ANALYSIS, OPTIMIZATION & EXECUTION OF GENERAL PURPOSE MULTIMEDIA APPLICATIONS ON SUBWORD VLIW DATAPATHS

by

Majd F. Sakr

BS, University of Pittsburgh, 1992

MS, University of Pittsburgh, 1995

Submitted to the Graduate Faculty of

the School of Engineering in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2003

UNIVERSITY OF PITTSBURGH

SCHOOL OF ENGINNERING

This dissertation was presented

by

Majd F. Sakr

It was defended on

November 18, 2003

and approved by

James T. Cain, Professor, Electrical Engineering

Ronald Hoelzeman, Associate Professor, Electrical Engineering

Ivan Kourtev, Assistant Professor, Electrical Engineering

Donald M. Chiarulli, Professor, Computer Science

Dissertation Director: Steven P. Levitan, Professor, Electrical Engineering

ANALYSIS, OPTIMIZATION & EXECUTION OF GENERAL PURPOSE MULTIMEDIA APPLICATIONS ON SUBWORD VLIW DATAPATHS

Majd F. Sakr, PhD

University of Pittsburgh, 2003

In this thesis we evaluate the characteristics of multimedia applications and propose a Multiple Instruction Stream Multiple Data Stream (MIMD) subword Very Long Instruction Word (VLIW) datapath that overcomes the limitations of current architectures to effectively execute multimedia applications.

The characteristics of multimedia applications differ from these of conventional technical applications, in that multimedia applications are highly parallel, computationally intensive, and use low precision (subword) data that can be streaming in nature.

Conventional architectures with fixed-width datapaths cannot effectively perform the required computation. Current solutions employ media-centric single instruction stream multiple data stream (SIMD) based instruction set extensions. However, compilers cannot automatically target these instructions and have to either use hand-tuned assembly libraries or compiler intrinsics. These restrictions limit the full exploitation of the available parallelism in general purpose media applications implemented using high-level programming constructs, which limits performance gains.

We alleviate these restrictions by allowing the compiler to target a flexible MIMD datapath with support for subword execution. Also, we enable the compiler to better exploit the parallelism within the media applications by introducing simple code transformations. To measure the effectiveness of our solution, we evaluate the performance of multimedia applications on the proposed subword MIMD VLIW datapath. This evaluation is performed on a set of multimedia benchmark kernels by analyzing compiler transformations and optimizations and then compiling and executing each kernel using the set of techniques that best target our architecture.

The result is an architecture and an analysis methodology that exploits parallelism across a wide range of multimedia applications by providing better performance and enhanced applicability which in turn enables the required realism in multimedia applications running on general purpose processors.

DESCRIPTORS

Computer Architecture
General Purpose Processors
Multimedia Applications
Processor Design
Subword Datapath
VLIW Microprocessor

TABLE OF CONTENTS

Page

LIST OF TABLES

Table No. Page

LIST OF FIGURES

Figure No. Page

FOREWORD

Needless to say, without the guidance, help and support of many kind hearts and bright minds, this work would not have been possible. I have been very fortunate to have met some special people who have had a significant effect on me, professionally and otherwise, while pursuing this work. It is an impossible task to list everyone and I thank you all.

A special thanks to my advisors, Prof. Steven Levitan and Prof. Donald Chiarulli for their continued guidance, motivation and support over the years. Steve Levitan is surely a special mentor, his high standards, commitment to science, limitless patience and caring create an infectious environment where learning and improvement are primary. I learned a lot from Don Chiarulli, especially from his creativity in finding solutions and enjoyment in performing the work.

I am thankful to my committee members, Professors James T. Cain, Ronald Hoelzeman and Ivan Kourtev. Their feedback and suggestions have significantly improved the dissertation.

I am especially grateful to Prof. Rastislav Bodik for his contributions to the work and, more importantly, his unwavering support without which this dissertation would not have been completed. Ras' clear vision, hard questions and zealous approach shaped my understanding of computer architecture in a fundamental way. Ras always managed to find the time to discuss this work and provided encouragement no matter how far or how busy he was.

