorithm, the beam is the range of scores outside of which current states or paths will not be extended. This is usually done by applying a factor to the best score of all paths reaching a given point and discarding all paths whose scores are less than the resulting score.

Continuous Acoustic Model. An HMM-based acoustic model, whose output density functions are Gaussian Mixture Models which each triphone has its own separate weighted Gaussian distributions.

Decoding. Translation of a message into codewords. In the case of speech, refers to the translation from a representation of a speech signal into a sequence of words or linguistic units. Usually, but not always, the input representation is the acoustic signal (for example, we can also speak of decoding a word lattice). Often used synonymously with search.

HMM. Hidden Markov Model. A statistical model used to exemplify a process which evolves over time, where the exact state of the process is unknown, or "hidden".

Hypothesis. A single sequence of words or linguistic units considered by a speech recognizer as the result of decoding an input utterance. May also refer to one particular component of such a sequence, as in a "word hypothesis".

Lattice. A directed acyclic graph representation of the set of hypotheses generated by a speech recognizer, where both word identities and timing information are represented.

MFCC. Mel-Frequency Cepstral Coefficients, the coefficients of the cepstrum of the short-term spectrum, downsampled and weighted according to the mel scale, a frequency scale thought to represent the sensitivity of the human ear.

PTM Acoustic Model. Phonetic Tied-Mixture. An HMM-based acoustic model, whose output density functions are Gaussian Mixture Models in which triphones that belong to the same basephone share a single set of Gaussian distributions.

Real-time factor. Often abbreviated as xRT, the measure typically used to report the performance of a speech recognizer. This is calculated as the ratio between the amount of time required to decode an utterance and the length of the utterance. For example, a real-time factor of 0.4 xRT means that each second of audio requires 0.4 seconds to decode (lower RTF means faster decoding).

Search. In automatic speech speech recognition, the process of searching the set of linguistic or symbolic representations of an utterance for one (or more) which are considered the most probable by the statistical models used by the recognizer. Often used synonymously with decoding.

Semi-continuous Acoustic Model. An HMM-based acoustic model, whose output density functions are Gaussian Mixture Models which share a single set of Gaussian distributions.

Senone. The two most common forms of parameter tying are state tying and mixture tying. In the former, the state output distributions of all triphones are mapped to a set of acoustic clusters, usually known as senones. Each senone consists of a complete mixture distribution with its own set of Gaussian parameters.

Triphone. In large-vocabulary continuous speech recognition, it is common practice to use context-dependent sub-word units as the basic units of recognition. These are typically phoneme-like units, consisting of a single phoneme in a particular phonetic context. Most frequently, the context is defined by the identities of the phonemes immediately preceding and following, in order to model coarticulatory effects. This unit is known as a triphone.

Tied. When multiple HMM states in an acoustic model share a set of parameters, these states are said to correspond to a single "tied" state. This is usually done for states that, despite having different symbolic representations, are acoustically similar. This allows for more compact and efficient acoustic models and better use of sparse training data.

Utterance. The longest segment of speech operated on at one time by an automatic speech recognizer. Typically corresponds to an uninterrupted phrase, sentence, or paragraph spoken by a single speaker.

Vector Quantization. The representation of a continuous vector space with a discrete set of prototype vectors or codewords.

Viterbi Algorithm. A dynamic programming algorithm to find the probability of the most likely state sequence.

WER (Word Error Rate). Number of insertions plus deletions plus substitutions that are required to convert the recognized word sequence into the reference word sequence, divided by the total number of words of the reference utterance.

This thesis is dedicated to my parents
for their endless love, support and encouragement.

Contents

List of Figures

List of Tables

1 Introduction

This chapter presents the background and motivation behind this work, a brief description of related work, and an overview of the main proposals and contributions of this thesis.

1.1 Motivation

The way people interact with devices nowadays is dramatically changing. Traditional input methods like keyboards or mouse are left behind and during the last decade we experienced a tremendous revolution in the area of human/machine interaction with the introduction of touch screens. However, a new avenue is being explored towards even more intuitive interfaces based on image recognition and speech recognition. This technology is emerging as a critical component in data analytics for a wealth of media data that is being generated everyday. Commercial usage scenarios are already appearing in industry such as broadcast news transcription [43, 72], voice search [88], automatic meeting diarization [10] or real-time speech translation in video calls [91]. Moreover, voice-based user interfaces are becoming increasingly popular, significantly changing the way people interact with computers.

Assistant tools like Google Now [4] from Google, Siri [11] from Apple or Cortana [64] from Microsoft rely on ASR as the main interaction system with the user on mobile devices. We can only expect a broader adoption of this interaction system with the anticipated popularization of IoT devices and wearables. On the one hand, companies like Google or Apple are actively improving their software to allow users to interact with tablet and mobile devices using voice. However, the computation requirements needed to enable speech recognition and natural language processing for large grammars and dictionaries are very high. These requirements make solutions to rely on the internet to offload part of the computation to servers in the cloud even for basic control instructions.

Although this approach is functional, its end-user experience is negatively impacted by the limited response time in situations of slow internet connection and the impossibility of interacting with the device even for local commands or dictation if the user is in a place with no internet coverage.

There is a large class of applications which cannot depend on a data connection, or where security or privacy concerns make it undesirable to entrust voice input to a remote server. In the consumer realm, this class includes in-car voice control, speech translation, music and contact search, and potentially other types of on-device search. It is clear that the use cases as well as the functional requirements of server-based and on-device speech recognition are quite different. However, these use cases are exclusive neither in time nor space, and it is no longer uncommon to encounter both types of speech recognition operating on the same device. One may, for example, use the on-device speech recognition on the Apple iPhone for dialing and other basic control tasks, while using Google's server-based voice search application on the same phone for Internet and local search and Nuance's server-based dictation application for composing e-mail and text messages.

Nowadays, mobile device hardware has advanced to the point where it is capable of handling medium vocabulary speech recognition tasks in real-time. This raises the possibility of providing much lower latency for speech recognition by processing audio data immediately as it becomes available. However, for the best recognition accuracy, and for inherently network-based tasks such as Internet search, it continues to make sense to perform recognition remotely, allowing much more substantial computational resources to be devoted to the problem.

Since the advent of speech recognition, the huge amount of computation required for an accurate large vocabulary speech recognition system was a serious problem that researchers and users were faced to. Hence, there are numerous methods and proposals to reduce the computation in many aspects. Reducing execution time and improving accuracy are two important goals for speech recognition researchers. However, these two goals are usually contradictory and we faced a trade-off between accuracy and performance. On the other hand, on mobile devices power consumption due to limited resources of energy is a very important issue. Accuracy in ASR systems depends on how much time we can devote to the computation. If we can spend a large amount of time computing, we can implement more sophisticated models ending up on better accuracy. However, the execution time needed to get reasonable accuracy on these toolkits when executed on power constrained cores is prohibitive. Figure 1.1 shows a projection of the expected accuracy with respect to the amount of time we devote to computation in a Tegra X1 mobile CPU and GPU. Time is normalized to real-time showed as 1 in the y-axis, which represents the maximum amount of time we could spend if we want to honor real-time constraints. As the figure shows, the higher the accuracy of the ASR system, the higher the time required to decode one second of speech.

This figure clearly shows that even modern mobile processors are unable to run sophisticated ASR models in real-time. Therefore, still improving the performance and energy-efficiency of ASR systems remains a challenging problem.

In this thesis, we focus on improving performance and energy-efficiency of ASR for mobile devices. ASR is already an essential feature for smartphones and tablets. Furthermore, speech technology will be of special interest for wearable computing, becoming a hard requirement for most mobile devices such as smart watches. Due to their tiny form factor, wearable devices are extremely constrained in terms of area and power and, therefore, they cannot afford complex

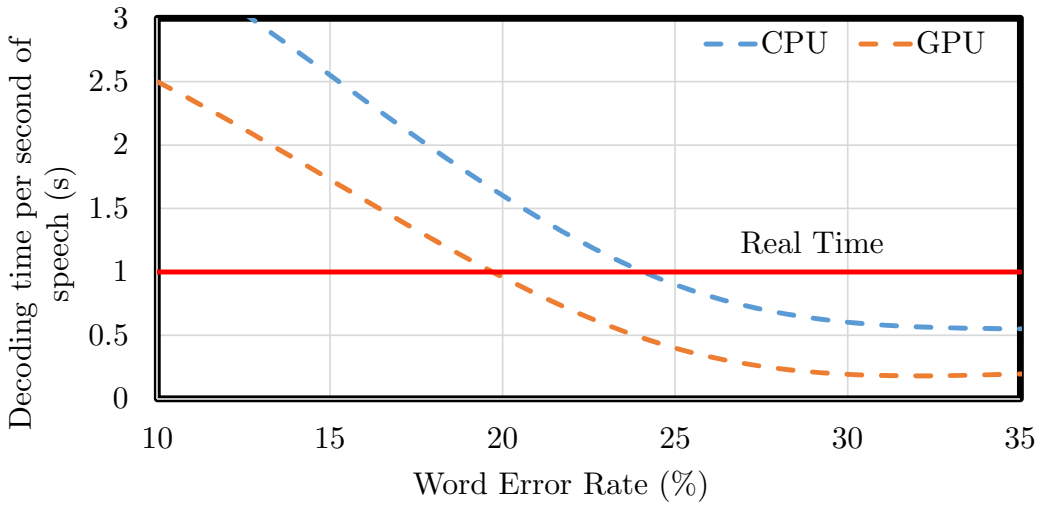


Figure 1.1: Decoding time for one second of speech vs accuracy of the Pocketsphinx ASR system running on a mobile CPU, ARM Cortex-A57, and a mobile GPU, NVIDIA Tegra X1.

hardware solutions like GPUs. Hence, we believe there is a strong case for real-time and energy-efficient ASR on mobile CPUs, which is the focus of this work.

The objective of speech recognition is the transcription of acoustic signals into a sequence of words. A state-of-the-art ASR pipeline consists of three main stages which are *Feature Extraction*, *Acoustic Scoring* and *Search Engine*. First, the input audio signal is split in frames, where each frame represents a 10 ms interval of the speech signal. Next, the *Feature Extraction* stage transforms the digitized audio samples within a frame into a vector of features. These features are then converted into a sequence of phonemes by an *Acoustic Model* in the *Acoustic Scoring* stage. The baseline ASR system employed in this thesis uses Gaussian Mixture Models (GMMs) [81] for acoustic scoring, i. e. each phoneme is modeled as a mixture of Gaussian functions. Finally, the *Search Engine* transforms the sequence of phonemes into a sequence of words by executing the Viterbi beam search. Chapter 2 presents a background for the algorithms used for automatic speech recognition and Pocketsphinx, our baseline ASR system.

Acoustic scoring is key for the accuracy of the ASR systems. Therefore, accurate ASR systems employ sophisticated large acoustic models. In such systems, for every frame of speech, hundreds of thousands of multi-dimensional Gaussians have to be evaluated. Therefore, GMM evaluation becomes the main bottleneck.

1.2 Problem Statement, Objectives and Contributions

The objective of this thesis is to propose novel techniques that address the issues in ASR systems, in order to improve their performance and energy-efficiency. In first place, we propose a set of software-based optimizations for Gaussian evaluation which is the main bottleneck in ASR system. In second place, our analysis shows that running the optimized ASR software in a

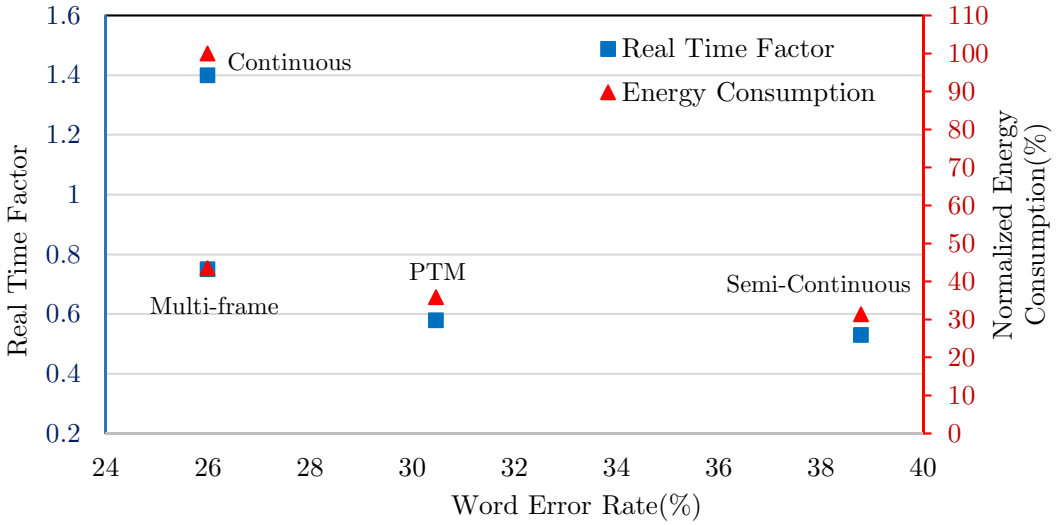


Figure 1.2: Real Time Factor and Normalized Energy consumption vs Word Error Rate for different acoustic models in Pocketsphinx running on an ARM Cortex-A57 mobile CPU. Multi-frame shows our optimized decoder. Energy consumption for different acoustic models are normalized with respect to the continuous acoustic model.

modern CPU results in considerable stalls in the rename stage of the CPU pipeline due to lack of physical registers. To overcome this issue, we propose a novel register renaming technique at microarchitecture level to improve the performance of register-intensive applications like GMM computation. In third place, we design an ASIC for Gaussian evaluation to improve the performance and energy-efficiency of ASR systems by orders of magnitude. The following sections outline the problems we are trying to solve, describe the approach we take to solve the problem and provide a comparison with related work, highlighting the novel contributions of this thesis.

1.2.1 Software-based Optimizations for ASR Systems

ASR applications deliver real-time, large vocabulary, speaker independent speech recognition. However, supporting accurate real-time ASR comes at a high energy cost. To illustrate this problem, Figure 1.2 shows the Real Time Factor (RTF) vs Word Error Rate (WER) for different configurations of Pocketsphinx [47], a widely used open source toolset for ASR, running on an ARM Cortex-A57 mobile CPU. As it can be seen, increasing accuracy causes a huge slowdown: the continuous acoustic model reduces error by 5 percentage points with respect to the simpler PTM (Phonetic Tied-Mixture) [53] model, while producing a slowdown of 2.4x and increasing energy consumption by 2.78x. Note that this is not only the case for Pocketsphinx, as other ASR toolsets exhibit similar power/performance trade-offs [35].

Previous solutions improve ASR performance by reducing the amount of computation required to convert the speech signal to words. One commonly used strategy is to simplify the acoustic model. Phonemes are typically modeled by using Gaussian Mixture Models (GMMs). The semi-continuous and PTM acoustic models of Pocketsphinx, included in Figure 1.2, significantly constrain the number of Gaussians in each mixture to improve performance. Although highly effective,

these approaches reduces accuracy to a large extent, as simple acoustic models cannot capture the complexity of speech.

Recognizing that accuracy is probably the most important parameter in an ASR system, we take a completely different approach as we improve performance while achieving the same accuracy of the baseline configuration. We boost GMM computation performance by applying low-level optimizations to the software in order to maximize the usage of the available CPU resources.

Based on an extensive analysis of the power/performance behavior of Pocketsphinx, we present several software optimizations in this work. The GMM evaluation, i. e. the acoustic model, is the main bottleneck consuming more than 80% of the execution time. Furthermore, our detailed analysis for mobile processors show that the main sources of stalls in the CPU are branch mispredictions and accesses to main memory. Finally, the power breakdown for the same CPU clearly shows that the DRAM is the main energy consumer.

Our software optimizations target the problems identified during the analysis. Regarding branch misprediction penalties, we show how the GMM evaluation code can be refactored to remove the most critical branches. As regards to the DRAM, we propose a multi-framing scheme where means and variances of a Gaussian are fetched once in the CPU caches and reused for evaluating the Gaussian in multiple frames of speech, improving the locality of memory accesses.

The main hurdle for implementing multi-framing in modern ASR systems is the interaction with lazy GMM evaluation. Due to the huge search space, ASR systems employ aggressive pruning to achieve real-time performance by dynamically discarding unlikely interpretations of the speech signal. Because of the pruning, only a subset of the Gaussians is active for a given frame of speech whereas the likelihoods of the other, inactive, Gaussians are not required. Pocketsphinx employs lazy GMM evaluation to avoid computing and fetching from memory inactive Gaussians. Combining lazy GMM evaluation with our multi-framing scheme is challenging as only the active Gaussians for the first frame in the batch are known. In this work, we propose a novel prediction scheme of active Gaussians that is highly effective and allows the use of both lazy GMM evaluation and multi-framing, substantially reducing main memory bandwidth usage.

Finally, we introduce SIMD instructions in Pocketsphinx to improve the performance and energy efficiency of the GMM computation. Furthermore, we propose a novel memory layout to store the means and variances that increases the amount of vectorizable code. Figure 1.2 shows that our optimized decoder, labeled as *Multi-frame*, provides the same accuracy as the continuous acoustic model while achieving performance and energy consumption close to the simpler PTM and semi-continuous acoustic models. Chapter 3 extensively describes the analysis, our proposals and the results in this work. This work has been published in the IEEE Transactions on Multi-Scale Computing Systems (TMSCS) Journal [95]:

- Hamid Tabani, Jose-Maria Arnau, Jordi Tubella, Antonio González, " Performance Analysis and Optimization of Automatic Speech Recognition," Multi-Scale Computing Systems (**TMSCS)**, *IEEE Transactions on,* 2017, DOI: 10.1109/TMSCS.2017.2739158.

1.2.2 Modifying Microarchitecture Design

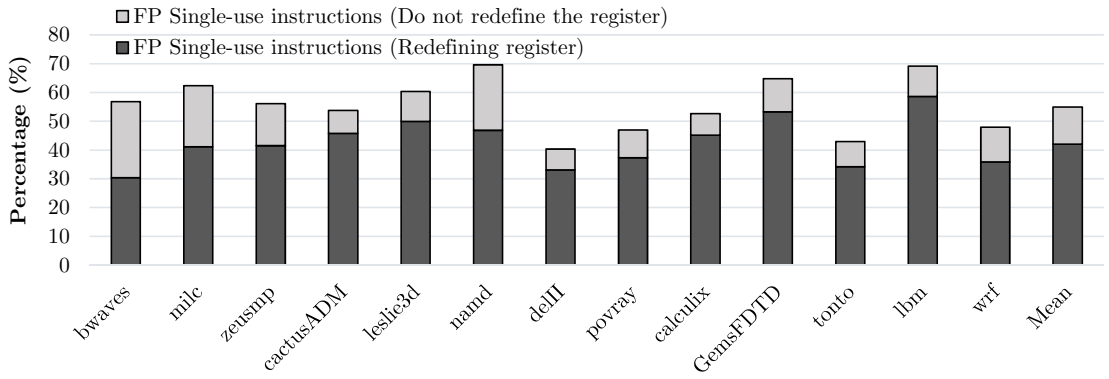
Although our proposals at software level improve the performance and energy-efficiency of the ASR system significantly, further improvements can be achieved by understanding the performance limiting factors at microarchitecture level. In this thesis, we try to further improve the performance of GMM evaluation by applying changes at the microarchitecture level. By running the optimized and vectorized GMM code that reduces branch mispredictions and cache misses significantly, the register file becomes under pressure and results in considerable stalls in the renaming stage restricting the overall performance of the processor.

Dynamically-scheduled superscalar processors exploit instruction-level parallelism (ILP) by reordering and overlapping the execution of instructions in an instruction window. The number of instructions that can be executed in parallel is highly dependent on the instruction window size and, thus, wide issue processors require a large instruction window [101]. However, a large instruction window has some implications in other critical parts of the microarchitecture, such as the size of the physical register file [33]. In this thesis, we are concerned with this issue. On the other hand, in spite of being able to execute instructions out-of-order, the amount of ILP that current superscalar processors can exploit is significantly restricted by data dependences. Due to the limited number of architectural registers, at some point compilers start reusing them, which may cause name dependences (write-after-read and write-after-write dependences). Dynamic renaming schemes eliminate these name dependences by assigning a new storage location to the destination register of each instruction. This increases the amount of independent instructions that can be executed in parallel, which results in an increase in the ILP.

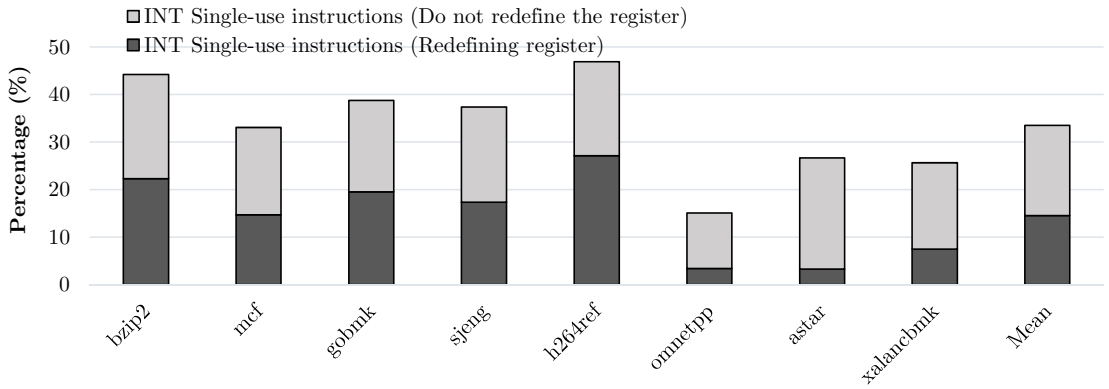
Larger instruction windows require a higher number of physical registers. However, increasing the size of the register file is challenging and has important implications in terms of energy consumption, access time and area [33].

This work is motivated by the observation that, in a significant percentage of instructions with a destination register, this register has a single consumer. Figure 1.3(c) shows that more than 57% of the instructions in GMM exhibit this property. For other benchmarks we show that more than 50% of the instructions in SPECfp and more than 30% in SPECint exhibit this property. In this case, the RAW dependence between producer and consumer will force sequential execution of the two instructions. In addition, since there is only a single consumer then no other instruction will require the value produced by the first instruction. Therefore, producer and consumer can safely use the same physical register as their destination.

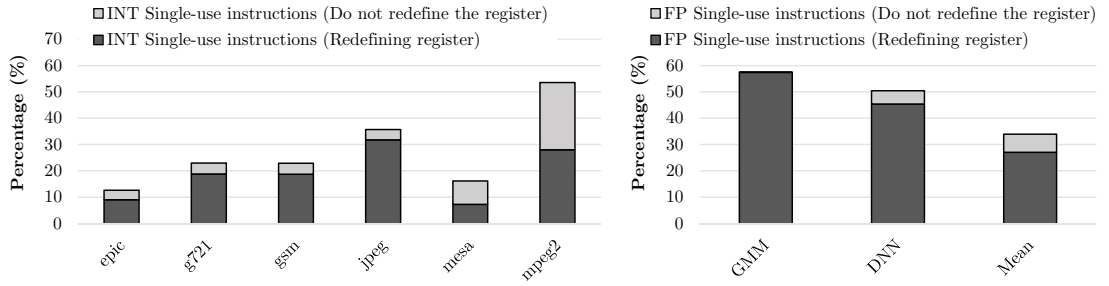
In this work, based on the aforementioned observation, we propose a register renaming scheme that implements this reuse of registers in dynamically scheduled processors that implement precise exceptions. We show that identifying single-use registers can be accomplished with simple hardware. To be able to recover the state of the processor in a branch misprediction, interrupt or exception, we propose to use a multi-bank register file with check-pointed register banks using shadow cells [32]. Check-pointed registers are allocated whenever it is predicted that a register might be reused. Hence, the former value of a register can be recovered in an event of branch misprediction, interrupt or exception. Chapter 4 explains our proposed register renaming scheme and provides the details of our analysis on various benchmarks. This work has been published in the 24th International Conference on High Performance Computer Architecture (HPCA) [97]:



(a) SPECfp



(b) SPECint



(c) Mediabench and Cognitive applications

Figure 1.3: Percentage of instructions with a destination register that are the only consumers of the value of a register. We distinguish between instructions that redefine the single-use register and instructions that redefine a different logical register.

- Hamid Tabani, Jose-Maria Arnau, Jordi Tubella, Antonio González, " A Novel Register Renaming Technique for Out-of-Order Processors, " High Performance Computer Architecture **(HPCA)**, *24th International Conference on*, Feb. 2018, Vienna, Austria.

1.2.3 A Hardware Accelerator Design for Acoustic Scoring in ASR systems

As discussed in section 1.2.1, the most compute intensive component of the majority of ASR systems, including Pocketsphinx, is the acoustic scoring. In an ASR system, the input audio signal is split in frames, typically of 10 ms duration. For each frame of speech, the acoustic scoring computes the likelihood, a.k.a. score, that the frame is part of a particular phoneme, for all potential phonemes in the language. Despite the increasing popularity of DNNs, GMMs are used by the majority of ASR systems including Pocketsphinx, Sphinx4, HTK [106], Julius [54] and some decoders of Kaldi. GMM evaluation is typically the main bottleneck in these systems. Our measurements on an NVIDIA Tegra X1 show that acoustic scoring takes more than 80% of the decoding time in Pocketsphinx.

Figure 1.4 shows the power dissipation and execution time for the GMM evaluation stage, i. e. acoustic scoring, on a mobile CPU and GPU. These numbers were collected using the standard acoustic model of Pocketsphinx, that requires the evaluation of 160k 36-dimensional Gaussian distributions for every frame of speech (10 ms). The mobile CPU dissipates 2.4 W, and it barely reaches real-time performance as it takes 1.08 s to process one second of speech. The mobile GPU achieves real-time performance, but at the cost of increasing power dissipation significantly (9.2 W), which in turn results in shorter operating times per battery charge. To illustrate this issue, for a typical smartphone battery of 11.55 WHr (41580 J) [8] the operating time when running ASR software on the CPU would be less than 4.8 hours, whereas it would be reduced to less than 1.25 hours when using the GPU. Note that CPU and GPU are not the only battery consumers in a smartphone, so these numbers are upper bounds of the operating time.

In this work, we present a hardware accelerator for GMM computation that includes innovative techniques to improve the energy-efficiency of ASR systems. Hardware acceleration is an effective approach to achieve high-performance and low-power ASR in mobile devices.

Our accelerator implements a lazy evaluation scheme that computes Gaussian distributions on demand instead of evaluating all the Gaussians on every frame, reducing the amount of computation. Our lazy evaluation scheme introduces and implements in hardware our novel technique to predict the active Gaussians in the next frames of speech, (as introduced in section 1.2.1), in order to apply lazy evaluation for multiple frames (batching).

On the other hand, our accelerator includes a novel clustering technique that provides 8x reduction in the size of the acoustic model. Our clustering works independently for each dimension of the Gaussian distributions, and we show that it provides better accuracy than the straightforward approach of clustering the entire GMM together.

Finally, we show that as a side effect, the proposed clustering results in a huge amount of redundant computations, and we introduce a novel memoization scheme that drastically reduces the number of floating point operations. Our scheme pre-computes and stores in hardware all

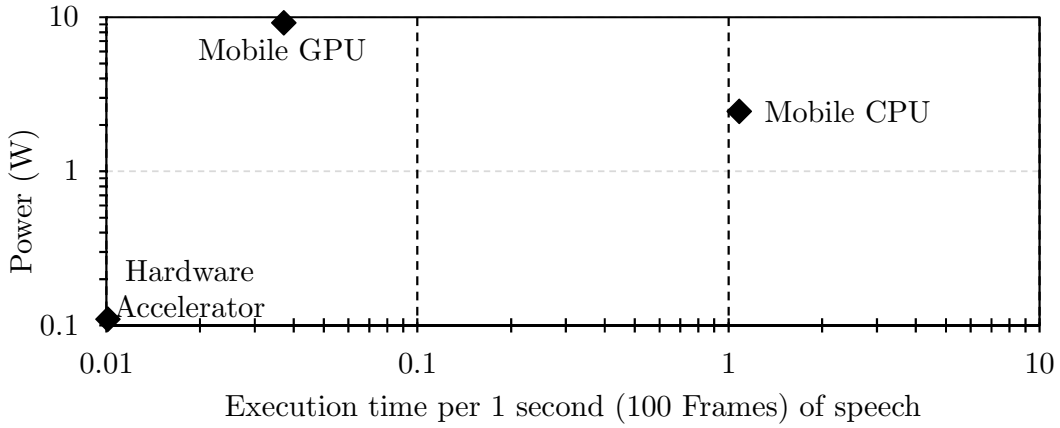


Figure 1.4: Power dissipation vs execution time for a mobile CPU, ARM Cortex-A57, a mobile GPU, NVIDIA Tegra X1, and a hardware accelerator for acoustic scoring.

possible results at the beginning of each frame, instead of probing/updating a memoization table for every individual floating point operation. Figure 1.4 shows the power and performance of our GMM accelerator. As it can be seen, the accelerator achieves higher performance than the mobile GPU at ultra low power, as low as 110 mW.

Chapter 5 presents in detail our proposed hardware accelerator. This work has been published in the 26th International Conference on Parallel Architectures and Compilation Techniques (PACT) [96]:

- Hamid Tabani, Jose-Maria Arnau, Jordi Tubella, Antonio González, " An Ultra Low-power Hardware Accelerator for Acoustic Scoring in Speech Recognition," Parallel Architecture and Compilation Techniques **(PACT)**, *26th International Conference on*, Sep. 2017, Portland, USA.

1.3 State-of-the-art in Performance and Energy Improvement for ASR Systems

Improving performance of GMM computation for speech recognition has attracted the attention of the research community the last few years. In this section, we review some related work which are proposed in order to improve performance and energy efficiency of ASR systems at software level or by proposing hardware designs. Later, we will present alternative types of acoustic models that have been proposed in order to reduce the complexity of GMM evaluation or to improve the accuracy of the ASR system.

1.3.1 Software Solutions

Regarding software improvements, most proposals focus on reducing the amount of computation at the cost of increasing Word Error Rate. Partial Distance Elimination [76, 19] employs the top N Gaussians with highest likelihood to compute senone[1] score instead of using all the Gaussians. Our work in this thesis is different as one of our objectives is to maintain accuracy and boost performance by exploiting the Vector Processing Unit (VPU) and saving bandwidth with an approach to evaluate scores of several frames of speech which is called multi-framing approach.

The use of SIMD instructions for GMM computation has been subject of research for several years [52, 74, 20]. These proposals do not evaluate the impact of SIMD on energy consumption. Moreover, these works do not evaluate the interaction of SIMD instructions with hardware prefetchers. Furthermore, the efficiency of SIMD instructions is limited as these works remain significant amount of code unvectorized. We address this problem in our work by proposing a novel memory layout to store the GMM parameters which significantly increases the amount of vectorizable code.

Gupta et al. [39, 40] propose a chunk-based technique to compute GMMs that is similar to our multi-framing approach. They target GPU architectures and very small vocabulary ASRs so that they can exploit temporal locality in on-chip memory. Our proposals are different in several ways. First, we target CPUs instead of GPUs. Second, our baseline employs much bigger vocabulary (130k words vs 5k) and acoustic model (164k Gaussians vs 15k), so our datasets significantly exceed the capacity of the on-chip caches.

Tan et al. [99] explain in detail how Automatic Speech Recognition can be presented in mobile devices and also over communication networks. In case the ASR is provided in the network, the voice is captured in user's device while it is processed in the cloud. Although complex and more accurate ASR systems can be executed in the cloud, for many tasks it is prefered to provide ASR in user's device due to indeterminate response-time in the cloud, inaccessibility to the network or security reasons.

Dixon et al. [29] implement the GMM evaluation using matrix multiplication and it is considered as state-of-the-art implementation of the GMM evaluation. Their approach uses a batch size of 256 frames to evaluate the GMMs. This is mainly to improve the efficiency of the matrix multiplication scheme. The main drawback of using large batch sizes is increasing the response time of the ASR system. Response time is a key factor for real-time applications. Furthermore, not all the senones are active every frame and the number of active senones varies smoothly from one frame to the next. Our analysis shows that in average less than 50% of senones remain active due to pruning of less probable hypotheses. Our methods can simply skip inactive senones and their corresponding Gaussians which results in a significant speedup and energy reduction and in section 3.4.1 we show that our proposed methods outperform their implementation.

A batching technique is proposed in [25, 49] to reduce the memory bandwidth. These works use small batch sizes of 4 and 8 in order to reduce the latency caused by batching. Our work is different as we combine batching with lazy evaluation by using a novel prediction scheme of active senones. Moreover, our technique not only avoids computing the inactive senones, but also

[1] State output distributions of all triphones are mapped to a set of acoustic clusters, named senones. Each senone consists of a complete mixture distribution with its own set of Gaussian parameters.

eliminates memory accesses for inactive senones in a batch. These architectures implement the straightforward GMM evaluation, whereas our design includes several optimization techniques, such as clustering and memoization, to reduce memory requirements and improve performance and energy efficiency significantly.

1.3.2 Alternative Acoustic Models

Some works tried to reduce the complexity of acoustic scoring by reducing number of Mixtures of Gaussians. Phonetic-Tied-Mixture (PTM) [55] and semi-continuous [46, 45, 30] acoustic models are among the main contributions. However, they are increasing the WER to a large extent. In PTM acoustic model [53, 87] triphones that belong to the same basephone share one GMM, reducing the number of Gaussians evaluated for acoustic scoring manifold. In semi-continuous acoustic models all the senones share one GMM, reducing the number of Gaussians evaluated for acoustic scoring in two orders of magnitude.

Some recent works proposed subspace GMMs [77, 78], to perform better than semi-continuous AMs with less available resources for training. Our experiments show that for large-vocabulary ASRs in comparison with a continuous acoustic model, an state-of-the-art PTM and semi-continuous acoustic models increase the WER by 5.5% and 12% respectively[2]. In this thesis, we focus on continuous acoustic models which are the most accurate types of acoustic models.

1.3.3 Clustering Schemes

Since the GMM parameters cannot be stored to on-chip caches due to their large sizes, ASR systems cannot simply exploit temporal locality. Therefore, off-chip memory accesses to read GMM parameters occupies significant percentage of the memory bandwidth and it consumes most of the energy. This makes main memory and off-chip memory accesses a performance bottleneck while consuming significant amount of energy. Hence, reducing the size of GMM parameters has been exploited using different approaches. Clustering GMM parameters are among the most popular approaches as they achieve considerable reduction in the size of GMM parameters by representing the parameters as indexes referring to a codebook of centroids.

There are several proposed scalar and vector clustering techniques [84, 19, 28]. Sub-vector clustering methods results in higher reduction in GMM parameter's size while producing thousands of vectors of centroids which increase the complexity of evaluating the Gaussians. Although these techniques significantly reduce the size of the acoustic models, they significantly reduce the accuracy of the ASR system. We explored data patterns in acoustic model parameters and based on that we proposed a scalar per-GMM component clustering technique which achieves significant reductions in the size of acoustic model with a negligible impact on the accuracy of the ASR system. Considering the same compression ratio, our scheme outperforms previous techniques providing significantly less impact on the accuracy of the ASR system.

[2] Experiments done by running entire LibriSpeech test corpus on the latest version of Pocketsphinx (version 5-prealpha) using 70k-word language model and 130k-word dictionary using latest PTM, semi-continuous and continuous acoustic models.

1.3.4 Hardware Solutions

Improving performance and reducing energy consumption of acoustic scoring using especial hardware designs has attracted the attention of the research community in recent years. Several hardware solutions for GMM evaluation have been proposed using ASICs or FPGAs.

A hardware co-processor is proposed in [60] to boost the performance of the GMM computation in Sphinx3. Reducing the size of the mantissa from 23-bits to 15-bits and 12-bits is proposed in [22] to reduce the acoustic model size, providing a compression ratio of 1.39x and 1.6x respectively. The technique is evaluated using a small vocabulary size of 5000 words, whereas we propose a novel clustering technique to achieve 8x reduction in acoustic model size with a 130k words vocabulary size.

Hardware/software co-design schemes are proposed in [23, 90, 109] to improve the performance of the ASR system. Our design improves performance by two orders of magnitude in comparison with their work. Furthermore, we proposed several techniques to reduce the energy consumption, memory footprint and memory requirements while using much larger datasets.

Our GMM accelerator is different from previous proposals as it includes lazy evaluation combined with batching, clustering and memoization, that provide significant performance and energy improvements in comparison with the previous designs.

1.3.5 Deep Neural Networks for Acoustic Scoring

GMM has been the mainstream machine learning technique for implementing the acoustic model in speech recognition systems for decades. The vast majority of ASR systems, such as Sphinx4, Pocketsphinx, Kaldi or HTK, provide an implementation of acoustic scoring based on GMMs. In recent years, the use of Deep Neural Networks (DNNs) for acoustic scoring [42] has become very popular due to its high recognition accuracy. Unlike the conventional idea that these are two competing approaches for speech recognition, recent research has shown that GMMs and DNNs complement each other. Swietojanski et al. [94] propose an acoustic model that combines a GMM and a DNN, achieving higher accuracy than the DNN alone. Rath et al. [83] present a hybrid and stacked ASR system that also combines a DNN with a GMM to improve recognition accuracy. These ASR systems combine the frame-level acoustic scores computed separately by a DNN and a GMM. Other hybrid systems, like the tandem approach [31] or the bottleneck approach [108], employ DNNs as feature extractors for a GMM. Yu et al. [107] explain in detail fuse DNN and GMM systems. Recently, Tachioka et al. [98] proposed to use DNN and GMM-based ASR systems to address variety of noises in a noisy environment. Their results show that combining these approaches increases the accuracy of the ASR system significantly.

To sum up, GMMs are still among the most common techniques for implementing the acoustic scoring in ASR systems. Furthermore, previous work has shown the synergy between GMM and DNN techniques. Therefore, we believe that improving the performance and energy-efficiency of GMMs will be of special interest for future speech recognition systems.

1.4 Thesis Organization

The remainder of this thesis is structured as follows:

Chapter 2 provides basic background information on the speech recognition algorithms. We mainly describe the ASR toolkit that we employ in this thesis and we detail its different stages.

In Chapter 3, first we present the extensive analysis of the ASR toolkit running on various general-purpose mobile and desktop processors. Second, we highlight the bottlenecks and the main performance and energy consumption determinative factors. Third, we present our software-based proposals to alleviate these bottlenecks.

In Chapter 4, we show that for register-intensive application like GMM evaluation, traditional register renaming schemes are very conservative which make the registers a limiting factor for the performance of out-of-order processors. We present a novel register renaming technique in order to reduce the pressure on the register file. We provide an extensive analysis on variety of benchmarks to motivate our proposed register renaming technique.

Chapter 5 presents our proposed hardware accelerator for GMM evaluation in ASR systems. First, we propose a baseline accelerator for Gaussian evaluation in acoustic scoring. Later, we propose our new scheme for clustering GMM parameters and the way we integrate it in the baseline accelerator. Furthermore, we introduce our new lazy Gaussian evaluation and a memoization scheme which provides significant performance and energy improvements.

Finally, Chapter 6 outlines some of the future steps and open research areas and summarizes the main conclusions of the thesis.

2

Background on Speech Recognition

This chapter presents a brief background to understand the state-of-the-art algorithms used for automatic speech recognition. Due to the objectives of this thesis, we will focus on hybrid HMM-based statistical modeling techniques for automatic speech recognition, which is the mainstream approach for ASR systems. Therefore, first, we provide a comparison between three different state-of-the-art ASR systems motivating the choose of our baseline ASR system.

Figure 2.1 shows decoding time per second of speech, memory footprint and accuracy for three popular open-source ASR systems when running on an ARM Cortex-A57. Pocketsphinx [48] uses Gaussian Mixture Models (GMMs) for acoustic scoring, whereas Kaldi [79] and Eesen [63] employ a Deep Neural Network (DNN) and a Recurrent Neural Network (RNN) respectively. We use the most recent pretrained acoustic models provided in their respective websites [1]. As it can be seen in Figure 2.1, Pocketsphinx achieves much lower decoding time than the DNN-based systems, at the cost of a small increase in WER. Furthermore, it exhibits a small memory footprint of less than a hundred MB, whereas Kaldi and Eesen require hundreds of MB. These numbers suggest that Pocketsphinx is the most suited approach for ultra low-power, low-cost devices, due to its lower computational and storage demands, so we chose it as the baseline for this thesis.

The objective of speech recognition is the transcription of acoustic signals into a sequence of words. Pocketsphinx, similar to almost all the state-of-the-art ASR systems, consists of three main stages: *Feature Extraction*, *Acoustic Model* and *Search Engine*. In the following sections we describe this pipeline.

[1] Higher accuracy can be achieved with even larger acoustic models and multiple rescoring passes [6], but they result in much larger memory footprint, energy consumption and decoding time.

