Parallelizing the GSM Speech Compression Algorithm on Raw

by

Vishal Kapur

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Electrical [Computer] Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

May 24, 2002

Copyright 2002 Vishal Kapur. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and
gen and distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.

Author

Department of Electrical Engineering and Computer Science

May 24, 2002

Certified by

Prof. Anant Agarwal
Thesis Supervisor

Accepted by

Arthur C. Smith
Chairman, Department Committee on Graduate Theses
Parallelizing the GSM Speech Compression Algorithm on Raw

by

Vishal Kapur

Submitted to the
Department of Electrical Engineering and Computer Science

May 24, 2002

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Electrical [Computer] Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

The GSM speech encoding algorithm, a widely used compression algorithm in mobile communications, was parallelized on Raw, a highly distributed hardware architecture with low latency communication. Parallelization architectures designed included a 2-stage pipeline and a 5-node satellite-hub model for distributing the time-intensive correlation calculations. Two additional architectures integrated these techniques: a 2-stage pipeline with a 4-node correlation parallelization, and an expanded 3-stage pipeline with a 9-node correlation parallelization. The expanded design took advantage of efficient switch-level data routing in the Raw network in order to distribute processing to non-local nodes. Results for the 3-stage, 9-node parallelization showed a factor of 3.6 speedup gain. A factor of 3.8 gain was achieved after optimizing data transfer. Future work includes extending the implementation to the entire GSM protocol, and utilizing available Raw resources to enhance the GSM speech compression algorithm with minimal performance loss.

Thesis Supervisors: Prof. Anant Agarwal and Dr. Volker Strumpen
Acknowledgements

First I would like to thank Dr. Volker Strumpen, who spent a great deal of time with me to guide, discuss, and teach. I would also like to thank Prof. Anant Agarwal for his support of the project. Also, thanks to Jutta Degener for her assistance in deepening my understanding of the algorithm, as well as suggesting paths for exploration. In addition, I am grateful for the financial support from the Electrical Engineering department.

Finally, thanks to my family and friends for helping me keep my perspective throughout the last five years.
Contents

1 Introduction 5

2 Goals 7

3 Background 8
  3.1 GSM Speech Compression Algorithm 8
    3.1.1 GSM Mobile Communications Standard 8
    3.1.2 GSM Speech Encoder Overview 10
    3.1.3 Linear Predictive Coding (LPC) 11
    3.1.4 Long-Term Prediction (LTP) 15
    3.1.5 Residual Pulse Encoding (RPE) 17
    3.1.6 Summary of the GSM Encoder 17
  3.2 Raw Microprocessor Architecture 18

4 Designs 22
  4.1 Pipeline Design 22
    4.1.1 Design Motivation 23
    4.1.2 Design Choices 23
    4.1.3 Expected Gains 26
    4.1.4 Limiting Factors 26
  4.2 Parallelization of Long-Term Parameter Computations 26
    4.2.1 Design Motivation 27
    4.2.2 Design Choices 29
    4.2.3 Expected Gains 30
  4.3 Integrated Designs 31
    4.3.1 Design Motivation 31
    4.3.2 Expected Gains 33
    4.3.3 Expanding the Integrated Design 33
  4.4 Data Transfer Optimizations 36
    4.4.1 Latency Hiding 36
    4.4.2 Loop Unrolling 37
  4.5 Opportunities for Parallelization in the Decoder 38

5 Results 39
  5.1 Notes on Implementation 39
  5.2 2-stage Pipeline 40
  5.3 Parallelization of Long-Term Parameter Calculations 41
  5.4 Integrated Designs 42
    5.4.1 2-stage Pipeline, 4-node Parallelization 42
    5.4.2 3-stage Pipeline, 9-node Parallelization 43
    5.4.3 3-stage, 9-node with Optimized Data Transfer 45

6 Conclusions 46
1 Introduction

Data compression has become an integral part of many digital systems, particularly those which utilize a network to transmit data. With bandwidth and transfer rates limited in such systems, compression of data is key for wide-scale implementation and scalability. Typical examples of such bandwidth-limited systems are video transmission over the internet and speech transmission over a wireless network.

The challenge with data compression is to balance the computational complexity of the algorithm with the compression ratio. More computationally demanding algorithms can often provide high quality speech at a high compression rate, but at the same time induce larger computation delays [Garg99]. The converse principle usually holds as well—i.e., less demanding algorithms have the advantage of lower delays, but sacrifice either quality of data, bandwidth, or both. In terms of a communication system, this delay might manifest itself as a noticeable lag or interruption during communication. Furthermore, a complex algorithm can result in higher power consumption; for handheld devices such as mobile cell phones this is clearly undesirable. As a result, an important constraint in developing a compression algorithm is its computational complexity. The ideal compression algorithm would be able to provide a high compression rate, minimize loss in quality, and minimize computational demands.

This study exploited the massive parallelism and low communication latency of the Raw microprocessor architecture in order to parallelize the GSM speech compression algorithm. The Raw architecture features a mesh of independent processing
units that are able to communicate through a low latency network. To allow applications to take advantage of this architecture, the components of the hardware are exposed to software compilers, allowing for the flexibility of developing applications in software while rivaling the speed of custom hardware implementations. In addition, by providing for a high I/O bandwidth the architecture is particularly suited for multimedia streaming applications such as speech compression.

The GSM speech compression algorithm was parallelized on the Raw architecture by exploiting the availability of two forms of parallelism:

- **Pipeline Parallelism.** The GSM algorithm processes speech data in discrete segments called frames. We can exploit pipeline parallelism here since these frames, for the most part, are processed independently of one another.

- **Process Parallelism.** A major component of the algorithm are time-intensive correlation calculations. These calculations might be viewed as being “process-parallel.” In other words, we distribute identical copies of the data to multiple independent processing streams, each of which then calculates correlations for a unique range of the data.

Implementation on a Raw simulator suggests that the integration of these two approaches leads to significant speedup, up to a factor of 3.8, in the compute time of the GSM encoding algorithm. Crucial elements in achieving this performance gain were the low latency communication between Raw processing nodes, and the efficient routing in the Raw network at the switch level. The results of this study address the challenge of achieving high quality compression with reasonable computational load.
Future work includes extending the current implementation to a hardware-based system, implementing the full GSM protocol in real-time on a Raw-based system, and improving on the speech quality offered by GSM by taking advantage of readily available Raw processing resources.