I have had the pleasant opportunity to have met some exceptional people, my current colleagues, Jose Martinez, Leo Selavo, Jason Boles, David Reed, Amit Gupta, Craig Windish and Michael Bails, past colleagues, Joumana Ghosn, Peter Tino, Atif Memon and Jim Plusquellic and old friends, Mounzer Fatfat, Marj Henningson, Marwan Azzam, and Caesar Azzam. Thanks for all the wonderful discussions and great times. A big thanks to Sandy Weisberg for all her caring and support and for making sure that I do graduate.

Some very special friends and colleagues have had a tremendous impact on many aspects of my life during this time, Samer Saab, Chakra Chennubhotla, and Lisa Minetti, there are no words to express my gratitude.

A very special thanks to the woman in my life, Lisa Graff, whose love and support always injected happiness during the good times and the tough times. Her enthusiasm, commitment and appreciation for art have inspired me in more ways than she knows.

I am deeply indebted to my family. Thanks to my brothers and sisters, Amer, Ghada, Riad, Najwa, and their families for their unquestioned love, continued encouragement and support. Finally, and most importantly, I would like to thank my father and mother, Fouad and Hiam Sakr, the examples they have set and their commitment to education have instilled in me important values, the love for life and the pursuit of knowledge. Nothing would have been possible without their love, guidance and infinite support.

1.0 INTRODUCTION

In this dissertation, we propose a solution to effective execution of multimedia based applications in a general purpose processor environment by employing a subword Multiple Instruction Stream Multiple Data Stream (MIMD) Very Long Word Instruction (VLIW) datapath. Current general purpose workloads have shifted towards multimedia applications, therefore, general purpose processor architectures must adapt to satisfy the computational requirements of this new workload.

Enabled by technological advancements as well as high network bandwidths, the type of computation we perform using general purpose microprocessors is changing. Consistently, more and more of the dynamic compute cycles are spent on executing multimedia based applications(1,2)*. The characteristics of these applications differ from conventional technical applications in that they are highly parallel, computationally intensive, and use variable precision (subword) data that is streaming in nature.

Ideally, an effective processing solution that targets these applications has several key components. First, a compiler that can extract the parallelism within the media applications and schedule the required execution in a manner that fully utilizes the hardware available by the processor. Second, a memory system that can effectively fetch and store streaming data with minimum address calculation and memory alignment overhead, since a stream needs to be identified by the address of its location in memory and its size. Finally, a flexible datapath that aids the compiler in the scheduling task and then efficiently executes the sequence of operations to satisfy the required computation.

*Parenthetical references placed superior to the line of text refer to the bibliography.

However, the design of a general purpose processor is a careful trade-off between cost, flexibility and performance. The processor must be capable of executing a wide range of general purpose applications in an effective manner while maintaining a reasonable cost. Furthermore, implementations of general purpose applications are not typically tuned or optimized to target a specific processor. Functionality, inter-operability and stability are usually a higher priority for developers of general purpose applications than optimizations to target a specific hardware platform. Hence, the compiler is burdened with the challenge to extract and exploit any parallelism within these applications.

Conventional architectures are not capable of satisfying the required computation of multimedia applications since they were designed to target applications with fixed-precision or fixed data width operations, highly irregular code on data that exhibits high degrees of locality. In anticipation of this shift in the workload to multimedia applications, microprocessor designers, motivated by the need for enhanced performance at a minimal cost, introduced media-centric Single Instruction Multiple Data (SIMD) based instruction set extensions(3). These instructions execute the same operation on multiple subword data elements requiring little control overhead and hence reduced die area cost. However, this solution is incremental due to the fact that current compilers cannot automatically target these SIMD instructions(4). SIMD execution is restrictive in that it requires that all parallelism in multimedia applications match the SIMD execution model. This requirement limits the full exploitation of available parallelism and further performance gains. Also, since the compiler cannot effectively solve the difficult problem of transforming non-SIMD code to target a SIMD architecture, the developer is burdened with this task. This architectural solution enables parallel subword execution in general purpose processors, however, the above problems have resulted in the limited use of multimedia instruction set extensions in general purpose applications.

An effective solution to this problem is one that takes into account the characteristics of multimedia applications and considers the processor design as well as compiler optimizations required to produce better performance.