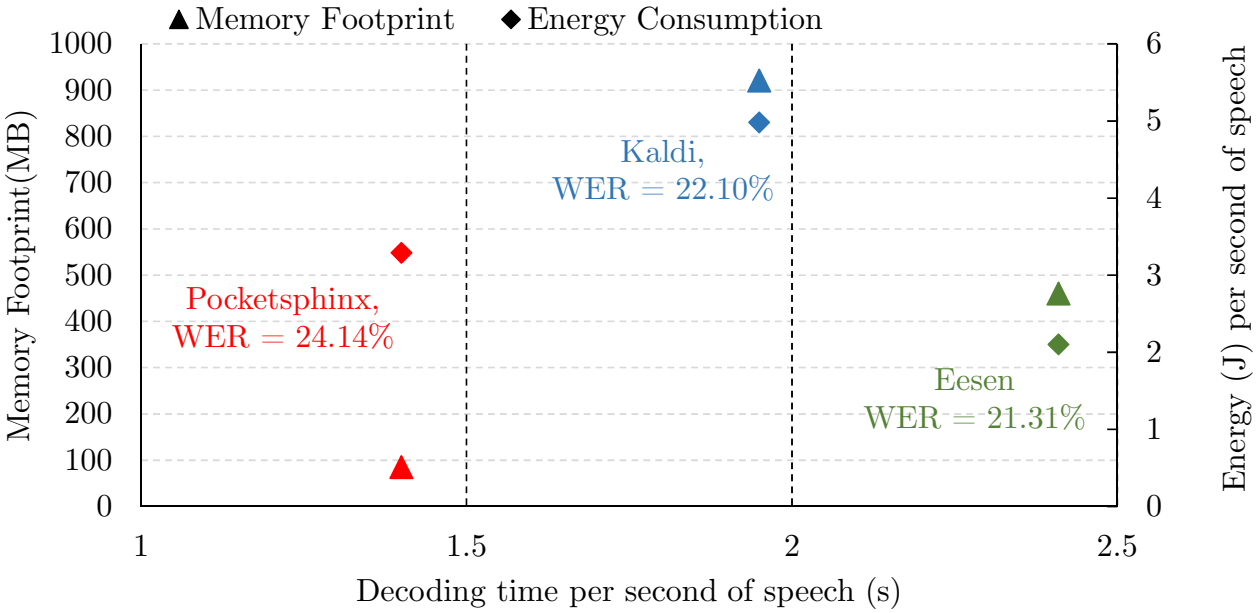


Figure 2.1: Decoding time, memory footprint and Word Error Rate (WER) for three popular ASR toolkits running on an ARM Cortex-A57. Results are collected using the most recent pre-trained models for each system.

2.1 Signal Processing and Feature Extraction

First, the input audio signal is split in frames, where each frame represents a 10 ms interval of the input signal. Next, the *Feature Extraction* component transforms the audio samples within a frame into a vector of features. Signal processing front-end differs in different speech recognition systems. For instance, in the latest continuous acoustic model in Pocketsphinx, the stream of 16-bit samples of audio, sampled at 16KHz or 8KHz, is converted into 12-element mel-frequency cepstral coefficients (MFCC) vectors in each 10 ms frame of audio. We represent the cepstrum vector at time x by $x(t)$ that contains individual element $x_k(t)$, $1 \leqslant k \leqslant 12$. This cepstrum vector is first normalized and 2 feature vectors are derived in each frame by computing the first and second order differences in time:

$$x(t) = Normalized\, Cepstrum\, Vector \tag{2.1}$$

$$\Delta x(t) = x(t+2) - x(t-2) \tag{2.2}$$

$$\Delta\Delta x(t) = \Delta x(t+1) - \Delta x(t-1) \tag{2.3}$$

Thus in every frame we obtain three vectors of 12 elements and these, ultimately, are the input to the speech recognition system. Figure 2.2 shows the feature extraction process [85].

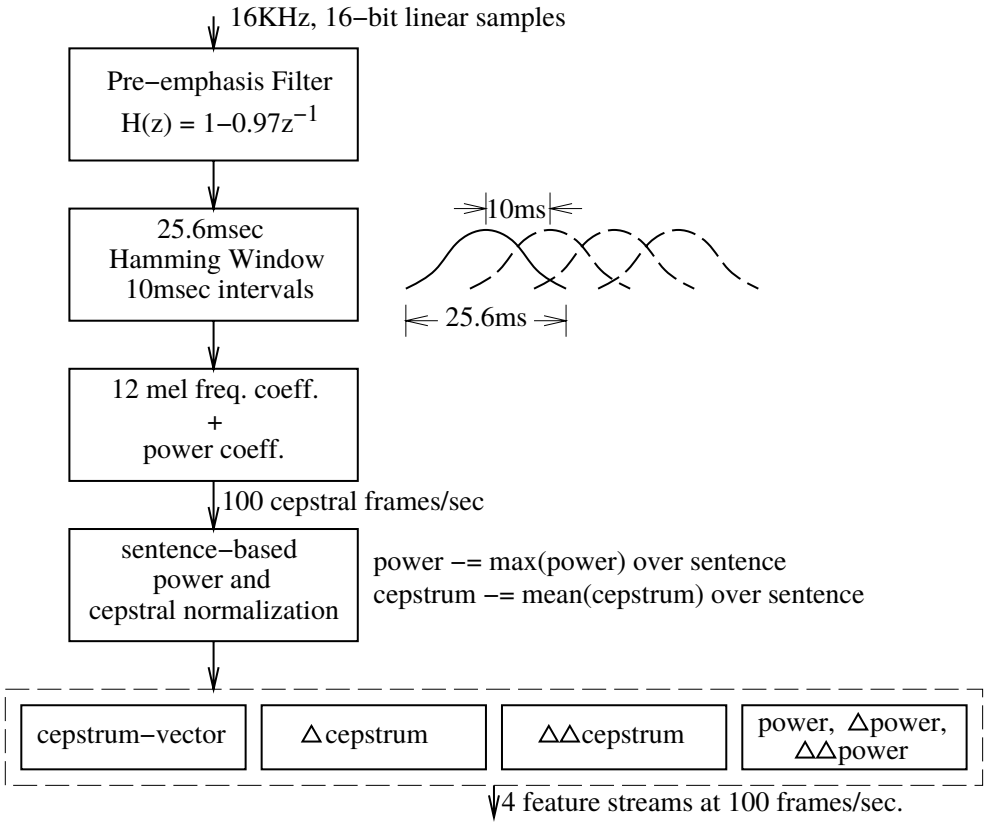


Figure 2.2: Feature Extraction Process [85].

2.2 Acoustic Model

2.2.1 Phones and Triphones

After extracting feature vectors from the audio signal, the features are then converted into a sequence of phonemes by the *Acoustic Model*. To accomplish this, there could be different models. One might wish to create word models from training data. However, in large vocabulary speech recognition, there are too many words to be trained. This problem is solved by modeling sub-word units. The important advantage of such modeling is to share modeled sub-units across different words. Phonetic models are the most frequently used sub-word models. One method, which was first proposed by IBM [14], is to model a phoneme (normally called a phone) influenced by its neighbor phones (i.e., a context-based phone). As an example, the AE phone in "man" sounds different from that in "lack". The former is more nasal.

ASR systems create acoustic models for sub-word units or phonemes. Due to co-articulation effects, the production of sound corresponding to a phoneme is influenced by neighboring phonemes. Hence, most large vocabulary speech recognition systems, including Pocketsphinx, use triphones [13]

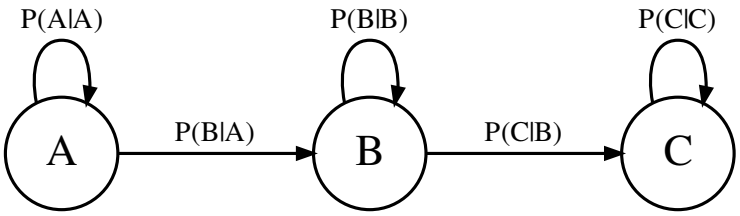


Figure 2.3: First-order Markov chain with three states A, B and C.

or context-dependent phone models as they are able to model such variations. There are only around 50 phonemes in spoken English, so there are around 50^3 triphones, but only a fraction of them are observed in practice.

2.2.2 HMM Modeling of Phones and Triphones

Most speech recognition systems use Hidden Markov Model (HMM) to represent the basic unit of speech. The way of using and training of HMMs is covered in literature [16, 15, 50, 82]. An HMM is a set of states connected by transitions as shown in Figure 2.3. Transitions model the emission of one frame of speech. Each HMM state has an associated *output probability* function that defines the probability of emitting the input feature observed in any given frame while taking that transition. The *output probability* for state i at time t is denoted by $b_i(t)$. Also, each HMM transition from any state i to state j has a static transition probability, which is denoted by a_{ij} and is independent of the speech input.

Each HMM state represents a small subspace of the overall feature space and its shape is complex enough to be accurately characterized by a simple mathematical distribution. Generally, the most common approach to model the state output probability is by a mixture of Gaussians which is known to be the most expensive part of an ASR system [60, 39]. In Pocketsphinx, *Acoustic Model* is based on Gaussian Mixture Models (GMMs) [81], i. e. each phoneme is modeled as a mixture of Gaussian functions.

For any HMM state s and feature stream f, the g-th component of the m-th GMM is a normal distribution with mean vector $\mu_{m,g}$ and standard deviation vector $\sigma_{m,g}$. Each mixture component also has a scalar mixture coefficient or weight $w_{m,g}$. Hence, the probability of observing a given frame of speech with feature vector x in the GMM m is given by Equation 2.4, where g ranges over the number of Gaussians in the mixture. The expression $\mathcal{N}(.)$ is the value of the Gaussian density function at x. In the speech input, x is a vector of features for one frame of speech and x_f is its f-th feature component.

$$b_s(x) = \prod_f \left(\sum_{g=0}^{N-1} w_{m,g}\mathcal{N}(x, \mu_{m,g}, \sigma_{m,g})\right) \tag{2.4}$$

For numerical stability, the multivariate Gaussian distribution $\mathcal{N}(.)$ is computed in log-space by using Equation 2.5. The dimensionality of the Gaussians is equal to the dimensionality of the feature vector x. During the recognition process, Equation 2.4 has to be evaluated for every GMM on a frame basis. In these equations, D, M and N are representing determinant, dimensionality of the feature vector and number of Gaussians respectively.

$$\mathcal{N}(x, \mu_{m,g}, \sigma_{m,g}) = D_{m,g} - \sum_{c=0}^{M-1} \frac{(x_c - \mu_{m,g,c})^2}{2\sigma^2_{m,g,c}} \tag{2.5}$$

In a fully continuous acoustic model each triphone has its own separate weighted GMM. However, computing such a big number of Gaussian functions is completely unfeasible for real-time speech recognition. In practice, multiple triphones share the same GMM to reduce the computational cost of the *Acoustic Model*. In the continuous model of Pocketsphinx HMM states are grouped into clusters called senones. All the triphones that belong to the same senone share the same GMM. Senones are just tied triphone HMM states. A context dependent HMM recognizer has a 3-5 state HMM for every context dependent phone. Conceptually, each different HMM state in each different phone HMM has its own Gaussian mixture model.

The latest generic acoustic model of Pocketsphinx (en-us-5.2) has 5138 senones. Although the continuous model offers the highest accuracy in Pocketsphinx, other acoustic models are included in an attempt to reduce the amount of computation. The PTM model has a GMM for each context-independent phoneme or basephone. Triphones that belong to the same basephone share the same GMM. Hence, the number of GMMs is reduced from 5138 in the continuous model to 42 in PTM, which is the number of basephones in this model. On the other hand, the semi-continuous acoustic model employs just one GMM that is shared by all the triphones, but each triphone has its own mixture coefficients, i. e. the weights applied to each Gaussian ($w_{m,g}$ in Equation 2.4) are different.

Furthermore, not all the senones are active for a given frame of speech, as some of them may be pruned away during the search process. Pocketsphinx only computes the GMM for active senones, as the acoustic likelihood for the other, inactive, senones is not required for the search. By doing this, the workload for the continuous acoustic model is reduced by approximately one half. In order to implement this optimization, the *Search Engine* generates a list of active senones based on the result of the pruning. The *Acoustic Model* uses this feedback to avoid GMM computation for senones that are inactive.

In this thesis, we use the latest continuous acoustic models of Pocketsphinx for our analysis and evaluations of our proposals. Note that all the proposed techniques are generally applicable to any acoustic model based on GMMs, independently of the specific parameters such as the number of senones or the number of Gaussians.

2.3 Language Models

In addition to an acoustic model to recognize the most likely phonemes, large vocabulary continuous speech recognition requires the use of language model or grammar to select the most

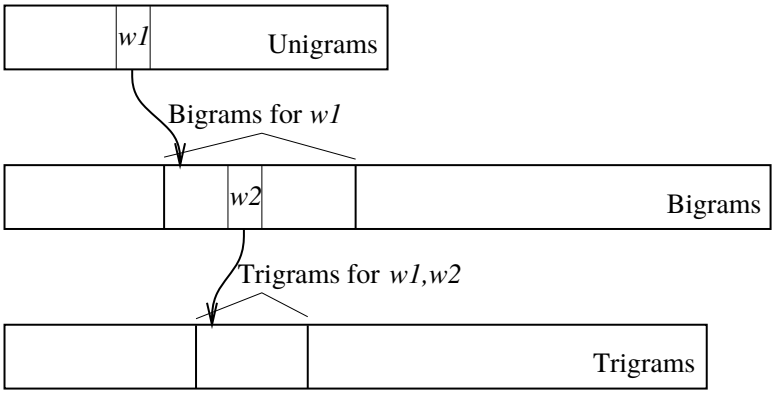


Figure 2.4: Language model structure in Pocketsphinx, our baseline ASR system.

likely word sequence among lots of alternative word hypothesis recognized during the search. While the acoustic model is the component of a speech recognizer which determines how closely a part of a spoken utterance matches a word or sequence of phones, the language model is the component which determines how likely a word or sentence is to have been spoken in the first place. The most obvious reason why this is necessary is that many words or sequences of words sound very similar, and it is not possible to decide among them without some prior knowledge of which ones are admissible or likely for a given language or domain. Mathematically, the language model is a statistical model of the probability distribution $P(S)$ over word sequences.

In a large vocabulary system, it is impossible to represent all possible context-free grammars or possible sequence of words compactly since there are too many of them. Similar to HMMs in acoustic modeling, practically all modern ASR systems use history-based or N-Gram models for language modeling. Another justification for the language model arises when we view recognition in terms of *Bayesian Decision Theory*. In this framework, speech recognition is the process of selecting between a (potentially infinite) set of alternate transcriptions for a sentence. Equivalently, we can think of this as a classification of the acoustic observation, where each class contains all the realizations of a particular sentence [27].

To model these conditional probabilities efficiently, the observation that the mutual information between a word and any predecessor word tends to decay with distance can be used. Therefore, we can consider all histories ending with the same m words to be part of an equivalence class. This is equivalent to making an assumption of conditional independence of the kind used in other Markov chains. Since the word and history taken together form a sequence of N words, we refer to a history-based model as an N-Gram model, though in fact, it is simply an $N-1$ order Markov chain over word sequences.

Because the number of words in a natural language is still extremely large, we encounter a number of unique problems in estimating the conditional probabilities which make up an $N-Gram$ model, and these are potential sources of modeling error. The most serious problem is that for any corpus of text, the majority of the words in the vocabulary may only occur a few times, and the overwhelming majority of N-Grams, that is, word sequences of length N, will never be observed. If

maximum likelihood estimation is used, the variance of the resulting probability estimates will be quite large, and the model will also assign zero probability to many plausible word sequences [27].

As the size of the basic vocabulary grows, N-Gram models encounter two issues with consequences for speech recognition. The first is that the number of possible N-Grams grows polynomially in the size of the vocabulary. As the number of N-Grams increases, so does the number of parameters to be trained, as well as the storage and memory space needed to store them. The second is that a larger vocabulary tends to increase the perplexity of the model, which has an adverse effect on the speed and accuracy of the recognizer, since it increases the size of the search space. Therefore, bi-gram and tri-gram grammars [85], consisting of word pairs and triples with given probabilities of occurrence can be created almost entirely automatically from a corpus of training text to be used as language models as Figure 2.4 shows.

2.4 Search Engine

The *Search Engine* transforms the sequence of phonemes into a sequence of words by executing the Viterbi beam search [81, 103, 104, 105] on a pre-compiled Hidden Markov Model (HMM) [81]. There are two main computational expensive components in speech recognition: acoustic probability computation and search. In the case of HMM-based systems, the former refers to the computation of the probability of a given HMM state emitting the observed speech at a given time. The later refers to the search for the best word sequence given the complete speech input. The search complexity is mostly unrelated to the complexity of acoustic models and it is heavily influenced by the size of the task. The search cost becomes significant for medium and large vocabulary recognition. In this thesis, the main focus will be on large vocabulary systems. The search problem has generally been addressed by Viterbi decoding [82, 100] using beam search [59].

2.4.1 Viterbi Decoding

Viterbi decoding is a dynamic programming algorithm that searches the state space for the most likely state sequence that accounts for the input speech. The state space is constructed by creating word HMM models from triphone HMM models, and all word HMM models are searched in parallel. The beam search heuristic is usually applied to limit the search by pruning out the less likely states due to huge number of states even in a medium vocabulary. Viterbi algorithm is a time-synchronous search algorithm in that it processes all states completely at time t before moving on to time $t+1$. The complexity of Viterbi decoding is N^2T (assuming each state can transition to every state at each time step), where N is total number of states and T is the total duration of the input speech in number of frames.

The abstract algorithm can be understood with the help of Figure 2.5. Y-axis represents the HMM states in the network and the other dimension is the time axis. Typically, there is one start state and one or more final states. The time-synchronous nature of the Viterbi search implies that the 2-D space is traversed from left to right, starting at time 0. The algorithm is summarized by the following expression where $P_j(t)$ is the path probability of state j at time t, a_{ij} is the static probability associated with the transition from state i to j, and $b_i(t)$, which is computed using

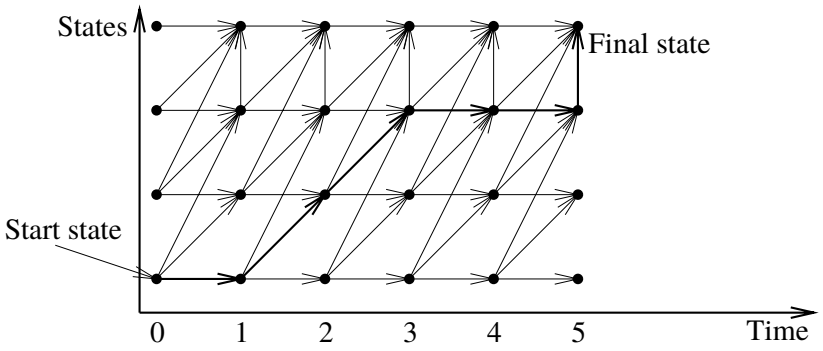


Figure 2.5: Viterbi search as Dynamic Programming [85].

GMM, is the output probability associated with state i while consuming the input speech at time t.

$$P_j(t) = max_i(P_i(t-1) \cdot a_{ij} \cdot b_i(t)), i \ \in set\ of\ predecessor\ states\ of\ j \tag{2.6}$$

The beam search heuristic reduces the average cost of search by orders of magnitude in medium and large vocabulary systems. The combination of Viterbi decoding using beam search heuristic is often referred to as *Viterbi beam search.*

2.4.2 Tree Structured Lexicons

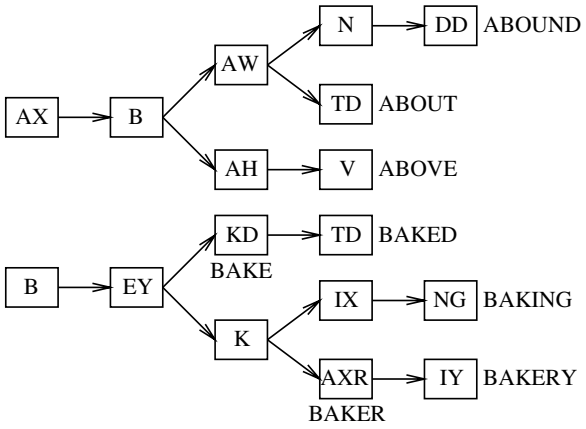
Even with the beam search heuristic, straightforward Viterbi decoding is still expensive. There exist numerous solution to reduce the search cost such as the use of a lexical-tree instead of a flat lexicon as Figure 2.6 shows. In such an organization, if the pronunciations of two or more words contain the same n initial phonemes, they share a single sequence of n HMM models representing that initial portion of their pronunciation. Lexical tree is used as a solution to eliminate three main sources of computational cost in other systems:

- Lexical tree introduces a high degree of sharing at the root nodes which results in significant reduction in the number of word initial HMMs that need to be evaluated every frame.
- The tree structure reduces the number of cross-word transitions by orders of magnitude which is always a dominant part of search in a systems based on flat lexicon.
- In lexical tree growth of the number of active HMMs and the number of word-transitions are much more slowly with increasing the vocabulary size in comparison with the flat lexicon.

In Figure 2.6, we saw the construction of a lexical tree of base phone nodes. However, it is preferred to use triphone acoustic models rather than simple base phone models for high recognition accuracy. Hence, the lexical tree has to be built out of triphone nodes rather than basephone nodes.

ABOUND	AX B AW N DD
ABOUT	AX B AW TD
ABOVE	AX B AH V
BAKE	B EY KD
BAKED	B EY KD TD
BAKER	B EY K AXR
BAKERY	B EY K AXR IY
BAKING	B EY K IX NG

(a) Pronunciation lexicon example



(b) Basephone lexical tree

Figure 2.6: A basephone lexical tree example.

This basically requires a trivial change to Figure 2.6, except at the roots and leaf positions of the tree (corresponding to word beginnings and endings), which have to deal with cross-word triphone models.

In a time-synchronous search, the phonetic right contexts are unknown since they belong to words that would occur in the future. Therefore, all phonetic possibilities have to be considered. This leads to a *right context fanout* at the leaves of the lexical tree.

On the other hand, the phonetic left context at the roots of the lexical tree is determined dynamically at run time, and there may be multiple contexts active at any time. However, a fanout at the roots, similar to that at the leaves, is undesirable since the former are active much more often. Therefore, cross-word triphones at the root nodes are modeled using the dynamic triphone mapping technique [27]. It multiplexes the states of a single root HMM between triphones resulting from different phonetic left contexts.

Figure 2.7 depicts the earlier example shown in Figure 2.6, but this time as a triphone lexical tree. The notation $b(l, r)$ in this figure refers to a triphone with basephone b, left context phone l, and right context phone r . A question-mark (?) indicates an unknown context that is instantiated dynamically at run time.

The degree of sharing in a triphone lexical tree is not as much as in the basephone version, but it is still substantial at or near the root nodes. The degree of sharing is very high at the root nodes, but falls off sharply after about 3 levels into the tree.

Pocketsphinx, our baseline ASR system, uses a lexical-tree. Lexical trees can be used to reduce the size of the search space. Since many words share common pronunciation prefixes, they can also share models to avoid duplication [85].

In a lexicon tree decoder, transitions between phones must also be treated separately from

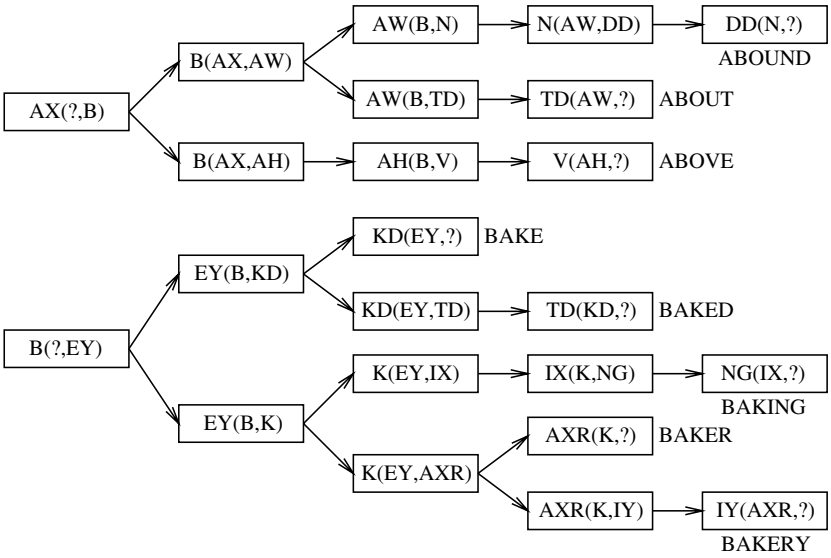


Figure 2.7: A triphone lexical tree example.

normal HMM transitions, with propagation of tokens through the lexicon tree. One specific problem with lexicon tree decoding arises from the fact that the identity of a hypothesized word is not known until a leaf node in the lexicon tree is reached. If a single, static lexicon tree is used, it is not possible to apply language model scores until the final phone of a word has been entered.

This results a similar problem to the one encountered with trigram scoring in flat lexicon search, except that it also occurs for bigram language models. As shown in Figure 2.8, only the predecessor word with the highest path score is propagated to the point where the language model score is applied. However it is possible, and in fact quite likely, that another predecessor word would have been preferred had the language model score been available at word entry [27]. In order to solve this problem, a multiple pass search strategy is proposed which is explained in the following section.

2.4.3 Multiple Pass Search

The output of speech recognition systems typically consist not only of a single hypothesized word sequence but also a word lattice which is an encoding of a very large number of alternative sentence hypotheses [73]. This word lattice is an approximate, finite representation of the space of sentences S which has been searched by the decoder. It is also frequently the case that speech recognition systems use a multi-pass search strategy [89], which implicitly involves a multi-stage reduction in the search space. For example, in the Pocketsphinx system [47] used as the baseline in this thesis, a three-pass search strategy inherited from the earlier SPHINX-II system [44, 85] is used.

In the Pocketsphinx decoder, an approximate first-pass search, using a static lexicon tree, is first used to generate a short-list of words at each frame. This search strategy suffers from widespread

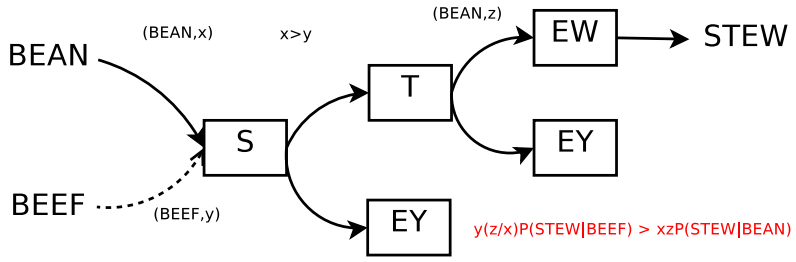


Figure 2.8: Search Problem: A single lexicon tree does not retain N-Gram history.

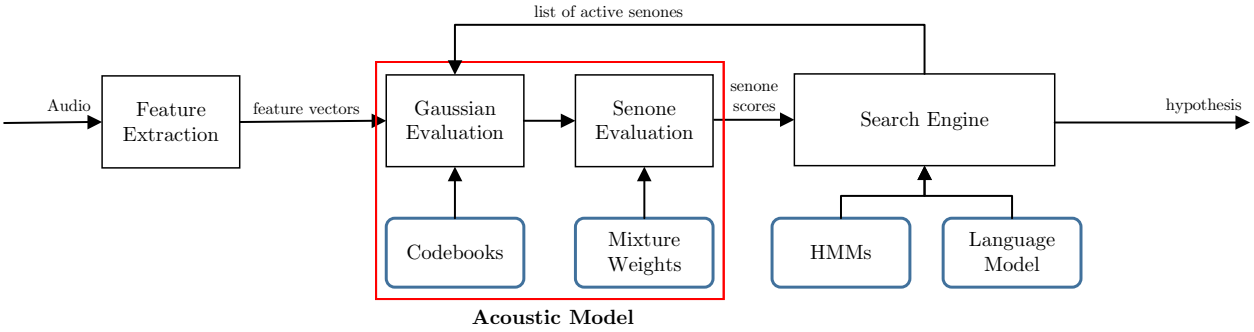


Figure 2.9: Pocketsphinx decoder architecture.

search errors. Therefore, the second pass uses a flat lexicon search, but uses the short-list generated in the first pass to restrict the set of words to be searched in each frame to a manageable number. However, this search algorithm uses an approximate trigram scoring technique, where only the best 2-word history is considered when applying language model scores at word transitions. To compensate for this, then the third pass performs an A* search over the resulting word lattice, allowing all trigram histories to be considered.

This organization of multiple passes is designed such that each successive pass is more exhaustive than the previous one, but also searches a more restrictive space of hypotheses. In this way, the known deficiencies of the earlier passes, such as the search error problem in the case of a static lexicon tree, and the approximate trigram problem in the case of a simple flat lexicon search, are corrected by a subsequent pass of search. In terms of computation requirements, the first pass is the most compute-intensive pass whereas the third pass (A* search) consuming less than 1% of the decoding time.

2.4.4 Pocketsphinx Architecture

The architecture of Pocketsphinx is largely inherited from Sphinx-II, its predecessor. A high-level overview of the Pocketsphinx decoder is shown in Figure 2.9.

A three-pass search strategy is implemented in Pocketsphinx consisting of the following stages:

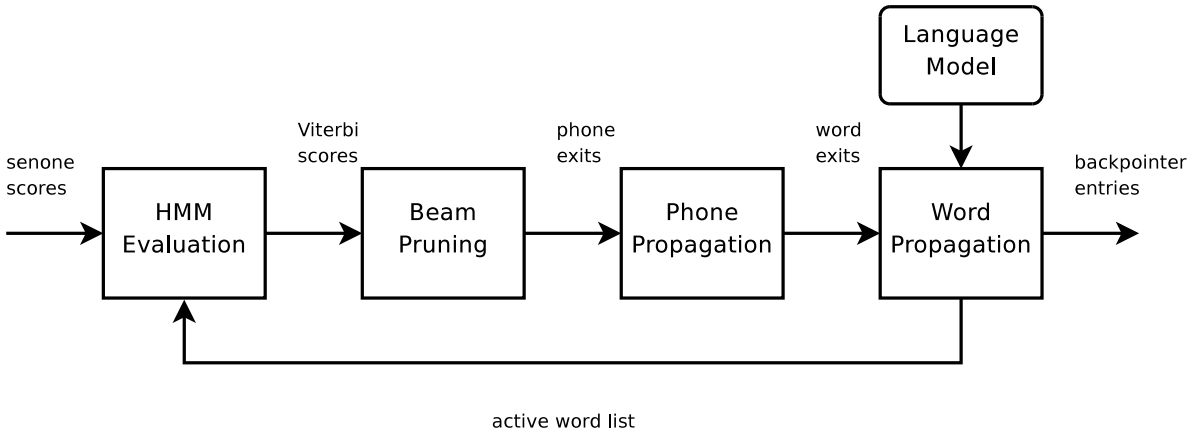


Figure 2.10: Forward search in Pocketsphinx [27].

- Forward-Tree - A Viterbi beam search using a lexicon tree.
- Forward-Flat - A Viterbi beam search using a flat lexicon.
- Best Path - A word graph search.

The general architecture of the two forward search passes is shown in Figure 2.10. The forward lattice from lexicon tree search is used to generate a frame-by-frame list of expansion words, which forms the dynamic vocabulary list which is searched by the flat lexicon search. Finally, A* search over the generated word lattice considers all the trigram histories and determines the most probable hypothesis.

In this thesis, unless explicitly stated otherwise, we use a generic English continuous acoustic model trained and provided by CMU (en-us-5.2), a 70K words trigram language model and a 130K word dictionary for our experiments. We focus on continuous acoustic models as they provide the highest accuracy in comparison with PTM and semi-continuous models. We use Pocketsphinx version *5-prealpha* and all our proposals and schemes in this thesis are implemented and evaluated using this version.

3

Software Optimizations

In this chapter, we provide our optimizations at software level in order to improve performance and energy-efficiency of our baseline ASR system. In section 3.1 we presents the results of the power/performance analysis of Pocketsphinx. Then, section 3.2 presents our software optimizations for GMM computation and section 3.3 describes our methodology. Section 3.4 discusses the performance and energy results. Finally, section 3.5 concludes the work presented in this chapter.

3.1 Energy-Performance Analysis

In this section, we provide a detailed power/performance analysis of a CPU when running Pocketsphinx with the continuous acoustic model. First, we describe the bottlenecks in the software and the main sources of stalls in the CPU pipeline. Second, we relate those CPU pipeline stalls with the source code of the GMM computation, identifying the branch instructions and memory accesses that cause such stalls. Finally, we complete the analysis with an energy characterization of Pocketsphinx.

3.1.1 Performance Characterization

We first profile the execution time of the different stages in an ASR pipeline. We run Pocketsphinx on an ARM mobile CPU with the parameters shown in Table 3.3. The results are provided in Figure 3.1. As it shows, the *Acoustic Model* takes up the bulk of execution time, as it requires 82.4% of the total time to convert the speech into words. On the other hand, the *Search Engine* and the *Feature Extraction* take 16.1% and 1.42% of execution time respectively. Therefore, *Acoustic Model* is clearly the main bottleneck.

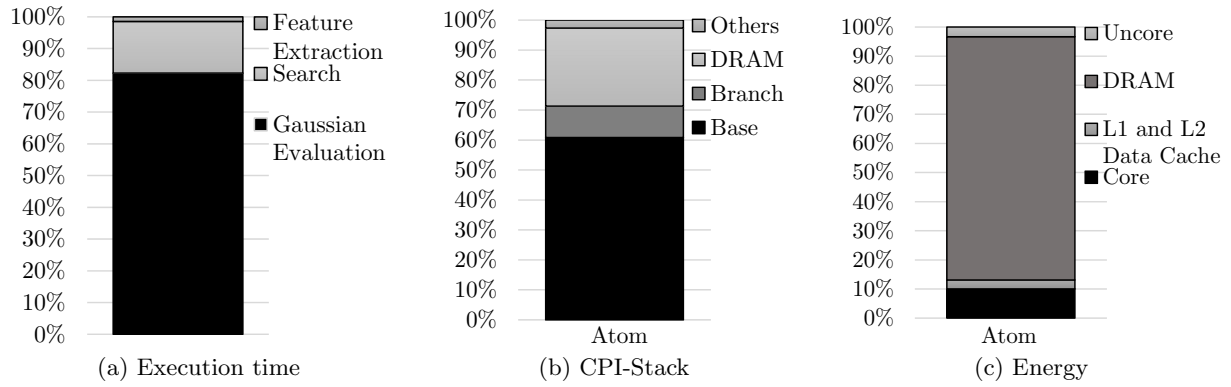


Figure 3.1: Summary of results for power/performance analysis of Pocketsphinx.

Figure 3.1(b) shows the CPI stack for an Atom-like processor, whose parameters are provided in Table 3.1, when executing the *Acoustic Model*. We run Pocketsphinx on the Sniper simulator [21] to generate the CPI stack. Accesses to off-chip system memory (DRAM) and branch mispredictions (Branch) are the main sources of stalls in the CPU pipeline as the CPI-stack shows. A more detailed analysis of the GMM evaluation code provides more insights on the sources of such CPU stalls.

Listing 3.1 shows the *Acoustic Model* implementation in Pocketsphinx. The function *GMM* is called for every senone, i. e. Gaussian mixture, on a frame basis. *GMM* evaluates the different Gaussian functions in the m-th mixture for a given feature vector x, implementing the computation described in Equation 2.5 with a few optimizations that work as follows. First, all the computations that do not depend on the input feature vector x are precomputed offline. The determinant vector, *det*, stores the value of $D_{m,g}$ for every Gaussian in the mixture. In a similar way, the matrix *vars* (short for variances) stores the result of $1/2\sigma_{m,g,c}$ for every component in the mixture. Regarding the second optimization, in an attempt to reduce the amount of computation only the N Gaussians with highest likelihood are used to compute the senone scores. The default value of N is 4 in Pocketsphinx. Therefore, the top 4 Gaussians are selected in *GMM* and only those four will be used to compute the final score. However, in order to select the top 4 Gaussians all the Gaussians in the mixture have to be computed. Nevertheless, computing all the components for each Gaussian might not be necessary as the likelihood is a continuously decreasing function. So if the likelihood of a Gaussian becomes smaller than the worst likelihood in the current top 4, we are sure this is not one of the 4 best Gaussians and, hence, we can stop computation for that Gaussian. Lines 9-10 of Listing 3.1 implement this optimization. Furthermore, lines 12-13 take care of inserting the Gaussian in the corresponding position in the sorted array of best Gaussians.

We computed the average number of iterations of the innermost loop in *GMM* function (lines 6-10). We found that on average 28.6 components are computed out of 36. This means a reduction of approximately 20% of the computation. However, this comes at the cost of having an if-sentence inside the innermost loop of the most critical function, which requires two extra x86 instructions per loop iteration to do a comparison and a branch. Moreover, we have another if-sentence after the innermost loop and extra code to insert the Gaussian in a sorted array. We found that the number of conditional branches when selecting the top 4 Gaussians increases by a factor of 1.42x with respect to a version that uses all the Gaussians. In addition, the total number of mispredicted

Listing 3.1: C-like pseudocode for acoustic model computation.

```
1  void GMM(int m, float *x, float *out) {
2    for (i = 0; i < top_N_Gau; i++)
3      out[i] = worst_Value;
4    for (g = 0; g < num_Gaussians; g++) {
5      float val = det[m][g];
6      for (c = 0; c < num_Components; c++) {
7        float diff = x[c] - means[m][g][c];
8        val -= diff * diff * vars[m][g][c];
9        if (val < gauval[top_N_Gau - 1])
10         break; // Not in top N
11     }
12     if (val >= gauval[top_N_Gau - 1])
13         InsertInSortedList(out, val);
14   }
15 }
```

branches increases by a factor of 1.79x. According to our experiments, these branches are the main responsible for the "Branch" section of the CPI stack in Figure 3.1(b). Section 3.2.1 targets this issue and proposes a method to remove branches inside the innermost loop.

3.1.2 Memory Characterization

The main source of CPU stalls is the latency of accesses to system memory, labeled as DRAM in Figure 3.1(b). Those memory accesses are mainly for fetching Gaussian parameters, i. e. *means* and *variances (vars)*. These parameters cannot be stored in the on-chip caches due to the big memory footprint. There are 5138 senones in the generic English continuous acoustic model of Pocketsphinx. Each senone has its own GMM that consists of 32 Gaussians of 36 components each one. The array *det* stores the determinant for each Gaussian, so its dimensions are 5138 *senones* $\times$ 32 *gaussians* and it requires approximately 0.63 MBytes. The dimensionality of *means* and *vars* matrices is 5138 *senones* $\times$ 32 *gaussians* $\times$ 36 *components*, so a total of 22.6 MBytes of system memory per matrix is required. Therefore, the total memory footprint for the Gaussian parameters is 45.7 MBytes.

As discussed in chapter 2, only Gaussian parameters for active senones in a given frame are fetched from memory. Figure 3.2 shows the number of active senones vs frames of speech. On average, 2675 of the senones are active for a given frame out of 5138, which means that 52% of the GMMs are evaluated per frame. This still requires a total of 23.8 Megabytes of memory per frame. Even if a senone is active for a sequence of consecutive frames, the CPU cannot exploit this frame-to-frame reuse due to the big memory footprint for processing one frame of speech. Figure 3.3 shows the bandwidth usage and memory footprint breakdown in Pocketsphinx. As it shows, acoustic model parameters consume less than 40% of the total required memory. However, as the figure shows, 79% of the memory bandwidth is used to fetch the GMM parameters due to the per-frame GMM evaluation. Note that the aforementioned bandwidth usage is only to fetch parameters of active senones.

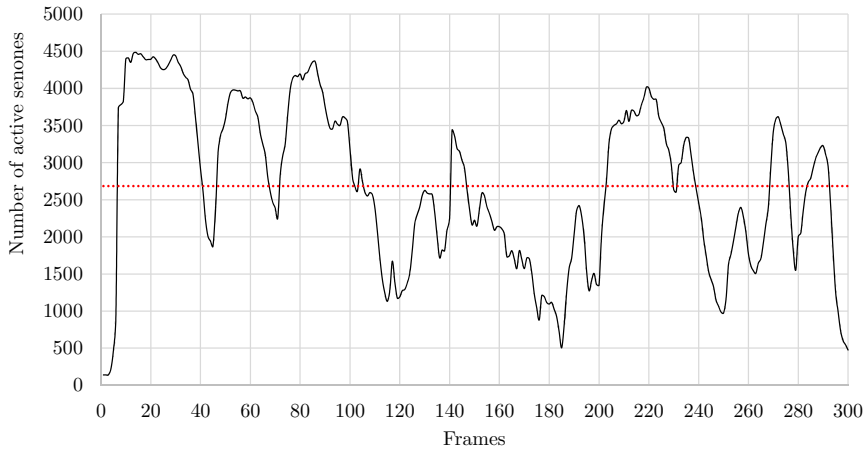


Figure 3.2: Number of active senones vs frames of speech. Red dotted line shows average number of active senones per frame, which is 2675 out of 5138.