The rest of this paper is organized as follows. Section 2 summarizes the goals of this study. Next, Section 3 provides a detailed mathematical analysis of the GSM encoding algorithm as well as further exploration into the Raw architecture. Section 4 then details the designs developed and implemented in this study for parallelizing the algorithm, and estimates the expected gain from each one. Finally, Section 5 presents the results for each of these designs, and Section 6 summarizes these findings and suggests future paths.

## 2 Goals

The goal of this study is to design architectures to exploit the opportunities for parallelism in the GSM speech compression algorithm, and to empirically evaluate and compare these architectures through implementation on the RAW architecture.

The central thesis is that the major opportunities for parallelism are in integrating the approaches of pipelining the algorithm and parallelizing the correlation calculations. Furthermore, effective use of low latency communication in Raw allows for an efficient, widely distributed parallelization with relatively low transfer overhead.
3 Background

The parallelized compression system developed in this study is composed of two components: one, the speech compression algorithm part of the GSM mobile communications standard; two, the Raw microprocessor architecture, which is utilized for a parallelized implementation of the algorithm.

3.1 GSM Speech Compression Algorithm

The GSM speech compression algorithm is part of the GSM mobile communications standard, an expanding and comprehensive standard for voice and data wireless communication. The algorithm is composed of three main modules: a short-term analysis (linear predictive coding), a long-term prediction algorithm, and an encoding algorithm (residual pulse encoding).

3.1.1 GSM Mobile Communications Standard

The Global System for Mobile communications (GSM) is a digital wireless communication standard for mobile phone networks [Garg99]. Although originally prevalent in Europe, today the GSM standard has expanded to 174 countries around the world, now reaching one in every ten people in the world [gsmworld.com]. GSM not only addresses wireless voice communication, but also provides data services such as text messaging and data transfer. An exciting standard currently in development is the 3GSM standard, which attempts to provide for high-bandwidth wireless services such as video-on-demand and high-speed internet access.
The GSM network consists of several major subsystems: the mobile station (i.e., the cell phone), the base station subsystem (radio link protocols), and the network subsystem (switching, call-routing, security, etc.) (see Figure 1). A crucial part of this system (and, in fact, any mobile communication system) is the speech compression that takes place in the cell phone before wireless transfer, and the decompression that occurs on receipt of the data at the base station. Speech compression (or speech "coding") is crucial for mobile communication systems since we would like to support as many users as possible given limited bandwidth resources (i.e., narrowband communication channels). At the same time we require that the transmitted speech is intelligible and relatively free of compression artifacts. Figure 1 illustrates how the GSM speech compression algorithm (both encoding and decoding) fits into the framework of the entire GSM network.
Figure 2: GSM Speech Encoder: Data Flow and Output Decomposition

### 3.1.2 GSM Speech Encoder Overview

The focus of this study is the GSM speech encoder, which is a part of the GSM full-rate speech compression algorithm, GSM 06.10 LPC-RPE. The GSM standard supports several speech codecs (i.e., coder/decoder algorithms), including full-rate, half-rate, and enhanced full-rate codecs [Garg99]. This study focuses on the full-rate codec due to its widespread availability [Degener94].

Figure 2 diagrams the full-rate speech encoder. The GSM full-rate codec achieves a 1:8 compression ratio. At its input, the algorithm takes speech input sampled at
8000 samples/sec, with each value digitized using 13 bits (this corresponds to a 104 kb/s input rate). The algorithm processes 20ms frames of the speech input at a time; each frame corresponds to 160 data samples. At its output, the speech codec produces 260 bits for every 20ms frame (13 kb/s).

GSM 06.10 utilizes a compression algorithm known as Linear Predictive Coder with Residual Pulse Encoding (LPC-RPE) [Eberspacher99]. This algorithm is a type of “hybrid” coder that integrates the approaches of linear predictive models and long-term prediction techniques in order to reduce the speech signal to a state in which it can be reliably encoded by a residual pulse encoding technique.

The followings sections will detail the algorithm modules for the GSM encoder, as shown in Figure 2:

- Linear predictive models (LPC) of speech are based on a physiologically plausible model of the human vocal tract, and are focused on short-term analysis of the speech signal.

- Long-term prediction (LTP) techniques exploit redundancy in the data that encompasses a wider range of the signal than the vocoder approach.

- Residual pulse excitation (RPE) is a technique to encode and compress a low-energy signal.

3.1.3 Linear Predictive Coding (LPC)

Linear predictive coding is based on a physiological model of human speech generation. Air passing through the vocal cords causes two main classes of vibrations: *voiced*
and *unvoiced* sounds. Voiced sounds are generated when the vocal cords vibrate periodically at a consistent pitch. On the other hand, unvoiced sounds result when the vocal cords do not vibrate, instead letting through an aperiodic, noisy signal. The vibrations of the vocal cords, whether voiced, unvoiced, or a mix of both types, then pass through the vocal tract; the result is speech. Furthermore, the shape of the vocal tract changes frequently (approximately every 10 msec to 100 msec - GSM uses 20 ms frames) throughout speech, in order to generate a wide variety of sounds [Phamdo00].

LPC models human speech production as the result of an excitation signal being passed through a time-varying digital filter (see Figure 3). In this model, the excitation signal represents the vocal cord vibrations, and the time-varying filter corresponds to the vocal tract.

The goal of LPC modeling is to determine the coefficients of the LPC filter and the corresponding excitation signal that estimates a segment of speech. In essence,
given some digitized speech signal as input (e.g., from a cell phone's microphone), we
traverse the filtering diagram in Figure 3 backwards. At the core of this approach are
the linear prediction techniques from digital signal processing (DSP) theory; these
techniques are used to approximate the coefficients for the LPC filter. We can then
approximate the excitation signal (also called the residual signal) by inverse filtering
the speech signal through the estimated LPC filter.

The advantage of performing this computation is that the excitation signal is
usually either periodic or random, implying that it requires less information to encode
(with minimal loss) than the full speech signal, and therefore presents an opportunity
for compression. As a result, instead of transmitting the entire signal, we transmit
only the encoded version of the residual signal and the coefficients for the LPC filter.
A more detailed mathematical analysis of the LPC approach follows.