Therefore, in this thesis we evaluate the characteristics of multimedia applications and propose a variable width augmented VLIW MIMD datapath that overcomes the limitations of current solutions and satisfies the requirements of general purpose multimedia applications. We evaluate our solution by simulating the execution of a set of multimedia benchmark kernels on the suggested subword datapath. We analyze the capabilities of available compiler optimizations in exploiting the available parallelism from the multimedia kernels. Finally, we introduce simple code transformations that allow the compiler to extract more parallelism from media applications resulting in enhanced performance.

The results are an architecture and an analysis methodology that exploit parallelism across a wide range of multimedia applications thus providing better performance.

1.1 PROBLEM STATEMENT

The questions that we address in this dissertation are, can a subword MIMD VLIW datapath coupled with a standard VLIW compiler yield high performance gains given the nature of parallelism exhibited in multimedia based workloads and what code transformations and compiler techniques are required in order to achieve better performance?

The goal of this thesis is to develop an architecture and compiler techniques that can exploit the parallelism in multimedia applications and achieve high performance.

1.2 STATEMENT OF WORK

In order to achieve the goals discussed above, we perform the following tasks:

- **Multimedia application analysis**: Given a set of multimedia applications, identify the kernels of these applications, analyze the code structure of the kernels through control-flow and data-flow analyses. Identify the breakdown of the operation types as well as the data-types of the operands. Perform an overall characterization of the multimedia applications studied.

- **Architectural decisions**: Evaluate the architectural benefits and drawbacks of VLIW processors. Suggest a classical VLIW datapath design and the architectural extensions required in order to achieve a subword MIMD VLIW datapath. Discuss the overhead of implementing the suggested datapath.

- **Compiler/Simulator infrastructure**: Identify a compiler infrastructure that can perform analysis of these applications, has the capability of extracting parallelism and performing optimizations on code segments. Further, the compiler must target the subword VLIW datapath. Finally, employ a simulator to allow the evaluation of the proposed architecture and code transformations on overall performance.

- **Performance evaluation (architecture)**: Study the effects on performance caused by changing the architecture of the datapath from a fixed-width to a variable-width datapath which is capable of performing MIMD subword operations on subword operands.

- **Performance evaluation, (compiler, code transformations)**: Study the efficacy of predicated execution in VLIW processors as well as hyperblock formation and other code manipulation techniques at exploiting the available subword parallelism in the multimedia kernels.

The results of this thesis is a novel subword datapath that enables the compiler to achieve better scheduling of multimedia applications as well as better exploitation of the available parallelism within the applications by employing compilation techniques and performing code transformations to achieve significant speedups across a set of multimedia applications.

1.3 DISSERTATION ROAD MAP

The dissertation is organized as follows: In Chapter 2.0, we discuss the background and motivation of this research. We start by presenting the characteristics of multimedia applications in section 2.1 and discuss why conventional processors need significant changes in order to match the media specific computation. In section 2.2 we discuss how SIMD instructions are being employed in general purpose processors. We then take a detailed look at the specific example of how the Intel MMX, SSE and SSE2 instruction set extensions are utilized and their limitations in achieving the desired performance across a wide range of multimedia applications, in section 2.3. In section 2.4, we list recent embedded processor architectures and discuss the difference between the embedded and general purpose domains. We summarize the current solutions and their limitations in section 2.5. In Chapter 3.0, we perform an extensive analysis on several implementations of multimedia applications and discuss their characteristics. Then in Chapter 4.0, we present a subword VLIW architecture and how it overcomes the limitations discussed in section 2.3 and propose our target architecture for this work, a subword VLIW datapath. We present our methodology and framework for our analysis in Chapter 5.0. We discuss our experimental analysis, results and present code transformation analysis in Chapter 6.0. Finally we present our conclusions in Chapter 7.0 and discuss possible future direction in Chapter 8.0.