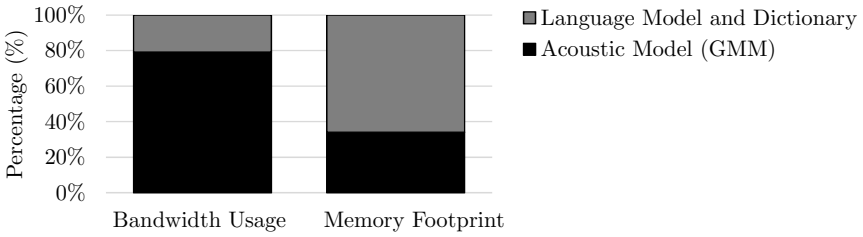


Figure 3.3: Memory bandwidth usage and memory footprint in Pocketsphinx.

CPU caches can still exploit some degree of spatial locality. Computing a GMM requires accessing in row-major order two 2D matrices of 32×36 floating point components (*means* and *vars* in Listing 3.1). Due to selection of the top 4 Gaussians some rows are not accessed completely, but still the average number of columns fetched is 28.61 and, hence, the access pattern to those matrices is very close to a pure row-major order.

The innermost loop of *GMM* function (lines 6-10 in Listing 3.1) includes three memory accesses for fetching the corresponding input vector x, the mean and the variance. The input vector of each frame requires just 144 bytes (36 fp elements) and exhibits high temporal reuse as it is the same for all the Gaussians. Memory accesses to *means* and *vars* matrices exploit spatial locality at the line level, so only the first access to a memory line misses in the L1. For a line size of 64 bytes, 16 elements are stored per line, so the miss ratio is 1/16. With those access patterns the miss ratio in the L1 is close to 4% as illustrated in Figure 3.4. Spatial locality is exploited only in the L1, and there is no temporal locality for means and variances, so the miss ratios for L2 and L3 caches are close to 100% if prefetchers are not used. Hardware prefetchers can significantly reduce the miss ratios for L2 and L3, as illustrated in Figure 3.4. This reduction in miss ratio provides 2.27x speedup. Due to their huge impact on performance, the baseline CPU configuration that we use for the experiments always employs hardware prefetchers.

Despite the good hit rate in the L1, an Atom-like processor is not able to completely hide the memory latency. Note that the FP to memory access ratio is significantly low for GMM

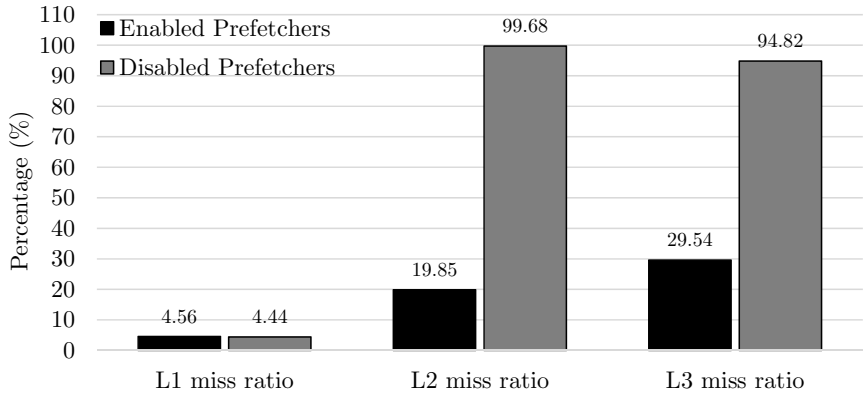


Figure 3.4: Miss ratios for different levels of memory hierarchy with and without prefetchers.

computation. Processing a component of a Gaussian requires fetching 12 bytes from memory (vector component, mean and variance) and only 4 fp operations are performed on these data. Therefore, the FP to mem ratio is 4 flops / 12 bytes = 0.33 flops per byte.

Regarding the memory bandwidth usage, we have previously mentioned that around 2675 senones are active on average. Each senone has a GMM with 32 Gaussians, and 28.6 components are processed on average for each Gaussian. So fetching the means requires reading 9.34 Megabytes from memory (2675 senones $\times$ 32 Gaussians/senone $\times$ 28.61 means/Gaussian $\times$ 4 bytes/mean). Fetching variances requires an equal amount of bytes from system memory. Furthermore, the determinants have to be read from memory, which require 0.33 Megabytes, as one floating point per Gaussian has to be fetched. So a total data of 19 Megabytes are required per frame. Since cache capacities are smaller (even for the last level cache), the 19 Megabytes are fetched from system memory every frame by regular memory accesses or prefetch requests. On the other hand, as one frame represents 10 ms of speech, in order to achieve real-time speech recognition at least 100 frames per second have to be processed. Hence, real-time ASR in Pocketsphinx requires a memory bandwidth of at least 1.85 Gigabytes/s only to fetch GMM parameters.

3.1.3 Energy Characterization

Figure 3.1(c) shows the energy breakdown for an Atom-like CPU when running Pocketsphinx. As it can be seen, the DRAM is the main source of energy drain, consuming 83.5% of the total energy. The CPU represents only 10% of the energy. The main consumers in the CPU are the FP units and the L1 data cache, requiring 3.1% and 3% of the energy respectively.

The poor temporal locality, which forces memory accesses to fetch Gaussian parameters from system memory on a frame basis, is the reason why DRAM consumes most of the energy. Section 3.2.4 proposes a technique to improve temporal locality in order to reduce the number of off-chip memory accesses and thus save DRAM energy.

Listing 3.2: GMM code without selection of *top N* Gaussians.

```
1  void GMM(int m, float *x, float *out) {
2    for (g = 0; g < num_Gaussians; g++) {
3      float val = det[m][g];
4      for (c = 0; c < num_Components; c++) {
5        float diff = x[c] - means[m][g][c];
6        val -= diff * diff * vars[m][g][c];
7      }
8      gauval[g] = val;
9    }
10 }
```

3.2 Pocketsphinx Optimizations

In this section, we describes our optimizations to boost GMM evaluation performance and reduce energy consumption.

3.2.1 Removing Branches in GMM Evaluation

As discussed, Pocketsphinx selects the *top N* Gaussians to reduce the number of iterations in the innermost loop of *GMM* function (see lines 6-10 in Listing 3.1) by 20.5%. However, it requires an additional branch instruction, with its corresponding compare instruction in x86, to branch outside the loop when a Gaussian is discarded. In other words, this optimization reduces the number of components computed, with a subsequent reduction in the number of floating point (FP) operations, but at the expense of increasing the number of conditional branches and compare instructions.

We used a refactored version of the *GMM* function to remove all the conditional branches that are due to the selection of *top N* Gaussians as Listing 3.2 shows. In this version each iteration is simpler as we remove the conditional branch and compare instructions that are required to check whether the Gaussian must be discarded. We refer to this version as *All32*, as all the 32 Gaussians in the mixture are always used to compute senone scores.

Listing 3.3 shows the x86 assembly code for the innermost loop of both implementations, the baseline version using the top 4 Gaussians and *All32*. Top 4 requires 4 FP operations (2 subtractions and 2 multiplications) per loop iteration. Furthermore, 2 conditional branches are included, one for branching at the beginning of the loop and one for branching outside the loop. As 28.6 iterations are performed on average, this version executes 114.44 FP operations and 57.22 conditional branches per senone.

On the other hand, *All32* version requires the same 4 FP operations, but only one conditional branch is executed per loop iteration. Since 36 iterations are performed per senone, *All32* executes 144 FP operations and 36 conditional branches. Therefore, this implementation increases FP operations by 25.8%, but it reduces conditional branches by 37%. We found that *All32* outperforms

Listing 3.3: x86 code GMM evaluation.

```
1  Top N Gaussians:          All 32 Gaussians:
2  add 0x4, rax              movss (rdi,rax,4),xmm0
3  ucomiss xmm2,xmm1         subss (rcx,rax,4),xmm0
4  jb 42a0cd                 mulss xmm0,xmm0
5  movss (rdx,rax,1),xmm0    mulss (r15,rax,4),xmm0
6  cmp rsi,rax               add 0x1,rax
7  subss (rcx,rax,1),xmm0    cmp eax,esi
8  mulss xmm0,xmm0           subss xmm0,xmm1
9  mulss (r15,rax,1),xmm0    jg 429fe0
10 subss xmm0,xmm1
11 jne 42a010
```

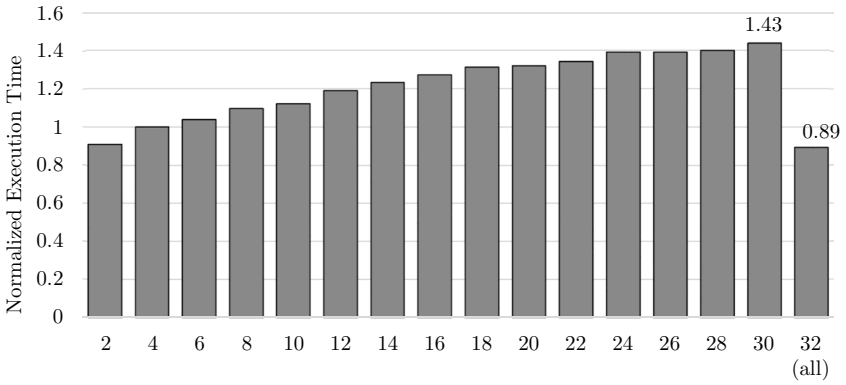


Figure 3.5: Normalized execution time for different *top N* Gaussians with respect to *top 4*.

the version that selects top 4 Gaussians, as reported in section 3.4.

Using the *top N* Gaussians was proposed as an optimization [76] in 2001, targeting significantly different pipelines for which FP operations were more expensive and branches were less costly than in today's microprocessors. Our experimental results show that this technique is not beneficial for modern CPUs, as these CPUs excel in FP performance but conditional branches are one of the main sources of stalls.

Figure 3.5 shows the normalized execution time for selecting different top Gaussians with respect to the baseline (top 4). As the figure shows, selecting more number of top Gaussians results in significant increase in the execution time. While selecting the *top N* Gaussians seems to reduce the floating-point operations, a significant increase in the number of conditional branches results in more stalls and performance degradation.

By using *All32* version, percentage of execution time due to GMM evaluation is reduced to 72% of the total execution time. Although removing the branches results in considerable performance improvements, GMM evaluation is still the main bottleneck in Pocketsphinx.

Listing 3.4: Vectorized version of GMM evaluation, using Intel SSE.

```
1  void GMM(int m, float *x, float *out) {
2    for (g = 0; g < num_Gaussians; g++) {
3      __m128 vval = {0, 0, 0, 0};
4      float *gaum = means[m][g];
5      float *gauv = vars[m][g];
6      for (c = 0; c < num_Components; c += v_Size) {
7        __m128 vx     = _mm_load_ps(x + c);
8        __m128 vmeans = _mm_load_ps(gaum + c);
9        __m128 vvars  = _mm_load_ps(gauv + c);
10       vx = _mm_sub_ps(vx, vmeans);
11       vx = _mm_mul_ps(vx, vx);
12       vval = _mm_fmadd_ps(vx, vvars, vval);
13     }
14     gauval[g] = det[m][g] - AddComponents(vval);
15   }
16 }
```

3.2.2 Vectorization

SIMD or vector instructions are widely supported by modern processors. All major vendors include SIMD instructions in their ISAs (e.g. Intel SSE/AVX [58] or ARM Neon [2]). The VPU increases performance due to its higher FP throughput. Furthermore, it improves energy efficiency by reducing the number of instructions fetched and decoded, since multiple FP operations are packed in one vector instruction.

GMM evaluation code is a good candidate for vectorization as the innermost loop iterations are independent. However, each iteration only includes four FP operations that must be executed sequentially and, hence, they cannot be packed in the same SIMD instruction. Due to this lack of independent FP operations, the innermost loop cannot be efficiently vectorized as it is. One effective way of exposing more independent FP instructions is using loop unrolling. The unrolled and vectorized version of the code is shown in Listing 3.4. We use unrolling factor equal to the SIMD width, i. e. vector size. In addition, we exploit FMA instructions to merge the last two FP operations performed in the scalar loop, in order to further reduce instruction count.

Our experimental results, discussed in section 3.4, show that the SIMD version provides significant speedups and energy savings. However, this version requires a horizontal reduction to add the components of the vector register *vval* (see line 14 in Listing 3.4). This reduction implies scalar operations on individual components of a vector. The presence of scalar instructions constrains the speedups achieved by vectorization.

3.2.3 Improved Memory Layout

In the SIMD version, multiple components of a Gaussian are computed at the same time by using multiple SIMD lanes. For a vector size of 4, each one of the 4 SIMD lanes computes and

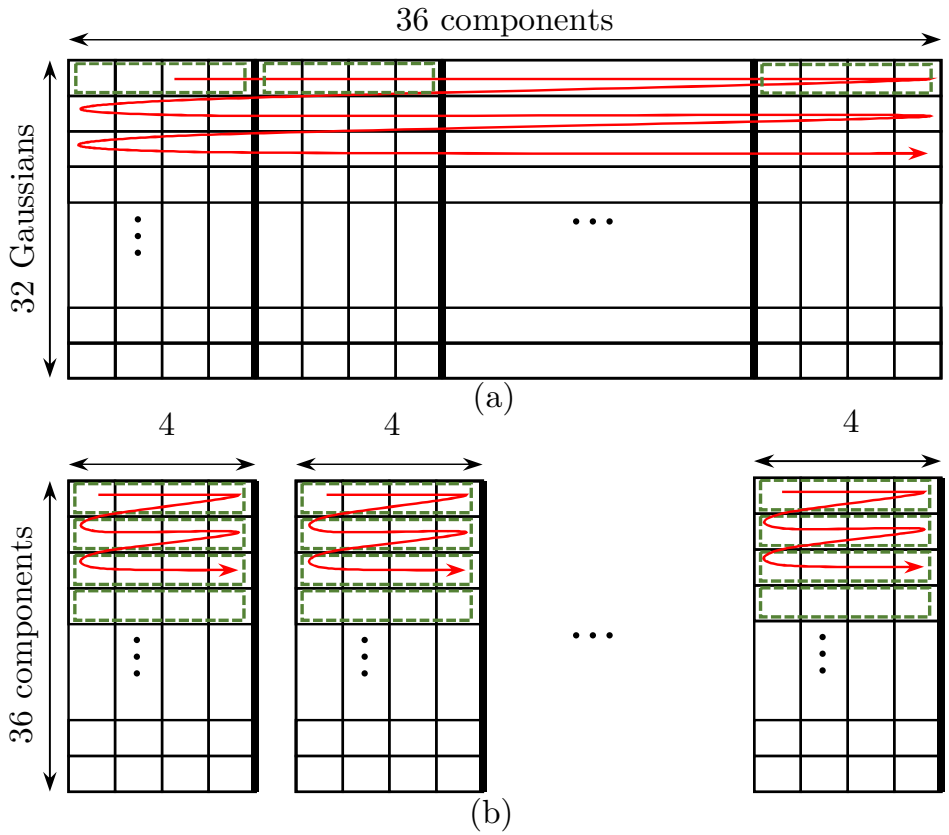


Figure 3.6: Memory layout and memory access pattern. (a) the baseline and (b) the transposed layout.

accumulates the values for 9 components. In order to get the acoustic likelihood, the aggregated value for the 36 components has to be computed, so we have to add the 4 partial results obtained by the 4 SIMD lanes. This reduction is performed using scalar instructions. To maximize the impact of vectorization, the amount of scalar code must be reduced as much as possible.

In an attempt to maximize the amount of vectorizable code, we propose to use the SIMD lanes to compute multiple Gaussians at the same time, instead of processing multiple components of one Gaussian. By doing this, the accumulated value on each SIMD lane is the final likelihood for one Gaussian and, hence, no horizontal reduction is required. This implementation requires some changes to the memory layout of *means* and *vars* matrices, as it is not possible to fetch the same component (or column) for different Gaussians (or rows) with one vector load, as they are not stored consecutively in memory.

We propose to change the memory layout for these matrices to the one illustrated in Figure 3.6(b). Transposing the matrix solves the problem with the vector load, as now we fetch consecutive columns (or Gaussians) in the same row (or component). However, by transposing the matrix we lose some spatial locality, as the traversal is performed in column-major order. Note that each SIMD lane has to process and accumulate results for one column. Our solution to this problem is to split the matrix in smaller sub-matrices with a number of columns equal to the SIMD width.

Listing 3.5: SSE version of GMM with improved memory layout.

```
1  void GMM(int m, float *x, float *out) {
2    for (g = 0; g < num_Gaussians; g += v_Size) {
3      __m128 vval = {0, 0, 0, 0};
4      float **gaum = means[m][g/v_Size];
5      float **gauv = vars[m][g/v_Size];
6      for (c = 0; c < num_Components; c++) {
7        __m128 vx     = _mm_load_ps1(x + c);
8        __m128 vmeans = _mm_load_ps(gaum[c]);
9        __m128 vvars  = _mm_load_ps(gauv[c]);
10       vx = _mm_sub_ps(vx, vmeans);
11       vx = _mm_mul_ps(vx, vx);
12       vval = _mm_fmadd_ps(vx, vvars, vval);
13     }
14     __m128 vdet = _mm_load_ps(&(det[m][g]));
15     vval = _mm_sub_ps(vdet, vval);
16     _mm_store_ps(gauval + g, vval);
17   }
18 }
```

By doing this we store the data in memory in the same order it is accessed by the application, maximizing spatial locality.

The proposed memory layout eliminates the horizontal reduction, increasing the amount of vectorizable code. Therefore, we reduce the number of scalar instructions to a large extent, achieving significant performance and energy savings over the first SIMD version as shown in section 3.4. On the other hand, we found that GMM evaluation benefits from larger SIMD widths, achieving better results when using AVX (SIMD width of 8) than SSE (SIMD width of 4).

The proposed memory layout, in addition to eliminating the horizontal reduction, makes vectorization independent of the feature vector length. In other words, various sizes of feature vectors in different acoustic models will not affect the vectorizable code since we are not computing multiple components at the same time anymore. Therefore, we reduce the number of scalar instructions to a large extent, achieving significant performance and energy savings over the first SIMD version. The impact of this technique will be more obvious using larger SIMD widths.

The SIMD implementation of GMM computation using the new memory layout is included in Listing 3.5. Note that the loop unrolled in this version is the outer loop, which is the one that iterates on the Gaussians. After processing all the components each element of *vval* vector contains the total aggregated value for one Gaussian. We can subtract these values from the corresponding determinants and store the final likelihoods in memory using SIMD instructions (see lines 15-16 in Listing 3.5). Note that no horizontal reduction is required in this version. Therefore we reduce the number of scalar instructions to a large extent, achieving significant performance and energy savings over the first SIMD version, as detailed in section 3.4.

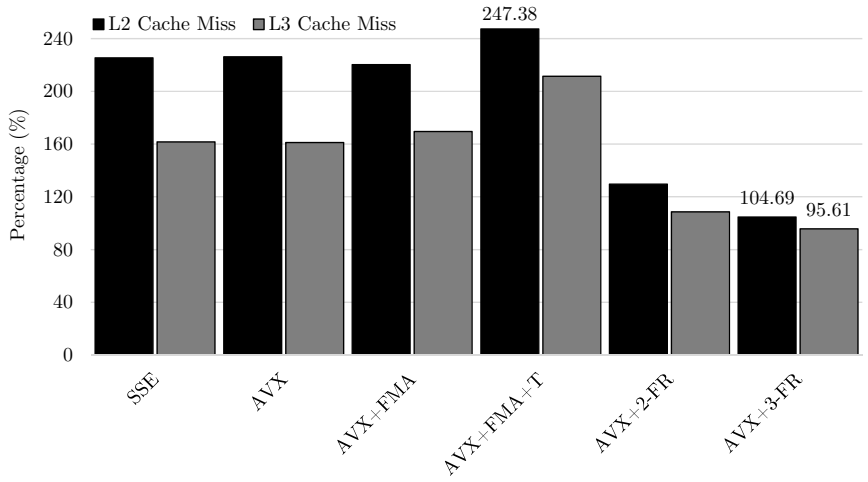


Figure 3.7: Number of L2 and L3 cache misses normalized to the baseline.

3.2.4 Multi-Frame Gaussian Evaluation

The power and performance analysis of Pocketsphinx, presented in section 3.1.2, identified system memory as the main source of both CPU stalls and energy drain. The SIMD implementation presented in section 3.2.3 exacerbates the problem due to memory latency. Due to the higher throughput of the VPU, the pressure on the memory subsystem increases because data is requested earlier than in the scalar implementation. Therefore, the prefetcher has less time to bring the data from memory and it is not able to achieve timeliness, which leads to an increase in the number of L2 and L3 misses, as shown in Figure 3.7.

As we describe in section 3.1, most of the memory bandwidth is used to fetch Gaussian parameters, i. e. means and variances. Those accesses exhibit poor temporal locality due to the big size of the dataset for one frame of speech. CPU caches cannot exploit frame-to-frame reuse and, hence, Gaussian parameters have to be fetched from system memory on a frame basis.

One approach to reduce the number of accesses to system memory is to evaluate Gaussians for multiple frames at the same time. By using multi-framing, means and variances for one Gaussian are fetched once in the on-chip caches and are used to evaluate the Gaussian in several frames of speech. For example, if we merge Gaussian evaluation for two frames, then the bandwidth usage is reduced by approximately one half, as means and variances are fetched from memory every other frame. Moreover, the number of FP operations per memory access doubles, alleviating the pressure on the memory subsystem and especially on the prefetcher.

The main hurdle for implementing the multi-frame approach is the lack of information about active senones for subsequent frames. The list of active senones is generated by the search after the pruning and it is only available for the current frame. We could decouple GMM evaluation from the search by ignoring the list of active senones and computing GMM for all of them. By doing this we could implement the multi-frame approach, but the workload would increase by 2x, as only half of the senones are active on average. Such a huge overhead renders this naive approach completely ineffective.

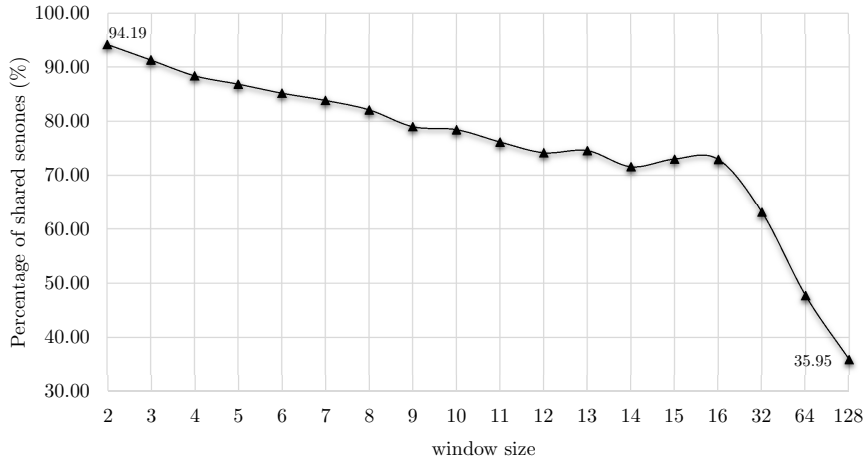


Figure 3.8: Percentage of shared active senones among consecutive frames for different window sizes.

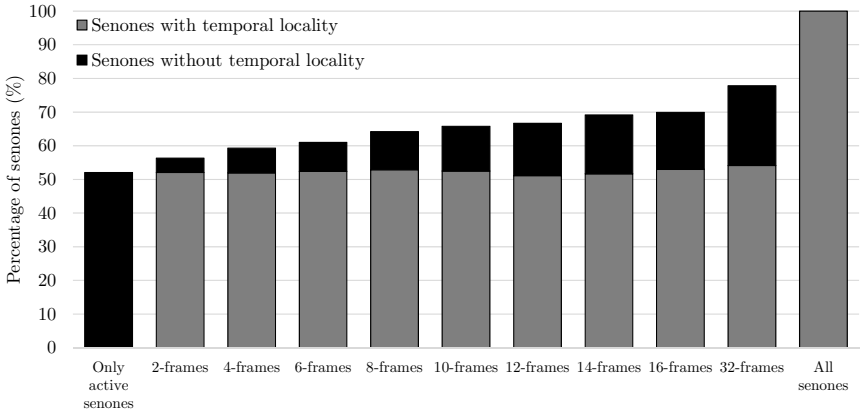


Figure 3.9: Gray bars show the percentage of senones computed with the multi-frame version, i. e. percentage of senones computed exploiting temporal locality. Black bars show the percentage of senones computed with the original single-frame code, i. e. the percentage of senones for which temporal locality is not exploited. For the versions computing 2-32 frames at a time, the black bars correspond to the mispredicted senones.

An analysis of the locality of active senones reveals a better strategy. The speech signal is quasi-stationary when considering small intervals, so the list of active senones tend to be similar for consecutive frames. Figure 3.8 shows the percentage of active senones that are shared among consecutive frames, for different window sizes. As we can see, considering a window of 2-3 frames more than 90% of the active senones are shared among those frames.

We use this observation to implement a simple prediction scheme. Our scheme predicts that the active senones for the next N - 1 frames will be the same as those in the current frame. GMM computation is triggered just once every N frames, using the multi-framing approach of fetching parameters once and computing the Gaussians for N frames. In addition, we include a

Listing 3.6: Multi-frame implementation of GMM evaluation.

```
1  void GMM(int s, float **x, float **gauval) {
2    for (g = 0; g < num_Gaussians; g += v_Size) {
3      __m128 vval[num_Frames];
4      memset(vval, 0, num_Frames * 16);
5      float **gaum = means[s][g/v_Size];
6      float **gauv = vars[s][g/v_Size];
7      for (c = 0; c < num_Components; c++) {
8        __m128 vmeans = _mm_load_ps(gaum[c]);
9        __m128 vvars  = _mm_load_ps(gauv[c]);
10       for (f = 0; f < num_Frames; f++) {
11         __m128 vx = _mm_load_ps1(x[f] + c);
12         vx = _mm_sub_ps(vx, vmeans);
13         vx = _mm_mul_ps(vx, vx);
14         vval[f] = _mm_fmadd_ps(vx,vvars,vval[f]);
15       }
16     }
17     __m128 vdet = _mm_load_ps(&(det[s][g]));
18     for (f = 0; f < num_Frames; f++) {
19       vval[f] = _mm_sub_ps(vdet, vval[f]);
20       _mm_store_ps(gauval[f] + g, vval[f]);
21     }
22   }
23 }
```

recovery mechanism to handle mispredictions. Two types of mispredictions are possible. The first type happens when we predict a senone as active but it is inactive. In this case we do not have to trigger any action, but we pay the overhead of computing an acoustic score that is not used during the search process. The second type happens when we predict a senone as inactive but it is active. In this case we trigger the single-frame version of Gaussian Evaluation to compute the score for the mispredicted senone. Figure 3.9 shows that this scheme is very effective as the number of mispredictions is very small (mispredicted senones correspond to the black bars, whereas correctly predicted senones are shown in gray bars), so we can exploit temporal locality for most of the senones. As we can see in both Figure 3.8 and Figure 3.9, increasing the number of frames computed at a time, i. e. the window size, increases the number of mispredicted senones, as the speech signal changes significantly at long distances. We obtained the best performance and power results by using small windows of 2-3 frames, since bigger windows suffer the overhead of mispredicted senones and provide diminishing returns in memory bandwidth savings. The multi-frame implementation of GMM computation is included in Listing 3.6.

3.3 Evaluation Methodology

We have evaluated our proposed techniques using two high-end Intel desktop CPUs, whose parameters are shown in Table 3.2. We use PAPI [69] hardware performance counters on Haswell

Core	Atom-like OoO, 2-wide, 1.33 GHz
L1-D Cache	24KB, 6-way, 64B lines, LRU
L2 Cache	1MB, 16-way, 64B lines, LRU
Main Memory	4 GB, 12.8 GB/s bandwidth
Branch predictor	Pentium M branch Predictor 15 cycles misprediction penalty
Technology	22 nm
Prefetchers	GHB, prefetch-degree = 2, table-size = 512

Table 3.1: Hardware parameters employed for the simulations.

	Haswell	**Skylake**
Core	Intel i5-4210M	Intel i7-6700
L1, L2, L3	32KB, 256KB, 3MB	64KB, 256KB, 8MB
Frequency	3.2 GHz	4.2 GHz
Main Memory	8GB DDR3	32GB DDR4

Table 3.2: Intel Haswell and Skylake parameters.

CPU	4x ARM Cortex A57
L1, L2	32KB L1, 2MB L2
Technology	20 nm
Frequency	1.9 GHz
Main Memory	4 GB LPDDR3

Table 3.3: Mobile CPU Parameters.

and Skylake processors to measure execution time, whereas we employ Intel RAPL [102] library to collect energy consumption. Furthermore, we have evaluated our optimizations on a low-power mobile ARM Cortex-A57 CPU with parameters shown in Table 3.3. We use the NVIDIA Tegra X1 [7] SoC to collect execution time. To measure energy consumption, we read the registers of the TI INA3221 power monitor included in the NVIDIA Jetson TX1 platform, in order to obtain power dissipation by monitoring CPU power rail as described in [62].

On the other hand, we use Sniper simulator [21] to collect further information of the CPU pipeline, including a complete CPI stack. We model a modern out-of-order mobile CPU, similar to an Atom Bay Trail processor [5]. The parameters for the experiments are included in Table 3.1. We use McPAT [57] to estimate energy consumption of the Atom-like processor. We use GCC version 4.8 in the x86 and ARM platforms and we employ -O3 optimization level.

We use standard audio files commonly employed to test ASR systems as our datasets. More specifically, we use LibriSpeech [75] test-clean corpus including 5.4 hours of audio files.

Our baseline for the experiments is the unmodified Pocketsphinx GMM implementation. We have evaluated the effect of our techniques on five different continuous acoustic models trained in Sphinx. The parameters of the different acoustic models are shown in Table 3.4.

Acoustic Model	#Mixtures	#Gaussians	Dimension
English 1	5138	32	36
English 2	6126	32	39
German	6198	16	29
Russian	5147	32	36
Greek	5102	32	36

Table 3.4: Parameters for the different continuous acoustic models that are employed to evaluate our proposed techniques.

3.4 Experimental Results

In this section, we analyze the performance and energy efficiency of the optimizations presented in section 3.2. The baseline configuration for all our experiments is the unmodified Pocketsphinx. Figure 3.10 shows the speedup and normalized energy achieved by all the optimizations on an Intel Haswell CPU, using the English1 acoustic model with parameters shown in Table 3.4. Note that we build each optimization on top of the previous one and the performance and energy improvements are measured for the entire application. The speedups and energy savings are reported for the entire ASR pipeline required to convert the speech into words, including the Feature Extraction and Viterbi Search in addition to the GMM evaluation. *All32* configuration improves performance by 12.8% and saves 11.2% energy by removing conditional branches in the innermost loop of GMM evaluation. *All32* reduces conditional branches by 37% with respect to the baseline, at the cost of increasing FP operations by 25%. Our results show that this is a good trade-off for modern CPUs, as the penalties introduced by branches are bigger than the cost of the extra FP operations.

The results for the straightforward SIMD implementation, introduced in section 3.2.2, are included for *SSE* and *AVX*. *SSE* employs a SIMD width of 4 and achieves 72.8% speedup and 34.4% energy savings. *AVX* version slightly improves the results to 76.9% speedup and 40.8% energy savings by using a SIMD width of 8. The use of Fused Multiply-Add (FMA) instruction further improves speedup to 78.8% and energy savings to 47.2% as shown in configuration *AVX+FMA*. The speedups of the SIMD version come from the higher FP throughput of the VPU. The energy savings come from the smaller execution time (static energy) and the reduction in instruction count (dynamic energy), as multiple scalar operations are packed in just one SIMD instruction.

Configuration *AVX+FMA+T* implements the improved memory layout presented in section 3.2.3. By using this layout for matrices that store means and variances the amount of vectorizable code increases, further improving speedup to 96% and energy savings to 47.5%.

Finally, configurations *AVX+2FR* and *AVX+3FR* implement the multi-framing scheme presented in section 3.2.4 using a window of 2 and 3 frames respectively to compute the Gaussians. *AVX+3FR* achieves the best results, providing 2.63x speedup and 61% energy savings. This multi-framing scheme reduces the number of accesses to system memory as means and variances are fetched every 3 frames instead of being fetched on a frame basis. This reduces DRAM energy and also improves performance by alleviating the pressure on the memory subsystem, as memory latency is the main performance limiting factor (see section 3.1.2).

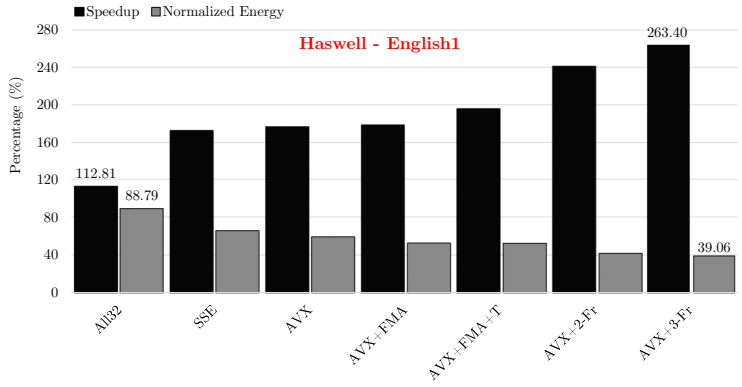


Figure 3.10: Speedup and normalized energy on Intel Haswell CPU. Baseline is unmodified Pocketsphinx with English1 acoustic model.

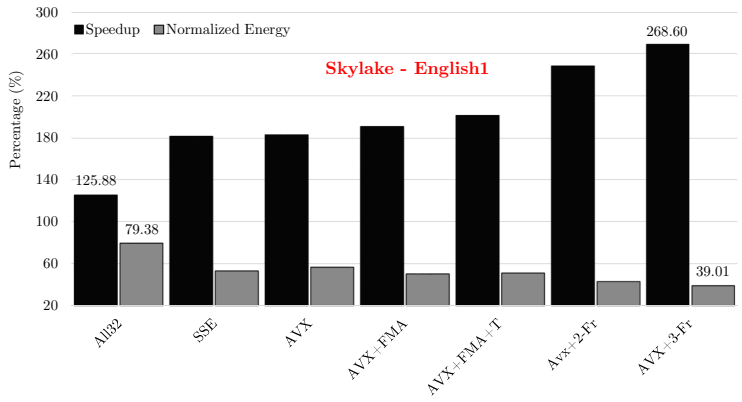


Figure 3.11: Speedup and normalized energy on Intel Skylake CPU. Baseline is unmodified Pocketsphinx with English1 acoustic model.

The multi-framing approach uses a prediction schemes that assumes that the active senones do not change for a group of N consecutive frames. This prediction is very effective for small sizes of N, such as 2 or 3, as the speech signal is quasi-stationary when considering small intervals. We found that 254 senones are mispredicted on average per frame out of 2929 senones computed, so misprediction rate is only 8.6%. In the single-frame version 9344 bytes/senone are fetched from system memory: 128 bytes for determinant array, 4608 bytes for means matrix and 4608 for variance matrix. Since 2675 senones are active on average, 23.83 Megabytes are fetched from system memory per frame. With *AVX+3FR* configuration, determinants, means and variances are fetched from system memory every 3 frames. Hence, the amount of data accessed per frame is reduced to 23.83/3 = 7.94 Megabytes. For multi-framing, we also have to consider memory accesses to compute mispredicted senones: 254 mispredicted senones/frame × 9344 bytes/senone = 2.26 MBytes. So the total amount of data fetched from memory per frame with *AVX+3FR* is 10.2 Megabytes, a reduction of 57.1% with respect to the single-frame version.

We tested the multi-framing approach using bigger numbers of frames, from 4 to 16. However, we obtained better results for small windows of just 2-3 frames for several reasons. First, as we increase the number of frames we get diminishing returns in bandwidth savings and, therefore,

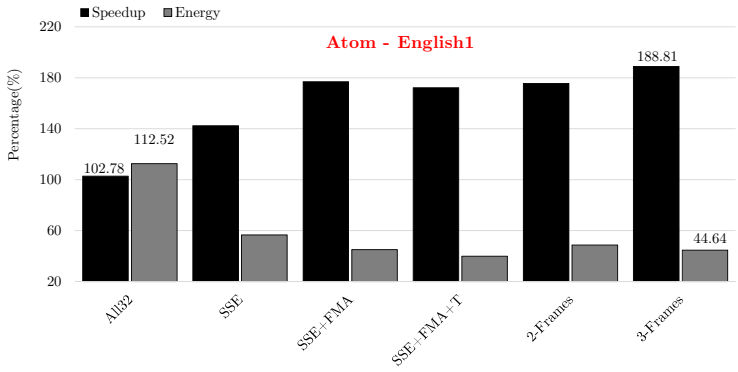


Figure 3.12: Speedup and normalized energy on Intel Atom CPU. Baseline is unmodified Pocketsphinx with English1 acoustic model.

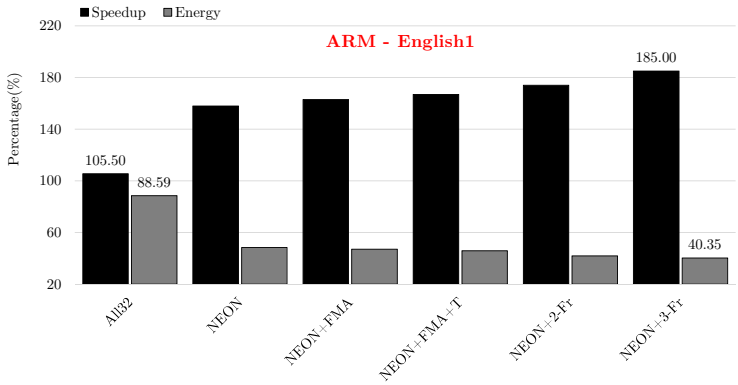


Figure 3.13: Speedup and normalized energy on ARM Cortex-A57 mobile CPU. Baseline is unmodified Pocketsphinx with English1 acoustic model.

in speedups and energy savings. Second, the number of mispredicted senones increases for bigger windows of frames (see Figure 3.8), increasing the overheads. Third, all the data for computing a Gaussian in multiple frames can be stored in the vector register file for small windows, but for a big number of frames L1 must be used, increasing memory pressure.

The proposed optimizations achieve similar speedups and energy savings when they are applied on an Intel Skylake CPU (Intel latest microprocessor), as illustrated in Figure 3.11. The best configuration, *AVX+3FR*, provides 2.68x speedup and 61% energy savings.

Since on of the main targets of Pocketsphinx are mobile devices, we have also evaluated our optimizations on a state-of-the-art ARM mobile CPU. The performance and energy results are shown in Figure 3.13. ARM CPUs take advantage of NEON extensions, which provide vector instructions with SIMD width of 4. Removing conditional branches provides 5.5% performance improvement while reducing energy consumption to 11%. Similar to the Intel desktop CPUs, removing conditional branches at the cost of increasing FP operations is also a good trade-off in the ARM CPU. We get the best results with the multi-frame implementation with window size of 3 frames, that provides 1.85x speedup and 59.65% energy savings.

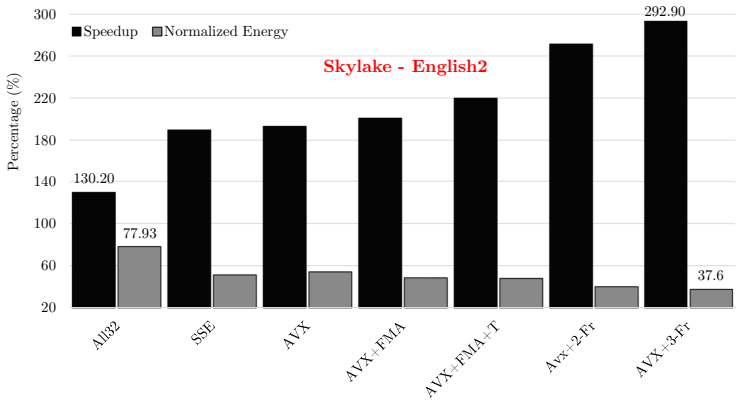


Figure 3.14: Speedup and normalized energy on Intel Skylake CPU, using English2 acoustic model. Baseline is unmodified Pocketsphinx.

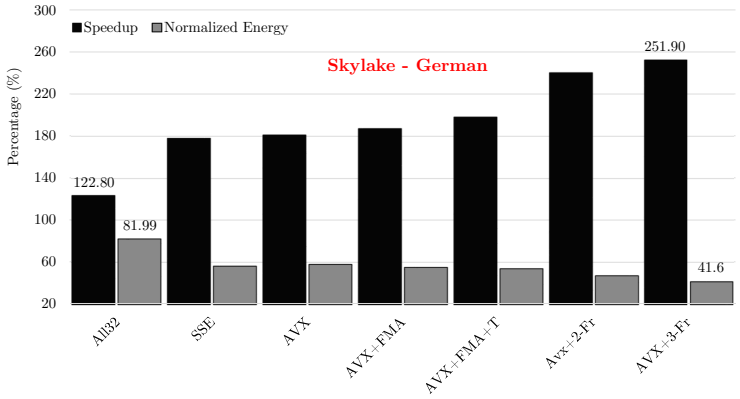


Figure 3.15: Speedup and normalized energy on Intel Skylake CPU, using German acoustic model. Baseline is unmodified Pocketsphinx.