Calculation of the LPC Filter Coefficients. The general problem of deter-
mining filter coefficients (specifically, for an all-pole filter) from output data is called
linear prediction, or linear predictive coding (LPC); extensive results from DSP theory
exist for this problem. The expression for the LPC filter is given by:

$$ H(z) = \frac{1}{1 - (a_1 z^{-1} + a_2 z^{-2} + \ldots + a_K z^{-K})} \quad (1) $$

The z-transform of the speech signal is then the excitation signal $E(z)$ filtered
through $H(z)$:
\[ S(z) = E(z) \cdot H(z) = \frac{E(z)}{1 - \sum_{i=1}^{K} a_i z^{-i}} \]  \hspace{1cm} (2)

Expressed as a difference equation, the filter takes the form:

\[ e[n] = s[n] - \sum_{i=1}^{K} a_i s[n - i] \equiv s[n] - \hat{s}[n] \]  \hspace{1cm} (3)

In the case of GSM, a filter length of \( K = 8 \) is used. This implies that the current sample of the signal will be estimated based on the last 8 samples.

From Equation 3, it is evident that the LPC filter (defined by its coefficients \( a_i \)), predicts the current value of the speech signal, \( s[n] \), using a linear combination of the previous \( K \) values to construct an estimate \( \hat{s}[n] \equiv \sum_{i=1}^{8} a_i s[n - i] \). The excitation signal \( e[n] \), then, can also be thought of as the error between the actual and predicted values [Huerta00].

We then define the mean squared error as:

\[ E = \sum_{n} e^2[n] = \sum_{n} (s[n] - \sum_{i=1}^{8} a_i s[n - i])^2. \]  \hspace{1cm} (4)

We would like to minimize the mean-squared error, \( E \), in order to determine the optimal set of filter coefficients \( a_i \). We can do this by taking the partial derivative of \( E \) with respect to each \( a_i \). This results in a set of 8 linear equations, which can be solved in a number of ways. An efficient method, which is used by GSM, is called the Levinson-Durbin recursion [Huerta00]. Having done this recursion, we now have the optimal set of filter coefficients \( a_i \) for the current frame.
Calculation of the Short-Term Residual. The inverse filtering operation is then straightforward. Equation 3 gives the expression for the excitation signal $e[n]$, also called the short-term residual (see Figure 2). Note that we now have all necessary data to complete this computation: the speech data $s[n]$ is given, and $\hat{s}[n]$ can now be computed. Given the short-term residual, we can proceed to the long-term analysis.

3.1.4 Long-Term Prediction (LTP)

LTP identifies long-term redundancy in the speech data, in order to enhance the quality of LPC-encoded speech. While LPC provides a high rate of compression, the resulting quality of speech can be unsatisfactory. This occurs because the residual signal produced from the LPC algorithm is not as simple as the voiced and unvoiced signals shown in Figure 3. As a result, compressing the residual data causes a noticeable loss in speech quality.

To address this issue, techniques have been developed to identify long-term predictive properties of the signal. Although the physiological basis is not as well-defined as the LPC model, such long-term correlation may arise from the quasi-periodicity of the voiced excitation signals [Huerta00]. Long-term analysis can be viewed as a second filtering operation, cascaded with the LPC filter of Figure 3; the resulting system is shown in Figure 4. In the frequency domain, we can consider the long-term analysis as modeling the finer details of the speech signal's spectrum; in contrast, the short-term analysis models a coarser envelope of the spectrum [Huerta00]. Finally, we note that the cost of a higher quality output is the additional bits required for
transmitting the LTP parameters.

**Calculation of the Long-Term Parameters.** GSM uses a long-term prediction (LTP) technique that analyzes four 5ms subsections of each frame. For each subsection, it identifies lag and gain parameters that give the maximum correlation with adjacent 5ms subsections. Equation 5 describes the algorithm used to compute these parameters:

\[ \beta = \max \left( \sum_{k=0}^{39} s[k] \cdot dp[k - \lambda] \right), \lambda : [40...120] \]  

Here the \( s \) array denotes the current 40-sample block of speech, and the \( dp \) array denotes the previous 120-sample block of speech. The gain, \( \beta \), is determined by the maximum correlation. Also define \( \lambda^* \), the desired lag, as the value of \( \lambda \) where the maximum correlation is achieved. These parameters then determine a filter which can be expressed as:

\[ P(z) = \frac{1}{1 - \beta z^{-\lambda^*}} \]  

Figure 4 illustrates how \( P(z) \) and \( H(z) \), the filter determined by LPC, are integrated.

![Figure 4: LPC Model with LTP Module](image-url)
**Calculation of the Long-Term Residual.** The short-term residual signal (from the short-term analysis) is then inverse filtered through $P(z)$, resulting in the long-term residual (see Figure 2). Since we have removed more information from the residual, we expect it to be relatively simple and low in total energy, preparing it for the next stage, residual pulse encoding.

### 3.1.5 Residual Pulse Encoding (RPE)

We can now encode (i.e., compress) the residual signal without significant loss in quality. This is done by residual pulse encoding (RPE), which essentially subsamples the residual by discarding two out of every three values. In addition, a coarse quantization is applied.

### 3.1.6 Summary of the GSM Encoder

The GSM frame is now complete: we have computed the LPC filter coefficients, the LTP lag/gain, and the encoded residual (see Figure 2). The resulting data for transmission is highly compressed (a 1:8 ratio), and also preserves speech quality. The cost, of course, comes in the form of computational complexity.

It is important to note that this algorithm is a lossy compression algorithm, As a result, decoding a GSM-encoded frame will not exactly reproduce the original speech. The advantage of lossy compression is that it often provides higher compression rates; at the same time, however, one must consider speech quality, and make certain that intelligible speech is produced [Vercoe, Gardner, et al]. In general, the GSM algorithm successfully achieves this goal with minimal speech distortion.
Although the focus of this study is the GSM encoder, the GSM decoder is also an integral part of a GSM network (see Figure 1). The decoding process is essentially the inverse of the encoding process: given the long-term residual and LTP gain/lag, the short-term residual is reconstructed; given the short-term residual and the filter coefficients, the desired speech is then estimated.

3.2 Raw Microprocessor Architecture

![Figure 5: A Raw Processor](adapted from Waingold, 1997)

The Raw microprocessor architecture is utilized in this study in order to implement parallelizations of the GSM speech compression algorithm. The Raw architecture (see Figure 5) utilizes the concept of replicating and connecting simple processing units to produce a flexible, scalable, and highly parallel architecture [Waingold97]. In addition, the hardware is exposed to the software system, so that compilers can take full advantage of the architecture by optimizing resource allocation. Finally, the availability of a high I/O bandwidth to the system makes Raw ideal for studying the
parallelization of a speech compression algorithm.

The architecture is composed of two essential units that are of relevance to the parallelization effort in this study. The first are the processing units, or "tiles." Each tile has a processor and a memory capacity for data and instructions (see Figure 5). Parallelism between tiles is achieved by allowing each tile its own independent instruction stream and memory space [Waingold97].