2.0 BACKGROUND AND MOTIVATION

In the past decade, microprocessor designs have targeted two application domains, technical and scientific applications for desktop computer systems and transaction processing as well as file serving for server computer systems(1). There is a growing consensus that the target domain is shifting to multimedia applications which will become the prevalent domain for future computer systems(2). These changes are primarily in the workload as well as the increase in global network bandwidth and capabilities. The workload changes are due to the fact that audio/visual realism found in general purpose multimedia applications such as video conferencing, video authoring, visualization, virtual reality modeling, 3D graphics, animation, realistic simulation, speech recognition, and broadband communication promise to deliver better efficiency and effectiveness to a broad range of computer users. The processing is performed on visual and auditory data, such as, images, frames of video, 3-D objects, animated graphics, and audio. Multimedia applications may also have to satisfy a real-time constraint, such as quality of service. Furthermore, computer networks are now capable of delivering this visual and auditory data in real time.

This trend will continue due to the use of intelligent network processors as well as increases in network bandwidth. The rate of increase in network bandwidth is one order of magnitude per new network generation. Currently, a Giga Bit (100MB/sec) optical or copper network can deliver data at rates equivalent to that of an ATA-100 IDE hard drive. The high network bandwidths extend the reach of multimedia applications for streaming data beyond the local hard drive and thus increase the popularity and usefulness of these applications. Hence, with the changes in workload and network capabilities, the challenge posed to computer systems is to perform multimedia processing in an effective manner in order to enable the new level of realism required from current and emerging multimedia applications(5,6).

Enabling this new level of audio/visual realism requires fast execution of algorithms such as encoding and decoding functions in compression algorithms for visualization and for lowering communication bandwidth requirements, data encryption and decryption algorithms to ensure security, translation of geometric models to realistic graphics, as well as analysis algorithms, such as detection and matching. These algorithms in multimedia applications are inherently parallel, computationally intensive, and perform a regular set of operations on large data sets. These characteristics are very similar to those of floating point scientific applications. However, the difference in multimedia computation is the real-time component as well as the characteristics of the data being processed. The data has two distinct features, varying precision requirements as well as being streaming in nature. In the next subsection, we illustrate the general characteristics of media applications in order to understand the challenges(2,5) they pose and to evaluate current microprocessor and compiler designs(6) solutions at addressing these challenges. We dedicate Chapter 3.0 to the detailed analysis of the characteristics of a specific set of multimedia applications.

2.1 CURRENT SOLUTIONS TARGETING MULTIMEDIA APPLICATIONS

Multimedia applications usually contain one or more code kernels which account for the majority of all dynamic instructions executed(2). The processing performed by these kernels can be characterized as inherently parallel, computationally intensive, where the code has regular control structures, consisting of operations performed on large sets of contiguous, streaming, low precision (sub-word) data(5,6).

First, the parallelism stems from the fact that these applications perform the same set of operations on large independent data sets. Therefore, these operations can be performed in parallel. Second, a large set of compute intensive operations are typically performed on the data while the control structure of the code is not complex. The control structure is usually regular with little branching as compared to typical integer applications. Third, the input and output data is characterized as streaming, for example streaming video frames or audio samples, which exhibit reduced temporal locality compared to technical computation. Streaming data is operated on and then discarded, or the result is stored straight back to memory or sent onto the network as an output data stream. Fourth, the precision requirements of the data elements vary for different types of data, such as 8-bits for image pixels and 16-bits for audio samples. In general, the data sizes vary from 8, 16, 32, and 64-bit integer elements to single (32-bit) and double precision (64-bit) floating point elements. In summary, these are highly parallel, compute intensive, bandwidth hungry kernels.

The conventional architectures of general purpose processors are not equipped to process multimedia applications effectively because of fixed-width datapaths and the architecture of memory hierarchy. The datapaths are usually fixed to operations performed on 32-bit word or 64-bit double word operands depending on the processor implementation. These datapaths are inefficient at performing computation on subword operands, for example using a 64-bit wide datapath to perform an 8-bit addition. Besides limited performance, given the parallel nature of multimedia applications, using these datapaths for multimedia processing is an inefficient use of the processor's die area especially if the subword application exhibits high degrees of parallelism. Further, the memory interface of general purpose processors does not match the data requirements of multimedia applications due to two factors. First, similar to the datapath inefficiency, data is accessed as a single 32-bit or 64-bit value which is a significant waste of the valuable memory bandwidth when accessing subword data. Second, the effectiveness of a memory hierarchy relies on high data locality, however, given the streaming nature of multimedia data, caches tend to be continuously polluted by streaming data.(7)

The shift in the workload to multimedia applications has forced microprocessor manufacturers to address some of the above limitations of general purpose processors at effectively executing such application types. As a solution, general purpose microprocessor manufacturers have augmented their conventional microprocessor designs with multimedia enhancements. The enhancements are in the form of a new set of instructions and functional units that better target media processing. We discuss these solutions and their limitations in detail in Section 2.2 and expand on a specific solution, the Intel media instruction set extensions in Section 2.3.