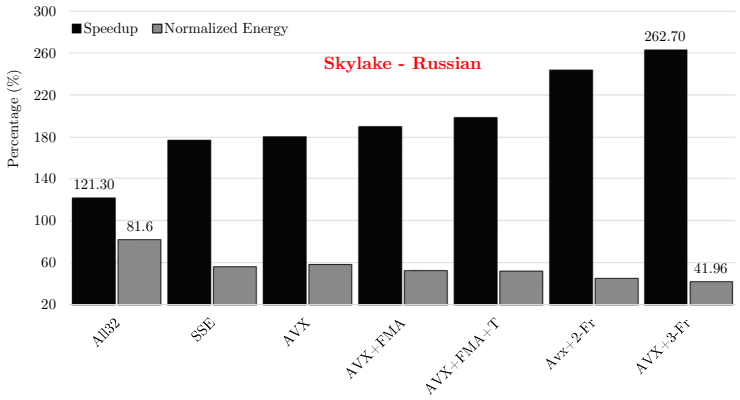


Figure 3.16: Speedup and normalized energy on Intel Skylake CPU, using Russian acoustic model. Baseline is unmodified Pocketsphinx.

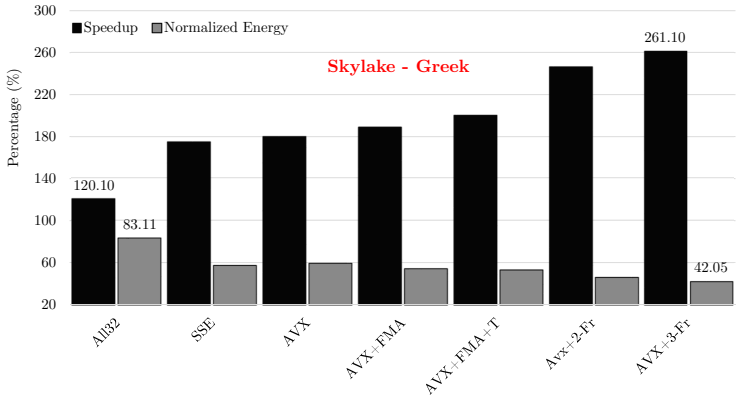


Figure 3.17: Speedup and normalized energy on Intel Skylake CPU, using Greek acoustic model. Baseline is unmodified Pocketsphinx.

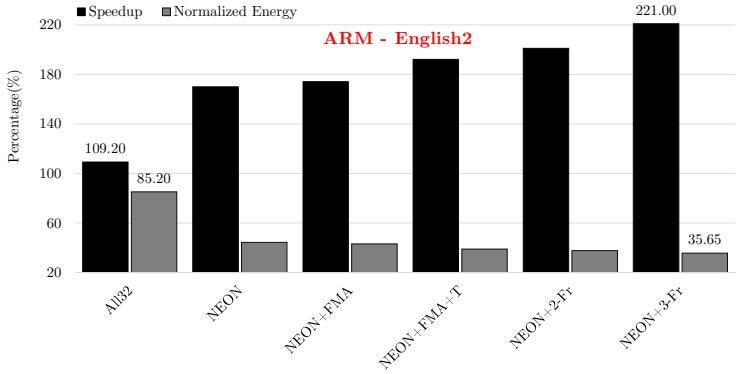


Figure 3.18: Speedup and normalized energy on ARM Cortex-A57 mobile CPU, using English2 acoustic model. Baseline is unmodified Pocketsphinx.

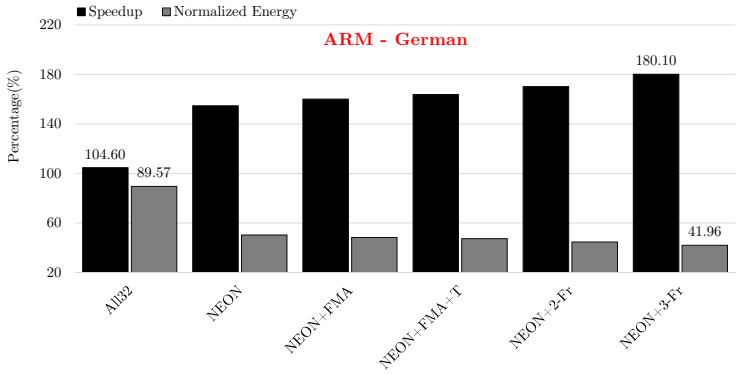


Figure 3.19: Speedup and normalized energy on ARM Cortex-A57 mobile CPU, using German acoustic model. Baseline is unmodified Pocketsphinx.

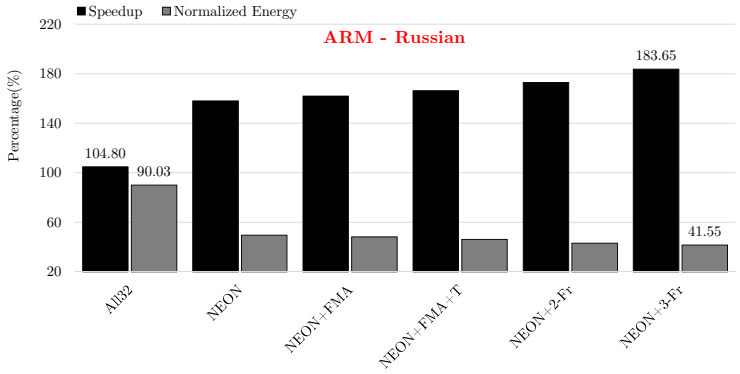


Figure 3.20: Speedup and normalized energy on ARM Cortex-A57 mobile CPU, using Russian acoustic model. Baseline is unmodified Pocketsphinx.

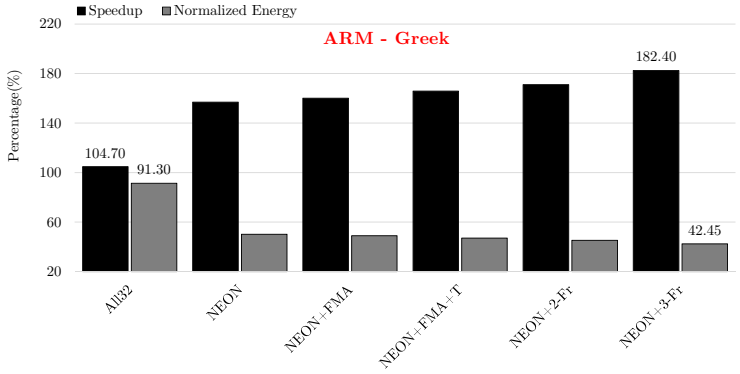


Figure 3.21: Speedup and normalized energy on ARM Cortex-A57 mobile CPU, using Greek acoustic model. Baseline is unmodified Pocketsphinx.

Regarding the use of desktop processors in the evaluations, we consider that it is also important to optimize ASR for high-end processors. Energy consumption is also an important issue for desktops, due to heat dissipation, and for servers, as it affects the cost of operating data centers.

To illustrate the general applicability of our techniques, we have evaluated the speedups and energy savings for different acoustic models. Figure 3.14, Figure 3.15, Figure 3.16 and Figure 3.17 show the results on the Intel Skylake CPU for the English2 ,German, Russian and Greek acoustic models respectively, whose parameters are shown in Table 3.4. As it can be seen, our techniques provide substantial speedups and energy savings for acoustic models with different parameters that target different languages. The configuration *AVX+3FR* achieves the best results, the speedups for the different acoustic models range between 2.51x (German) and 2.92x (English2). Regarding the energy savings, *AVX+3FR* reduces energy by 62.4% and 58.4% for English2 and German respectively.

Finally, Figure 3.18, Figure 3.19, Figure 3.20 and Figure 3.21 show the speedups and energy savings on the ARM mobile CPU for the English2, German, Russian and Greek acoustic models respectively. The results are similar to the benefits reported for English1 in Figure 3.13: our

techniques provide consistent performance improvements and energy savings for different acoustic models.

We have also evaluated our optimizations on an Atom-like mobile CPU by running simulations on Sniper. The performance and energy results are shown in Figure 3.12. Note that Atom does not support AVX. Therefore, we are constrained to SSE and vector size of 4 for the experiments on this CPU. On Atom CPU, our optimizations achieve similar results to the ones obtained in Haswell and Skylake, but some differences arise. First, *All32* configuration achieves a small speedup of 2.7%, but the energy increases by 12.5%. Second, the configuration that employs the improved memory layout (*SSE+FMA+T*) reduces energy by 6% with respect to straightforward SIMD implementation (*SSE+FMA*), but performance decreases by 4%. As in Skylake and Haswell, on Atom the multi-frame implementation with window size of 3 frames achieves the best results, providing 1.88x speedup and 55.3% energy savings. As Figure 3.12 shows, a slight increase can be seen in the energy consumption of *All32*. This outlier is indeed due to the McPAT power model. McPAT accurately accounts for the energy of the extra FP operations and cache accesses in this configuration. However, it does not properly model the cost of a recovery from a branch misprediction. Energy for flushing the pipeline or recovering the register map table is not considered in McPAT.

3.4.1 Comparison With Other GMM Implementations

GMM is a machine learning technique used in a wide range of applications in different areas including, for example, speech recognition or image recognition. A popular implementation of GMM evaluation consists on using matrix-matrix multiplication as described in [29]. In this implementation, the Gaussians and the input features are represented as 2D matrices and the acoustic scores are obtained by performing matrix multiplication, typically by using the BLAS specification for dense linear algebra. This is the approach employed in other speech recognition toolkits like Kaldi [79].

In this section, we compare the performance and energy consumption of our GMM implementation with the matrix multiplication approach. We use OpenBLAS library [1], a high-performance implementation of the BLAS specification, to implement Pocketsphinx's acoustic model by leveraging the high-performance implementation of the SGEMM operation (single-precision general matrix multiplication). In our experiments, we use single-threaded OpenBLAS implementation. Figure 3.22 shows the speedups of different GMM implementations of the last acoustic model for English language in Pocketsphinx. Baseline is the unmodified Pocketsphinx implementation. Our version of GMM outperforms the matrix multiplication approach in the Intel and ARM platforms. We achieve 10%, 22% and 3% speedup when running on ARM, Haswell and Skylake CPUs respectively, when using a SIMD width of 4 (same than OpenBLAS). On the other hand, Figure 3.23 shows the normalized energy for the same configurations and CPUs. Our GMM implementation provides higher energy savings than the version based on matrix multiplication.

Our optimized decoder achieves higher performance and energy-efficiency than the GMM implementation based on matrix multiplication due to several reasons. First, instruction mix analysis of the different implementations reveals that 45% of the instructions in our proposed methods are SIMD instructions, whereas 38% of the instructions in matrix multiplication are vector instructions.

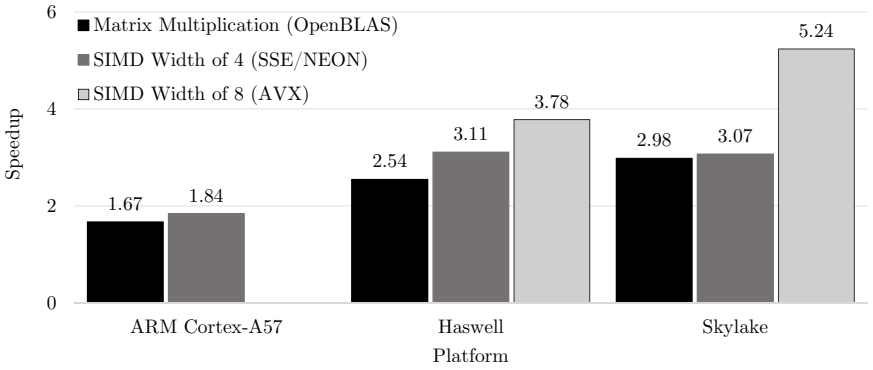


Figure 3.22: Speedup achieved by matrix multiplication technique using OpenBLAS library vs our proposed techniques running on different platforms. Baseline is unmodified Pocketsphinx. All the configurations implement the same acoustic model for English language.

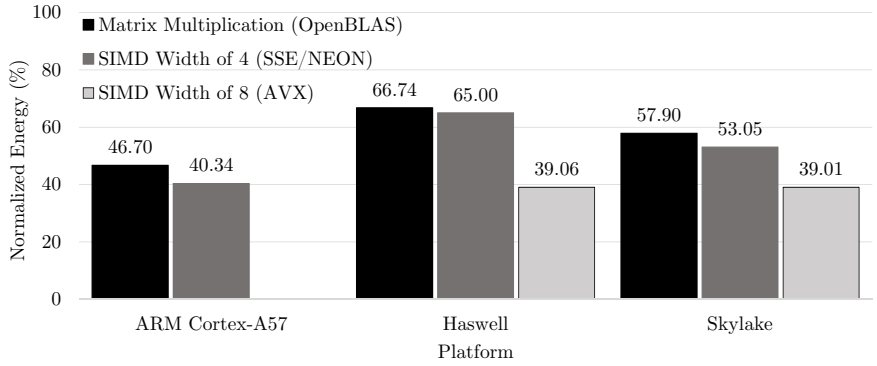


Figure 3.23: Normalized energy consumption achieved by matrix multiplication technique using OpenBLAS library vs our proposed techniques running on different platforms. Baseline is unmodified Pocketsphinx.

Second, matrix multiplication implementation requires a preprocessing stage to prepare the matrix of input features. Although the matrix with the Gaussians, i. e. means and variances, is static and can be initialized offline, the matrix with the input features has to be created on-the-fly for each frame of speech as described in [29]. This preprocessing represents a non-negligible overhead. In comparison with the matrix multiplication, our *SSE+FMA+T* implementation requires 13% less instructions.

3.4.2 Discussion

Our proposed methods are applicable for any acoustic model based on Gaussian Mixture Models, so it works for speech in any language. In this work, we use Pocketsphinx as our baseline ASR system since we target mobile platforms, but our proposed techniques can be used in the acoustic models available in Sphinx 4, Kaldi, Julius or HTK. We believe speech recognition will be a feature supported by the majority of computing devices in the near future, and acoustic models will evolve

towards more complex ones for the sake of better accuracy.

On the other hand, the proposed techniques can also be used for other applications that are relevant for mobile devices, especially in the area of computer vision. GMMs are employed for image segmentation [34, 38], image retrieval [86], tracking people in images [67] or detecting and tracking moving objects in video sequences [24].

3.5 Conclusions

In this chapter we present the result of energy and performance analysis of our baseline ASR system when running on modern CPUs. Gaussian evaluation of the acoustic model is the most computationally expensive component, representing more than 80% of the total execution time. We show that most of the CPU stalls are due to mispredicted branches and accesses to main memory. Furthermore, we show that DRAM is the main source of energy consumption.

To improve the performance and energy efficiency of Gaussian evaluation, we propose multiple optimizations to alleviate the bottlenecks identified in the analysis. First, we remove conditional branches from the innermost loop of the Gaussian evaluation code, achieving 12% speedup and 11% energy savings for a generic English acoustic model running on a Haswell processor. Second, we employ a multi-frame Gaussian evaluation scheme with prediction of active senones to reduce off-chip memory accesses by 57.1%.

Finally, we used SIMD instruction in the VPU and a new memory layout to further boost Gaussian evaluation and improve energy efficiency. Our implementation using SIMD instructions and multi-frame Gaussian evaluation achieves up to 2.92x speedup and 62.4% energy savings on an modern Intel Skylake desktop CPU. Furthermore, it obtains 1.88x and 1.85x speedup and reduces energy consumption by 55% and 59% on an Atom-like and a modern ARM mobile CPUs respectively. All the performance improvements and energy savings are achieved without any loss in the accuracy of the ASR system.

4

A Register Renaming Scheme for Out-of-Order Processors

In the previous chapter, we presented several optimizations at software level in order to improve the performance and energy efficiency of ASR systems. We observe that running the optimized software on modern processors puts significant pressure on the register file which results in considerable stalls at the renaming stage, limiting the performance of the processor. In this chapter, we focus on improving the performance of modern out-of-order processors at microarchitecture level by proposing a technique to reduce this pressure on the register file.

First, we present an analysis of several benchmark suites that shows a high opportunity to reuse physical registers, by exploiting the large percentage of single-use values. Then, we propose a novel renaming technique that reduces the pressure on the register file by enabling physical register sharing. We use a cost-effective register file design with check-pointed and conventional registers to recover the state of the processor in event of branch mispredictions, interrupts or exceptions. We show that not only for our optimized ASR application the proposed scheme provides significant improvements, but also for SPEC benchmarks the proposed register renaming scheme results in 6% speedup with no area overhead, or 10.5% reduction in register file size for the same performance.

In this chapter, section 4.1 reviews the traditional register renaming techniques and section 4.2 presents the analysis that motivates this work. In section 4.3, we describe our register renaming scheme in detail. Section 4.4 presents our evaluation methodology, and the experimental results are provided in section 4.5. Section 4.6 reviews some related works and finally, section 4.7 sums up the main conclusions of this work.

4.1 Register Renaming

Register renaming is key for the performance of out-of-order processors. Instructions, after being decoded, are kept in the reorder buffer until they commit. The size of the reorder buffer determines the maximum number of in-flight instructions. These instructions are usually called instruction window.

The goal of renaming is to remove register name dependences, write-after-read and write-after-write dependences for the instructions in the instruction window. This is achieved by allocating a free storage location for the destination register of every new decoded instruction. The most common solution to provide the storage locations is a merged register file [37]. In this case, there is a physical register file that contains more registers than those defined in the ISA, which are referred to as logical registers. A register map table is used to manage the translations from logical to physical identifiers. When an instruction commits, the physical register allocated by the previous instruction with the same logical destination register is freed. This has become the adopted approach of practically all current microprocessors, due to its energy efficiency, and is the baseline technique assumed in this work. Other schemes such as renaming through the reorder buffer [92] are much less commonly used nowadays, since they tend to be less energy efficient.

In the merged register file organization, a number of physical registers close to the number of logical registers plus the window size is required since the majority of the instructions have a destination register. A number of physical registers equal to the number of logical registers is needed to keep the committed state of the processor. In addition, for every instruction whose destination operand is a register, an additional register is allocated when it enters the window at rename stage and a physical register is released when it leaves the window at commit stage.

Renaming schemes are conservative to guarantee correct execution. In conventional schemes, a physical register is released when the instructions that redefines the corresponding logical register commits. In this manner, it is guaranteed that there is no other potential consumer of this value stored in the physical register being released, since the redefining instruction is no longer speculative. Note that many cycles may happen between the last read of the register and its release, which leads to underutilization of the register file. This results in an unnecessary increase of the register file pressure.

4.2 Motivation

As discussed in previous chapter, we applied several software optimizations to the baseline ASR code to improve its overall performance running on CPUs. Although our optimizations are very effective, our analysis show that the optimized code significantly increases the pressure on the register file and causes many stalls in the renaming stage due to lack of physical registers. Figure 4.1 shows the activity breakdown in the renaming stage of the pipeline for both baseline and our optimized code. For this experiment we model a processor with the configuration in Table 4.1. It should be mentioned that to clearly show the pressure increase on the register file, we consider a floating-point register file with only 48 registers. In the baseline code, main memory and

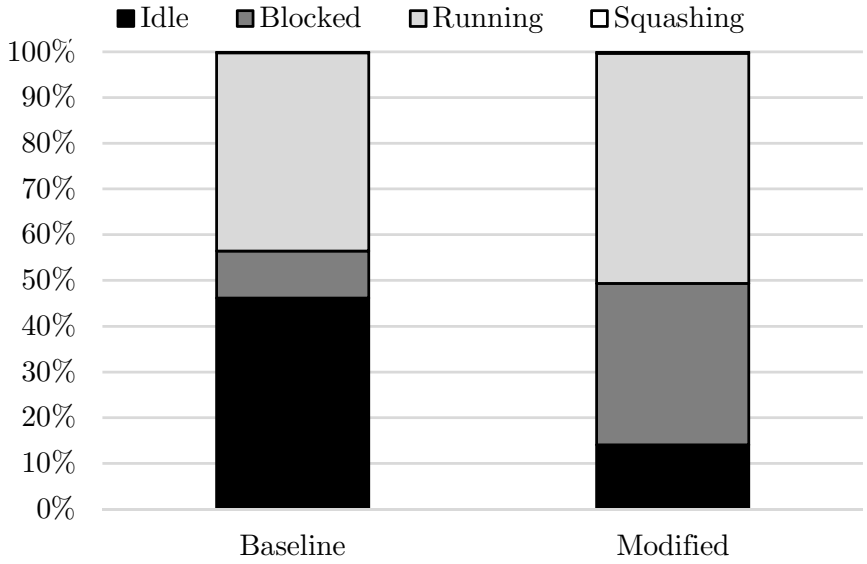


Figure 4.1: Renaming stage activity breakdown for the baseline GMM evaluation code and the optimized version.

Listing 4.1: ARM assembly code for the innermost loop of GMM evaluation.

```
1    I1:  add     x4, x7, x1
2    I2:  add     x3, x1, x6
3    I3:  add     x2, x1, x5
4    I4:  add     x1, x1, #0x4
5    I5:  cmp     x1, x8
6    I6:  ld1     {v3.4s}, [x4]
7    I7:  ld1     {v0.4s}, [x3]
8    I8:  ld1     {v2.4s}, [x2]
9    I9:  fsub    v0.4s, v3.4s, v0.4s
10  I10:  fneg    v2.4s, v2.4s
11  I11:  fmul    v0.4s, v0.4s, v0.4s
12  I12:  fmla    v1.4s, v2.4s, v0.4s
13  I13:  b.ne    I1
```

branches were the main performance limiting factors whereas in the modified version their effect dramatically reduced. However, as Figure 4.1 shows, running the modified GMM code, results in considerable stalls in the renaming units due to lack of physical registers. In order to find an efficient solution to alleviate this issue, we first analyze the assembly code for the innermost loop of the optimized GMM evaluation.

As Listing 4.1 shows, all the instructions *I6, I7, ..., I12* are vector instructions and require a new physical register. The register written by instruction *I7* is only used by instruction *I9* and similarly, the register written by instruction *I9* is only used by instruction *I11*. This is the

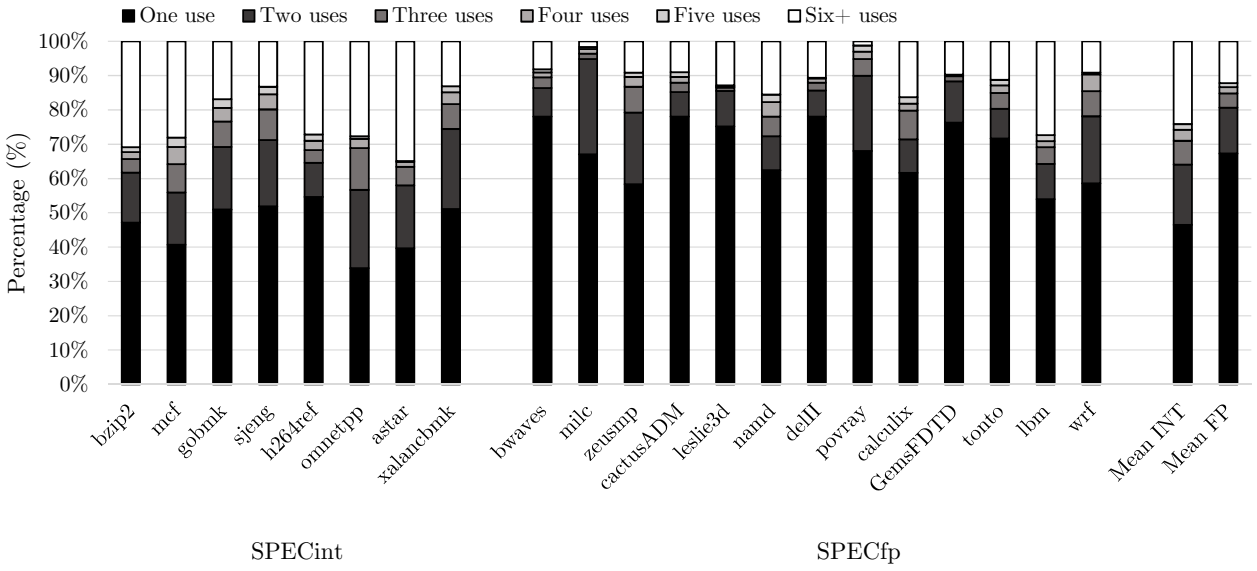


Figure 4.2: Percentage of registers consumed one, two, three, four, five and six or more times. Most of the values are consumed just once in SPEC.

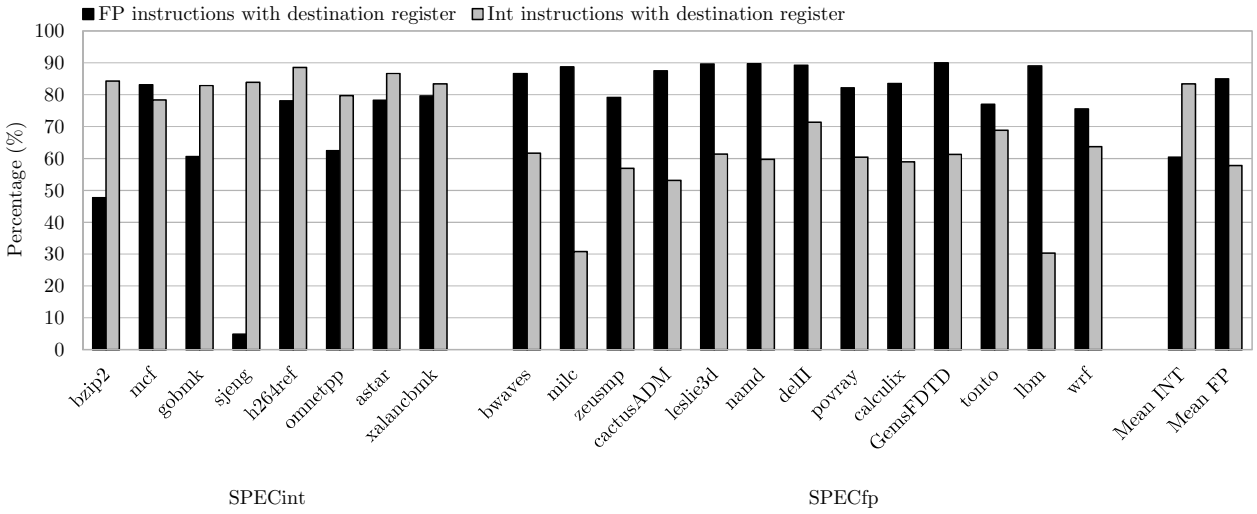


Figure 4.3: Percentage of Floating point and Integer instructions that require a new physical destination register.

case for instructions *I8* and *I10*. In other words, the values of these registers have only a single consumer and the consumer is the redefining instruction which guarantees that there will be no other consumers. Hence, as far as the true dependences between the aforementioned instructions are maintained, they can share a physical register. Therefore, only four vector registers are required instead of seven which can significantly reduce the pressure on the register file.

We have done an extensive analysis to study the register usage in various set of benchmarks. This work is primarily motivated by the observation that, in a significant percentage of instructions, the value stored in a register has only a single consumer (see Figure 4.2). In conventional register renaming schemes, the single-consumer instruction allocates a new physical register for the destination register. However, in this case, once the value of the source register is read by the consuming instruction, the same physical location can be used to write the result since no more consumers need the previous content of the register. In other words, the source register can be reused for the destination instead of allocating a new one. As we show in Figure 1.3, for GMM application, more than 57% of the instructions have this property. Similarly, for SPEC2006 benchmarks, this happens for more than 50% and 30% of SPECfp and SPECint instructions respectively. Note that, in average, more than 85% of the instructions require a physical register as a destination as we see in Figure 4.3.

We often find chains of instructions where a given logical register is both the destination operand and a single consumer operand of it. This is the case of instructions *I1, I4, I5* and *I6* in Figure 4.5. In this case, all the instructions in the chain can share the same physical register as destination, further reducing the pressure on the register file.

Following this idea of physical register sharing for the instructions in a chain, Figure 4.4 shows the percentage of instructions that can avoid allocating a new physical register if each register can be reused up to one, two, three or an unlimited number of times. Note that Figure 4.4 only considers instructions with a destination register, i.e. instructions that require an allocation of a physical register at renaming. For this reason, the percentage of *One Reuse* in Figure 4.4 is not equal to the category *One use* in Figure 4.2, as it does not include single-uses performed by stores, comparisons and other instructions without a destination register. However, since the majority of instructions include a destination register, still more than 50% of the instructions in SPECfp and more than 30% of the instructions in SPECint can reuse a physical register multiple times. More specifically, 32.3%, 12.3% and 5.9% of the instructions in SPECfp can reuse a physical register up to one, two and three times respectively. Two reuses means a chain of three instructions whose only consumer is the next instruction in the chain. Similarly, three reuses means a chain of four instructions sharing the same physical register as destination. On the other hand, only 4.1% of the instructions can reuse a physical register more than three times, i.e. chains of more than four instructions are unusual. Regarding SPECint benchmarks, 22%, 5.2% and 2.3% of the instructions can reuse a physical register up to one, two and three times respectively, while only 1.2% of the instructions can reuse a physical register more than three times.

4.3 Renaming with Physical Register Reuse

In this section, we present our novel register renaming scheme for out-of-order processors that exploits physical register sharing to reduce the pressure on the register file. First, we illustrate the technique through an example. Second, we provide the implementation details of the technique, describing the changes to the different hardware structures of the processor, such as the register map table or the issue queue. Finally, we extend our renaming scheme to support precise exceptions, describing the required support in the register file.

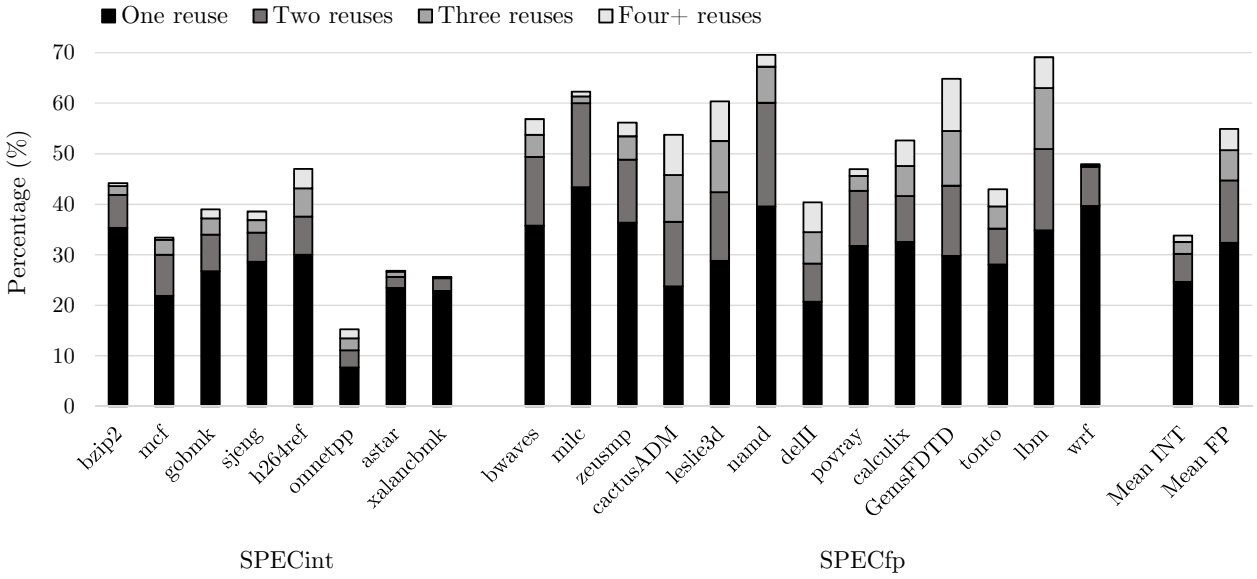


Figure 4.4: Percentage of instructions that can reuse a physical register, if a register can be reused up to 1, 2, 3 or an unlimited number of times. Note that we only consider instructions with a destination register.

4.3.1 Proposed Register Renaming Technique

For the sake of clarity, we first explain the proposed register renaming scheme through an example. Figure 4.5 presents the assembly code for several instructions of an application similar to the innermost loop of GMM evaluation. To execute the eight instructions in this example, conventional renaming schemes allocate eight different physical registers, one per instruction, as illustrated in Figure 4.5(a). The outcome of the register renaming and the step-by-step updates to the register map table are also shown in Figure 4.5(a). As we can see in this example, four different physical registers are employed for the same logical register, *r1*.

In this example, instructions *I1, I4, I5* and *I6* form a chain of read-after-write dependences that guarantees that they will be executed in program order and, therefore, they will write their results in order in the register file. In addition, each instruction is the only consumer of the previous one. In this case, the same physical register can be used as the destination for these four instructions, since the RAW dependence guarantees that an instruction produces its value before the next instruction in the chain is issued and, moreover, the single-use condition guarantees that the value is not used by any other instruction. In other words, there is no other instruction reading the value between the producer and the consumer in this chain of instructions that may introduce a WAR hazard. Hence, there is no requirement of keeping the four values alive in different physical registers for correct program execution.

In this work, we propose to reuse the same physical register in the aforementioned condition, i.e. when two instructions exhibit a RAW dependence and the second instruction is the only consumer of the value. To leverage physical register sharing, we introduce a new hardware structure: the

Result of Renaming

					dst	src1	src2
①	I1:	add	r1	← r2, r3	P1	P2	P3
②	I2:	ld	r3	← m(x1)	P5	-	-
③	I3:	mul	r2	← r3, r4	P6	P5	P4
④	I4:	add	r1	← r1, r4	P7	P1	P4
⑤	I5:	mul	r1	← r1 , r1	P8	P7	P7
⑥	I6:	mul	r1	← r1 , r3	P9	P8	P5
⑦	I7:	add	r5	← r1 , r2	P10	P9	P6
⑧	I8:	sub	r2	← r5 , r1	P11	P10	P9

Register Map Table

	PR
r0	-
r1	P1 P7 P8 P9 ① ④ ⑤ ⑥
r2	P2 P6 P11 0 ③ ⑧
r3	P3 P5 0 ②
r4	P4 0
r5	P10 ⑦

I1:	add	r1	← r2, r3	1 new register
I2:	ld	r3	← m(x1)	1 new register
I3:	mul	r2	← r3, r4	1 new register
I4:	add	r1	← r1, r4	1 new register
I5:	mul	r1	← r1 , r1	1 new register
I6:	mul	r1	← r1 , r3	1 new register
I7:	add	r5	← r1 , r2	1 new register
I8:	sub	r2	← r5 , r1	1 new register
				8 new registers

(a) Register renaming with traditional schemes.

Result of Renaming

					dst	src1	src2
①	I1:	add	r1	← r2, r3	P1.0	P2.0	P3.0
②	I2:	ld	r3	← m(x1)	P5.0	-	-
③	I3:	mul	r2	← r3, r4	P6.0	P5.0	P4.0
④	I4:	add	r1	← r1, r4	P1.1	P1.0	P4.0
⑤	I5:	mul	r1	← r1 , r1	P1.2	P1.1	P1.1
⑥	I6:	mul	r1	← r1 , r3	P1.3	P1.2	P5.0
⑦	I7:	add	r5	← r1 , r2	P6.1	P1.3	P6.0
⑧	I8:	sub	r2	← r5 , r1	P7.0	P6.1	P1.3

Register Map Table

	PR
r0	-
r1	P1 ①
r2	P2 P6 P7 0 ③ ⑧
r3	P3 P5 0 ②
r4	P4 0
r5	P6 ⑦

Physical Register Table

	Read bit	2-bit Counter
P1	0 1 ① ⑦	0 1 2 3 ① ④ ⑤ ⑥
P2	0 1 0 ①	0 0
P3	0 1 0 ①	0 0
P4	0 1 0 ③	0 0
P5	0 1 ② ③	0 ②
P6	0 0 1 ③ ⑦ ⑧	0 1 ③ ⑦
P7	0 ⑧	0 ⑧

I1:	add	r1	← r2, r3	1 new register
I2:	ld	r3	← m(x1)	1 new register
I3:	mul	r2	← r3, r4	1 new register
I4:	add	r1	← r1, r4	
I5:	mul	r1	← r1 , r1	
I6:	mul	r1	← r1 , r3	
I7:	add	r5	← r1 , r2	
I8:	sub	r2	← r5 , r1	1 new register
				4 new registers

(b) Register renaming with our proposed scheme.

Figure 4.5: Step-by-step renaming of several instructions, including the changes in the Rename Table and Register Map Table. In this example, we assume that physical registers *P2, P3* and *P4* have been previously assigned to *r2, r3* and *r4* respectively. (a) conventional renaming scheme, (b) the proposed renaming scheme.

Physical Register Table (PRT). The PRT contains one entry per physical register, as shown in Figure 4.5(b). Each PRT entry includes one *Read bit* and a *2-bit Counter*. The *Read bit* is used to identify the first consumer of a register. If the *Read bit* is set it indicates that the physical register is the source operand for an in-flight or a committed instruction in the pipeline. On the contrary, if the *Read bit* is clear, it indicates that no consumer of the value has been found (i.e. fetched and renamed) yet.

When the first consumer of a register is being renamed, in order to reuse the source register for the destination register, we also have to verify that there will be no future consumers. In case the consumer is also redefining the first-use register, it is guaranteed that there will be no younger consumer of the value. For example, instruction *I5* in Figure 4.5 satisfies this property, as it is the only consumer of *r1* and it also redefines *r1*. In case the instruction being renamed is not the redefining instruction (see instruction *I8* in Figure 4.5), a simple single-use predictor is employed to decide whether the same register is reused or a new physical register is allocated. Section 4.3.4 provides more details about the single-use predictor and the actions taken in case of misprediction.

On the other hand, the *2-bit Counter* keeps track of the number of instructions sharing the same physical register, and it is used to maintain the true dependences. Due to the sharing of registers, the same name, i.e. the same physical register ID, is used to identify different values produced by different instructions. Therefore, it is not possible to correctly identify which instructions have to be woken up in the issue stage when a value is produced using only this ID. To avoid this ambiguity, we append the *2-bit Counter* to the register ID, so the source or destination of an instruction is specified as the N-bits of the physical register ID plus the two bits of the counter. The *2-bit Counter* is increased each time the same physical register is reused and it identifies the different versions of this register. In this manner, up to four instructions can share the same physical register but yet RAW dependences can be identified, as different instructions produce or wait for different versions of the register. As shown in Figure 4.5(b), both instructions *I5* and *I6* take P1 as source operand, but they wait for version one *(P1.1)* and version two *(P1.2)* respectively. When instruction *I4* produces *P1.1* the issue logic only wakes up *I5*, that is the instruction waiting for version one.

This scheme can be generalized to employ an *N-bit counter* to allow up to 2^N instructions to share the same physical register as destination. Note that when the counter is saturated we cannot longer reuse the register, as it would not be possible to differentiate its multiple versions to keep track of the RAW dependences. We found that in SPEC benchmarks it is uncommon to have more than three reuses (see Figure 4.4). Furthermore, additional bits represent larger overheads in the PRT and the issue queue. We found that a 2-bit counter provides a good trade-off between the degree of physical register sharing and cost.

Figure 4.5(b) shows the proposed renaming technique step-by-step. In this case, instructions *I1, I4, I5* and *I6* share the same physical register *P1*. Our renaming scheme only requires four physical registers, instead of the eight employed by the conventional approach. Next sections provide further details on the renaming technique, describing how the processor operates and the changes required to the different hardware structures. In the following we will explain in more detail the actions and changes applies to each stage of the pipeline.

Renaming Source Registers

Every time a source operand of an instruction is renamed, the Register Map Table is accessed as in the conventional approach to get the physical register ID that is assigned to the logical register. Next, the physical register ID is used to index the PRT. The *Read bit* of the corresponding entry is set to indicate that an in-flight instruction will read the value stored in the register. In addition, the *2-bit counter*, that indicates the most recent version of the register, is read from the PRT, so the renaming logic provides the physical register ID plus the *2-bit counter*.