The second essential component of the Raw architecture is the network interconnect that connects the tiles in a two-dimensional mesh. The goal of this network is to provide high-bandwidth, low latency communication amongst the tiles [Taylor01]. The interconnect is composed of a static network, over which communication is scheduled statically, and a dynamic network; the focus of this study is the static network. The network design also allows for extensive scalability, since local latencies between tiles are kept low and constant.

Each tile also consists of a switch that interfaces the local processor to the static network. Switches can route data from (or to) the local tile onto multiple local directions on the network. They are also able to route data between two non-adjacent points on the network, for efficient non-local communication. Both of these capabilities of the switch were exploited in this study in order to increase the efficiency and the expansiveness of the parallelization.

There are several advantages of a distributed architecture such as Raw. First, since tile interconnects are local, and therefore short, inter-tile communication latencies are small. This implies that applications can be parallelized on Raw with little overhead
for communication startup and synchronization [Taylor01]. This is in contrast to, for example, a multiprocessor system, where communication latencies often limit the effectiveness of parallelized computation. A second advantage of the Raw architecture is that in addition to distributed computation, the memory system is also distributed. With local memories, we expect significantly shorter memory access latencies, as well as higher memory bandwidth across all tiles [Babb99]. One can also envision advantages for Raw in terms of ease of scalability and flexibility.

In addition to its unique hardware design, Raw also provides for a unique software system. To take advantage of the hardware, applications must have access to the low-level components, namely the local memories, processors, and switches. In other words, a compiler for Raw will be able to take a sequential program and parallelize it across the available Raw tiles, scheduling static communication as necessary. The compiler can also cluster computation and associated memory on the same (or nearby) tiles. The exposed architecture therefore allows for application-specific parallelization, which can potentially provide for significant performance gains over sequentially implemented software [Babb97]. Furthermore, this implies that it is feasible to not only have the flexibility and power offered by software, but also rival the speed of custom hardware implementations.

Finally, the Raw architecture is ideally suited for applications that can take advantage of the high data bandwidth available to the I/O system as well as the high computational parallelism. Signal processing applications for video and audio usually fit this framework very well. Streaming multimedia data utilizes the high I/O
bandwidth; in addition, signal processing algorithms are often highly parallelizable. Raw, then, is an efficient and effective platform on which develop a parallelized speech compression algorithm.
4 Designs

There were four parallelization architectures developed in this study:

1. Pipeline (Section 4.1)

2. Locally distributed parallelization of the correlation calculations (Section 4.2)

3. Integration of the first two designs (Section 4.3)

4. Expanded integrated design utilizing both locally and non-locally distributed parallelization (Section 4.3.3)

The following sections detail these designs and analyze their expected performance gains. In addition, Section 4.4 discusses methods for reducing data transfer overhead.

Also note that the parallelization effort in this study was restricted to the GSM encoder; Section 4.5 discusses parallelization opportunities in the GSM decoder.

4.1 Pipeline Design

The pipelined version of the GSM encoding algorithm is shown in Figure 6.
4.1.1 Design Motivation

A pipelined design takes advantage of the independent processing of adjacent frames of the signal. (However, this independence of frame processing does not extend to all levels of the algorithm; see next section for details).

4.1.2 Design Choices

The major design choices in utilizing a pipeline is how to distribute the computation in the algorithm amongst the \( N \) stages, and how to choose \( N \). These choices follow from a mathematical analysis of pipeline gains, which says that gain is maximized when the computation loads of the stages are the same.

\[
\begin{align*}
X_1 & \rightarrow X_2 & \cdots & \rightarrow X_N \\
X_i & : \text{ computational load on stage } i
\end{align*}
\]

Figure 7: Generic Pipeline: \( N \) Stages

A generic pipeline is shown in Figure 7. In general, the gain from utilizing a pipeline is constrained by the maximum computation time among the \( N \) stages:

\[
G_P = \frac{1}{x_M}, x_M \equiv \max(x_1, x_2, \ldots, x_N),
\]

where \( G_P \) is the factor of improvement (i.e., reduction) in computation time, and \( x_M \) is the maximum computation load. From Equation 7, it is clear that in order to maximize the effectiveness of the pipeline, the maximum load amongst the \( N \) stages,
$x_M$, should be minimized. We can see that this is achieved when:

$$x_1 = x_2 = \ldots = x_N = \frac{1}{N} \quad (8)$$

In other words, in order to achieve maximum throughput through the pipeline, and therefore maximum gain, each stage should consume an equal fraction of the compute time. This results in a maximal gain $G_P = N$.

From this mathematical analysis, it is clear that ideally we would like to choose the “breaks” in the algorithm such that the computational loads in the stages are equal (or as close as possible). In addition, we would like to choose the number of stages, $N$, in such a way that exploits the benefits of a longer pipeline, while weighing the cost of additional data transfer overhead.

A statistical profiling of the algorithm was done to determine the distribution of computational load among the various modules. The results are shown in Table 1.

To determine the number of stages and the distribution of workload between those stages, we must analyze the data dependencies between the various modules in the

<table>
<thead>
<tr>
<th>GSM Module</th>
<th>Computational Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preprocessing</td>
<td>5.48 %</td>
</tr>
<tr>
<td>LPC Analysis</td>
<td>7.82 %</td>
</tr>
<tr>
<td>Short-Term Filtering</td>
<td>25.07 %</td>
</tr>
<tr>
<td>LTP Parameter Calc.</td>
<td>49.21 %</td>
</tr>
<tr>
<td>LTP Filtering</td>
<td>3.5 %</td>
</tr>
<tr>
<td>RPE</td>
<td>9.11 %</td>
</tr>
<tr>
<td>File I/O</td>
<td>0.32 %</td>
</tr>
</tbody>
</table>
GSM algorithm (see Figure 2). The goal is to identify the finest "grain" at which we can divide the algorithm without violating these dependencies.

There is, in fact, a strong data dependency between "sub-frames" in the LTP and RPE loop. This loop traverses the full 160-sample block in 4, 40-sample blocks, or sub-frames. For each block, LTP parameters are calculated and the long-term residual is calculated and encoded. The dependency arises in the calculation of the LTP parameters (see Equation 5). This calculation, as discussed in Section 3.1.4, uses an array \((dp)\) of the previous 120 sample values. The dependency issue is that after the parameters for the first sub-frame have been calculated, they must be used in order to "reconstruct" the 40 sample values that will be used in the next sub-frame's calculation, as a replacement for the actual sample values. This reconstruction is done to be consistent with the symmetrical calculation done by the decoder, which necessarily will reconstruct the values based on the LTP parameters it has been given [Degener].