In another approach, there has been a surge in building multimedia specific processors such as Sony's EmotionEngine(15,16), MicroUnity's MediaProcessor(17), NVIDIA's Vertex Engine(18) and Transmeta's recent TM6000 SOC(19). These processors target primarily the home entertainment and video game market by achieving higher performance for a specific set of multimedia algorithms such as the MPEG-2 decoding accelerator found in the EmotionEngine. Furthermore, the application-specific media processors' priority on offering very high performance for general purpose multimedia applications is not high since they target the embedded system market. In our analysis, we focus on solutions targeting general purpose multimedia applications. We illustrate several emerging embedded processors in Section 2.4 and discuss the differences between these solutions and general purpose processors.

Beyond the solutions that target the processor architecture, there are many new cache memory designs intended to resolve the pollution of conventional caches by media stream memory accesses. Streaming media applications exhibit reduced temporal locality when accessing memory as well as reduced spatial locality when accessing 2D data such as images. The most popular solution splits the cache into two, a spatial locality and a temporal locality cache(20,21,22). These approaches statically partition the available die area into the two caches. Other designs propose dynamically reconfigurable cache partitioning in order to better target the specific cache requirements of the processor and software applications(23,24). Another solution allows cache bypassing(25) or provide instructions that can specify the cache level when performing loads and stores(26). Finally, several software solutions attempt to increase the efficiency of conventional caches, one approach exploits hardware prefetching for allocating cache blocks of data that exhibits 2D spatial locality(27). In our work, we do not attempt to solve the memory access issues when executing streaming media application. We focus on the design of the datapath design and its effect on compiler optimizations.

In order to gain a better understanding of the above mentioned processor architectures, in the next three sections, we evaluate the media centric instruction set architecture (ISA) extensions introduced by mainstream microprocessor manufacturers and discuss their limitations. We also list the emerging VLIW and other embedded processors and discuss the implications of the differences between the embedded and general purpose processor domains.

2.2 MULTIMEDIA APPLICATIONS AND SIMD ISA EXTENSIONS

With the current growth of interest in multimedia applications, general purpose microprocessor manufacturers offered multimedia enhanced versions of their processors. The instruction set architecture (ISA) was augmented by adding a set of multimedia instruction set extensions. The multimedia instructions perform a single operation on several subword data elements concurrently using the Single Instruction Multiple Data (SIMD) compute model. This decision was made since the multimedia kernels perform the same set of operations on large data sets and, hence, exhibit a large amount of parallelism.

Examples of these instruction set extensions include the AMD's 3DNow!(8), Motorola's AltiVec for the PowerPC(9), Intel's MMX(10) and SSE(11) extensions for its IA-32 processors and IA-64 extensions(11), Compaq's MVI(13) extensions for the Alpha processors and SUN's VIS(14) extensions for the SPARC processors. A more complete list of multimedia ISA extensions is shown in Table 1.

These instructions include: subword parallel arithmetic instructions, such as addition, subtraction, multiplication; data manipulation, rearrangement and precision conversion instructions such as packing and unpacking operations, interleaving operations, shuffle and rotate operations; comparison instructions; logic instructions; and complex instructions such as multiply and accumulate and pixel distance computation instructions. Further, some instructions are needed to convert data between fixed width registers and subword registers. Finally, there are special memory access operations such as cache-bypassing stores and cache-level specific prefetch instructions.