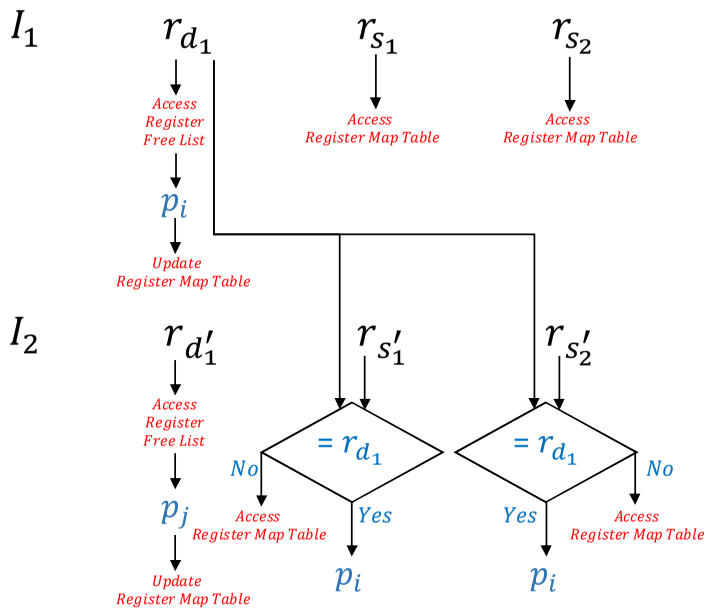
Renaming Destination Registers

Our renaming scheme tries to reuse some of the source registers as the destination for the instruction being renamed, in order to avoid an allocation of a new physical register. For this purpose, the renaming logic checks first the *Read bit* of the source registers in the PRT. If this bit is zero for some of the sources, it means the instruction being renamed is the first consumer of the value stored in that register. To identify single-use condition, the renaming logic also checks whether the instruction is the last consumer of the value. To this end, the source register ID is compared with the destination register ID of the instruction. If the source register matches the destination, it is guaranteed that the instruction is the last consumer of the value. On the contrary, we check whether or not the register was predicted as single-use by the predictor described in section 4.3.4.

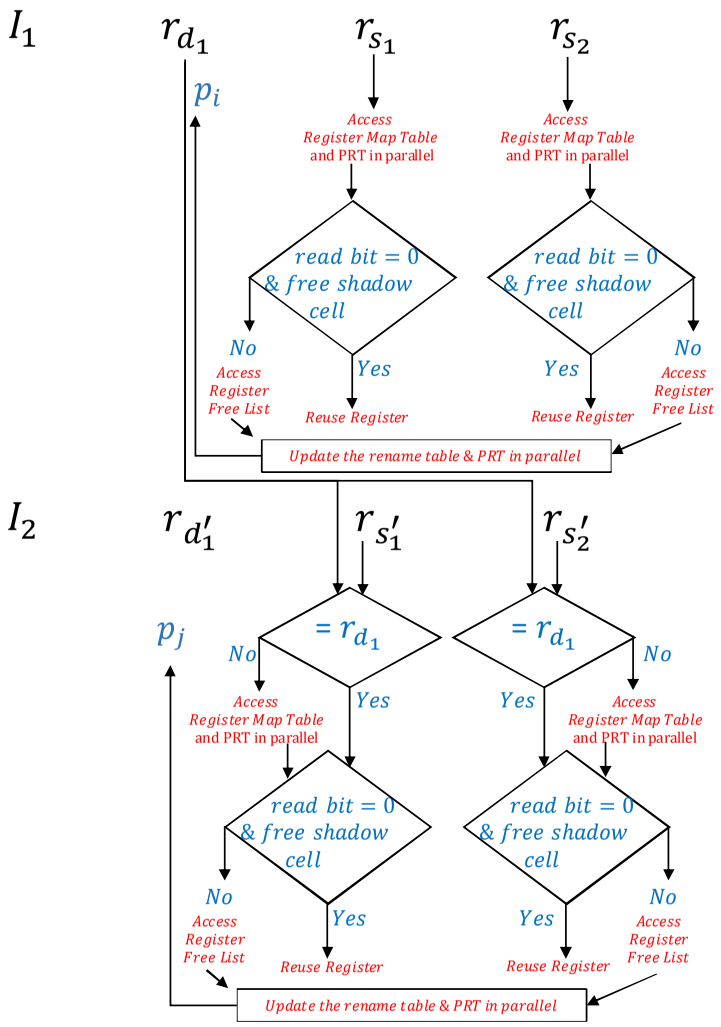
On the other hand, the *2-bit counter* of the source register is also accessed to verify that it is not saturated, i.e. that there are versions of the register available and, hence, the processor will be able to maintain the RAW dependences for another reuse. If the instruction is identified as the single consumer of the source register and the *2-bit counter* is not saturated, the source physical register is reused as the destination of the current instruction being renamed, and no allocation of a physical register is performed. The corresponding entry in the Register Map Table is updated to map the logical register to the physical register being reused. In addition, the *Read bit* is set to zero and the *2-bit counter* is increased in the corresponding PRT entry.

In case the instruction cannot be identified as the single consumer of a source register or the *2-bit counter* is saturated, a new physical register is allocated. The Register Map Table entry is updated with the ID of the allocated register, whereas the *Read bit* and the *2-bit counter* are set to zero in the PRT.

Modern processors are able to rename multiple instruction per cycle. Therefore, the logic must check for RAW dependences between instructions renamed in the same cycle. In our renaming scheme if a register is being reused, the PRT and the Register Map Table have to be updated accordingly in order to maintain the true dependences. Figure 4.6 shows the operations for renaming two instruction in one cycle in our scheme and in the baseline renaming scheme. Figure 4.7 shows a simplified logic for renaming these two instructions which demonstrates the feasibility of implementing our renaming technique. Note that modern processors are able to rename multiple serially-RAW-dependent instruction per cycle.

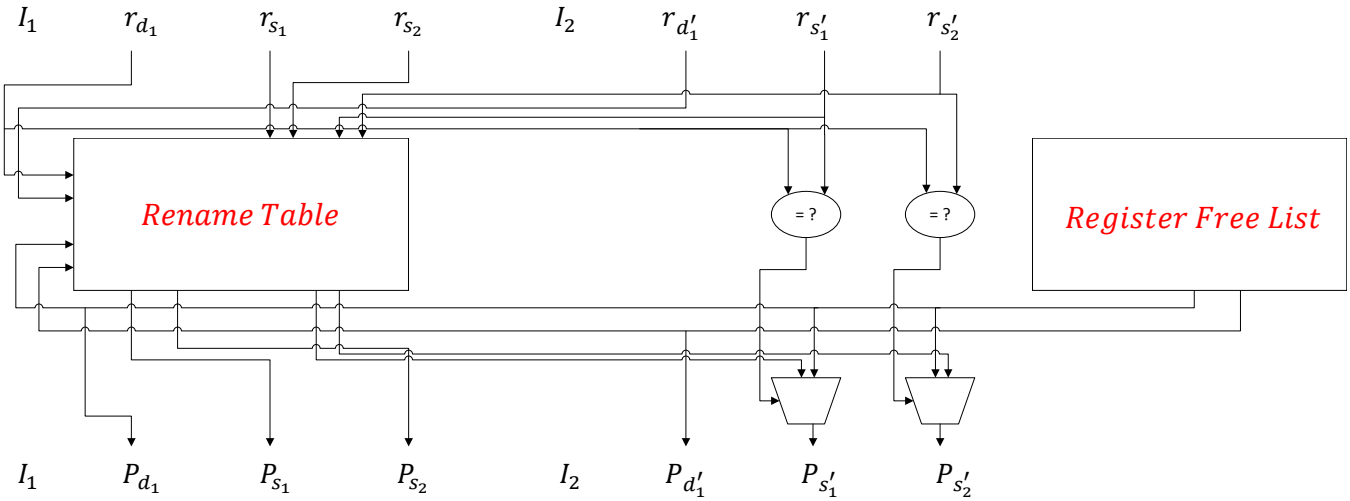


(a) *Baseline*

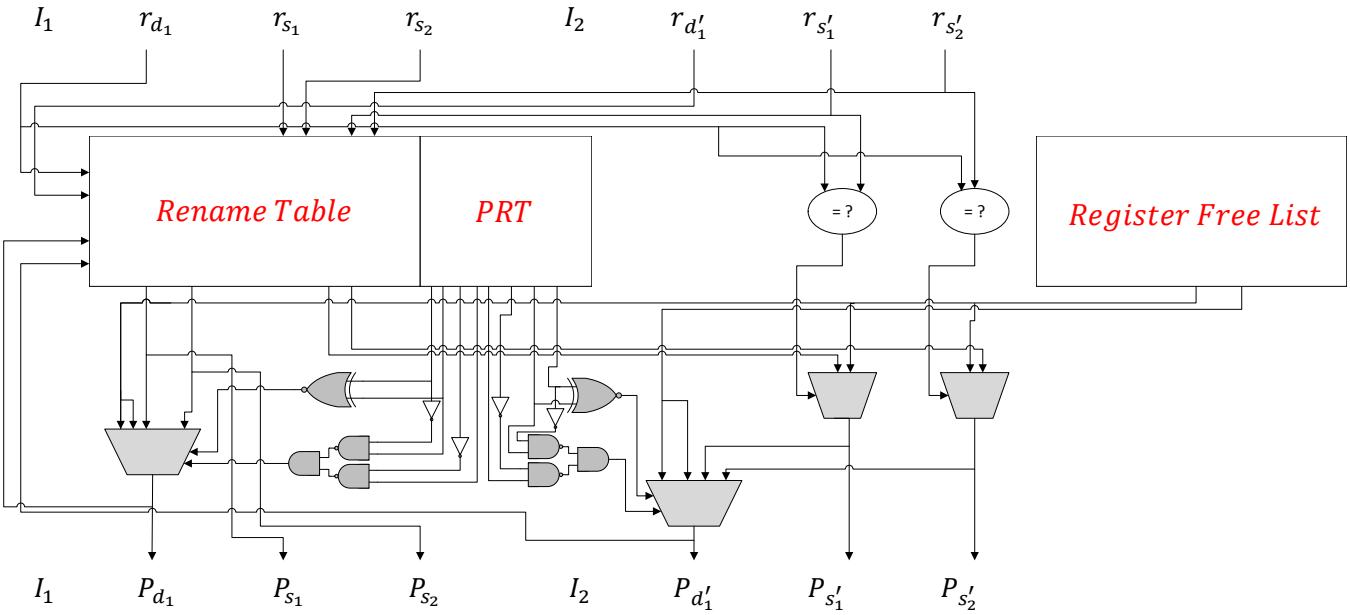


(b) *Our Technique*

Figure 4.6: Functional diagram for renaming two instructions in one cycle in the baseline register renaming scheme and in our scheme.



(a) Renaming two instructions in the baseline schemes



(b) Renaming two instructions with our proposed scheme

Figure 4.7: A logic implementation for renaming two instructions in one cycle in the baseline register renaming scheme and in our scheme.

Releasing a Physical Register

When a physical register is being reused, no register allocation is required for the new instruction. This technique is equivalent to a release-on-rename scheme. Although no modification is done to the list of free registers, the result of the technique is identical to releasing the physical register and immediately allocating it to the new instruction.

In case the physical register cannot be reused, a new one is allocated. The old register is released when the redefining instruction commits. Therefore, if a physical register can be reused the technique mimics the behavior of a release-on-rename scheme, otherwise it works as the conventional release-on-commit approach.

Lack of Physical Registers

Conventional renaming schemes stall when the list of free registers is empty. In our approach, the renaming will be blocked only when there is no available physical register and there is no possibility of reusing a register as described in Section 4.3.1. Our experimental results presented in Section 5.4 show that our scheme is very effective avoiding stalls in the renaming stage due to lack of registers.

4.3.2 Mispredictions, Interrupts and Exceptions

Modern out-of-order processors use dynamic speculation to issue an instruction before it is known whether or not the instruction should be executed. In our renaming scheme, multiple instructions may reuse the same physical register, and each instruction in a chain of reuses overwrites the value produced by the previous instruction. Since values in a shared physical register are speculatively overwritten, it is necessary to recover the previous value of a register in case of a branch misprediction or an exception between two instructions in the chain of reuses.

In the example of Figure 4.5, assume that instruction *I2* causes a TLB miss or raises a page fault exception and, when the exception is triggered, instruction *I4* has already written its result in P1. In this case, the previous value of P1, i.e. the value produced by instruction *I1*, must be recovered before invoking the exception handler to maintain precise exceptions.

In order to deal with branch mispredictions and support precise exceptions, the different versions of a shared register must be kept. However, this requires extra storage and increases the pressure on the register file, which is exactly what our renaming technique is trying to avoid by using physical register sharing. Shadow bit cells are a cost-effective solution to keep previous values of a register in a check-pointed register file [51]. Shadow copies of a register introduce a small overhead since they are independent of the number of ports, i.e. they are not directly accessible. Previous values of a shared register are only required in the infrequent case of a branch misprediction or exception, whereas only the last version is required for normal execution. Therefore, the most recent version is stored in the normal bits, that are directly accessible, whereas older versions of a shared register are stored in the shadow bits with a small area overhead, and recovered when

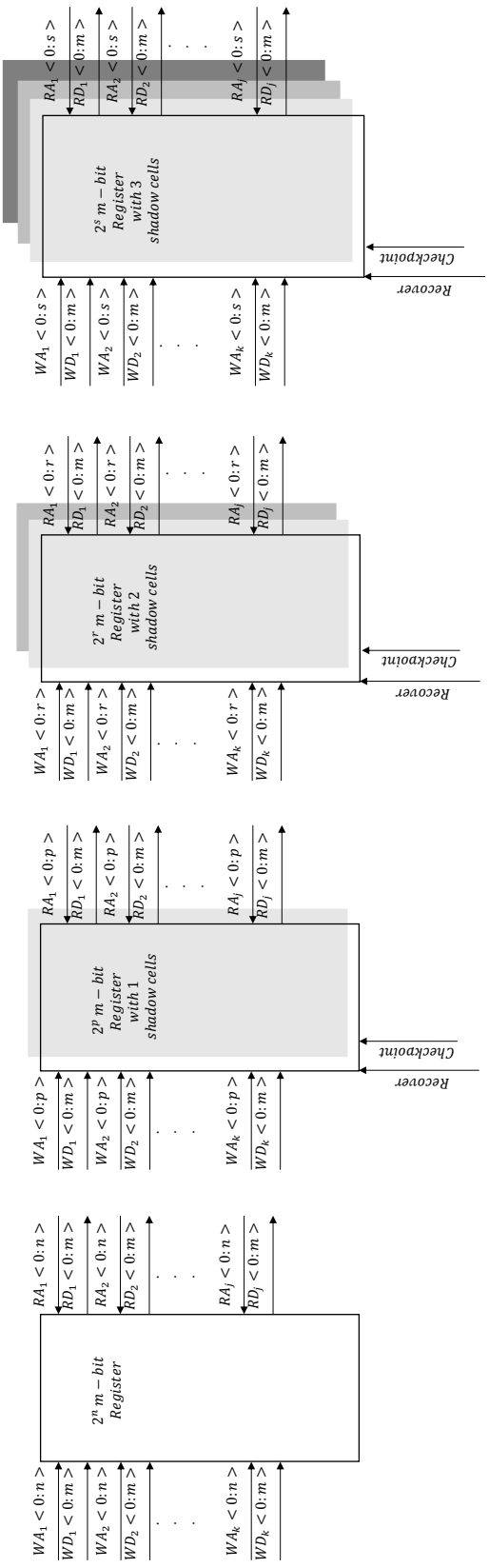


Figure 4.8: The structure of a four-bank *m-bit* register file with j read ports and k write ports. (In order from left to right) A bank with single-bit-cell, one, two and three shadow cells.

necessary.

In event of an interrupt or an exception the entire pipeline is flushed. Before the interrupt or the exception handler can be invoked, all logical registers must reflect their state before the interrupt or the exception. To this end, the processor consults the rename and retirement map tables and any entry that differs indicates a logical register whose correct state needs to be recovered from the shadow cells. Although this recovery process may take a few cycles more than in the baseline, the infrequent nature of interrupts and exceptions make this cost negligible.

4.3.3 The Register File

As described in the previous section, our renaming scheme employs a check-pointed register file with shadow bit cells to store the different versions of a shared physical register. With a *2-bit counter* in the PRT, our scheme allows up to three reuses of the same physical register, which means that up to three shadow copies must be kept in the register file. Hence, a straightforward implementation would include three shadow cells for each physical register.

Although each shadow copy represents a minor overhead [51], including three shadow copies for each physical register is not cost-effective, since all the copies are not required most of the time. As shown in Figure 4.4, most of the registers require zero or just one shadow copy, whereas chains of two or three reuses are much less common. Therefore, we propose to split the register file in four banks, where each bank includes registers with zero, one, two or three shadow copies respectively. By using this organization, our scheme is able to cover most of the cases while it avoids the extra cost of including three shadow copies in all the registers. Next sections provide further details on the implementation of the register file.

The Register File Design And Its Mechanism

To reduce the latency and increase the efficiency in area, register files are implemented in multiple banks [26]. In this work, we propose to have a multi-bank register file in which some of these banks have registers with one, two or three shadow cells embedded.

In such registers, each traditional register bit-cell is backed-up by pairs of cross-coupled inverters which are connected to the main bit-cell using a pass transistor. Shadow cells are accessed only through the main SRAM-cell of the register. Therefore, they do not require any additional read or write ports. A single-bit cell with n embedded shadow cells can hold up to $n+1$ different contents. The processor can simply manage these contents. At the *write* stage, the value of a register is stored in a shadow cell and a *recover* command copies back the content of a particular shadow cell to the main storage of the register. A physical register can be reused only if it has free shadow cells to store the previous content of the register.

Figure 4.8 shows the structure of four different banks of a register file with j read ports and k write ports. As it shows, the first bank in the left has single-bit-cell registers while the other banks have register cells with one, two and three embedded shadow cells. The design of a register bit cell with one shadow cell is shown in Figure 4.9. As this figure shows the shadow cell is only accessible

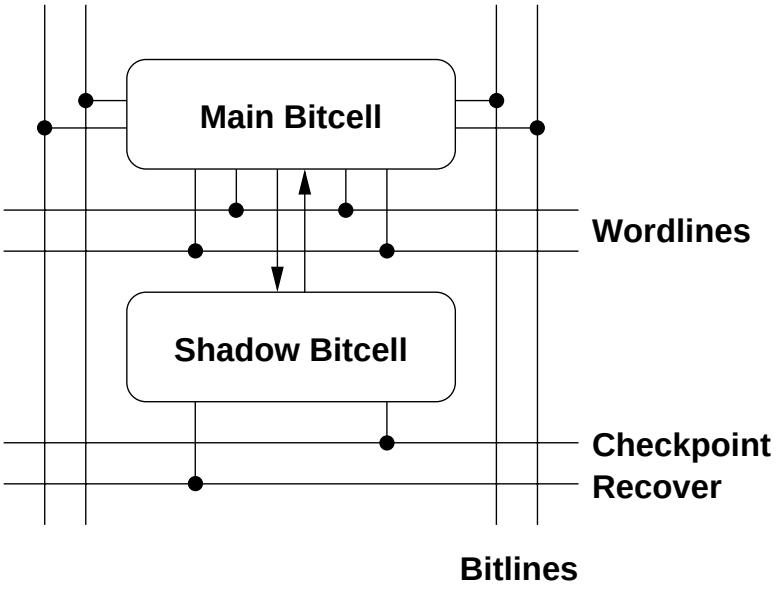


Figure 4.9: The design of a register bit cell with one shadow cell [51].

through the main cell.

Since the shadow cells are accessed only through the main bit cell, the additional area required by the shadow cells is independent of the number of register ports (see Figure 4.9). Therefore, the area overhead of the shadow bits becomes relatively smaller as the number of ports increases. Furthermore, we have observed that most of the registers do not need to have shadow cells. For this reason, the majority of the registers do not have shadow cells which helps to reduce the overhead significantly. We employ a set of banks with conventional single-bit-cell registers which includes the majority of the registers and few banks with registers that have embedded shadow cells.

When a new physical register is allocated, the register type predictor (described in section 4.3.4) predicts the expected use of it (single-use/multiple use; number of reuses), and allocates it in the corresponding bank according to this prediction. When a single-use register is written, the previous content of the register is stored in the shadow cell according to the *2-bit* counter of the register ID. In event of branch mispredictions, exceptions or interrupts, a recover instruction is issued for each register that needs its previous value to place them back to the main storage of the registers. In order to recover the value from the registers with more than one shadow cells, the added 2-bit to the register ID determines the correct shadow cell to recover the previous content of the register. If the physical register does not have enough shadow cells to store the previous content of the register, the register cannot be reused and a new register is allocated at the renaming stage.

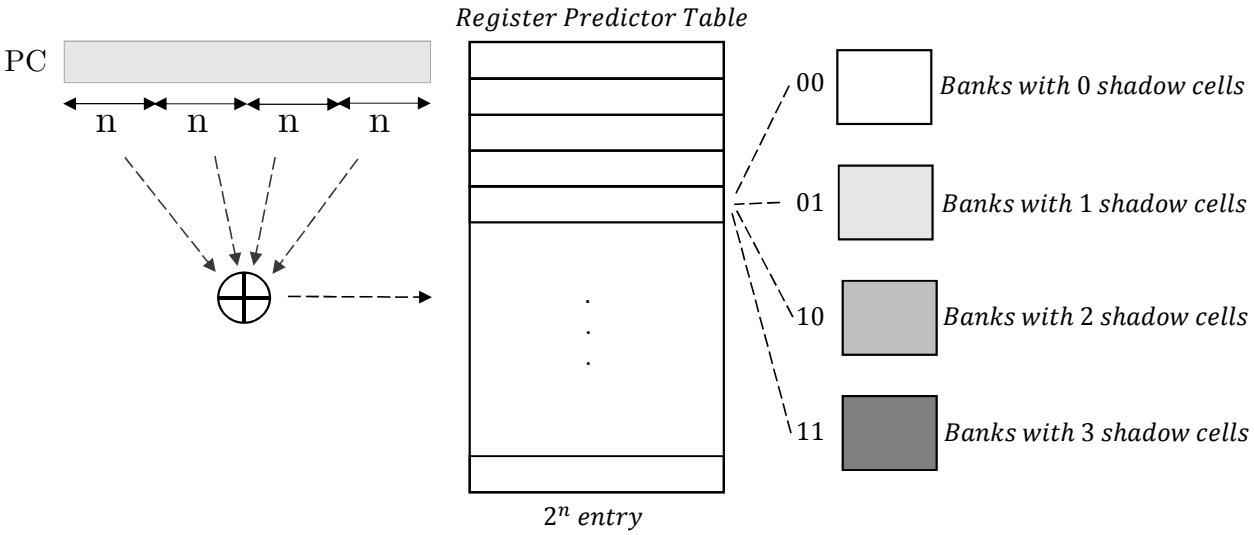


Figure 4.10: The design of the proposed register type predictor.

Impact on the Performance

According to our analysis, there is a very small increase, less than 1%, in the access time of the register file with the shadow cells due to longer word select and bit lines, as reported elsewhere (for instance, see [32]). The reason for the slight delay is that in the design of the registers with shadow cells, no gate capacitance is added.

Whenever a register is to be written in the *write* stage, according to its register ID the value of the register is stored in parallel to the appropriate shadow cell, so no extra latency is added to the write operation. In event of branch mispredictions, exceptions and interrupts there may be some registers whose previous values need to be recovered from the shadow cells. Therefore, recovering the state of the processor in such events may take few cycles more with respect to the baseline. In our experiments, we have taken this into account.

4.3.4 Register Type Predictor

When an instruction is being renamed, if it needs a new physical register (i.e., the source registers cannot be reused because they do not have shadow cells available or they are not the first use), a hardware predictor determines the type of the register which should be assigned to it. We design a simple 2-bit entry predictor which predicts the type of the register that should be allocated. Using the PC of the instruction, a simple hashing function determines an entry in the register predictor table, as Figure 4.10 shows. Then, according to the value of the entry, if there is a free register of that type, a new physical register is allocated. In the register predictor table, *00* indicates a normal register (i.e., it implicitly predicts that the register is not single-use) whereas

01, 10 and *11* indicate registers with 1, 2 and 3 shadow cells respectively (that is the register is predicted to be reused).

If there are no free registers of the predicted type, a register with the closest number of shadow cells will be allocated. In case there is no free register of any type, renaming will stall as in the conventional scheme.

Renaming Registers Using the Predictor. Whenever a source register is renamed, if the *read bit* is clear (which shows the register does not have previous consumers) and the register has free shadow cells, the physical register allocated to the source operand will be reused as a new physical register for the destination. Physical register ID can indicate the bank and accordingly type of the register. Thus, by knowing type of the register and the *2-bit Counter* we can determine whether or not the register has free shadow cell. Note that a register with shadow cells implicitly indicates that the register has been predicted to be single-use. Accordingly, the register map table and the PRT will be updated and the *2-bit Counter* will be incremented.

Releasing a Register Using the Predictor. Whenever a physical register is released, the entry in the register predictor table that has been used to allocate this register is updated to reflect the actual number of reuses. If not all the allocated shadow copies have been used, the value of the corresponding entry in the register predictor table is decremented. A register may be released in the commit stage or during instruction squashes. Whenever we allocated a physical register, we keep the entry of the register predictor table in the PRT so that when the register is being released, we know which entry of the predictor table should be updated.

On the other hand, if a register that is predicted to be single-use (i.e. has been allocated in a bank with 1,2 or 3 shadow copies) is detected to be used more than once, the corresponding entry in the predictor is reset to zero.

Finally, if a register predicted to be single-use is tried to be reused (i.e. it is the source operand of an instruction and it is the first use) but there are no shadow cells available, the register is not reused and the corresponding entry in the predictor is increased, so that next time it allocates a register with a greater number of shadow copies.

Handling Single-Use Mispredictions

A register predicted to be single-use may be reused and later encounter that the single-use prediction was wrong because there is an additional use. This is illustrated in Figure 4.11. Register r_1 is predicted to be a single-use register. Therefore, its physical register (P_1) is assigned to register r_x. Later, while renaming instruction *3*, it is discovered that r_1 is mispredicted to be a single-use register. Now, since the physical register P_1 has been assigned to another register, the register r_1 in instruction *3* cannot be renamed to P_1 as it holds the value of r_x. Since register r_x is renamed to P_1 and there might be instructions before instruction *3* that uses r_x, it is more cost effective to rename further consumers of r_1 to a new physical register. Therefore, the value of the r_x which was in physical register P_1 needs to be moved to this new physical register.

For this purpose, we propose to use two different micro-operations, depending on whether

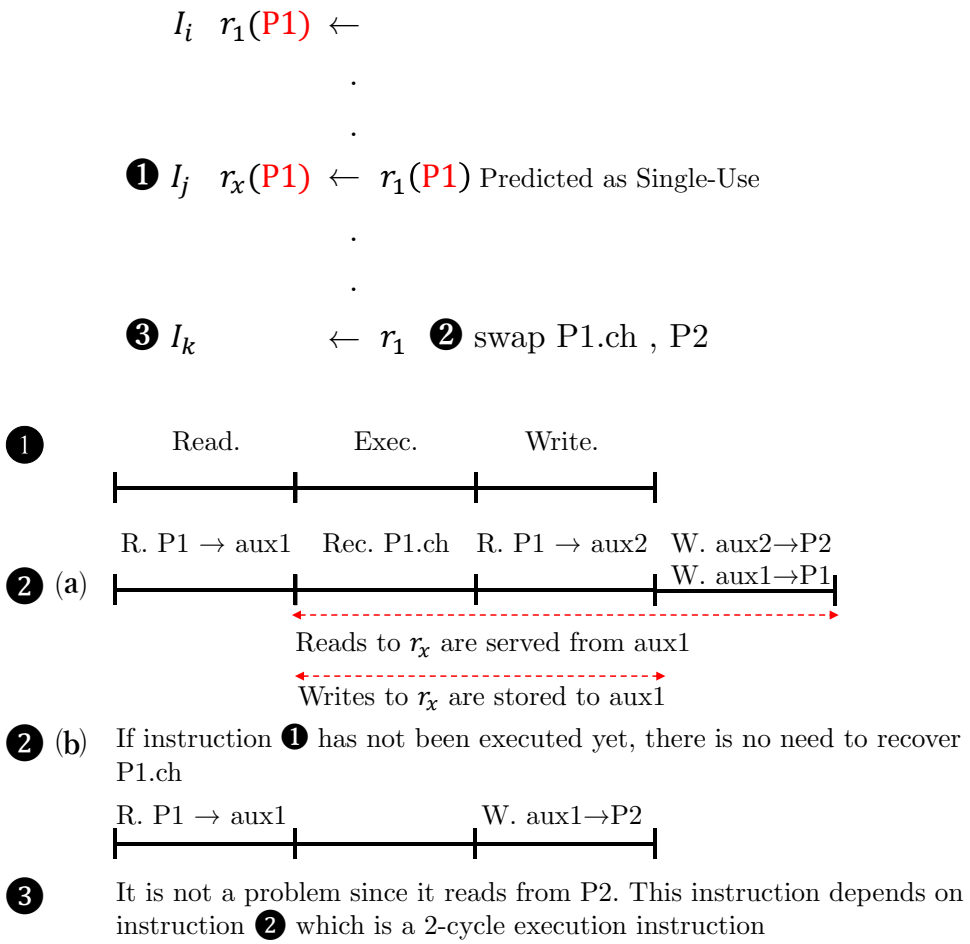


Figure 4.11: Timing for the proposed micro-operations to move the value of the check-pointed register.

instruction *1* has been executed when instruction *3* is being renamed or not, which move the value of r_1 to a new physical register (instructions *2(a)* and *2(b)* in Figure 4.11). If instruction *1* has been executed before the instruction *3* is being renamed, the value of r_1 is check-pointed in a shadow cell of P_1 ($P_{1.ch}$). Hence, as instruction *2(a)* in Figure 4.11 shows, first, the current value of P_1 is stored in an auxiliary register. Then, the check-pointed value in P_1 ($P_{1.ch}$) is being recovered and moved to another auxiliary register. Finally, last step is to move back the correct value of P_1 and move the recovered value of r_1 to the new physical location. If instruction *1* has not been executed before the instruction *3* is being renamed, the value of r_1 is not check-pointed and it only needs to be moved to the new physical register (see the timing for instructions *2(b)* in Figure 4.11).

	Configuration	
Core		Fully Out-of-Order
	ISA	ARMv8
	Core Frequency	2.0 GHz
	ROB size	128-entry
	Issue Queue	40-entry Fully Out-of-Order
	Width	3-Width Decoder, 3-Width Instruction Dispatch
	TLB	48 KB, 3-Way TLB
		48-Enrty Fully-Associative L1 TLB
Caches	**L1 Data**	32 KB, 2-Way, 1 Cycle
	L1 Instruction	48 KB, 3-Way, 1 Cycle
	L2	1 MB, 16-Way, 12 Cycles
		64 Bytes Cache Line Size
Prefetcher	**Type**	Stride Prefetcher (Degree 1)
	Branch Target Buffer (BTB)	2K
	Fetch Queue	32-Instructions
	Misprediction Penalty	15 Cycles
DRAM	**Frequency**	DDR3 1600 MHz
	Number of Ranks	2 Ranks/Channel
	Number of Banks	8 Banks/Rank, 8 KB Row Size.
	Other parameters	$t_{CAS} = t_{RCD} = t_{RP} = \text{CL} = 13.75$ ns
		$t_{REFI} = 7.8$ μs

Table 4.1: System Configuration.

4.4 Methodology

4.4.1 Simulation Environment

In this work, we model an out-of-order ARM processor using the GEM5 [17] cycle-accurate simulator with the parameters presented in Table 4.1. This processor uses a merged register file and releases a physical register when the redefining instruction commits. The simulator models in detail both the baseline and our proposed technique. We use CACTI 6.5 [70] to estimate area of the register files, PRT, issue queue and the predictor. CACTI 6.5 supports different models for SRAMs, DRAMs and register files. We specify different parameters including the number of read/write ports, number of banks, technology, bus width etc. to estimate the area, power and latency of different configurations.

4.4.2 Benchmarks

We use SPECfp and SPECint from SPEC CPU2006 [41] benchmarks for our experiments. We run the benchmarks using the *ref* inputs provided in the SPEC software package. All the SPEC benchmarks have been compiled using GCC 4.8.4 with *-O3 -mtune=armv8* optimization flags. For each benchmark 5 billions committed instructions has been simulated. In addition to SPEC2006 CPU benchmarks, we use Mediabench [56] benchmark suite.

Units	Configuration	Area (mm^2)
Integer Register File (64-bit registers)	128 Registers	0.2834
Floating-point Register File (128-bit registers)	128 Registers	0.4988
PRT	**Overhead**	5.08 E-04
Issue Queue	**Overhead**	1.48 E-03
Register Predictor	**Overhead**	3.1 E-03
Total Overheads		5.085 E-03

Table 4.2: Area for the register file, register map table, issue queue and the register predictor.

In addition to the SPEC benchmarks which are widely used, we use our optimized GMM application, described in the previous chapter, and a Deep Neural Network (DNN) application which are among the main kernels commonly used in many of machine learning applications. We add these two benchmarks in addition to the SPEC2006 and Mediabench.

4.5 Experimental Results

In this section, we evaluate the proposed register renaming technique explained in Section 4.3. In our experiments, we consider the overheads of the proposed renaming technique in order to make a fair comparison with respect to the baseline system, including the area of the PRT, register files, issue queue and the predictor. First, we evaluated the area for these units in the baseline and later we reevaluated the area considering the applied changes in these units. We want to make comparison with the same area. To this end, considering the area of overheads, we adjust the number of registers in the register file for our renaming scheme in such a way that the total area becomes the same as the baseline register file.

The additional area required by the shadow cells is independent of the number of register file ports and it becomes relatively smaller as the number of register file ports increases. Besides, since we use registers with shadow cells only for a small percentage of the registers, their area overhead becomes very small (normally in the order of a few physical registers). Unless stated otherwise, comparisons between our technique and the baseline are performed assuming the same total area for both, including the overheads.

4.5.1 Size of Different Banks in the Register File

Overheads of the proposed scheme

In the first place, we calculate the overheads that our scheme adds to the baseline system. Regarding the issue queue and PRT table, we consider the changes in the sizes of these units with respect to the baseline. Furthermore, the register type predictor has a table with the size of 1K bits. Table 4.2 shows the summary of area overheads in different modified units for our scheme. As it is shown, the overheads are small in comparison with the size of the register file.

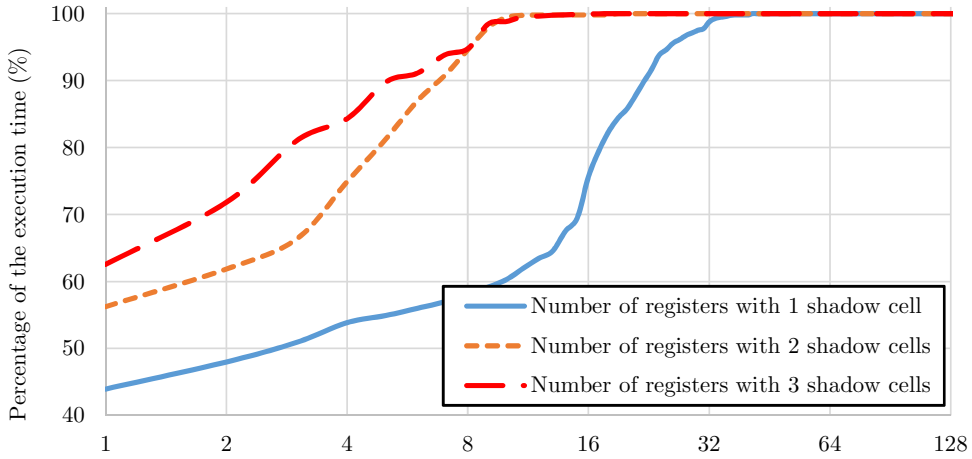


Figure 4.12: Number of physical registers with 1, 2 and 3 shadow cells needed to cover different percentages of the SPECfp execution time.

Register file configuration for the baseline	**Register file configuration for our scheme** 0-sh, 1-sh, 2-sh, 3-sh, Total				
48	28	4	4	4	64
56	28	6	6	6	82
64	36	6	6	6	90
72	36	8	8	8	108
80	42	8	8	8	114
96	58	8	8	8	130
112	75	8	8	8	147

Table 4.3: Register File Configuration.

Adjusting the sizes of each bank in the register file

Considering the aforementioned overheads, we performed a sensitivity analysis to identify the most convenient size of the different banks in the register file. For this study, we assumed an unbounded number of registers with up to three shadow cells. Figure 4.12 shows different number of physical registers with different number of shadow cells to cover different percentages of the SPECfp execution time. Hence, based on this study we tune the number of the registers of each bank in the floating-point register file. Similarly, we tune the number of registers in each bank for the integer register file. Table 4.3 shows the equivalent sizes of the register file that we consider. For each particular size of the register file in the baseline system, a hybrid register file configuration of the same area has been considered to evaluate the proposed scheme.

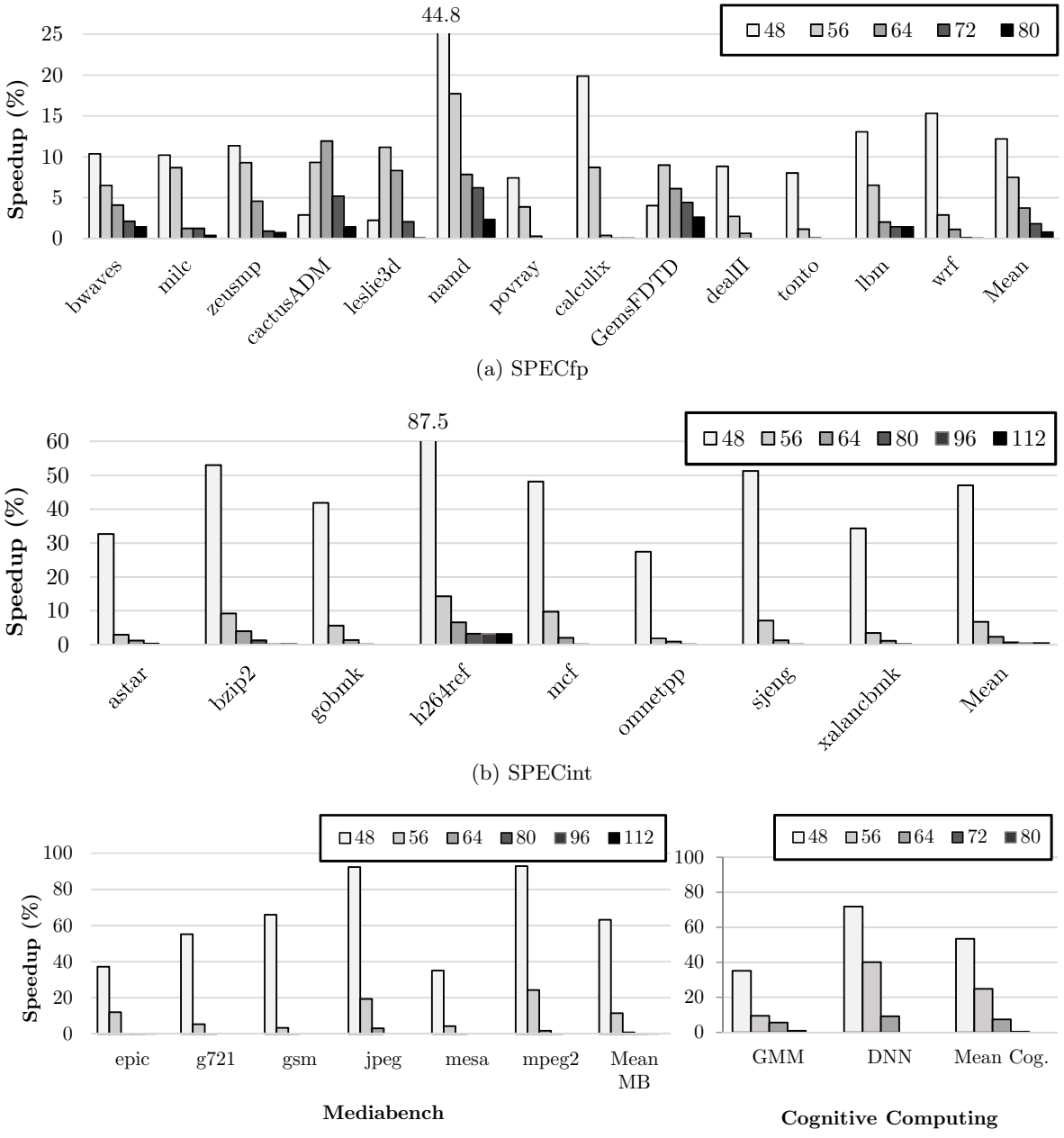


Figure 4.13: Speedups achieved for each benchmark with respect to the baseline system for different sizes of the register file. The proposed system has a multi-bank register file with 4 banks: a conventional one and banks with 1, 2 and 3 shadow cells. A register can be reused up to 3 times.

4.5.2 Performance Improvement

We present the performance results for the proposed register renaming technique in comparison with the baseline system. We assume a register file with four types of banks as shown in Figure 4.8. Figure 4.13 presents the performance improvements with respect to the baseline for different sizes of the register file. Note that the integer and floating-point register files are decoupled. Hence, for integer benchmarks we consider different sizes of the integer register file whereas for floating-point benchmarks we measure performance for different sizes of the floating-point register file.

As Figure 4.13(a) shows, for SPECfp benchmarks, the proposed technique provides 12.2%, 7.5%, 3.75%, 1.83% and 0.82% performance improvements on average using a register file of equivalent size for our proposed technique. Similarly, for SPECint benchmarks, the proposed technique provides 47%, 6.76%, 2.29%, 0.67% and 0.41% performance improvements on average with respect to the baseline for different number of physical registers in the register file (see Figure 4.13(b)). As the results show, for small register files, the benefits are high. As the register file size increases, the benefits decrease since the register file becomes less critical and a more effective use of it has smaller benefits.