As a result, this implies that we cannot pipeline the LTP/RPE loop. Similarly, we cannot parallelize this loop. However, there is an opportunity to parallelize calculations within the loop; this approach is discussed further in Section 4.2.

Taking into consideration the previous dependency discussion and Table 1, it is clear that we can achieve maximal gain using \(N = 2\); i.e., we can do no better by increasing \(N\) in this case. The choice of breaks in Figure 6 show the most effective ways to balance the load between the 2 stages.
4.1.3 Expected Gains

From Figure 6, the highest load on any stage is \( x_M = x_2 \approx 0.61 \). Given Equation 7, this implies that the expected performance gain, \( G_P \), is:

\[
G_P = \frac{1}{x_M} = \frac{1}{0.61} = 1.64
\]  

(9)

4.1.4 Limiting Factors

The main limiting factor to the effectiveness of the pipeline, aside from how the workload is distributed, is the overhead due to the communication latency between stages. A high communication latency decreases the throughput of the pipeline, thereby reducing the gain.

Fortunately, Raw is meant to provide low latency communication between tiles (see Section 3.2). In addition, we can alleviate this limitation by taking advantage of the Raw architecture's ability to overlap data transfer with data computation. This technique is called latency hiding, and is discussed in detail in Section 4.4.1. We can also utilize a technique called loop unrolling, which reduces the overhead of sending signals to the switch that tell it to perform a data transfer; Section 4.4.2 expands on this approach.

4.2 Parallelization of Long-Term Parameter Computations

The architecture for parallelizing the long-term (LT) parameter computations is shown in Figure 8. Specifically, these are the correlation calculations that take place
during the long-term prediction calculations, in order to determine the LTP lag and gain parameters (see Section 3.1.4).

4.2.1 Design Motivation

Parallelization of the LT parameter calculations is motivated by the statistical profile shown in Table 1. The profile suggests that the LT parameter calculation module consumes the highest percentage of the computational load (~ 50%), thus making it a prime candidate for parallelization.

The parallelization takes advantage of the fact that the correlation calculations for different lags are independent of each other. As a result, we can divide the work of calculating the correlation at different lags between several processing nodes. So,
modifying the original algorithm (see Equation 5), we have:

\[
max(\sum_{k=0}^{39} s[k] \cdot dp[k - \lambda]), \lambda : [\lambda_{start} \ldots \lambda_{end}]
\]  

(10)

The parallelization then divides the full range of \( \lambda : [40 \ldots 120] \) equally among the available processing nodes. For example, with a 5-node parallelization, we might assign \( \lambda_{start} = 40, \lambda_{end} = 56 \) to Node 1.

The LTP algorithm essentially looks for the lag producing the maximum correlation value. Since each processing node finds a “local” correlation maximum, these local maximums are then compared at the central “hub” node (see Figure 8) to produce the global maximum. Also, in order to do these calculations, each node requires the 40-sample array of current signal values (i.e., the \( s \) array), and the 120-sample array of previous signal values (i.e., the \( dp \) array).

The central hub node plays several roles in this design, including mediation of data transfer to/from the satellites, as well doing the non-parallelized computations. The hub node acts as a “source node” by transferring the initial data to the satellite nodes, as well as receiving results from each node. As previously mentioned, the central node also acts as a “merge node” by integrating the local results from the satellite nodes and determining the global result. Finally, as denoted in Figure 8, the central node must also execute the remainder of the unparallelized modules of the GSM algorithm (i.e., LPC analysis, filtering, RPE, etc.).
4.2.2 Design Choices

The main design choice in parallelizing the correlation calculations is determining the number of processing nodes to utilize. On the one hand, the benefit of adding additional nodes is clearly to achieve a higher performance gain. On the other hand, however, there are several factors that can potentially limit adding new processing nodes.

First, we might expect each additional processing node to induce additional overhead for data transfer. The Raw architecture, however, alleviates this problem by providing a mechanism for efficient routing of data to multiple destinations simultaneously. The switch on each Raw tile (see Figure 5) has the capability to multi-cast data in multiple directions on the network. Since the data sent to each auxiliary node is the same, we can route each word of data to any number of locally adjacent tiles within one cycle. This takes advantage of the communication parallelism available in the Raw network in order to alleviate the significant cost of adding new nodes.

Second, the topology of the Raw architecture makes non-local data transfer less efficient than data transfer between adjacent processing nodes. Inherently, the Raw architecture is meant to minimize local communication latencies. However, this suggests that ideally we would like to restrict data transfer between the central hub and the satellite nodes to occur locally. Note that this limits the number of processing nodes to 5, including 4 adjacent tiles (arranged in the “star” formation of Figure 8) and the central node.

Figure 9 shows several topologies which can be ruled out by this constraint. These
topologies are less efficient than the star formation since they require data for calculation (and the results from computation) to be passed through multiple nodes. This is avoided by the star formation since the data required for calculation is sent directly to each node.

That being said, the efficient routing at the switch level minimizes the cost incurred by sending data to non-local tiles. Section 4.3.3 describes the technique used to exploit this to produce an expansive, efficient design.

4.2.3 Expected Gains

We can apply Amdahl's Law to determine the maximum ideal gain that can be achieved by parallelizing the correlation calculations. Given an infinite number of processors, Amdahl's Law predicts the ideal gain $G_C$ as:

$$G_C = \frac{1}{s}$$ (11)
Where $s$ is the fraction of the program that is strictly serial, i.e., not parallelizable, or not being considered for parallelization. The profiling in Table 1 shows that the correlation calculations consume $\frac{1}{2}$ of the total compute time, implying that $G_C = 2$, i.e., we expect a maximum speedup improvement of a factor of 2. Of course, we cannot use an infinite number of processors, so there is a practical upper limit to achieving this ideal.

4.3 Integrated Designs

The integrated design combines the approaches of the pipeline and the parallelization of the correlation calculations, resulting in the architecture shown in Figure 10. The expanded design in Section 4.3.3 takes further advantage of the efficient switch-level data routing in Raw in order to expand the parallelization beyond just locally adjacent nodes.

4.3.1 Design Motivation

The motivation for integrating the first two approaches is clearly to take better advantage of the parallelization opportunities by combining their gains. Aside from simply linearly combining the gains from the two approaches, integrating the approaches may also lead to "nonlinear" speedup gains by allowing for more efficient usage of the pipeline. For example, parallelizing the correlation calculations reduces the load on Stage 2, the compute intensive stage (see Figure 6). As a result, we expect the workload distribution in the pipeline to become more balanced, reducing the workload $x_M$.
on the limiting stage and thereby leading to a higher pipeline gain. So, feedback from the gain in one technique (correlation parallelization) can lead to further gain in the second technique (pipelining).