Table 1 Instruction Set Extensions by major microprocessor manufacturers

Processor	ISA Extension	Capability
AMD	3DNow!	**Floating point** operations on 4x16, 2x32 and 1x64-bit operands using a 64-bit datapath.
Hewlett-Packard PA-RISC	MAX-1, MAX-2	**Integer operations** on 4x16-bit operations using a 64-bit datapath.
Silicon Graphics, MIPS-64	MIPS-3D	**Floating point** operations on 2x32-bit single precision operands using a 64-bit datapath.
Motorola, PowerPC	AltiVec	**Integer operations** on 16x8, 8x16, 4x32-bit operations using a 128-bit datapath. **Floating point** operations on 4x32-bit single precision operands using a 128-bit datapath.
Intel IA-32	MMX, SSE, SSE-2	**Integer operations** on 8x8, 4x16, 2x32, 1x64-bit operations using a 64-bit datapath. **Integer operations** on 16x8, 8x16, 4x32, 2x64-bit operations using a 128-bit datapath. **Floating point** operations on 4x32-bit single precision, 2x64-bit double precision operands using a 128-bit datapath.
Intel IA-64	A combination of MAX-1, MAX-2, MMX, SSE, SSE-2	**Integer operations** on 8x8, 4x16, 2x32, 1x64-bit operations using a 64-bit datapath. **Floating point** operations on 2x32-bit single precision operands using a 64-bit datapath.
Sun UltraSPARC	VIS	**Integer operations** on 8x8, 4x16, 2x32-bit operands using a 64-bit operand.

These instructions execute using a SIMD execution model, however, they differ from the execution performed in conventional vector architectures. There are two main types of vector architectures(28,29), the *vector-register machine* and the *memory-memory vector machine*. In the vector-register machine(31), all vector operations are between vector registers except for load/ store operations. In the memory-memory vector machine, all vector operations are from memory to memory. The first vector machines introduced were of this type. All current mainstream vector machines, however, use the vector-register architecture. In a vector-register machine, the running time of a vector operation has two components, the *start-up time* and the *initiation rate*. Since the vector functional units are usually pipelined, the start-up time is the time required (that is, the number of clock cycles) to fill the pipeline of the functional unit. The initiation rate is the number of clock cycles required to produce a result. Hence the execution time for a vector instruction is the startup time added to the multiplication of the initiation time by the length of the vector.

The media centric ISA extensions listed above execute in a pure SIMD fashion. The operations are performed at once on all data elements that constitute a vector operand. For example, a SIMD addition operation on an eight element 8-bit vector operand is achieved using a typical 64-bit add while ignoring the carry bit at the variable width, 8-bit, boundary. It should be noted that some of the implementations of functional units are pipelined but not all.

Although processors with multimedia ISA extensions have been on the market for several years, the extensions are not widely used because there are no commercial compilers or even mature research compilers with optimizations that can effectively target the new hardware(33,34). This is due to several reasons, the high level languages used to implement most applications do not support vector types, subword types, or vector operations. The SIMD instructions are customized with specific applications in mind, in other words, they lack variable precision (subword) operation generality. The instructions introduced are very specific to what the independent software vendors (ISV) asked the processor designers to make available to them. Hence, the compiler has to transform non-SIMD code to SIMD instructions, which is a challenging task. The quick solution was to develop assembly libraries of popular functions and make them available to code developers. The libraries enhanced the use of the new SIMD instructions however in a very limited fashion because of the limited functionality available in the libraries as well as problems with the interface to these functions. Therefore, the use of SIMD ISA extensions was limited to developers performing assembly language implementations which required very detailed understanding of the algorithm being implemented as well as the SIMD instruction set and the underlying hardware. Hand optimizations using assembly code need to be updated manually in order to take advantage of new capabilities introduced with new processors. Hence, the limited use of these SIMD ISA extensions implies limited exploitation of available parallelism in multimedia applications.

The limited targetability of this hardware is not due to a lack of interest or effort, it is due to the fact that identifying and transforming a code segment to target the SIMD execution units is a challenging task. The limitation is due to the restrictions applied on executing instructions in the SIMD unit. The restrictions are due to operand-size specific operations, restrictions on data configuration and layout in memory that must be met in order to enable the use of SIMD operations. Hence, augmenting the processor by allowing very specific concurrent execution without developing the capability of the compiler in order to achieve real speedups across applications of interest is not an effective solution.