For Mediabench and cognitive computing benchmarks, the proposed scheme provides significant performance improvements, as Figure 4.13(c) shows.

Figure 4.14 shows the average committed instructions per cycle (IPC) for the baseline and the proposed scheme. We plot the average IPC for the SPECfp and SPECint benchmarks shown in Figure 4.13(a) and Figure 4.13(b) respectively. The X axis represents the number of physical registers in the baseline. For the proposed technique, we assume a register file of equivalent area, taking into account its overheads. As it can be seen, our scheme can achieve the same performance as the baseline with a significantly lower number of registers. For instance, our technique with a floating-point register file equivalent to 56 registers achieves the same IPC as the baseline with 64, which represents a saving of 13% in area.

4.5.3 Analysis on Register Type Predictor

As described in section 4.3, the proposed technique uses a simple predictor with 512 entries that predicts the most likely reuse for each register and it is the configuration assumed in the experiments of this work. In case the prediction corresponds with the actual number of reuses, we count it as a hit; otherwise it is counted as a miss.

As we showed in Figure 4.3, more than 85% of the instructions, in average, require a destination register for which in our scheme we use the register predictor. Therefore, a register may be predicted to be reused correctly or incorrectly. On the other hand, the predictor may predict not to reuse a register which again can be a correct or an incorrect prediction. Figure 4.15 shows a breakdown for all the instructions of SPECint and SPECfp benchmarks. As this figure shows, despite the instructions that do not have a destination register, the rest will fall into one of these four categories.

Using the predictor two different kind of misprediction may occur. First, there may be a possibility to reuse a single-use register, however, due to an incorrect prediction, the register is not

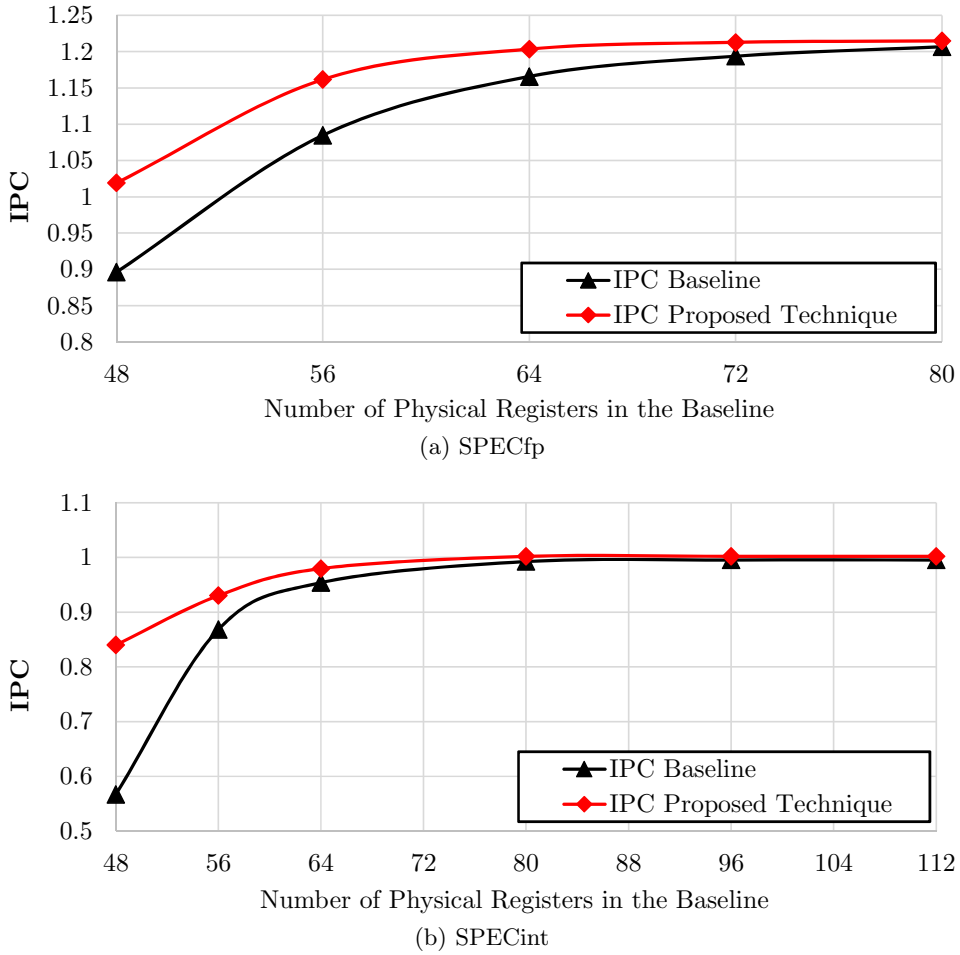


Figure 4.14: IPC of the proposed scheme and the baseline for different sizes of register files in the baseline and in the proposed technique.

reused. In this case, an opportunity of reusing a register is lost but no further actions are required. On the other hand, a register may have more than a consumer and it is predicted as a single-use register. In this case, as the register is reused incorrectly, the previous value of the register needs to be recovered as discussed in section 4.3.4.

4.5.4 Complexity of the Proposed Register Renaming Scheme

Our register renaming technique introduces a small overhead. The new hardware structures included are fairly small: the PRT requires 384-bits and the register predictor table contains 1 Kilobits. Regarding the register file, shadow copies are much cheaper than regular registers as they are not connected to read/write ports. Furthermore, only a small number of registers include shadow copies as shown in Table 4.3. Finally, the issue queue requires 4 additional bits per entry, which is small compared to the information already stored in modern out-of-order processors.

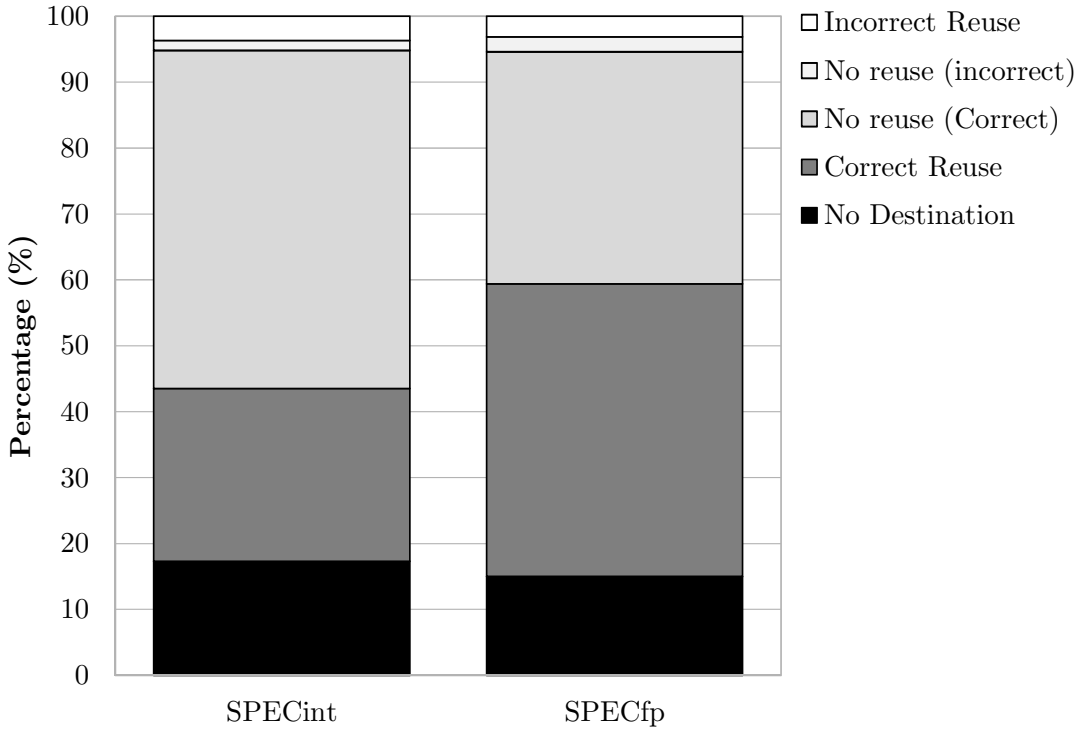


Figure 4.15: Accuracy of the register type predictor.

Regarding the renaming logic, our scheme adds an indirection from Register Map Table to the PRT and it requires an access to the predictor table. Handling dependencies among instructions renamed in the same cycle is not a problem, since out-of-order processors already handle this case and include additional checks and bypasses. Our scheme only requires extra checks to the Read-bit and 2-bit counter. Although this extra complexity in the renaming might impact the total delay of the rename stage, we assume our technique has no impact on cycle time for two reasons. First, some of the latencies can be overlapped, for example Register Map Table and predictor table can be accessed in parallel. Second, renaming is not typically in the critical path of modern out-of-order processors. In the worst case, we can further pipeline the renaming, since adding one stage to the front-end results in negligible impact on the overall IPC as reported elsewhere (see for instance [93]).

Finally, all comparisons in the work are done for configurations with the same area, in order to show that our renaming scheme is better than simply adding more registers.

4.6 Related Work

Using physical registers in a more efficient way has been the goal of many researches in the past. The most similar works to the technique presented in this thesis are those that try to delay the allocation of registers or anticipate its release.

Monreal et al. [65] proposed a register renaming technique based on virtual-physical registers [36]. By employing virtual-physical registers, their approach postpones the physical register allocation until the corresponding instruction finishes its execution. Their approach has a significant cost due mainly to the requirement of two separate register map tables, and two mapping operation per each destination register (from logical to virtual-physical, and from virtual-physical to physical).

Several works have been proposed to early release a register. Moudgill et al. [68] suggested to release physical registers as soon as the last instruction that redefines a register commits. The last-use tracking is based on counters which record the number of pending reads for every physical register. This initial proposal did not support precise exceptions since the counters were not correctly recovered when instructions were squashed. Later, Akkary et al. [9] proposed to improve the Moudgill scheme by adding an unmapped flag for each physical register, which is set when a subsequent instruction redefines that logical register. Then, a physical register can be released once its usage counter is zero and its unmapped flag is set. Moreover, for proper exception recovery of the reference counters, when a check-point is created, the counters of all physical registers belonging to the check-point are incremented. Similarly, when a check-point is released, the counters of all physical registers belonging to the check-point are decremented. In addition to the overheads that these techniques impose, they release a register far later than its actual lifetime.

Monreal et al. [66] proposed two different schemes to release a register. The first scheme waits for a redefining instruction to become non-speculative before releasing the previous version of its logical register. The second adds a new queue with multiple levels corresponding to the unconfirmed branches in the reorder buffer (ROB). Registers are released when the redefining instruction becomes non-speculative and the last instruction using the physical register has committed. On a branch misprediction, the relevant levels in the release queue are squashed. The downside of these techniques is that no recovery mechanism is in place to retain values released early. In the event of an exception or interrupt it would be impossible to reconstruct the precise processor state. They also need to add many large structures to the processor so that the status of redefining and last-use instructions can be maintained, increasing the complexity of the pipeline.

Ergin et al. [32] introduced a check-pointed register file to implement early register release. In their approach, a register is being deallocated immediately after the instruction producing the register's value commits itself and all potential consumers of this value have started execution before the redefining instruction is known to be non- speculative. To support branch misprediction, precise exceptions and interrupts, their proposal save the register value into the shadow bit-cells of the register where it can be accessed easily in such events.

Jones et al. [51] proposed a compiler-based technique for early register release. The compiler defines the points where registers will no longer be used and can be safely released. To guarantee that the state of the processor can be safely recovered after an interrupt or an exception, they used a check-pointed register file similar to the register file proposed in [32]. This scheme requires compiler support and changes in the ISA in addition to the overheads of the shadow cells in the entire register file.

Quinones et al. [80] proposed two techniques to release registers in out-of-order processors with register windows. The proposed techniques are based on the observation that when none of

the instructions of a procedure are currently inflight, all mappings of the procedure context are not needed. Therefore, these mappings and their associated physical registers can be released. Although their approach is beneficial, it is specific for processors with register windows, which are not common nowadays.

Note that previous work on early register release like [68] and [66] do not support precise exceptions. Precise exceptions is a must for modern out-of-order processors and, hence, these previous techniques cannot be implemented on modern CPUs. Work in [80] is specific to processors with register windows which are very uncommon nowadays. Finally, work in [51] requires compiler support and changes to the ISA. ISA extensions become legacy and make the technique less attractive. In comparison, our scheme supports precise exceptions and does not require any ISA extension. In short, previous techniques are not suitable for modern processors and, to the best of our knowledge, our technique is the only one that can reuse a physical register as early as the last use of this register is renamed, as opposed to other techniques that need to wait at least until the last use instruction commits.

4.7 Conclusions

In this chapter, we show that running the optimized GMM application puts a significant pressure on the register file and causes stalls in the renaming stage, limiting the performance of the processor. We observe that in GMM code the value generated by more than 57% of the floating-point instructions has only a single consumer. In addition, we show that for our benchmark suites, in average, the value generated by more than 50% of the floating-point instructions in SPECfp and more than 30% of the integer instructions in SPECint and mediabench are consumed only by one instruction.

To exploit this property, we propose a register renaming technique for out-of-order processors that allows to reuse single-use registers for the destination operand, instead of allocating a new physical register. We employ a multi-bank register file in which some banks have registers with integrated shadow cells, to recover the state of the processor in the event of branch mispredictions, exceptions and interrupts. We present the design of a simple register predictor which is used to allocate the most appropriate type of physical register for each destination register. We show that the proposed technique provides up to 38% speedup for the GMM application. Similarly, we achieve 6% speedup, in average, for the SPEC2006 benchmarks, considering same area in hardware. Alternatively, using the proposed register renaming technique, the same performance as the baseline can be achieved while reducing the area of the register file by 10.5%.

5

Hardware Accelerator Design

Previous chapters have explored ASR improvements by optimizing the software and modifying the microarchitecture of general purpose processors. To dramatically improve ASR energy efficiency, in this chapter, we present the design of our proposed hardware accelerator for GMM evaluation which is a more efficient but less flexible approach.

We propose a baseline hardware accelerator for GMM evaluation, which is the main bottleneck of ASR systems. The accelerator implements in hardware a lazy computation scheme that evaluates Gaussian distributions on demand, reducing the amount of computation by more than 50%. We show how lazy evaluation combined with batch processing of multiple frames by predicting active Gaussians, as presented in chapter 3, can be implemented in hardware. Fetching GMM parameters from the main memory is the main performance bottleneck and consumes significant amount of energy. We introduce a novel clustering scheme to reduce the size of GMM parameters. A thorough analysis of the impact of clustering in accuracy, power and performance is presented, in order to select the most efficient configuration. Later, we show that the use of clustering increases the degree of redundancy, and a novel memoization scheme is proposed to remove the redundant floating-point operations.

In this chapter, section 5.1 presents the basic accelerator design and the lazy evaluation scheme. Section 5.2 introduces our clustering techniques and the memoization scheme. Section 5.3 describes the evaluation methodology and the experimental results are provided in Section 5.4. Finally, Section 5.5 sums up our conclusions.

5.1 Hardware Accelerated Acoustic Scoring

In this section, we present a high-performance and low-power accelerator for GMM evaluation, with the aim of improving the energy-efficiency of acoustic scoring in mobile ASR systems. The accelerator focuses on the GMM evaluation since it is the main performance and energy bottleneck as reported in chapter 3.

We first introduce a base design of the accelerator, that consists on a straightforward hardware implementation of the GMM evaluation method described in Chapter 2. Next, we present a lazy Gaussian evaluation scheme implemented on top of the base design, in order to avoid GMM computation for inactive senones.

5.1.1 GMM Accelerator

We first review the data and the computations required for GMM evaluation before presenting the architecture of the accelerator. GMM evaluation consists of computing Equation 5.1 for each multidimensional Gaussian distribution in the acoustic model. Regarding the data, each Gaussian requires multiple memory accesses to fetch the determinant ($D_{m,g}$), the input features (x_c), the means ($\mu_{m,g,c}$) and the variances ($\sigma^2_{m,g,c}$). To speed-up GMM computation the value $1/2\sigma^2_{m,g,c}$ is precomputed offline, as it does not depend on the input x_c, and fetched from memory instead of the variance. Therefore, processing one component of a Gaussian distribution requires one subtraction, two multiplications to compute the square and multiply the result by $1/2\sigma^2_{m,g,c}$, and one subtraction to perform the accumulation. Once all the components have been computed, the result of the Gaussian distribution has to be written in memory.

$$\mathcal{N}(x, \mu_{m,g}, \sigma_{m,g}) = D_{m,g} - \sum_{c=0}^{M-1} \frac{(x_c - \mu_{m,g,c})^2}{2\sigma^2_{m,g,c}} \tag{5.1}$$

As discussed in previous chapters, we use the latest acoustic model for generic English language in Pocketsphinx which consists of 5000 mixtures of Gaussians or senones, where each mixture has 32 Gaussians and each Gaussian includes 36 components. Therefore, the memory footprint is 44.55 MB: 21.9 MB for means, 21.9 MB for the precomputed $1/2\sigma^2_{m,g,c}$ and 0.61 MB for the determinants, since parameters are stored as 32-bit floating-point numbers.

ASR systems typically compute GMM for batches of multiple frames in order to save memory bandwidth. Performing GMM evaluation for multiple frames in parallel improves temporal locality, as the means and variances can be fetched once from the off-chip system memory and reused for multiple frames, instead of fetching GMM parameters on a frame basis. Note that most of the memory bandwidth is employed for fetching Gaussians. To calculate the acoustic scores for one second of audio (100 frames), even by using lazy evaluation scheme, more than two gigabytes needs to be fetched from the main memory.

Figure 5.1 shows the initial architecture of the GMM accelerator. Its pipeline works as follows. First, the *Frame Fetcher* unit is triggered to read the input features for all the frames in the batch

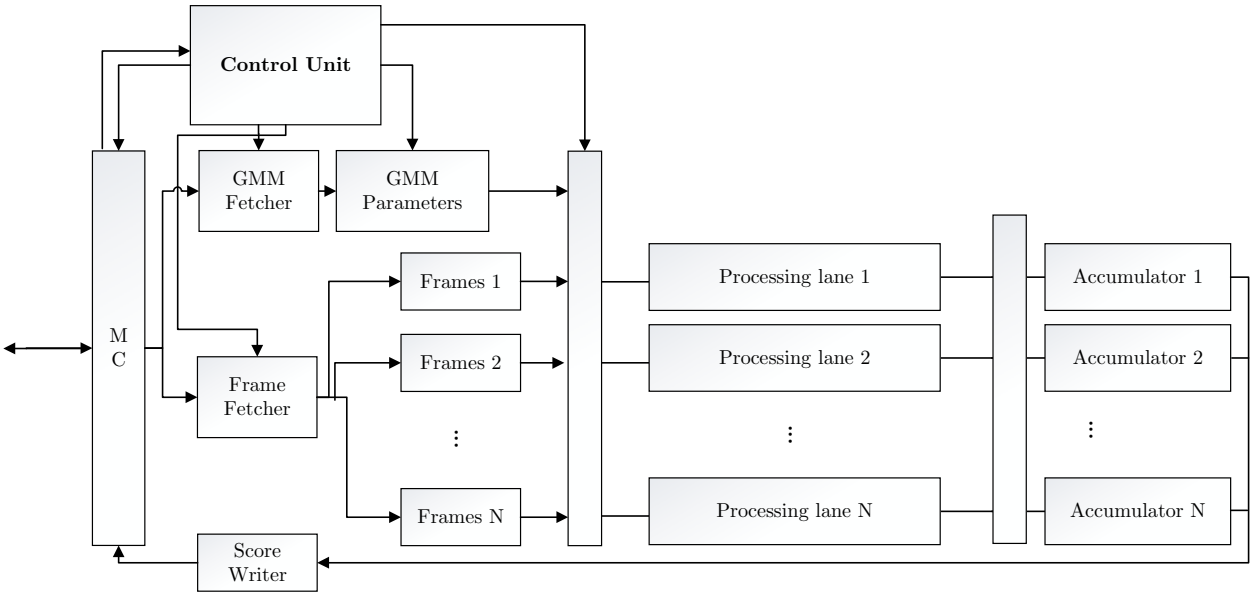


Figure 5.1: Architecture of the baseline GMM accelerator.

from main memory. The features are stored in an internal SRAM memory with multiple banks named *Frames 1*, ..., *Frames N* in Figure 5.1. Typical ASR systems employ between 30 and 40 features per frame. We design our accelerator to support up to 64 features. Therefore, the size of each bank depends on the number of frames and the numbers of banks:

$$Bank\ Size = \frac{F\ frames \times 256\ bytes/frame}{B\ banks} \tag{5.2}$$

For example, for a batch size of 128 frames and 8 banks, the size of each bank is 4 KB. Once all the features have been fetched from main memory and stored in the on-chip SRAM, the *GMM Fetcher* unit is triggered to read the parameters for the first Gaussian. These parameters, i. e. means and variances, are stored in the *GMM parameters* SRAM memory. The accelerator is designed to support Gaussian distributions with up to 64 dimensions and, hence, one Gaussian requires up to 512 bytes of storage. The *GMM parameters* SRAM memory includes two banks of 512 bytes to store the current and the next Gaussian distribution. By using the two banks the accelerator can overlap the latency for fetching the next Gaussian with the computations for the current Gaussian.

Once the means and variances are available in the on-chip SRAM, the Gaussian is evaluated for all the frames in the batch by using the *Processing lanes* and the *Accumulators*. Figure 5.2 shows the pipeline for performing GMM computations. Each *Processing lane* includes SIMD FPUs to process four components of a Gaussian in parallel for one particular frame. The computation of the Gaussian works as follows. First, 4 means and 4 variances are fetched from the *GMM Parameters*, by performing one access of 256 bits, and broadcast them to the different *Processing lanes*. In parallel, a 128-bit fetch is performed in each of the *Frames 1* ... *Frames N* SRAM memories to read 4 features for each frame. The *Processing lane* receives its 4 corresponding features and the

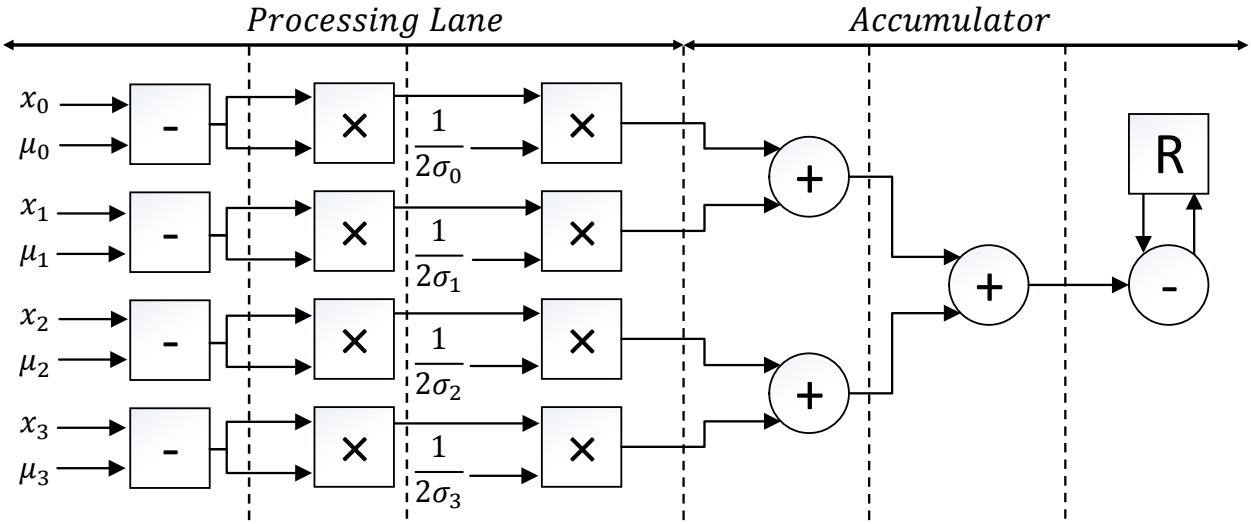


Figure 5.2: Pipeline of the *Processing lane* and *Accumulator* in the GMM accelerator.

4 means and performs the first subtraction, $x_c - \mu_{m,g,c}$. Next, a SIMD FP multiplier is used to compute $(x_c - \mu_{m,g,c})^2$. In the next stage, another SIMD FP multiplier is used to multiply the previous result by the variance.

The *Accumulator* employs a reduction tree to compute the summation of the 4 components, as shown in Figure 5.2. Furthermore, it performs the final subtraction to update the score of the Gaussian. This process is repeated until the current Gaussian is evaluated for all the frames in the batch. Finally, the *Score Writer* unit stores the scores for the different frames in main memory.

The accelerator is able to overlap memory accesses with computations to a large extent. To this end, the processing of a Gaussian is split in three stages: fetching means and variances from memory, evaluating the Gaussian for all the frames in the batch and writing the scores in main memory. The three stages are pipelined, so the accelerator fetches the next Gaussian from memory while it evaluates the current Gaussian and writes the scores for the previous Gaussian.

The capacity of the accelerator to hide memory latency depends on the batch size as illustrated in Figure 5.3. This graph shows the time required for memory transfers and computations for 128 frames of speech in a version of the accelerator with 8 processing lanes, using the acoustic model of Pocketsphinx that contains 160k 36-dimensional Gaussians. As it can be seen, the memory transfer time is significantly reduced when increasing batch size, as the Gaussians are fetched once from main memory and reused for multiple frames. For a batch size bigger than 16 frames the memory transfers can be completely hidden with useful computations. Bigger batch sizes improve temporal locality and increase the FP to MEM ratio, improving the capacity of the accelerator to tolerate memory latency. However, it also increases the latency of the ASR system, as more frames of audio have to be buffered before the recognition process starts. For example, a batch size of 128 frames introduces a delay of 1.28 seconds (10 ms per frame). Therefore, the batch size cannot be arbitrarily increased for online speech recognition systems.

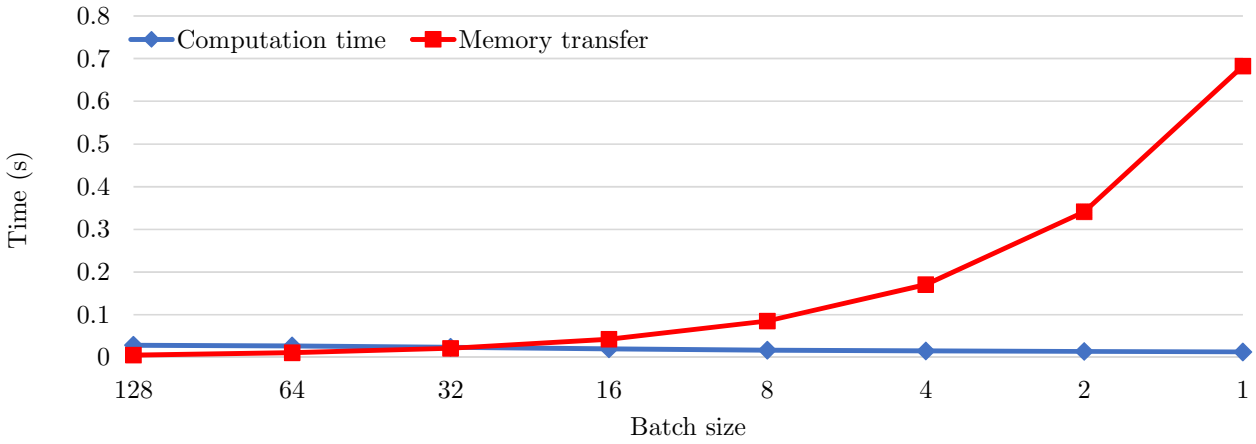


Figure 5.3: Execution time for memory transfers and computations for 128 frames of speech, using different batch sizes from 1 to 128.

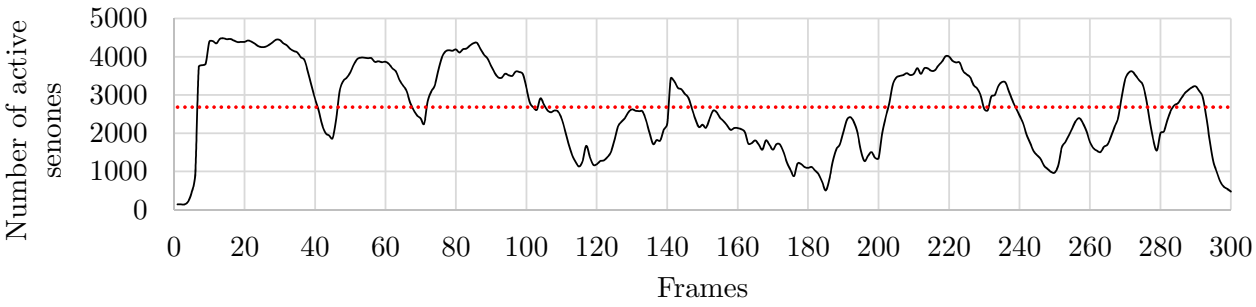


Figure 5.4: Number of active senones vs frames of speech. Red dotted line shows average number of active senones per frame.

5.1.2 Lazy GMM Evaluation

The result generated by the GMM accelerator is consumed by the *Search Engine*, the next stage in the pipeline of an ASR system. The *Search Engine* employs the acoustic scores to perform a search on a graph-based recognition network, in order to find the sequence of words with maximum likelihood. As discussed in Chapter 2, for large vocabulary ASR it is unfeasible to explore the entire search space due to its huge size. ASR systems prune away paths that are very unlikely to match the input stream. Due to the pruning, the scores of some mixtures or senones are not used by the *Search Engine* and, hence, GMM computation for these inactive senones can be skipped. Figure 5.4 shows the number of active senones in several frames of speech.

Many ASR systems, such as Pocketsphinx, generate a list of active senones, i. e. senones whose acoustic scores are required to perform the search, for every frame of speech. We extend our GMM accelerator to access this list of active senones in order to reduce the amount of computation and the memory bandwidth usage. Instead of blindly computing all the senones, the GMM accelerator computes Gaussians on demand as required by the *Search Engine*. This version of the accelerator

includes a SRAM memory to store the list of active senones. Assuming a batch size of F frames, this memory stores F bits for every senone. This bitmask indicates whether the senone is active on each one of the frames in the batch.

A senone can only be completely skipped if the bitmask indicates that it is inactive in all the frames of a batch. In that case, means and variances for its N Gaussian distributions (e.g. 32 in Pocketsphinx) are not fetched from main memory. Moreover, all the associated GMM computations are avoided. However, if the senone is active in at least one frame then the GMM parameters have to be fetched from main memory. In that case, the accelerator can still avoid some computations by disabling the *Processing lanes* for frames where the senone is not active.

Figure 5.5 shows the percentage of active senones in Pocketsphinx when processing the test set of LibriSpeech corpus [75] (5.4 hours of speech). The bigger the batch size, the higher the likelihood that a senone is active in at least one frame. For a batch size of 128 frames more than 90% of the Gaussians have to be processed, but for modest batch sizes of 4-8 frames there is a large percentage of senones that are inactive in all the frames of the batch. Note that online ASR systems prefer small batch sizes to reduce the response time.

The memory bandwidth required for fetching the list of active senones is significantly smaller than the bandwidth used for fetching Gaussians. For example, for Pocketsphinx acoustic model (5000 senones) and a batch size of 8 the list of active senones contains 40k bits (4.8 KB). By fetching this information the accelerator is able to completely skip more than 40% of the senones (see Figure 5.5), so the lazy GMM evaluation scheme avoids fetching 17 MB from main memory per batch. Moreover, the amount of floating point computations is reduced by 57%.

Note that the list of active senones produced by the *Search Engine* is only available for the current frame and, hence, it is not clear how to apply lazy evaluation when processing frames in batches. For this reason, software solutions that implement lazy evaluation, such as Pocketsphinx, process frames sequentially, whereas systems that exploit batching have to compute all the Gaussians for every frame. In this thesis, we propose a novel approach to combine the benefits of both lazy evaluation and batching (see section 1.2.1). The speech signal is quasi-stationary when considering a short interval of time. We exploit this behavior to implement a simple prediction scheme in hardware: the accelerator predicts that the active senones in the next N - 1 frames will be the same as in the current frame, being N the batch size. This simple scheme achieves 94% and 83% of accuracy for batch sizes of 3 and 8 frames respectively. We check the prediction with the list of active senones when it is available for each frame, and we only compute sequentially the senones that are mispredicted. For example, for a batch size of 8 only 17% of the senones are computed sequentially, whereas 83% employ batching. Therefore, our scheme avoids computing all the senones by using lazy evaluation, but it is still able to apply batching to a large extent, achieving substantial improvements in performance and energy consumption.

5.2 Clustering and Memoization

Online ASR systems require small batch sizes of just a few frames of speech in order to achieve high responsiveness. Moreover, our lazy GMM evaluation scheme presented in Section 5.1.2 also

Figure 5.5: Percentage of active senones for different batch sizes. A senone is considered active if it is active in at least one of the frames of a batch.

requires small batches to skip a substantial percentage of the Gaussian distributions. The main disadvantage of small batch sizes is that the memory transfer time cannot be completely hidden by computations, as shown in Figure 5.3.

Most of the memory bandwidth in GMM evaluation is employed for fetching means and variances. For example, for Pocketsphinx acoustic model and a batch size of 8 frames, 21.9 MB of means and 21.9 MB of variances have to be fetched from memory for every batch. In comparison, only 0.61 MB, 1.12 KB and 4.8 KB have to be fetched for determinants, input features and list of active senones respectively. Therefore, 98.61% of the memory bandwidth is used for reading means and variances.

In this section, we first explore different alternatives for clustering the GMM parameters, with the aim of reducing the memory bandwidth usage for GMM evaluation. We show that the best clustering scheme provides 8x bandwidth reduction, with a negligible loss in accuracy. We next extend our GMM accelerator introduced in Section 5.1 to support clustered GMMs, to reduce its memory requirements and increase the overlap of memory accesses and computations. Finally, we show that the clustering scheme increases the amount of redundant computations to a large extent, and propose a memoization technique that avoids 74.88% of the floating point operations, which translates into important savings in energy consumption.

5.2.1 Clustering GMM Parameters

Reducing the size of the datasets for ASR is key for both performance and energy consumption. For small batch sizes used in online ASR systems, the performance of acoustic scoring is mainly constrained by memory transfers as illustrated in Figure 5.3 and Figure 5.6. Furthermore, off-chip memory accesses are known to be one of the main sources of energy consumption in mobile devices and, hence, memory bandwidth reductions translate into important energy savings.

Compressing the GMM parameters can potentially alleviate the pressure on the memory subsystem. Since accuracy is also an important requirement for end-user applications based on speech

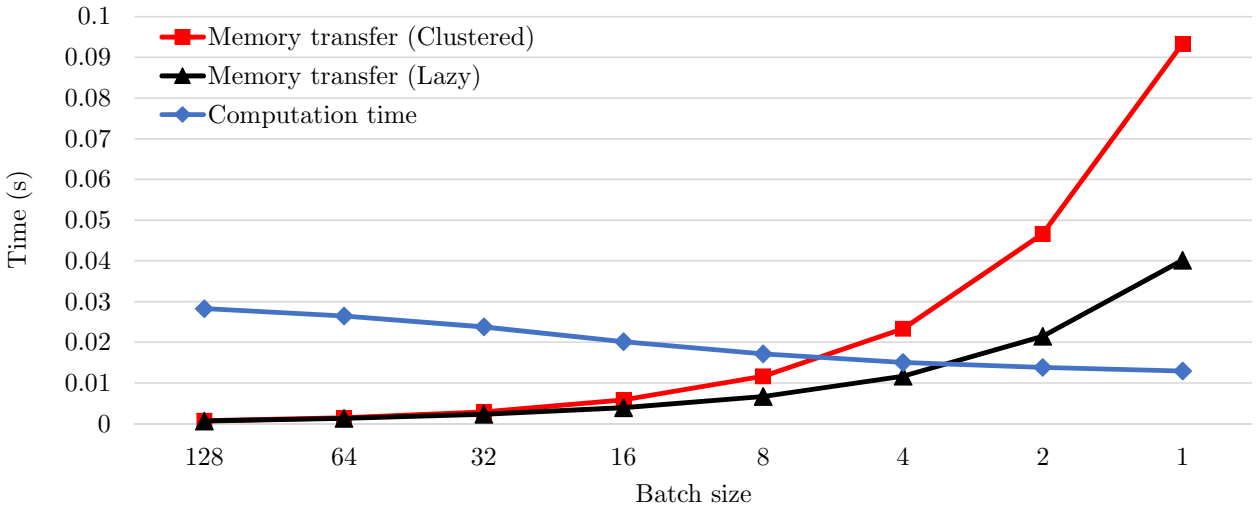


Figure 5.6: Time needed to transfer the data from main memory to the chip for various batch sizes for baseline (considering all senones as active), lazy evaluation and clustered data.

recognition, we first evaluated lossless compression methods. Unfortunately, lossless compression algorithms provide very modest compression ratios, even when using sophisticated software-based techniques that are not amenable for real-time systems. We obtained the best results with *7zip* algorithm, that provides 1.08x reduction in the size of Pocketsphinx acoustic model, whereas we achieved smaller compression ratios with *bzip2*, *rar* and *zip*. The main reason for such low compression ratios is that the data is stored in floating point format. Lossless algorithms specifically tailored for IEEE floating point format can be employed, but the state-of-the-art techniques [12] provide only around 2x compression.

Both means and variances are represented as 32-bit single-precision floating point data type. Figure 5.7 shows the compression ratio achieved using different software compression algorithms to compress the means and variances. According to our analysis, variances are integer values and can be represented using only few bits of mantissa and exponent in the floating point data type. This is the main reason that such algorithms can compress variances very well. However, this is not the case for means and as the figure clearly shows, none of the software algorithms can compress the means properly. It should be mentioned that most of the software approaches are compressing the whole data and obviously decompressing part of the data may not be feasible. In other words, these algorithms are not amenable for online decompression in real-time. With all these, significantly higher compression ratios of 4-8x are required to solve the problems with memory bandwidth usage in our GMM accelerator.

We have exploited the GMM parameters data. We applied various sophisticated software techniques to compress the data using lossless compression algorithm. However, in the best case using *7zip* compression algorithm we only get 1.08x reduction for the means. The main reason for such a low compression ratio is that the data types are floating point and no special locality or a pattern could be found easily [1]. It should be mentioned that achieving an acceptable compression ratio with these type of algorithms results in a complex and time-consuming data decompression

[1] We obtained similar compression ratio using bz2, rar, zip and several well-known compression algorithms.

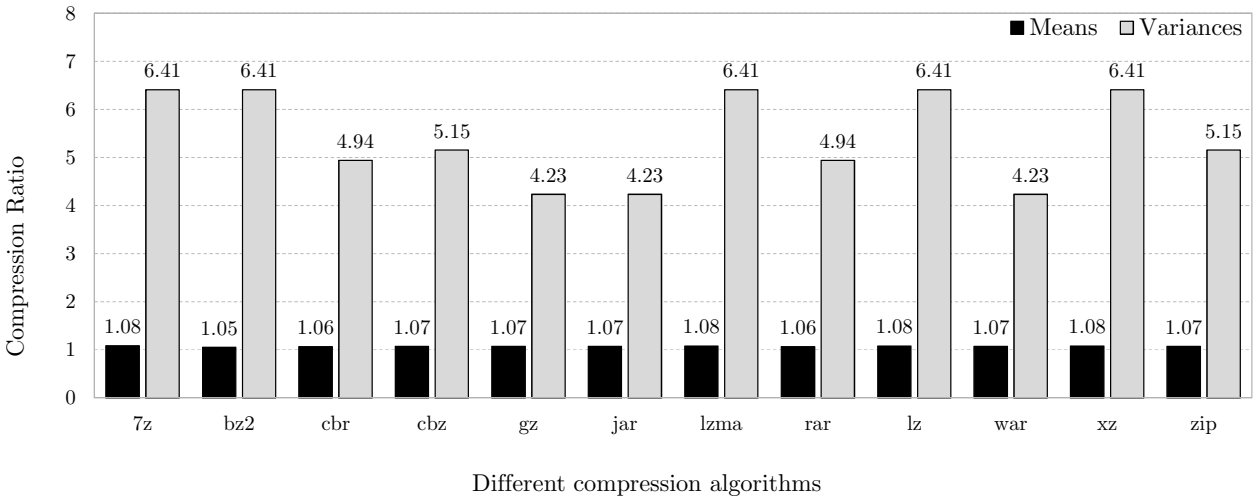


Figure 5.7: Compression ratio using different software compression algorithms to compress means and variances.

which causes a huge overhead.

As we show later in figure 5.13, even applying lazy evaluation, still main memory consumes considerable amount of energy. On the other hand, fetching huge amount of data limits the performance of the system due to bandwidth constraints. Several works have been done to alleviate this problem. Some works reduced the number of mixtures of Gaussians to reduce the computation complexity and memory requirement. However, these works reduced the accuracy of the ASR system to a large extent.