In addition, since the pipeline and correlation parallelizations are independent of one another, they can be integrated without modifying either one, making this approach straightforward to design. The resulting architecture is then simply constructed as shown in Figure 10. The first stage of the pipeline corresponds to the original pipeline stage in Figure 6. The second stage encompasses a parallelized version of stage 2 in Figure 6.

It should be noted that the number of nodes used for the correlation parallelization...
has been reduced. This is a function of the topology restrictions discussed previously—i.e., data transfer from the hub to satellite nodes has been kept local.

4.3.2 Expected Gains

Since the two approaches are independent of one another, we expect their gains to also contribute independently, aside from the nonlinear gains discussed previously. This suggests that the total expected gain, $G_I$, is the product of the individual gains:

$$G_I = G_P \cdot G_C = 3.3$$ (12)

Clearly this is an approximate estimate: while Equation 12 does not take into account the nonlinear gains from integration, where the increase in $G_C$ feeds back into the system and increases $G_P$. On the other hand, it optimistically estimates $G_C$ (see Equation 11).

4.3.3 Expanding the Integrated Design

As mentioned in Section 4.2.2, there is an opportunity to expand the parallelization of the correlation calculations beyond the use of just local satellite nodes. The motivation behind such an expansion is to take advantage of the efficient routing of data at the switch-level. If the overhead of routing data to nodes not adjacent to the central hub is not too large, then there is a possibility of increasing the total number of processing nodes utilized for the parallelization. In addition, we would like to utilize the as many of the 16 available Raw tiles as possible.
Figure 11 shows the design for an expanded integrated architecture that utilizes 11 of the 16 Raw tiles; 9 of these are used for the correlation calculations. The hub and the “Level 1” satellite nodes are the same as those used in the original integrated design (Figure 10). New to this design are the “Level 2” satellite nodes, which are all locally adjacent to one of the Level 1 nodes. Level 2 nodes were chosen by identifying all nodes that are 2 hops away from the central hub (and not occupied by another task). Figure 12 shows that the organization of the nodes used for the parallelization can be viewed as a tree. Level 2 nodes, then, are the “children” of Level 1 nodes.

Given this topology, the only problem remaining is how to integrate the results from all the nodes at the central hub. In general, the issue is that we cannot distinguish the identity of a set of results by the direction from which they arrive. For
example, the results from Node 6, a child of Node 3, will necessarily arrive to the hub from the same direction as the results from Nodes 3 and 7. This could potentially be solved by adding another word of data sent with the results to act as an identity marker.

Instead, we can exploit the nature of the computation to simplify the solution. We note that the goal of the LT parameter calculation is to find the global maximum correlation; i.e., non-maximum local correlations are irrelevant. As a result, we can use the Level 1 satellites to integrate the results from its Level 2 children and itself, in effect finding the local maximums for each sub-tree in Figure 12. We then find the global maximum of these new local maximums at the central hub. This approach avoids the use of extra identification markers, and is relatively straightforward to implement.

The final note to be made about the design in Figure 11 is the addition of a third pipeline stage. As became clear after this design was implemented with just 2 stages, the workload distribution of the pipeline significantly changes with the advent of
this expansive 9-node parallelization, to the point that the throughput-limiting stage becomes Stage 1 (see Section 5.4.2 for further details). As a result, we can further divide Stage 1 into two stages, to more efficiently distribute the workload and reduce the maximum load $x_M$. This plays on the previous discussion of nonlinear gains achieved by integration; in this case, however, the feedback from one technique’s gain prompts us to modify the architecture itself.

4.4 Data Transfer Optimizations

There are several optimizations which can alleviate the overhead of data transfer between processing nodes; this study focused on two techniques. The first technique, latency hiding, takes advantage of hardware support in Raw to overlap calculations with data transfers. The second technique is loop unrolling, which makes the transfer loop more efficient by sending a block of data transfers each time the switch is signaled.

4.4.1 Latency Hiding

Latency hiding minimizes communication overhead in a parallelized environment by overlapping communication with computation. In this study, latency hiding was utilized to reduce the overhead for communication between pipeline stages.

The concept behind latency hiding is to minimize data transfer overhead by overlapping data transfer with data computation [Strumpen94]. Conceptually, latency hiding is intuitive. However, there must be a mechanism for its implementation. Fortunately the Raw architecture provides such a mechanism. The use of special
assembly-level instructions allows for a value to be computed (e.g., through multiplication or addition) and simultaneously transferred over the static network. By using these instructions, this compute-transfer pair is significantly more efficient than doing the compute and transfer sequentially. This is, therefore, an effective mechanism for implementing latency hiding.

We can utilize this technique in the data transfer between pipeline stages. For example, in the original pipeline design (see Figure 6), we can hide much of the data transfer of the outputs from the LPC analysis (i.e., the reflection coefficients), and the short-term filtering (i.e, the short-term residual).

4.4.2 Loop Unrolling

Loop unrolling reduces the overhead of data transfer by “unrolling” the transfer loop.Ordinarily we would need to signal the switch each time we would like data to be transferred. However, this introduces essentially a 100% overhead, since for each word of data we actually send, we must also spend a cycle informing the switch. This can potentially be an issue when large amounts of data are transferred.

To alleviate this overhead, we can send a block of data each time the switch is signaled. So for example, we could send 16 words per signal. This reduces the overhead of the extra switch signal to only 6%. If the number of words to be transferred is not a multiple of the block size, we can handle the remaining transfers as we did before. Overall, we retain the generality in terms of the number of transfers we can perform, while also reducing the transfer overhead.
4.5 Opportunities for Parallelization in the Decoder

The main opportunity for parallelization in the decoder is pipeline parallelism; the process parallelism exploited for the encoder is not available. At the same time, the decoder is less compute-intensive than the encoder, reducing the need for parallelization.

As discussed in Section 3.1.6, the decoding algorithm is essentially the inverse of the encoder. Clearly, since data is still processed in frames, we can still utilize pipelining. The specifics of how the pipeline is best designed would depend on the computational profile of the decoder.