In summary, one of the limitations in utilizing SIMD ISA extensions is the basic assumption that all subword parallelism found in multimedia applications can be characterized to match the SIMD execution model only. This assumption restricts the programmer and the compiler by having to exploit the parallelism in multimedia applications specifically using SIMD instructions. In addition, there are other architectural based limitations to current processors. In order to discuss these architectural limitations, we need to evaluate a specific architecture and how the SIMD based functional units are embedded within it. Therefore, in order to understand these restrictions and limitations of extending the ISA to gain significant speedups, we take a detailed look at the

Intel MMX, SSE and SSE2 ISA extensions and their hardware implementation, since they offer a super set of all available SIMD operations available in general purpose processors as shown in table 1.

2.3 INTEL'S MMX, SSE AND SSE2 ISA EXTENSIONS

The designers of the Intel microprocessors incrementally added the SIMD functionality to their IA-32 bit general purpose processors (Table 2). They started by adding the MMX extensions(35,36,37) which are 64-bit integer instructions on packed operands. The MMX unit does not alter the state of the processor and, hence, requires no operating system (OS) support. Further, none of the operations raise any exceptions.

The Pentium is an in-order superscalar machine, the next generation processor introduced, the Pentium Pro, was more revolutionary by implementing a dynamic out of order (OOO) execution pipeline. The PII combined the out of order execution in the Pentium Pro with the MMX unit. The PIII offered the SSE extensions(38,40,41), which are packed floating point instructions for 32-bit single precision operands. The SSE unit is 128-bits wide, has a control and state register and the floating point operations could raise arithmetic exceptions. Hence, the PIII required some OS modifications and support. The NetBurst architecture(42,43) was introduced in the P4 processor, it is a 20 stage hyper pipeline with a trace cache and double clocked integer ALUs, along with the SSE2 extensions. The 128-bit SSE2 extensions perform packed 128-bit integer operations as well as packed double precision floating point operations. In the next subsection we take a close look at each of the SIMD extensions and then end this section by discussing the advantages and limitations of these processor enhancements.

Table 2 The incremental introduction of SIMD instructions into the Intel Processors

Intel IA-32 Processor	ISA Extension	Capability
Pentium	MMX	47 integer instructions on packed 8, 16, 32 & 64-bit operands using a 64-bit datapath.
PII	Pentium Pro + MMX	Out of order (OOO) execution, superscalar processor with MMX instructions.
PIII	PII + SSE	70 floating point instructions on packed 32-bit single precision floating point operands using a 128-bit datapath.
P4	PIII + NetBurst + SSE2	144 integer and floating point instructions on packed 8, 16, 32 & 64-bit integer operands and packed 32 & 64-bit floating point operands.

2.3.1 The MMX instruction extensions

To enhance the performance of the Pentium processor, Intel introduced 47 new MMX instructions(44,45) which are parallel operations on packed eight 8-bit, four 16-bit, two 32-bit integer operands as well as operations on a 64-bit integer operand. The cost of adding the MMX instructions was approximately 10% of the die area. Eight 64-bit MMX registers are aliased to the eight x87 floating point (80-bit) registers, hence, no new hardware is required for the MMX register file. The EMMS (Empty MMX state) instruction is used to switch from MMX mode to x87 mode. The EMMS instruction can require up to 50 clock cycles to complete. The requirements of adding the MMX unit were that it maintains full software compatibility and does not require any OS support. Therefore, the MMX unit does not have a state, there are no control registers or any condition codes. Also the instructions could not raise any arithmetic exceptions. We discuss how overflow and underflow conditions are dealt with later in this section. For context switching support, the x87 FSAVE and FRSTOR instructions can be used to save and restore values in the MMX registers.

The SIMD operations can be performed on operands with varying precision. The four different data types introduced are shown in Figure 1.