Lossy compression algorithms can be used to achieve larger compression ratios, as long as they do not cause a significant decrease in accuracy. One of the most successful techniques for lossy GMM compression is clustering [84, 28, 18], K-means being the most popular algorithm. Clustering schemes are among the techniques to reduce the data size. Most successful methods apply nonlinear quantization schemes, e.g. K-means, and reduce the data size up to 4x without significant loss of the accuracy. In these techniques, indexes will be stored instead of means and variances and these indexes will be used to access a table of centroids to estimate the means and variances. With clustering, the floating point values are replaced by significantly smaller integer indices into a codebook of centroids. For example, when using K-means with 256 clusters each 32-bit FP value is replaced by an 8-bit index, providing close to 4x compression ratio (1 KB is required for the codebook of 256 FP values).

On the other hand, instead of clustering individual scalar values, vector clustering for GMM can be used as described in [84] to further increase compression ratio. With vector clustering multiple FP values are represented with just one index. Pocketsphinx acoustic model consists of 160k 36-dimensional Gaussians. Clustering entire Gaussians only requires one index for every 36 FP values. However, the bigger the length of the vector the higher the accuracy loss and, hence, larger codebooks are required to achieve accuracy levels close to the uncompressed dataset. Note that the codebook of centroids is also part of the compressed dataset and it must be considered for computing compression ratio. In order to achieve a better trade-off between compression ratio and

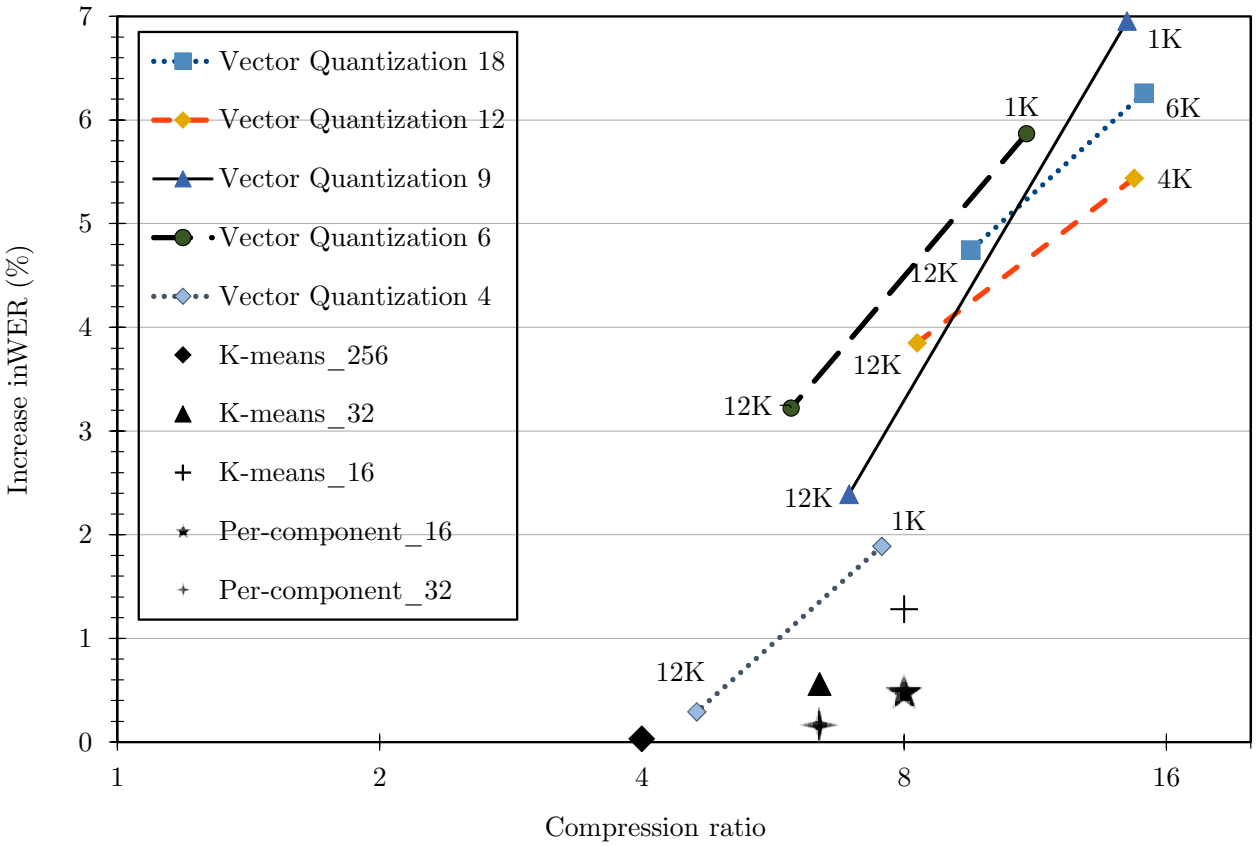


Figure 5.8: Compression ratio vs increase in Word Error Rate (WER) for different clustering algorithms.

accuracy, vector clustering can be implemented at the sub-Gaussian level, where each Gaussian is split in two 18-dimensional vectors, or three 12-dimensional vectors, etc.

We have implemented both vector and scalar clustering for GMM parameters and we have analyzed their impact on compression ratio and accuracy. The results are shown in Figure 5.8. Regarding vector clustering, we consider vector sizes of 4, 6, 9, 12 and 18. In addition, we change the number of clusters, i. e. the codebook size, between 1K and 12K. As it can be seen, for the same number of clusters the bigger the vector size the bigger the compression ratio, but the bigger the accuracy loss. Vector clustering achieves compression ratios close to 16x, but at the cost of a significant decrease in accuracy (more than 5% in absolute WER).

On the other hand, scalar clustering achieves better accuracy than vector quantization methods. Figure 5.8 depicts configurations using scalar clustering with K-means algorithm, for three different codebook sizes: 16 ("K-means_16"), 32 ("K-means_32") and 256 centroids ("K-means_256"). The bigger the codebook size the smaller the accuracy loss, but the smaller the compression ratio as indices require more bits. The configuration with 256 clusters provides a reduction of the dataset close to 4x with a negligible impact on accuracy, 0.02% increase in WER. For a codebook of 16 centroids each index only requires 4 bits and, hence, compression ratio is close to 8x. However, WER is increased by more than 1.3%.

We have done a comprehensive analysis of different clustering schemes including vector and scalar quantization for several state-of-the-art acoustic models using hours of large vocabulary corpuses. Our analysis shows that both scalar quantization and the vector quantization techniques can achieve up to 4x reduction in GMM parameter's size without significant loss in accuracy of the ASR system (see Figure 5.8).

We have done various analysis in order to achieve further data size reduction meanwhile maintaining the accuracy. Using the existing methods, we achieved up to 8x reduction in the size of the GMM parameters. Our results show that using K-means algorithm with 8 clusters for means and 32 clusters for variances (which together could be packed in an 8-bit index) the absolute WER increases by 1.3% while it achieves 8x reduction in the parameter's size. Using 32 clusters for the means and 64 clusters for the variances provides 5.8x reduction in the size of the GMM parameters. Note that as the means are usually in a much narrower range in comparison with the variances, they can be represented in less number of clusters than the variances. On the other hand, unexpectedly, vector quantization techniques reach this accuracy with less compression ratio and thousands of clusters [2].

For further analysis, we used data visualization to explore data patterns. We observed that the same component of different Gaussian distributions tend to exhibit similar values, in both means and variances (see Figure 5.9). In other words, the distribution of the values of one component in various Gaussians has much smaller standard deviation. Using this property in GMM parameters, which is observed in many acoustic models, we introduce a new clustering algorithm that it exploits this similarity. We call this technique *per-component clustering*. Based on the aforementioned observation, in this technique, first, we separate the GMM parameters of each component into different groups. Next, we apply K-means algorithm for each of the separated groups. By doing this we have indexes and centroids per each group of data. The benefit of this technique is that we have different clusters for each group.

Increasing the number of clusters results in better accuracy, as the distances from the original values to the centroids of the clusters tend to be smaller. However, it also results in smaller compression ratio, since more bits are required to represent the indices. In this thesis, we propose a novel strategy to increase the number of clusters without increasing the size of the indices. For an acoustic model with N Gaussians and C components per Gaussian, the dataset consists of an $N \times C$ matrix. In our scheme we apply clustering for each Gaussian component separately generating C different codebooks, one for each column. By doing this, the number of centroids is increased by a factor of C, but the index size remains the same as the size of each individual codebook is not increased. For example, assuming an acoustic model with 36-dimensional Gaussians and 16 clusters, we apply scalar K-means for each column of the matrix and generate 36 codebooks with 16 centroids, which means a total of 576 centroids for the entire dataset. Nevertheless, the index size is still 4 bits since each dimension of the Gaussian is restricted to its corresponding codebook. Note that the target codebook can be obtained from the column of the matrix, so indices in column

[2]Our results show that using vector-quantization in order to maintain the accuracy of the baseline we need 12k clusters and 18 partitions which costs a big complexity in both software and hardware (in software it causes a significant slowdown due to large cache miss ratio and in hardware it requires large caches to store the centroids). These techniques perform better in small-vocabulary ASRs [84]. The compression ratio achieved by this technique in order to maintain the accuracy of the ASR system is 4.32x.

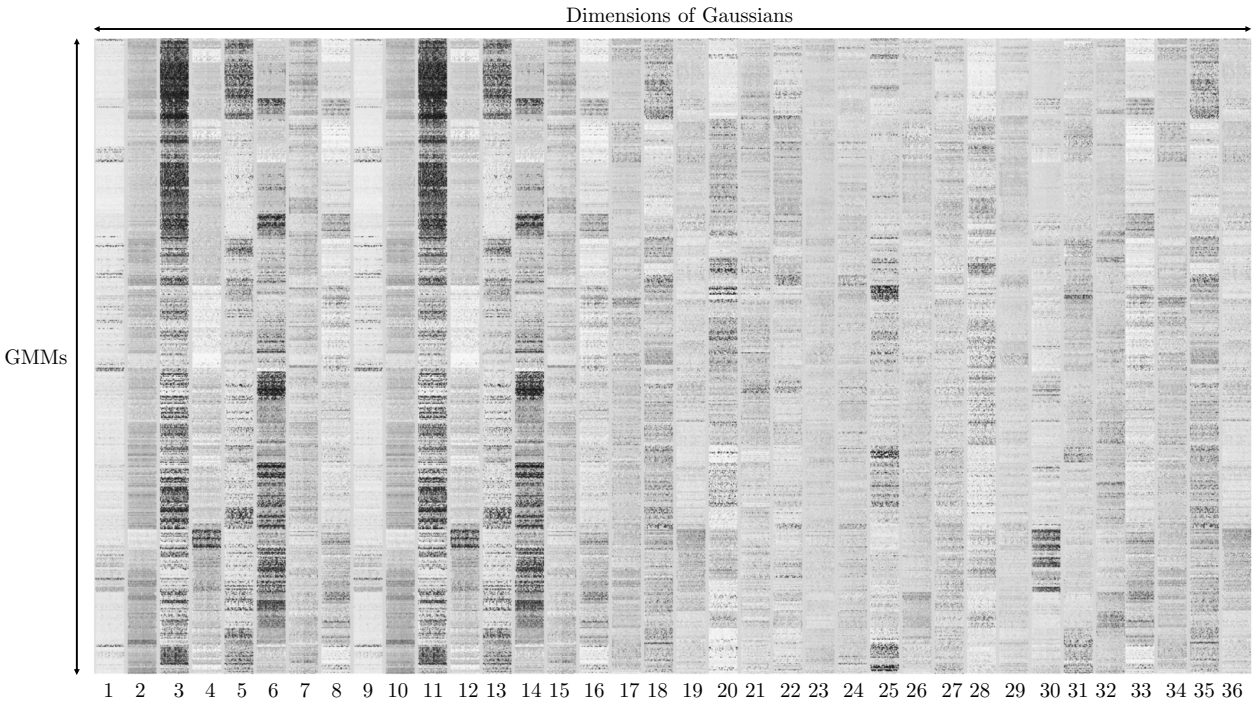


Figure 5.9: Grayscale visualized display of components of various Gaussian distributions which clearly shows the similarity of components.

c use codebook c and, hence, no additional information has to be stored to locate the codebook.

Figure 5.8 shows the results for our per-component clustering scheme, using 16 centroids per codebook ("Per-component_16") and 32 centroids ("Per-component_32"). Our scheme provides a better trade-off between compression ratio and accuracy than the schemes that apply K-means for the entire dataset with just one codebook. Compared to "K-means_256", "Per-component_16" doubles compression ratio, from 4x to 8x, despite increasing the number of clusters from 256 to 576, since each component is clustered separately and only 4 bits are required per index. Compared to "K-means_16", which also uses 4-bit indices, "Per-component_16" improves the increase in WER from 1.3% to 0.4% as it employs 576 centroids instead of 16 for the entire dataset. Therefore, per-component clustering achieves the same compression ratio as the schemes with a small number of clusters, while providing a level of accuracy close to the schemes with large codebooks.

Our experiments show that our per-component clustering scheme using 256 clusters for means and variances (8 bits instead of 64 bits), achieves close to 8x reduction in data size while increasing the absolute WER to 0.4% (see Figure 5.8). The error produced by our technique is 3.25x smaller than the alternative scalar clustering technique achieving the same compression ratio.

The per-component clustering generates highly tuned codebooks for each dimension. Figure 5.10 shows the Mean Squared Error (MSE) for each of the 36 dimensions in the acoustic model of Pocketsphinx, including our per-component clustering and the regular K-means with 16 clusters for the entire dataset. Per-component clustering introduces smaller errors for most of the dimensions, achieving higher accuracy with the same compression ratio.

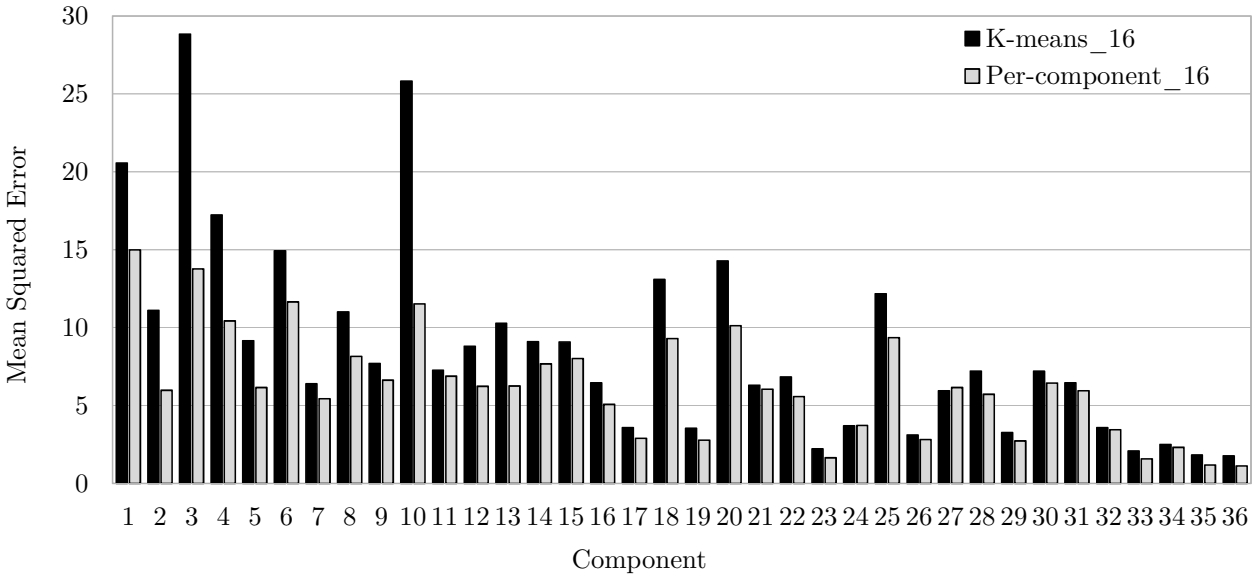


Figure 5.10: Mean Squared Error introduced by the clustering techniques for each component, or dimension, of the Gaussian distributions in Pocketsphinx English language acoustic model. Our per-component clustering provides smaller errors for most of the dimensions.

We have implemented in our GMM accelerator the per-component clustering with codebooks of 16 centroids. This configuration provides close to 8x compression ratio: 32-bit FP values are replaced by 4-bit indices, whereas up to 4 KB are used for codebooks (up to 64 dimensions multiplied by 16 centroids). For LibriSpeech test set (5.4 hours of audio) this configuration introduces a negligible error with respect to the uncompressed dataset, increasing WER by only 0.4%.

Our extensive experimental results show that our *per-component* clustering scheme delivers better accuracy in comparison with the proposed scalar and vector quantization techniques. We almost maintain the accuracy while reducing the GMM parameters up to 8x which achieves a huge reduction in main memory accesses and energy. We modify our baseline accelerator design to support lazy evaluation combined with clustering as illusterated in Figure 5.11. It should be mentioned that our accelerator supports both clustered and unclustered GMM parameters which can be easily configured by the user.

We extended our GMM accelerator to support clustering as illustrated in Figure 5.11. This version includes an on-chip SRAM memory, named *Centroids*, to store the codebooks. The accelerator supports Gaussians with up to 64 dimensions and codebooks with up to 16 centroids. The total size of the *Centroids* SRAM memory is 4 KB: 64 codebooks $\times$ 16 centroids/codebook $\times$ 4 bytes/centroid. The *Processing lanes* of the GMM accelerator require 4 means and 4 variances per cycle to achieve peak throughput. The *Centroids* memory is split in 4 banks of 1 KB with 2 read ports per bank, in order to fetch 4 centroids for means and 4 centroids for variances in one cycle. To avoid bank conflicts, codebooks for different components are interleaved with a factor of 4, as 4 consecutive components are processed per cycle.

When using clustering, the *GMM Fetcher* reads indices from main memory and stores them in

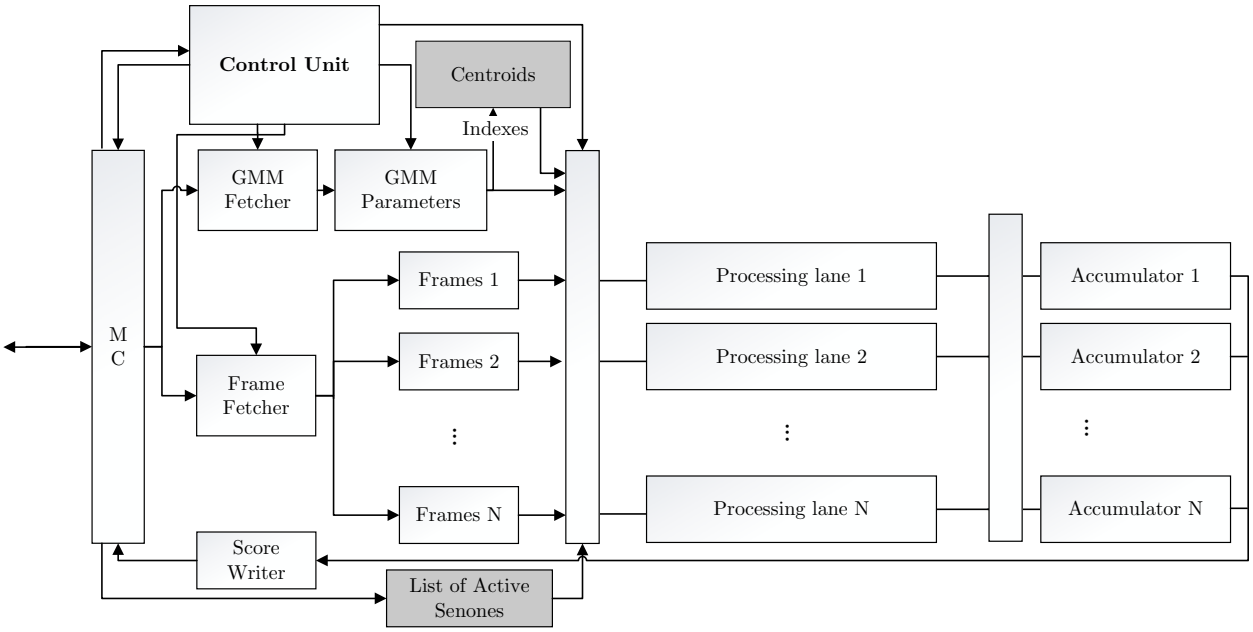


Figure 5.11: Architecture of the GMM accelerator including support for clustering and lazy evaluation.

the *GMM Parameters* SRAM memory. The pipeline of the accelerator includes an additional stage to fetch the centroids. In this new stage, the indices for means and variances read from the *GMM Parameters* memory are used to obtain the corresponding values from the *Centroids* memory. These centroids are then dispatched to the *Processing lanes*, the rest of the pipeline remains unmodified and works as described in Section 5.1.

The use of clustering provides a large reduction in memory bandwidth usage. For Pocketsphinx acoustic model, 43.8 MB are fetched from memory every batch to read means and variances when using the uncompressed dataset. Our clustering scheme reduces the size of the dataset to 5.5 MB, achieving a reduction in memory bandwidth close to 8x.

5.2.2 Memoizing GMM Computations

GMM evaluation for one frame of speech requires computing the expression $(x_c - \mu_{m,g,c})^2 \times (1/2\sigma^2_{m,g,c})$ for every component of every Gaussian (see Equation 2.5 in Chapter 2). The number of times the aforementioned expression is computed depends on the number of Gaussians in the acoustic model and the dimensionality of the Gaussians. For Pocketsphinx acoustic model, which contains 160K 36-dimensional Gaussians, this expression is evaluated 5.76 million times per frame of speech. However, when using our clustering scheme described in Section 5.2.1, each one of the 36 input features (x_c) can only be combined with 16 different means and 16 variances. The total number of unique evaluations is $36 \times 16 \times 16 = 9216$. Therefore, 99.84% of the evaluations of this expression are redundant, i. e. they use the same value of x_c, $\mu_{m,g,c}$ and $\sigma^2_{m,g,c}$ as a previously computed component.

In this thesis, we propose the use of memoization to avoid all these redundant computations.

Memoization is an optimization technique that avoids repeating the execution of redundant computations by reusing the results of previous executions with the same input values. The first time a computation is executed, its result is dynamically cached in a Look Up Table (LUT). Subsequent executions of the same computation will obtain the result from the LUT rather than recalculating it.

We extend our GMM accelerator to memoize the result of the expression $(x_c - \mu_{m,g,c})^2 \times (1/2\sigma^2_{m,g,c})$, which corresponds to the first subtraction and the two multiplications performed in the *Processing lanes* as illustrated in Figure 5.2. In this version of the accelerator, we decouple the *Processing lanes* from the *Accumulators*. The *Processing lanes* are first triggered to precompute all the unique combinations of the aforementioned expression. The output results of the *Processing lanes* are stored in a new on-chip SRAM memories, named *Memoization Buffers*, as illustrated in Figure 5.12. This SRAM memory stores the unique results in a 4D matrix with dimensions: $Num\_frames\_in\_batch \times Num\_features\_per\_frame \times Num\_means \times Num\_variances$. Our accelerator supports up to 8 frames per batch, 64 features per frame and 16 centroids for means and variances, so the total size for the *Memoization Buffers* is 512 KB.

Once all the unique computations are executed, the scores of the different Gaussians are computed for the given batch of frames. Evaluating a Gaussian consists on performing a summation of the corresponding results stored in the *Memoization Buffers*, using the *Accumulators*. The indices for mean and variance, together with the frame index and feature index are used to access the 4D matrix of precomputed results. Two new pipeline stages are included to perform this access as illustrated in Figure 5.12. In the first stage, the address of the precomputed result is obtained from the different indices. In the next stage, this address is employed for accessing the *Memoization Buffers* in order to get the precomputed result for the expression $(x_c - \mu_{m,g,c})^2 \times (1/2\sigma^2_{m,g,c})$. The values obtained from the *Memoization Buffers* are forwarded to the *Accumulators*, where they are employed to update the score of the Gaussian.

In summary, GMM evaluation requires four FP operations for each component of every Gaussian. The first three FP operations exhibit a high degree of redundancy, since only 0.16% of the computations are unique whereas 99.84% are redundant. We extend our accelerator to precompute these 0.16% unique FP values and store them in a relatively small table, so they can be later used for computing Gaussians.

In the base design of the accelerator, evaluating Pocketsphinx acoustic model for a batch size of 8 frames requires about 184 million FP operations: 8 $frames \times 160K$ *Gaussians* $\times 36$ *components* $\times$ 4 FP *ops*. When using memoization, we first precompute the unique values, this requires about 221K FP operations: 8 $frames \times 36$ *components* $\times 16$ *means* $\times 16$ *variances* $\times 3$ FP *ops*. In addition, to get the scores we have to perform the summation of 36 of these precomputed values, which requires 46.08M FP ops: 8 $frames \times 160K$ *Gaussians* $\times 36$ *components* $\times 1$ FP *op*. The total number of operations with the memoization scheme is 46.30M, i.e. 74.88% of the FP operations are avoided.

Computation reuse is lucrative only when the cost of accessing the structures used for memoization is smaller than the benefit of skipping the actual computation. According to our results obtained with CACTI and Synopsys Design Compiler at 28 nm, the energy for accessing one bank of the *Memoization Buffers* is much smaller than the cost of performing one FP subtraction and

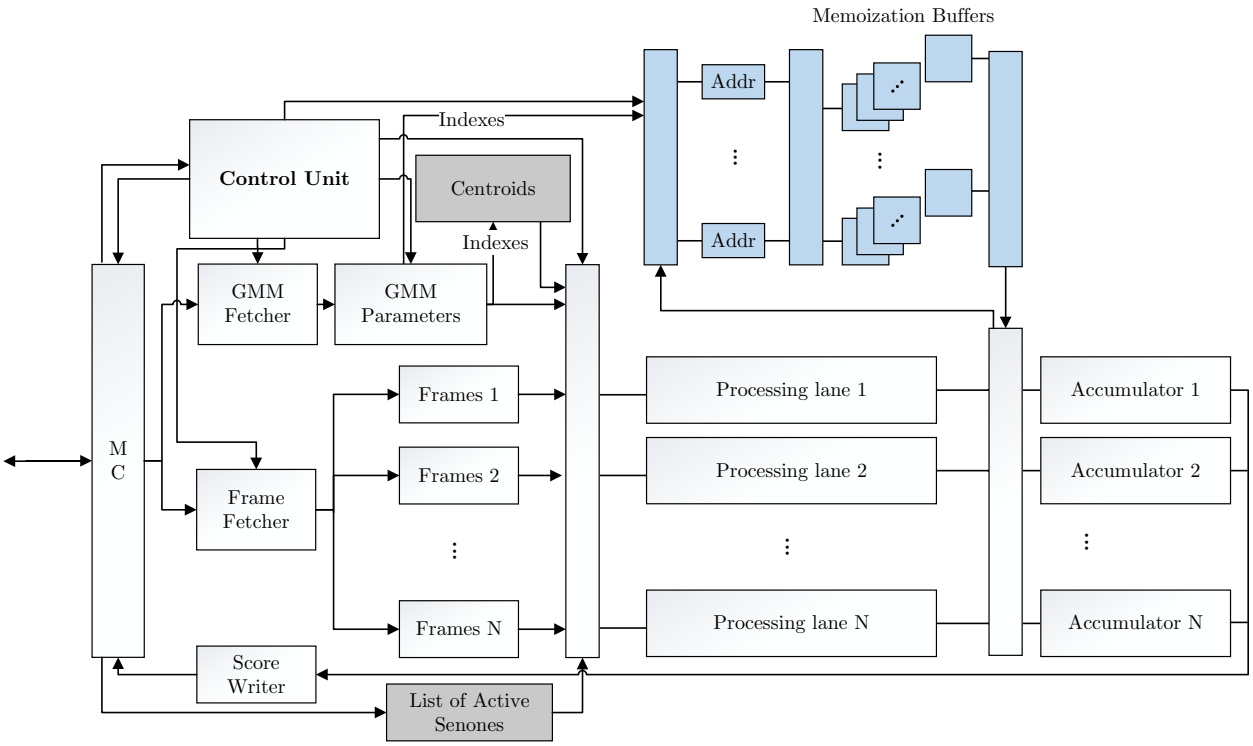


Figure 5.12: Architecture of the GMM accelerator including support for lazy evaluation, clustering and memoization.

two FP multiplications. Therefore, the cost of accessing the *Memoization Buffers* is substantially smaller than the cost of the avoided computations and, hence, the use of memoization improves the energy-efficiency of the accelerator as reported in Section 5.4.

In short, we show that our proposed clustering scheme performs very well while reducing the GMM parameters up to 8x. Using clustering scheme, the total possible values in a frame will be:

$$Total\ Possible\ Values = Components \times Clusters\_Means \times Clusters\_Variances \tag{5.3}$$

Therefore, all the operations will be among these few number of combinations. In other words, 99.84% of the computations are redundant. We explored this huge amount of redundancy and proposed a memoization technique to eliminate them.

5.3 Evaluation Methodology

We have developed a simulator that models the GMM accelerator described in secion 5.1. Furthermore, the simulator implements the lazy evaluation scheme presented in Section 5.1.2, our clustering technique (see Section 5.2.1) and the memoization scheme (see Section 5.2.2). The simulator works based on real trace inputs from the ASR system. We use this simulator for design space exploration and also for implementing the proposed schemes. We also implemented the fetch

Technology	28 nm
Frequency	1200 MHz
Number of lanes	8
Lane width	4
Parameter buffer	3 buffers, 1 KB each
Frame buffer	1 KB per each lane
Active senone	6 KB
Centroids buffer	1 KB
Write score	1 KB
Memoization buffers	512 KB
Functional units per lane	8 fp multipliers, 4 fp adders

Table 5.1: Hardware parameters for the proposed accelerator.

CPU	4x ARM Cortex A57 at 1.9 GHz
L1, L2	32KB L1, 2MB L2
Technology	20 nm

Table 5.2: Mobile CPU Parameters.

GPU	Tegra X1 Maxwell (GM204)
Streaming multiprocessors	256-core
Technology	20 nm
Frequency	1000 MHz
L1, L2 caches	24KB [61], 256KB

Table 5.3: Mobile GPU Parameters.

modules and the functional units pipelined and in different parallel lanes to increase the throughput. Table 5.1 shows the hardware parameters of the designed accelerator.

We use CACTI 6.5 [71] to estimate area and energy consumption of the different SRAM memories included in the accelerator. In addition, we have implemented in Verilog the rest of the components in the pipeline of the accelerator and we synthesized them using the Synopsys Design Compiler with a 28nm commercial library. The simulator provides the activity factors that are employed to estimate dynamic energy for the different components. We use the delay estimated by CACTI and the delay of the critical path reported by Design Compiler to set the target frequency so that the various hardware structures can operate in one cycle.

Regarding our datasets, we use the testset of audio files included in LibriSpeech [75] corpus, that consists of 5.4 hours of speech. On the other hand, we use the latest acoustic model for English language provided in Pocketsphinx. We also train several acoustic models using LibriSpeech train set to verify our proposed clustering scheme in addition to other acoustic models.

We compare our GMM accelerator with the performance of a software implementation running on a mobile CPU and a mobile GPU. We use NVIDIA Tegra X1 [7] as our baseline mobile platform. Tegra X1 includes an ARM CPU with parameters shown in Table 5.2 and a state-of-the-art mobile GPU with parameters provided in Table 5.3.

Regarding the software implementation, we use the method for acoustic likelihood computation based on matrix-matrix multiplication described in [29]. In this method, the Gaussians and the input frames are represented as 2D matrices and the acoustic scores are obtained by using the matrix multiply (SGEMM) operation of BLAS specification. For the CPU version, we use SGEMM implementation available in the OpenBLAS library [1]. For the GPU version, we employ the high-performance implementation of SGEMM provided in cuBLAS [3]. To measure energy consumption, we read the registers of the TI INA3221 power monitor included in the NVIDIA Jetson TX1 platform, in order to obtain power dissipation by monitoring CPU and GPU power rails as described in [62].

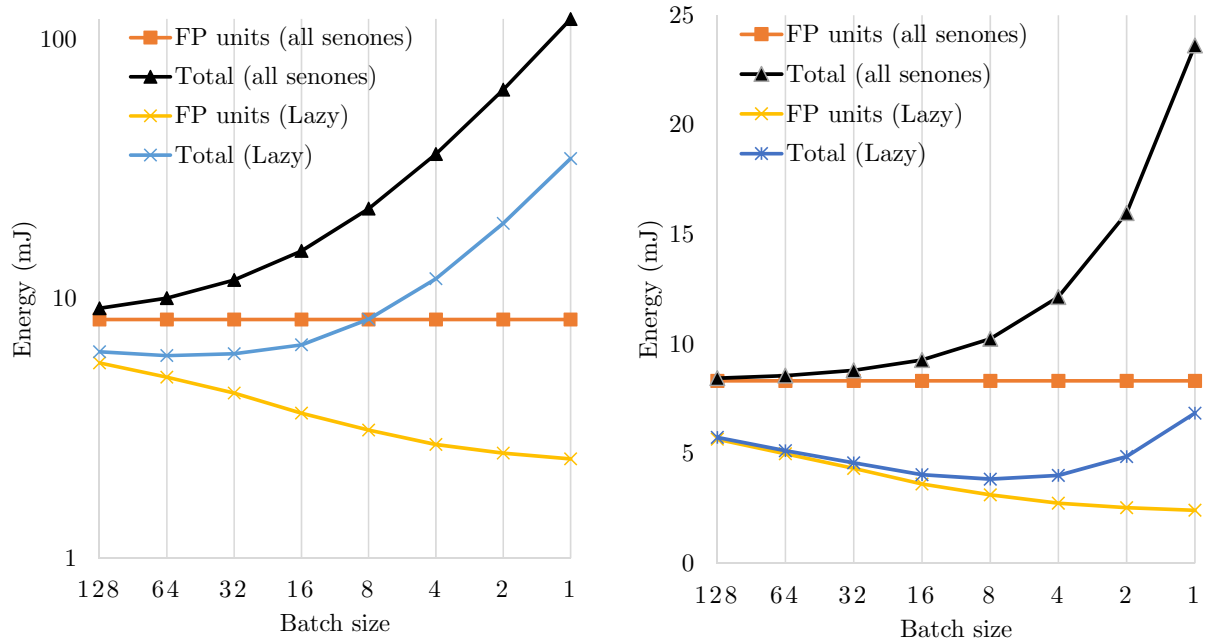
5.4 Experimental Results

In this section, we evaluate the performance and energy consumption of our GMM accelerator presented in Section 5.1. In first place, we analyze the impact of the batch size on the energy consumption of the accelerator. In second place, we compare our GMM accelerator with software solutions running on a mobile CPU and a mobile GPU. In our experiments we compare our designs with the software implementations running on a mobile CPU and a mobile GPU, these results are discussed in Section 5.3.

Figure 5.13 shows the energy consumption of the FP units and the overall accelerator versus batch size. The figure shows the total energy for processing 128 frames of speech. For the base design of the accelerator that computes all the Gaussians every frame, labeled as *all senones*, the energy for the FP units is constant as the amount of computation does not depend on batch size. However, overall energy is significantly reduced when increasing batch size due to the energy required for memory transfers. The bigger the batch size the smaller the memory bandwidth usage, as Gaussians are fetched once and reused for a bigger number of frames, improving temporal locality. Energy consumption of base design (*all senones*) exhibits similar behavior using both the original dataset and the clustered dataset, but the absolute energy consumption is smaller for the clustered dataset (Figure 5.13(b)) due to the memory bandwidth reduction achieved with clustering.

Regarding our lazy evaluation scheme, the energy required for FP computations increases with the batch size. As described in Section 5.1.2, the bigger the batch size the bigger the likelihood that a senone is active in at least one of the frames and, hence, the smaller the effectiveness in removing computations and memory accesses. For the original dataset (Figure 5.13(a)), overall energy is dominated by memory transfers and, hence, configurations with large batch sizes exhibit smaller energy consumption. However, for the clustered dataset (Figure 5.13(b)) the overall energy with the lazy scheme achieves the best results for a batch size of 8 frames. In this configuration, the memory bandwidth usage is reduced to a large extent due to the clustering, that reduces the size of the dataset by 8x, and the lazy scheme that avoids fetching GMM parameters for inactive senones.

On the other hand, Figure 5.13 also shows that both the lazy evaluation scheme and the clustering technique provide significant energy savings with respect to the base design for any batch size. Online ASR systems require small batch sizes of 4-16 frames to achieve high responsiveness, as buffering a large number of frames would introduce delays that are unacceptable for real-time



(a) Original acoustic model

(b) Clustered acoustic model

Figure 5.13: Energy consumption for floating point units (FP units) and entire GMM accelerator (Total), for the base design that evaluates all the Gaussians (all senones) and the accelerator using our lazy evaluation scheme (Lazy). (a) shows energy using the original uncompressed acoustic model, whereas (b) shows the results using our clustering scheme to reduce the size of the acoustic model.

systems. For small batch sizes, the configuration using lazy evaluation and clustering provides the lowest energy consumption.

Reducing the response time of the ASR systems is a key factor for end-users, specially for online speech recognition. We discussed the impact of large batch sizes on the latency of the ASR systems and also the overheads of extra computations. Figure 5.13(a) shows that reducing the amount of computations by applying lazy evaluation results in less total energy consumption. The benefit directly depends on the batch size. Large batch sizes increase the computation energy and latency while reducing the memory accesses. Figure 5.13(b) shows the total energy consumption considering computation and memory energy for clustered GMM parameters. As it shows, the energy consumption reduced manifold due to less memory accesses and lazy evaluation until batch size of 8. However, further reduction in the batch size results in extra memory fetches and more energy. We choose the batch size of 8 and the remaining results are provided by this batch size.

Figure 5.14 shows the speedup and the energy reduction achieved by the GPU and the different versions of our GMM accelerator with respect to a modern mobile CPU. The configuration named *GPU* corresponds to the mobile GPU with parameters shown in Table 5.3. *ASIC* corresponds to the base design of the GMM accelerator as described in Section 5.1.1. *ASIC+L* is the base design including the lazy Gaussian evaluation scheme presented in Section 5.1.2. *ASIC+L+C* is the GMM accelerator including lazy evaluation and the clustering technique introduced in Section 5.2.1.

Finally, *ASIC+L+C+M* configuration includes lazy evaluation, clustering and the memoization scheme described in Section 5.2.2.

The mobile GPU included in NVIDIA Tegra X1 provides 27.9x speedup and 14.6x energy savings with respect to the mobile CPU. The matrix multiply operation, that is used to implement the acoustic model in the software-based solutions, exhibits a high degree of data parallelism and, hence, it benefits from the large number of functional units provided by the GPU. In addition to the performance improvement, the GPU achieves high energy-efficiency for this data parallel code, reducing the overheads of instruction fetching and decoding by exploiting SIMD execution model.

On the other hand, the base design of the accelerator, named *ASIC* in Figure 5.14, provides 70.7x speedup and 691.5x energy reduction over the mobile CPU. The *ASIC* includes hardware specifically designed to accelerate GMM evaluation, avoiding the overheads of software implementations and delivering high-performance and energy-efficiency for acoustic scoring.