The parallelization of the correlation calculations does not transfer to the decoder, since the decoding algorithm does not involve correlations. Instead, the long-term calculations for the decoder synthesize the short-term residual, which is done simply by multiplying the LTP gain with the 40-sample block given by the LTP lag; as a result, the long-term analysis is a relatively small portion of the overall computation. The most time-intensive module of the decoding process is the short-term filtering. This module, however, has strong data dependencies between the calculation of successive values, and so cannot be parallelized or pipelined. Identifying additional opportunities for parallelism would require a deeper analysis of the decoding algorithm.

Overall, the decoding algorithm is significantly less computationally intensive than the encoder, and as such has both less need for parallelization and less opportunities for doing so.
5 Results

The designs detailed in Section 4 were implemented on a simulator of a Raw microprocessor, allowing for quantitative evaluation and comparison. Results for these implementations are found as follows:

1. Pipeline (Section 5.2)

2. Parallelization of Long-Term Parameters Calculations (Section 5.3)

3. Integrated Designs (Section 5.4)

5.1 Notes on Implementation

Implementation was done in C for a cycle-accurate simulator of a 16-tile, 4-by-4 Raw processor (see Figure 13). The designs illustrated in Section 4 are spread over such a processor straightforwardly, where every box or node in the illustration corresponds directly to a Raw tile. Also, speech input was retrieved from speech sound files.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

Figure 13: 16-tile Raw processor
A publicly available implementation for the GSM 06.10 speech codec has been developed by Jutta Degener [Degener]. This implementation was used as the base with which the parallelized architectures were constructed. Data communication between local tiles was statically scheduled (i.e., at compile-time) by issuing instructions to the local switches through their corresponding tile processors.

Also, note that baseline used for the results presented here is the compute time for the GSM encoder on one Raw tile. Table 2 lists these baseline results for three sound file sizes. Additionally, the numbers presented are the simulated clock cycles of the chip, which does not necessarily correspond to actual running time of the simulation.

### 5.2 2-stage Pipeline

Timing results for the 2-stage pipeline are shown in Table 3. Measurements for the data transfer overhead are also shown. Specifically, these are measurements of the total time spent for data transfer between the 2 pipeline stages (denoted as “Pipeline” transfer time). Also shown is the transfer overhead as a percentage of the total compute times.

Comparing the results for large files with the expected gain in Equation 9, we see that the empirical results match the estimate of 1.64 closely. The pipeline is able to
Table 3: 2-stage Pipeline Timing Results

<table>
<thead>
<tr>
<th>File Size</th>
<th>Speedup Gain</th>
<th>Compute Time (×10⁶ cycles)</th>
<th>Pipeline Transfer Time (×10⁶ cycles)</th>
<th>Transfer Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>19 frames</td>
<td>1.60</td>
<td>2.97</td>
<td>0.04</td>
<td>1.3 %</td>
</tr>
<tr>
<td>138 frames</td>
<td>1.64</td>
<td>20.94</td>
<td>0.28</td>
<td>1.3 %</td>
</tr>
<tr>
<td>205 frames</td>
<td>1.64</td>
<td>31.07</td>
<td>0.41</td>
<td>1.3 %</td>
</tr>
</tbody>
</table>

achieve its maximum throughput due to the low transfer overhead (∼1.3%).

Also note that the gain for the small file (19 frames) is lower than that for the larger files. This stems from the fact that the number of frames (19) is too small, relative to the number of nodes used to parallelize, to average out initial transients through the pipeline. This is a general trend found in the remainder of the results as well.

5.3 Parallelization of Long-Term Parameter Calculations

Timing results for the parallelization of the LT parameter calculations are shown in Table 4. Note that the transfer times shown here now correspond to the average time spent transferring data between the hub node and each of the satellite nodes (denoted as “S-H” transfer time).

Table 4: Long-Term Parameter Parallelization Timing Results

<table>
<thead>
<tr>
<th>File Size</th>
<th>Speedup Gain</th>
<th>Compute Time (×10⁶ cycles)</th>
<th>S-H Transfer Time (×10⁶ cycles)</th>
<th>Transfer Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>19 frames</td>
<td>1.28</td>
<td>3.72</td>
<td>0.16</td>
<td>4.2 %</td>
</tr>
<tr>
<td>138 frames</td>
<td>1.29</td>
<td>26.65</td>
<td>1.15</td>
<td>4.3 %</td>
</tr>
<tr>
<td>205 frames</td>
<td>1.28</td>
<td>39.70</td>
<td>1.72</td>
<td>4.3 %</td>
</tr>
</tbody>
</table>
In our estimate of the gains achieved by performing this parallelization (see Equation 11), we expected a gain of a factor of 2. Clearly we have not come close to that goal in using 5 nodes. This discrepancy arises from two factors. First, as suggested before, since the analysis for Equation 11 assumed an infinite number of processors, the estimate of a gain of 2 was overly optimistic and an upper bound. Second, there was a transfer overhead of \(\sim 5\%\) between the satellites and the hub, which counteracts the benefits of the parallelization.

5.4 Integrated Designs

The results for the two integrated designs are presented as follows:

1. 2-stage Pipeline + 4-node Correlation Parallelization

2. 3-stage Pipeline + 9-node Correlation Parallelization

5.4.1 2-stage Pipeline, 4-node Parallelization

Timing results for the integrated design with 4 nodes used for the correlation parallelization and 2 pipeline stages is shown in Table 5. Note that we now have two independent types of transfer overhead: one, between pipeline stages; two, between the satellite and hub nodes (for space considerations, only transfer overhead is shown here).

Our conjecture that integrating the two approaches will lead to a higher gains has proven to be correct, as we achieve about a factor of 2.3 gain. There is a significant discrepancy with our estimate of 3.3 in Equation 12, but clearly \(G_C\) was overestimated.
Table 5: LT Parameter Parallelization Timing Results

<table>
<thead>
<tr>
<th>File Size</th>
<th>Speedup</th>
<th>Gain</th>
<th>Compute Time (×10⁶ cycles)</th>
<th>Pipeline Transfer Overhead</th>
<th>S-H Transfer Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>19 frames</td>
<td>2.22</td>
<td></td>
<td>2.14</td>
<td>1.8 %</td>
<td>7.4 %</td>
</tr>
<tr>
<td>138 frames</td>
<td>2.32</td>
<td></td>
<td>14.81</td>
<td>1.9 %</td>
<td>7.8 %</td>
</tr>
<tr>
<td>205 frames</td>
<td>2.33</td>
<td></td>
<td>21.87</td>
<td>1.9 %</td>
<td>7.8 %</td>
</tr>
</tbody>
</table>

dear. If we compensate for that overestimation by using the results in Table 4, we get an expected gain of \(~2.1\).