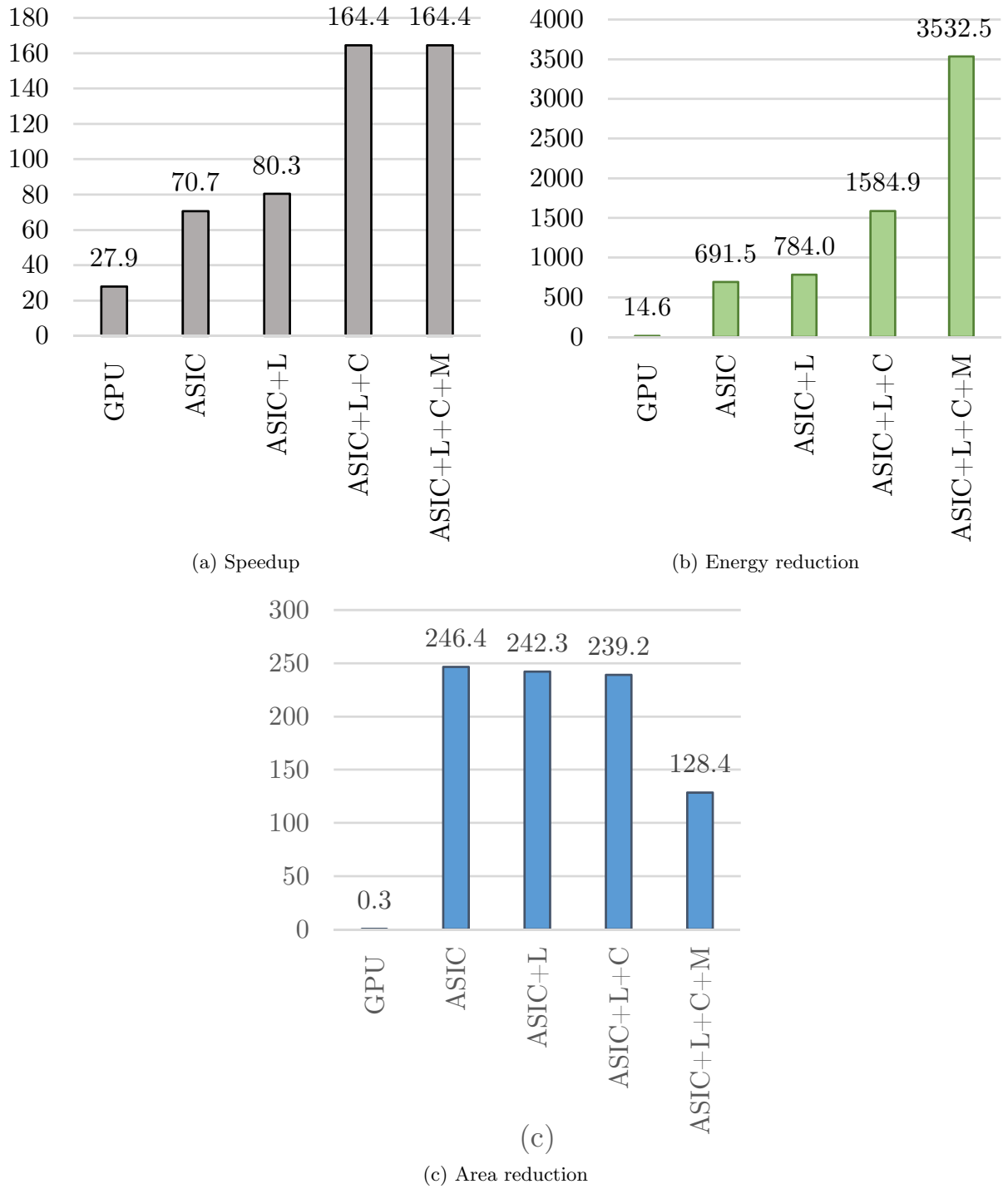
The lazy Gaussian evaluation scheme improves performance and energy-efficiency of the base design, achieving 80.3x speedup and 784x energy reduction over the mobile CPU. For a batch size of 8 frames, the lazy evaluation scheme avoids the computations and memory accesses for more than 40% of the Gaussians on average, as reported in Figure 5.5.

As Figure 5.14(a) and Figure 5.14(b) show, the *ASIC+L+C+M* design achieves 2.32x speedup over the baseline design while reducing the energy consumption by 5.1x. The main reason to achieve this reduction in energy is the memoization technique. On the other hand, applying clustering and lazy evaluation, which significantly reduce memory bandwidth usage, provide a significant speedup.

The lazy evaluation, *ASIC+L*, increase the performance by 13% in comparison with the baseline design. The main constraint which limits the performance is the memory bandwidth. Applying clustering eliminates the memory bandwidth bound and increases the performance by 2.04x in comparison with *ASIC+L*. In terms of energy, *ASIC+L+C* consumes 2.29x less energy since we mainly remove the extra computations by more than 50% using lazy evaluation and reducing the memory accesses and bandwidth by applying clustering. Note that using our prediction scheme presented in section 5.1.2 only computes sequentially the senones that are mispredicted. In our experiments for measuring the performance and energy consumption of the accelerator we take this into account.

Our clustering technique provides large improvements in performance and energy consumption. The *ASIC+L+C* configuration achieves 164.4x speedup and 1584.9x energy reduction over the mobile CPU. We use our per-component clustering with 16 centroids for means and 16 for variances as described is Section 5.2.1. The per-component clustering provides 8x reduction in memory bandwidth usage, which results in speedups, since memory transfer time is one of the main bottlenecks for small batch sizes as reported in Figure 5.3, and energy savings as off-chip memory accesses are one of the most expensive operations in terms of energy for mobile SoCs.

Figure 5.14 shows that the memoization scheme does not provide any performance benefit, as this technique targets energy savings. Memoization does not increase the throughput of the accelerator, we still have 8 *Accumulators*. In the base design, the input of the *Accumulators* comes from the *Processing lanes*, whereas with the memoization scheme the input comes from the *Memoization Buffers* as illustrated in Figure 5.12. Hence, we replace the three FP operations performed



(a) Speedup

(b) Energy reduction

(c) Area reduction

Figure 5.14: Speedup, energy consumption and area of the mobile GPU and the different versions of the accelerator in comparison with the baseline, a modern ARM A57 mobile CPU. In the performance and energy comparison results we just compare the GMM evaluation stage of the ASR pipeline.

Platform	4x ARM A57	Tegra X1	B+L+C+M
Year	2015	2015	2017
Platform Type	Mobile CPU	Mobile GPU	ASIC
Technology	20nm	20nm	28nm
Frequency (GHz)	1.90	1	1.20
Power (Watts)	2.45	9.22	0.11
GFLOPS/W	0.91	6.90	1465.36
Frames/s	95	2688	15802
Frames/J	37.9	558.5	134736.8

Table 5.4: Performance and power of different processors.

in the *Processing lanes* by one access to the *Memoization Buffers*, that requires substantially less energy as described in Section 5.2.2. Overall, 74.88% of the FP operations are replaced by memory accesses to the memoization table, improving the energy reduction over the mobile CPU from 1584.9x in the *ASIC+L+C* to 3532.5x in the *ASIC+L+C+M*. Note that the memoization scheme requires a first stage for creating the memoization table. However, this precomputation only takes a negligible 0.15% of total execution time, as the number of entries in the memoization table, i. e. the number of possible combinations of means, variances and input features, is fairly small when using the clustered acoustic model.

The *ASIC+L+C+M* design has a very similar performance to the *ASIC+L+C*. The throughput of the both designs is the same. In *ASIC+L+C+M*, first we precompute the non-redundant combinations (Equation 5.3) and store them in the lookup tables and after we start evaluating the scores. Once pre-computing the values is done, we power-gate the processing lanes, Frame buffers and Fetch Frames unit, which are shown in Figure 5.12. The time for precomputing the scores and storing them in the lookup tables is only 0.15% of the total execution time. On the other hand we remove the time needed to fill the pipeline stages. Regarding the energy consumption, the *ASIC+L+C+M* design consumes 2.23x less energy in comparison with the *ASIC+L+C* design. The main reason to reduce the energy is that we remove 99.84% of the redundant computations.

Regarding the area, Figure 5.14(c) clearly shows that the area required for the proposed accelerator is orders of magnitude smaller than a mobile CPU or a mobile GPU. The increase in the area required by the *ASIC+L+C+M* design in comparison with other designs is due to the memoization buffers. As Table 5.1 shows, the *ASIC+L+C+M* design has 512 KB memoization buffers to cache the result of unique computations for every batch of frames.

Table 5.4 shows power and performance of the different processors for GMM. Since each frame corresponds to 10 ms of speech, a real-time ASR system must be able to process at least 100 frames per second. The mobile CPU processes 95 frames/s while dissipating 2.45 W. The mobile GPU processes 2688 frames/s but at the cost of increasing power dissipation to 9.22 W. Our GMM accelerator achieves higher performance than the mobile GPU, processing 15802 frames/s while dissipating as low as 110 mW. Therefore, the accelerator provides a large improvement in performance per watt with respect to the mobile CPU and GPU.

To sum up the energy-performance analysis, Figure 5.15 plots energy reduction vs speedup for the GMM evaluation. The accelerator provides significantly higher energy-efficiency than the mobile processors. *ASIC+L+C+M* configuration achieves the best results, providing 5.89x speedup

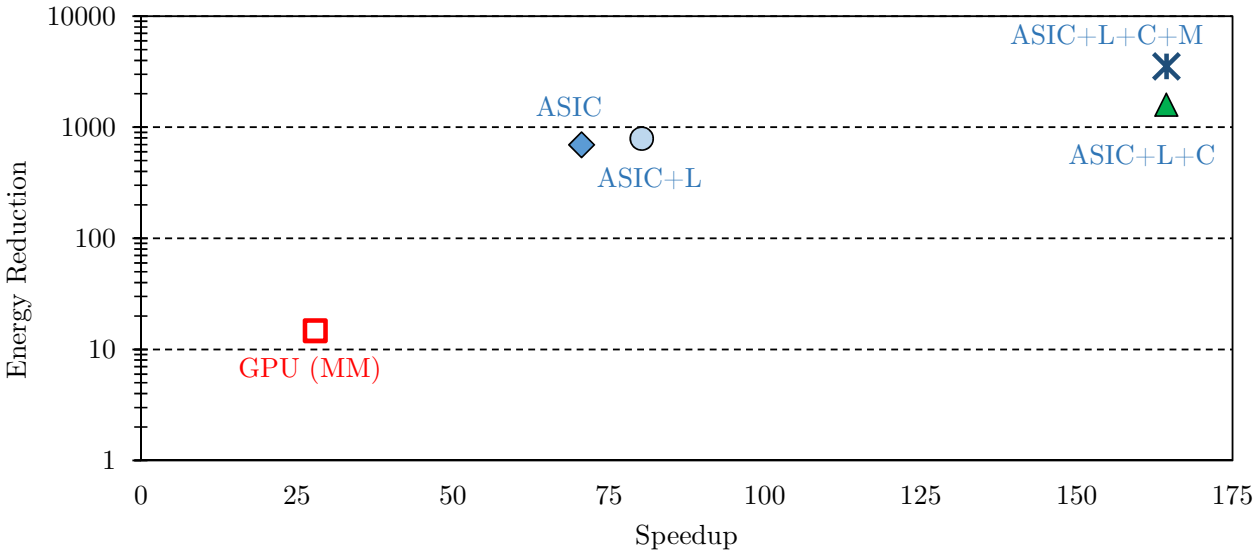


Figure 5.15: Speedup and energy reduction for different versions of the accelerator versus GMM evaluation in the mobile CPU and GPU.

over the mobile GPU, while reducing energy consumption by 241x and requires an area of 0.94 mm^2. The overall system achieves 4.66x speedup and 4.54x energy reduction for the entire ASR pipeline. We overlap memory transfers with computations to a large extent so they do not degrade the performance of the system.

Using accelerated acoustic scoring, Viterbi search becomes the most time and energy consuming stage of the ASR pipeline. Although Viterbi search can be implemented in software for CPUs or GPUs, it also can be implemented in hardware to improve performance and energy efficiency [105, 104].

5.4.1 Discussion

Off-chip memory accesses are clearly the main bottleneck for performance and also represent the main source of energy drain. Due to the large size of the acoustic model parameters, they can not stored on-chip. Using our clustering scheme we show that the ASR system achieves almost the same accuracy while reducing the acoustic model parameters manifold. We could further reduce memory energy by adding an eDRAM at the cost of larger area in the accelerator in order to store the GMM parameters on-chip. Note that in mobile and wearable devices main memory is limited and ASR systems are not the only consumers of the memory.

It should be mentioned that in the design of the accelerator, we implement each techniques on top of the previous. In other words, our latest version of accelerator, *ASIC+L+C+M*, supports clustering per se without memoization, and also unclustered GMM parameters. Our designs support all types of acoustic models which they differ only in the number of mixtures of Gaussians, Gaussian distributions per mixture and the Gaussian dimension.

Our accelerator supports any acoustic model based on Gaussian Mixture Models, so it works for speech in any language. Although Deep Neural Networks are a promising alternative for acoustic scoring, the vast majority of ASR software is still based on GMM. In this thesis, we use Pocketsphinx as our baseline ASR system since we target mobile platforms, but our accelerator also supports the acoustic models available in Sphinx 4, Kaldi, Julius or HTK. Moreover, we design our accelerator with enough throughput and storage to support in real-time larger and more accurate acoustic models that are likely to appear in the next years. We believe speech recognition will be a feature supported by the majority of computing devices in the near future, and acoustic models will evolve towards more complex ones for the sake of better accuracy.

GMM is a popular machine learning algorithm also used in different areas. The proposed GMM accelerator can be used for other applications that are relevant for mobile devices, especially in the area of computer vision. GMMs are employed for image segmentation [34, 38], image retrieval [86], tracking people in images or detecting and tracking moving objects in video sequences [24].

5.5 Conclusions

In this chapter, we present a custom hardware accelerator for large-vocabulary, speaker-independent, continuous speech recognition, motivated by the increasingly important role of automatic speech recognition systems in mobile and wearable devices with limited resources. Our proposed accelerator design includes innovative techniques to improve performance and energy efficiency of Gaussian evaluation. First, we implement in hardware a lazy evaluation scheme and a prediction technique to compute Gaussian distributions on demand, eliminating more than 50% of the computations for inactive senones. Second, we propose and employ a novel clustering scheme to reduce memory footprint and memory bandwidth usage by 8x with a negligible impact on the accuracy of the ASR system. Third, the accelerator implements a memoization scheme to remove all the redundant computations, avoiding 74.9% of the total floating point operations. The final design using the proposed clustering scheme and memoization achieves 5.89x speedup over an optimized and state-of-the-art software implementation running on a high-end mobile GPU, while reducing energy consumption by 241x.

6

Conclusions and Future Works

In this chapter the main conclusions and contributions of this thesis are presented, as well as some open-research areas for future work.

6.1 Conclusions

The purpose of this thesis is to characterize the limiting factors of state-of-the-art CPU architectures for ASR systems, providing efficient techniques to improve performance and energy-efficiency of ASR systems.

In this thesis, we present an energy/performance analysis of Automatic Speech Recognition (ASR) system when running on general purpose CPUs. We show that the Gaussian evaluation of the acoustic model is the most computationally expensive component, as it represents more than 80% of total execution time. Most of the CPU stalls are due to mispredicted branches and accesses to system memory. Regarding energy consumption, DRAM is clearly the main source of energy drain.

We propose several software optimizations to alleviate the bottlenecks identified in the analysis. First, we remove conditional branches from the innermost loop of the Gaussian evaluation code, achieving 12% speedup and 11% energy savings. Second, we propose a multi-frame Gaussian evaluation scheme with prediction of active senones which results in a reduction of off-chip memory accesses by 57.1%. In the next step, we exploit the Vector Processing Unit (VPU) via SIMD instructions and a new memory layout to boost Gaussian evaluation and improve energy efficiency. Using SIMD instructions in the implementation and our multi-frame Gaussian evaluation scheme provide 2.68x speedup and 61% energy savings on a modern Intel Skylake CPU. Similarly, on modern Atom and ARM mobile CPUs, it obtains 1.88x and 1.85x speedup and reduces energy

consumption by 55% and 59% respectively. Note that all the performance improvements and energy savings are achieved without any loss in the accuracy of the ASR system.

After optimizing and vectorizing the ASR application at software level, we observe that the register file faces more pressure as we remove other sources of stalls like main memory and branches. Based on an extensive analysis on various set of benchmarks, we show that not only for GMM application, but also in other benchmarks, in average, the value generated by more than 50% of the floating-point instructions and more than 30% of the integer instructions are consumed only by one instruction.

We highlighted the shortcomings and complexities of previous work and based on the aforementioned observation, we propose a novel register renaming technique for modern out-of-order processors that allows to reuse this single-use registers for the destination operand, instead of allocating a new register. To recover the state of the processor in the event of branch mispredictions, exceptions and interrupts, we employ a multi-bank register file in which some banks have registers with integrated shadow cells. A simple register predictor is proposed to allocate the most beneficial type of the register for each instruction. Considering similar costs in hardware, we show that the proposed technique provides up to 38% speedup for the GMM application. Moreover, it provides 6% speedup on average for the SPEC2006 benchmarks. Alternatively, the same performance as the baseline can be achieved while reducing the area of the register file by 10.5%.

Finally, we design and propose a custom hardware accelerator for large-vocabulary, speaker-independent, continuous speech recognition. Our accelerator focuses on the evaluation of the Gaussian Mixture Model (GMM) for acoustic scoring, as this stage is the main bottleneck in speech recognition systems. Our design includes innovative techniques to improve the energy efficiency of the proposed accelerator. It implements in hardware a lazy evaluation scheme and a prediction technique to compute Gaussian distributions on demand, avoiding more than 50% of the huge floating-point computations. Moreover, it employs a novel clustering technique to reduce memory bandwidth usage by 8x with a negligible impact on accuracy. For further reducing the energy consumption, the accelerator implements a memoization scheme to remove redundant computations, avoiding 74.9% of the floating point operations. The final design achieves 5.9x speedup over a state-of-the-art software implementation running on a high-end mobile GPU, while reducing energy consumption by 241x.

6.2 Contributions

In this thesis different schemes, from software level to microarchitecture level and ASIC design have been proposed to improve the performance and energy efficiency of ASR systems. The main contributions obtained through this dissertation are summarized as follows.

In first place, we have done a thorough analysis to identify the performance bottlenecks and the sources of energy drain when running ASR application on mobile and desktop CPUs. This thesis introduces several techniques at software level to improve the efficiency of ASR systems running on modern processors. We eliminate the stalls which branch mispredictions caused them by refactoring the code. By proposing a new memory layout to store the GMM parameters in the

main memory and changing the ways to process them, we efficiently exploit vectorization with no need to do horizontal reduction. Furthermore, we propose a simple and very efficient scheme to predict the active senones in small batches of frames. All these optimizations are implemented together and result in significant performance improvement and energy reduction on variety of mobile and desktop CPUs, as presented in Chapter 3. This work has been published in the IEEE Transactions on Multi-Scale Computing Systems (TMSCS) Journal:

- Hamid Tabani, Jose-Maria Arnau, Jordi Tubella, Antonio González, " Performance Analysis and Optimization of Automatic Speech Recognition," Multi-Scale Computing Systems (**TMSCS)**, *IEEE Transactions on,* 2017, DOI: 10.1109/TMSCS.2017.2739158.

In second place, we propose a novel register renaming technique for Out-of-Order processors, which is able to reuse single-use registers to reduce the pressure on the register file. Unlike previous works, our scheme is able to precisely recover the state of the processor after event of branch mispredictions, interrupts and exceptions. The proposed scheme is implemented by applying some changes to the renaming table, issue queue and register files while it does not need any changes in the compiler nor the ISA. We show that for cognitive benchmarks, mediabench and SPEC2006 benchmarks it provides considerable performance improvements, as reported in Chapter 4. This work has been published in the 24th International Conference on High Performance Computer Architecture (HPCA):

- Hamid Tabani, Jose-Maria Arnau, Jordi Tubella, Antonio González, " A Novel Register Renaming Technique for Out-of-Order Processors, " High Performance Computer Architecture **(HPCA)**, *24th International Conference on*, Feb. 2018, Vienna, Austria.

In third place, we design and propose a hardware accelerator for GMM evaluation. Our accelerator consumes less energy and outperforms CPUs and GPUs by orders of magnitude. Our accelerator implements in hardware our scheme to predict the active senones in a batch of frames. We provide a comprehensive study of different lossy and lossless compression schemes and an analysis of GMM parameters. We propose a novel clustering scheme which provides significantly higher $Compression/WER$ ratio in comparison with traditional schemes. Clustering GMM parameters results in a huge amount of redundant computations. We use this property to propose and implement a memoization scheme in our accelerator to further reduce the energy consumption by eliminating the redundant floating-point computations which is explained in Chapter 5. This work has been published in the 26th International Conference on Parallel Architectures and Compilation Techniques (PACT):

- Hamid Tabani, Jose-Maria Arnau, Jordi Tubella, Antonio González, " An Ultra Low-power Hardware Accelerator for Acoustic Scoring in Speech Recognition," Parallel Architecture and Compilation Techniques **(PACT)**, *26th International Conference on*, Sep. 2017, Portland, USA.

6.3 Open-Research Areas

One extension of the work proposed in this thesis would be to propose more sophisticated prediction schemes to predict the active senones in larger batch sizes. We show that the accuracy of our scheme dramatically reduces when we consider batch sizes of 32, 64, etc. Therefore, according to our results, it is only beneficial to do lazy GMM evaluation for small batch sizes. Although our scheme avoids fetching and evaluating inactive GMMs, however, a prediction scheme with higher accuracy for larger batch sizes can result in more performance and energy improvements.

GMM is a machine learning algorithm widely used in other areas. It is worth exploiting our prediction scheme in these other areas as lazy evaluation is a general technique and can avoid evaluating inactive GMMs. However, depending on different areas that GMMs are used, predicting the inactive Gaussians may not be the same. On the other hand, for speech, our prediction scheme can be used to avoid computing acoustic score for inactive senones/tokens in hybrid ASR systems based on DNNs or RNNs. Although the DNNs or RNNs are structurally different from GMMs, computing inactive senones in those systems can be eliminated.

Our methodology and the experiments to evaluate our register renaming scheme presented in Chapter 4 are only for single-threaded workloads. As usually more registers are needed for running multi-threaded workloads, we expect to see an increasing pressure on the register file while running these workloads. Therefore, our scheme can become more beneficial for these workloads. For instance, we expect the need of a restructured register type predictor in order to distinguish between the registers of different threads.

Another interesting extension to our proposed register renaming scheme would be to use the history of branch predictor to more accurately predict the type of the registers and improve the accuracy of the register predictor. Nowadays, processors employ sophisticated branch predictors which are quite accurate. Therefore, accurately predicting the correct program flow can improve the register allocation accuracy.

As discussed in section 1.3.5, several researches proposed that GMMs can be used in combination with DNNs to improve accuracy of the ASR system. There are various hardware accelerators proposed for neural networks. A future work to the proposed accelerator of this thesis could be a GMM+DNN accelerator in a single chip to perform both GMM and DNN evaluation. This design not only covers more variety of applications, but also it can be used to jointly calculate the acoustic scores using both GMM and DNN scores.

Bibliography

[1] An optimized BLAS library (OpenBLAS). http://www.openblas.net/.

[2] ARM NEON. http://www.arm.com/products/processors/technologies/neon.php.

[3] cuBLAS. http://docs.nvidia.com/cuda/cublas/.

[4] Google Now. https://en.wikipedia.org/wiki/Google_Now.

[5] Intel Atom Processor (Bay Trail). http://ark.intel.com/products/84311/Intel-Atom-Processor-E3805-1M-Cache-1_33-GHz.

[6] Kaldi ASR results. https://github.com/kaldi-asr/kaldi/tree/master/egs.

[7] NVIDIA Tegra X1. http://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf.

[8] The Samsung Galaxy S7 Review. http://www.anandtech.com/show/10120/the-samsung-galaxy-s7-review.

[9] Haitham Akkary, Ravi Rajwar, and Srikanth T Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 423–434. IEEE, 2003.

[10] X. Anguera, S. Bozonnet, N. Evans, C. Fredouille, G. Friedland, and O. Vinyals. Speaker diarization: A review of recent research. *IEEE Transactions on Audio, Speech, and Language Processing*, 20(2):356–370, Feb 2012.

[11] https://en.wikipedia.org/wiki/Siri.

[12] Angelos Arelakis, Fredrik Dahlgren, and Per Stenstrom. Hycomp: a hybrid cache compression method for selection of data-type-specific compression methods. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 38–49. ACM, 2015.

[13] L. Bahl, R. Bakis, P. Cohen, A. Cole, F. Jelinek, B. Lewis, and R.L. Mercer. Further results on the recognition of a continuously read natural corpus. In *ICASSP'80.*, volume 5, pages 872–875, Apr 1980.

[14] L Bahl, Raimo Bakis, P Cohen, A Cole, Frederick Jelinek, B Lewis, and R Mercer. Further results on the recognition of a continuously read natural corpus. In *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP'80.*, volume 5, pages 872–875. IEEE, 1980.

[15] James Baker. The dragon system–an overview. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 23(1):24–29, 1975.

[16] Leonard E Baum. An inequality and associated maximization thechnique in statistical estimation for probabilistic functions of markov process. *Inequalities*, 3:1–8, 1972.

[17] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[18] Enrico Bocchieri. Vector quantization for the efficient computation of continuous density likelihoods. In *Acoustics, Speech, and Signal Processing, 1993. ICASSP-93., 1993 IEEE International Conference on*, volume 2, pages 692–695. IEEE, 1993.

[19] Jun Cai, Ghazi Bouselmi, Yves Laprie, and Jean-Paul Haton. Efficient likelihood evaluation and dynamic gaussian selection for hmm-based speech recognition. *Computer Speech & Language*, 23(2):147–164, 2009.

[20] Patrick Cardinal, Pierre Dumouchel, and Gilles Boulianne. Large vocabulary speech recognition on parallel architectures. *Audio, Speech, and Language Processing, IEEE Transactions on*, 21(11):2290–2300, 2013.

[21] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2014.

[22] Dhruba Chandra, Ullas Pazhayaveetil, and Paul D Franzon. Architecture for low power large vocabulary speech recognition. In *2006 IEEE International SOC Conference*, pages 25–28. IEEE, 2006.

[23] Tao Chen, Jiawei Zheng, Xingsi Zhang, Shengchang Cai, and Yun Chen. A hardware accelerator for speech recognition applications. In *ASIC (ASICON), 2011 IEEE 9th International Conference on*, pages 760–763. IEEE, 2011.

[24] Prakash Chockalingam. *Non-rigid multi-modal object tracking using Gaussian mixture models*. PhD thesis, Clemson University, 2009.

[25] Anthony Chun, Jenny X Chang, Zhen Fang, Ravishankar Iyer, and Michael Deisher. Isis: An accelerator for sphinx speech recognition. In *Application Specific Processors (SASP), 2011 IEEE 9th Symposium on*, pages 58–61. IEEE, 2011.

[26] J-L Cruz, Antonio González, Mateo Valero, and Nigel P Topham. Multiple-banked register file architectures. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pages 316–325. IEEE, 2000.

[27] David Huggins Daines. *An Architecture for Scalable, Universal Speech Recognition*. PhD thesis, Carnegie Mellon University, 2011.

[28] Vassilios V Digalakis, Leonardo G Neumeyer, and Manolis Perakakis. Quantization of cepstral parameters for speech recognition over the world wide web. *IEEE Journal on selected areas in communications*, 17(1):82–90, 1999.

[29] Paul R Dixon, Tasuku Oonishi, and Sadaoki Furui. Harnessing graphics processors for the fast computation of acoustic likelihoods in speech recognition. *Computer Speech & Language*, 23(4):510–526, 2009.

[30] Jacques Duchateau, Kris Demuynck, and Dirk Van Compernolle. Fast and accurate acoustic modelling with semi-continuous hmms. *Speech Communication*, 24(1):5–17, 1998.

[31] Daniel PW Ellis, Rita Singh, and Sunil Sivadas. Tandem acoustic modeling in large-vocabulary recognition. In *Acoustics, Speech, and Signal Processing, 2001. Proceedings.(ICASSP'01). 2001 IEEE International Conference on*, volume 1, pages 517–520. IEEE, 2001.

[32] Oguz Ergin, Deniz Balkan, Dmitry Ponomarev, and Kanad Ghose. Increasing processor performance through early register release. In *Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. IEEE International Conference on*, pages 480–487. IEEE, 2004.

[33] Keith I Farkas, Norman P Jouppi, and Paul Chow. Register file design considerations in dynamically scheduled processors. In *High-Performance Computer Architecture, 1996. Proceedings., Second International Symposium on*, pages 40–51. IEEE, 1996.

[34] Rahman Farnoosh and Behnam Zarpak. Image segmentation using gaussian mixture model. *IUST International Journal of Engineering Science*, 19(1-2):29–32, 2008.

[35] C. Gaida, P. Lange, R. Petrick, P. Proba, A. Malatawy, and D. Suendermann-Oeft. Comparing open-source speech recognition toolkits. In *Technical report, Project OASIS DHBW Stuttgart, Stuttgart, Germany*, 2014.

[36] Antonio Gonzalez, Jose Gonzalez, and Mateo Valero. Virtual-physical registers. In *High-Performance Computer Architecture, 1998. Proceedings., 1998 Fourth International Symposium on*, pages 175–184. IEEE, 1998.

[37] Antonio Gonzalez, Fernando Latorre, and Grigorios Magklis. Processor microarchitecture: An implementation perspective. *Synthesis Lectures on Computer Architecture*, 5(1):1–116, 2010.

[38] Nicola Greggio, Alexandre Bernardino, Cecilia Laschi, Paolo Dario, and José Santos-Victor. Fast estimation of gaussian mixture models for image segmentation. *Machine Vision and Applications*, 23(4):773–789, 2012.

[39] Kshitij Gupta and John D Owens. Three-layer optimizations for fast gmm computations on gpu-like parallel processors. In *ASRU'2009*, pages 146–151. IEEE, 2009.

[40] Kshitij Gupta and John D Owens. Compute & memory optimizations for high-quality speech recognition on low-end gpu processors. In *HiPC'2011*, pages 1–10. IEEE, 2011.

[41] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[42] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[43] R. Hsiao, M. Fuhs, Q. J. Y. Tam, I. Lane, and T. Schultz. Handbook of natural language processing and machine translation. page 496–504, 2011.

[44] Xuedong Huang, Fileno Alleva, Hsiao-Wuen Hon, Mei-Yuh Hwang, Kai-Fu Lee, and Ronald Rosenfeld. The sphinx-ii speech recognition system: an overview. *Computer Speech & Language*, 7(2):137–148, 1993.

[45] Xuedong Huang, K-F Lee, and H-W Hon. On semi-continuous hidden markov modeling. In *Acoustics, Speech, and Signal Processing, 1990. ICASSP-90., 1990 International Conference on*, pages 689–692. IEEE, 1990.

[46] Xuedong D Huang and Mervyn A Jack. Semi-continuous hidden markov models for speech signals. *Computer Speech & Language*, 3(3):239–251, 1989.

[47] D. Huggins-Daines, M. Kumar, A. Chan, A.W. Black, M. Ravishankar, and A.I. Rudnicky. Pocketsphinx: A free, real-time continuous speech recognition system for hand-held devices. In *ICASSP*, May 2006.

[48] David Huggins-Daines, Mohit Kumar, Arthur Chan, Alan W Black, Mosur Ravishankar, and Alexander I Rudnicky. Pocketsphinx: A free, real-time continuous speech recognition system for hand-held devices. In *2006 IEEE International Conference on Acoustics Speech and Signal Processing Proceedings*, volume 1, pages I–I. IEEE, 2006.

[49] Ravi Iyer, Sadagopan Srinivasan, Omesh Tickoo, Zhen Fang, Ramesh Illikkal, Steven Zhang, Vineet Chadha, Paul Stillwell, and Seung Eun Lee. Cogniserve: Heterogeneous server architecture for large-scale recognition. *IEEE Micro, 31(3)*, 2011.

[50] Frederick Jelinek. Continuous speech recognition by statistical methods. *Proceedings of the IEEE*, 64(4):532–556, 1976.

[51] Timothy M Jones, MFR O'Boyle, Jaume Abella, Antonio Gonzalez, and Oguz Ergin. Compiler directed early register release. In *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, pages 110–119. IEEE, 2005.

[52] S. Kanthak, K. Schutz, and H. Ney. Using simd instructions for fast likelihood calculation in lvcsr. In *ICASSP*, 2000.

[53] A. Lee, T. Kawahara, K. Takeda, and K. Shikano. A new phonetic tied-mixture model for efficient decoding. In *ICASSP*, 2000.

[54] Akinobu Lee and Tatsuya Kawahara. Recent development of open-source speech recognition engine julius. In *Proceedings of APSIPA ASC 2009*, pages 131–137, 2009.

[55] Akinobu Lee, Tatsuya Kawahara, Kazuya Takeda, and Kiyohiro Shikano. A new phonetic tied-mixture model for efficient decoding. In *Acoustics, Speech, and Signal Processing, 2000.*

ICASSP'00. Proceedings. 2000 IEEE International Conference on, volume 3, pages 1269–1272. IEEE, 2000.

[56] Chunho Lee, Miodrag Potkonjak, and William H Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335. IEEE Computer Society, 1997.

[57] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO-42*, pages 469–480. IEEE, 2009.

[58] Chris Lomont. Introduction to intel advanced vector extensions. *Intel White Paper*, 2011.

[59] Bruce Lowerre. The harpy speech understanding system. In *Readings in speech recognition*, pages 576–586. Morgan Kaufmann Publishers Inc., 1990.

[60] Binu Mathew, Al Davis, and Zhen Fang. A low-power accelerator for the sphinx 3 speech recognition system. In *CASES'03*, 2003.

[61] Xinxin Mei and Xiaowen Chu. Dissecting gpu memory hierarchy through microbenchmarking. 2015.

[62] Merlin Friesen. Linux Power Management Optimization on the Nvidia Jetson Platform. http://events.linuxfoundation.org/sites/events/files/slides/Linux_Low_Power_ELC_SanDiego.pdf.

[63] Yajie Miao, Mohammad Gowayyed, and Florian Metze. Eesen: End-to-end speech recognition using deep rnn models and wfst-based decoding. In *2015 IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*, pages 167–174. IEEE, 2015.

[64] https://en.wikipedia.org/wiki/Cortana_(software).

[65] Teresa Monreal, Antonio González, Mateo Valero, José González, and Víctor Viñals. Dynamic register renaming through virtual-physical registers. *Journal of Instruction Level Parallelism*, 2000.

[66] Teresa Monreal, Víctor Viñals, Antonio González, and Mateo Valero. Hardware schemes for early register release. In *Parallel Processing, 2002. Proceedings. International Conference on*, pages 5–13. IEEE, 2002.

[67] Ghazaleh Moradiannejad. *People Tracking Under Occlusion Using Gaussian Mixture Model and Fast Level Set Energy Minimization.* PhD thesis, Citeseer, 2013.

[68] Mayan Moudgill, Keshav Pingali, and Stamatis Vassiliadis. Register renaming and dynamic speculation: an alternative approach. In *Proceedings of the 26th annual international symposium on Microarchitecture*, pages 202–213. IEEE Computer Society Press, 1993.

[69] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. Papi: A portable interface to hardware performance counters. In *Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[70] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to model large caches. *HP Laboratories*, pages 22–31, 2009.

[71] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to model large caches. *HP Laboratories*, pages 22–31, 2009.

[72] U. Nallasamy, I. Lane, M. Fuhs, M. Noamany, Y. Tam, Q. Jin, and T. Schultz. Handbook of natural language processing and machine translation. page 535–540, 2011.

[73] Stefan Ortmanns, Hermann Ney, and Xavier Aubert. A word graph algorithm for large vocabulary continuous speech recognition. *Computer Speech & Language*, 11(1):43–72, 1997.

[74] Jianlin Ou, Jun Cai, and Qian Lin. Using simd technology to speed up likelihood computation in hmm-based speech recognition systems. In *ICALIP*, pages 123–127, 2008.

[75] V. Panayotov, G. Chen, D. Povey, and S. Khudanpur. Librispeech: An asr corpus based on public domain audio books. In *ICASSP*, 2015.

[76] Bryan L Pellom, Ruhi Sarikaya, and John HL Hansen. Fast likelihood computation techniques in nearest-neighbor based search for continuous speech recognition. *Signal Processing Letters, IEEE*, 8(8):221–224, 2001.

[77] Daniel Povey, Lukáš Burget, Mohit Agarwal, Pinar Akyazi, Kai Feng, Arnab Ghoshal, Nagendra Kumar Goel, Martin Karafiát, Ariya Rastrow, Richard C Rose, et al. Subspace gaussian mixture models for speech recognition. In *2010 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 4330–4333. IEEE, 2010.

[78] Daniel Povey, Lukáš Burget, Mohit Agarwal, Pinar Akyazi, Feng Kai, Arnab Ghoshal, Ondřej Glembek, Nagendra Goel, Martin Karafiát, Ariya Rastrow, et al. The subspace gaussian mixture model—a structured model for speech recognition. *Computer Speech & Language*, 25(2):404–439, 2011.

[79] Daniel Povey, Arnab Ghoshal, Gilles Boulianne, Lukas Burget, Ondrej Glembek, Nagendra Goel, Mirko Hannemann, Petr Motlicek, Yanmin Qian, Petr Schwarz, et al. The kaldi speech recognition toolkit. In *IEEE 2011 workshop on automatic speech recognition and understanding*, number EPFL-CONF-192584. IEEE Signal Processing Society, 2011.

[80] Eduardo Quinones, Joan-Manuel Parcerisa, and Antonio Gonzalez. Early register release for out-of-order processors with register windows. In *Parallel Architecture and Compilation Techniques, 2007. PACT 2007. 16th International Conference on*, pages 225–234. IEEE, 2007.

[81] L. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, Feb 1989.

[82] Lawrence R Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[83] Shakti P Rath, Kate M Knill, Anton Ragni, and Mark JF Gales. Combining tandem and hybrid systems for improved speech recognition and keyword spotting on low resource languages. In *INTERSPEECH*, pages 835–839, 2014.

[84] Mosur Ravishankar, Roberto Bisiani, and Eric H Thayer. Sub-vector clustering to improve memory and speed performance of acoustic likelihood computation. In *EUROSPEECH*, 1997.

[85] Mosur K Ravishankar. *Efficient Algorithms for Speech Recognition.* PhD thesis, 1996.

[86] Zs Robotka and A Zempléni. Image retrieval using gaussian mixture models. *Annals Univ. Sci. Budapest, Sect. Comp*, 31:93–105, 2009.

[87] Ananth Sankar. A new look at hmm parameter tying for large vocabulary speech recognition. In *ICSLP*, 1998.

[88] Johan Schalkwyk, Doug Beeferman, Françoise Beaufays, Bill Byrne, Ciprian Chelba, Mike Cohen, Maryam Kamvar, and Brian Strope. *"Your Word is my Command": Google Search by Voice: A Case Study*, pages 61–90. Springer US, Boston, MA, 2010.

[89] Richard Schwartz, Long Nguyen, and John Makhoul. Multiple-pass search strategies. In *Automatic Speech and Speaker Recognition*, pages 429–456. Springer, 1996.

[90] David Sheffield, Michael J Anderson, Yunsup Lee, and Kurt Keutzer. Hardware/software codesign for mobile speech recognition. In *INTERSPEECH*, pages 627–631, 2013.

[91] https://www.skype.com/en/features/skype-translator/.

[92] Gurindar S Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE transactions on computers*, 39(3):349–359, 1990.

[93] Eric Sprangle and Doug Carmean. Increasing processor performance by implementing deeper pipelines. In *ACM SIGARCH Computer Architecture News*, volume 30, pages 25–34. IEEE Computer Society, 2002.

[94] Pawel Swietojanski, Arnab Ghoshal, and Steve Renals. Revisiting hybrid and gmm-hmm system combination techniques. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 6744–6748. IEEE, 2013.

[95] Hamid Tabani, Jose-Maria Arnau, Jordi Tubella, and Antonio González. Performance analysis and optimization of automatic speech recognition. *IEEE Transactions on Multi-Scale Computing Systems*, 2017.

[96] Hamid Tabani, Jose-Maria Arnau, Jordi Tubella, and Antonio Gonzalez. An ultra low-power hardware accelerator for acoustic scoring in speech recognition. In *Parallel Architecture and Compilation Techniques (PACT), 26th International Conference on*. IEEE/ACM, 2017.

[97] Hamid Tabani, Jose-Maria Arnau, Jordi Tubella, and Antonio Gonzalez. A novel register renaming technique for out-of-order processors. In *High Performance Computer Architecture (HPCA), 24th International Conference on*. IEEE/ACM, 2018.

[98] Yuuki Tachioka and Tomohiro Narita. Optimal automatic speech recognition system selection for noisy environments. In *Signal and Information Processing Association Annual Summit and Conference (APSIPA), 2016 Asia-Pacific*, pages 1–8. IEEE, 2016.

[99] Zheng-Hua Tan and Børge Lindberg. *Automatic speech recognition on mobile devices and over communication networks*. Springer Science & Business Media, 2008.

[100] Andrew Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE transactions on Information Theory*, 13(2):260–269, 1967.

[101] David W Wall. *Limits of instruction-level parallelism*, volume 19. ACM, 1991.

[102] V.M. Weaver, M. Johnson, K. Kasichayanula, J. Ralph, P. Luszczek, D. Terpstra, and S. Moore. Measuring energy and power with papi. In *ICPPW*, pages 262–268, 2012.

[103] Reza Yazdani, Jose-Maria Arnau, and Antonio González. Unfold: a memory-efficient speech recognizer using on-the-fly wfst composition. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 69–81. ACM, 2017.

[104] Reza Yazdani, Albert Segura, Jose-Maria Arnau, and Antonio Gonzalez. An ultra low-power hardware accelerator for automatic speech recognition. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–12. IEEE, 2016.

[105] Reza Yazdani, Albert Segura, Jose-Maria Arnau, and Antonio Gonzalez. Low-power automatic speech recognition through a mobile gpu and a viterbi accelerator. *IEEE Micro*, 37(1):22–29, 2017.

[106] S Young, G Evermann, M Gales, T Hain, D Kershaw, X Liu, G Moore, J Odell, D Ollason, D Povey, et al. The htk book (v3. 4). *Cambridge University*, 2006.

[107] Dong Yu and Li Deng. *Automatic speech recognition: A deep learning approach*. Springer, 2014.

[108] Dong Yu and Michael L Seltzer. Improved bottleneck features using pretrained deep neural networks. In *Interspeech*, volume 237, page 240, 2011.

[109] Kai Yu and Rob A Rutenbar. Acoustic scoring unit implemented on a single fpga or asic, January 28 2014. US Patent 8,639,510.