We can improve this estimate by considering the gains that arise from the feedback between the two sources of speedup. Specifically, the factor \(G_C\) modifies the workload balance in the pipeline, which then adds further gain. We originally had a workload of: Stage 1 at 39% and Stage 2 at 61%. With \(G_C = 1.3\), the workload changes to: Stage 1 at 46% and Stage 2 at 54%. As a result, from Equation 7, we have \(G_p = \frac{1}{0.54} = 1.85\). The adjusted estimate of the gain, then, is \(G_C = 1.85 \cdot 1.3 = 2.4\). After taking into account the transfer overhead of \(~8\%\), we can reconcile this analysis with the results in Table 5.

5.4.2 3-stage Pipeline, 9-node Parallelization

Timing results for the expanded integrated design with 9 nodes used for the correlation parallelization and 2 pipeline stages is shown in Table 6. Again, we have two independent types of transfer overhead. In addition, since we now have 3 pipeline stages, the pipeline transfer overhead shown is the average of the overhead between the two pairs of stages (since the amount of data transferred is identical in both cases).
Table 6: Expanded LT Parameter Parallelization Timing Results

<table>
<thead>
<tr>
<th>File Size</th>
<th>Speedup Gain</th>
<th>Compute Time (\times 10^6) cycles</th>
<th>Pipeline Transfer Overhead</th>
<th>S-H Transfer Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>19 frames</td>
<td>3.33</td>
<td>1.43</td>
<td>2.7 %</td>
<td>11.1 %</td>
</tr>
<tr>
<td>138 frames</td>
<td>3.57</td>
<td>9.60</td>
<td>2.9 %</td>
<td>12.0 %</td>
</tr>
<tr>
<td>205 frames</td>
<td>3.58</td>
<td>14.21</td>
<td>2.9 %</td>
<td>12.0 %</td>
</tr>
</tbody>
</table>

The expanded design proves to be the most effective design, showing a factor of \(\sim 3.6\) speedup gain. This is a significant improvement over the other approaches. The major source of this additional gain has come from the ability to add a third pipeline stage. This, in turn, was made possible by the effectiveness of the 9-node parallelization in significantly reducing the computational load imparted by the correlation calculations.

An interesting trend occurs in the data transfer times between the various approaches. We see that transfer overheads between the satellite and hub range from 5\% (in the original correlation parallelization) to 12\% in this design. Essentially what is occurring is straightforward: while the total compute times are decreasing, the overhead for transfer is staying constant. This suggests that there is room for improvement in these gains by decreasing these transfer times; the next section addresses this issue.

Finally, as mentioned earlier, the timing results from a 2-stage version of this design suggested that the design be expanded to 3 stages. Specifically, timing results for the 2-stage version showed that the workload distribution for the 2-stage version had Stage 1 at 60\% of the total compute time, and Stage 2 at 40\%. Clearly, since we
are able to easily divide the modules in Stage 1, there was an opportunity to expand the pipeline. After expansion to 3 stages, this distribution became: Stage 1 at 22%, Stage 2 at 40%, and Stage 3 at 38%.

5.4.3 3-stage, 9-node with Optimized Data Transfer

Table 7: Transfer Overhead and Speedup Gains after Loop Unrolling

<table>
<thead>
<tr>
<th>Block Size</th>
<th>Speedup Gain</th>
<th>Pipeline Transfer Overhead</th>
<th>S-H Transfer Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 words</td>
<td>3.80</td>
<td>1.0 %</td>
<td>4.6 %</td>
</tr>
<tr>
<td>40 words</td>
<td>3.81</td>
<td>1.0 %</td>
<td>4.1 %</td>
</tr>
<tr>
<td>160 words</td>
<td>3.81</td>
<td>1.0 %</td>
<td>4.0 %</td>
</tr>
</tbody>
</table>

Table 7 shows the results from implementing loop unrolling in order to optimize data transfer in the 3 stage, 9-node parallelization. The “block size” refers to the granularity to which we expand the loop; e.g., a block size of 8 words implies that 8 data words are transferred in succession, without the overhead of branching and looping.

The results show a significant improvement over the overhead in Table 6. The 160 word block, which essentially eliminates the branching overhead (since the array being transferred is 160 samples in length), shows both pipeline and satellite-hub overhead have been reduced by a factor of 3. In addition, the speedup gain has been improved to 3.8, versus 3.58 without the optimizations.
6 Conclusions

The results presented here suggest that we can achieve significant performance improvements in parallelizing the GSM speech compression algorithm on Raw. A 3.8 factor of improvement in the computation times can be achieved by taking advantage of both pipeline and process parallelism in the GSM encoder. This was made possible by the distributed processing and low latency communication that Raw provides. Additionally, we were able to exploit the efficient data routing in Raw at the switch level in order to expand the parallelism beyond local processing nodes.

There are several implications for these results. First, they suggest that a Raw-based system (for example, a handheld supported by a Raw-chip [Agarwal99]) could act as a software-based cell phone employing the GSM protocol, running GSM speech compression in real-time. In addition, we could potentially run several such compute-intensive applications simultaneously in real-time, given sufficient Raw fabric. Second, we can now qualitatively compare the performance of a Raw chip with a conventional microprocessor. For example, one could imagine a Raw chip clocked at a significantly lower rate than a conventional chip, that could match the performance of the conventional chip due to parallelizations. Note that this does not yet take into account the advantage Raw has in I/O bandwidth to the chip. Finally, this study has exposed several ways in which the GSM algorithm can be successfully parallelized, including pipelining and parallelizing the time-intensive correlation calculations.

There are also numerous future directions to be considered. The first extension would be to implement these designs on a Raw hardware emulator, and eventually
on an actual Raw chip. Such implementations could potentially take in live speech input from a microphone and output to a speaker, and would provide for a physical validation of the simulation results. Second, one could potentially conceive of ways that the GSM compression algorithm can be extended upon in order to improve speech quality or increase the compression ratio. For example, there is a large body of research on speech enhancement, which can reduce background noise in a speech signal [Compernolle96]. One could implement such an enhancement module, integrate it with the parallelized GSM (perhaps as an additional pipeline stage), and still operate with little or no reduction in performance. Finally, the scope of this study could be broadened by implementing the entire GSM protocol in software, in essence implementing the previously mentioned software-based cell phone.

This study takes a step towards answering the challenge presented earlier: i.e., achieving the ultimate goal of a compression algorithm that encodes at a high compression ratio, a high quality, and a low computational load. It does this by utilizing a powerful distributed architecture that could soon be a ubiquitous part of systems that need to adapt to both their environments and their users.
References