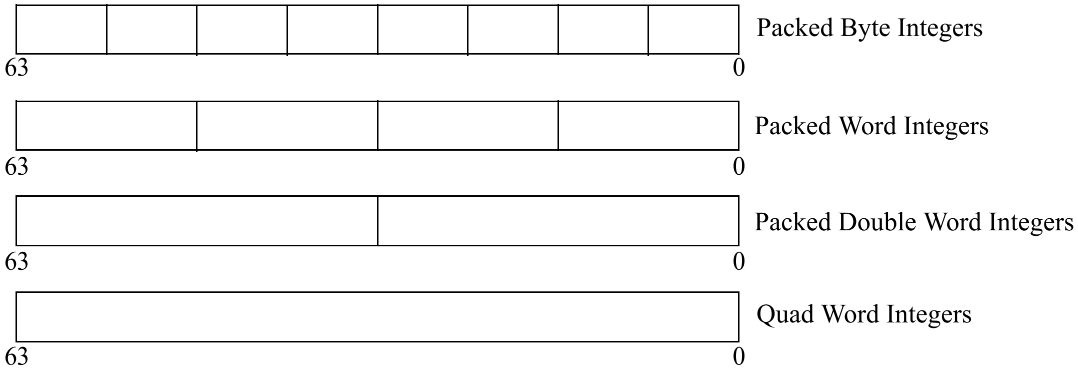


Figure 1 New data types introduced with the MMX instruction extensions.

Since the MMX unit cannot raise any numeric exceptions, three new types of arithmetic operations are introduced in the instruction set in order to eliminate the overflow and underflow conditions:

- **Wraparound** (W), truncate the result to the available bits in the result register, ignore the most significant bits that are out of range.
- **Signed Saturation** (S), saturate the result to either highest positive integer or smallest negative integer using the available bits in the result register.
- **Unsigned Saturation** (US), saturate the result to either the highest integer or zero.

This solution is sufficient because saturation arithmetic satisfies the majority of computation found in multimedia applications.

The types of operations available within the MMX instruction set can be categorized as follows: Data Transfer; Arithmetic; Comparison; Conversion; Unpacking; Logical; Shift; and the Empty MMX state instruction (EMMS)

The instruction set is mixed and not all types of operations are available for all operand widths. Instruction types are operand size specific as shown in Table 3 below:

Table 3 MMX Instruction Set Summary

<table>
<tr><th></th><th></th><th colspan="3">Wraparound</th><th colspan="3">Signed Saturation</th><th colspan="3">Unsigned Saturation</th></tr>
<tr><td colspan="2"></td><td>8bit</td><td>16bit</td><td>32bit</td><td>8bit</td><td>16bit</td><td>32bit</td><td>8bit</td><td>16bit</td><td>32bit</td></tr>
<tr><td>Arithmetic</td><td>Addition</td><td>✓</td><td>✓</td><td>✓</td><td>✓</td><td>✓</td><td></td><td>✓</td><td>✓</td><td></td></tr>
<tr><td></td><td>Subtraction</td><td>✓</td><td>✓</td><td>✓</td><td>✓</td><td>✓</td><td></td><td>✓</td><td>✓</td><td></td></tr>
<tr><td></td><td>Mult save low</td><td></td><td>✓</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td>Mult save high</td><td></td><td>✓</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td>Mult and Add</td><td></td><td>✓</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>Comparison</td><td>Compare Eq</td><td>✓</td><td>✓</td><td>✓</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td>Compare GT</td><td>✓</td><td>✓</td><td>✓</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>Conversion</td><td>Pack</td><td></td><td></td><td></td><td></td><td>✓</td><td>✓</td><td></td><td></td><td>✓</td></tr>
<tr><td>Unpack</td><td>Unpack High</td><td>✓</td><td>✓</td><td>✓</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td>Unpack Low</td><td>✓</td><td>✓</td><td>✓</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td colspan="11">Logical, Shift and Data Transfer Instructions</td></tr>
<tr><td></td><td colspan="2"></td><td colspan="2">8 bit</td><td colspan="2">16 bit</td><td colspan="2">32 bit</td><td colspan="2">64 bit</td></tr>
<tr><td>Logical</td><td colspan="2">AND</td><td colspan="2"></td><td colspan="2"></td><td colspan="2"></td><td colspan="2">✓</td></tr>
<tr><td></td><td colspan="2">AND NOT</td><td colspan="2"></td><td colspan="2"></td><td colspan="2"></td><td colspan="2">✓</td></tr>
<tr><td></td><td colspan="2">OR</td><td colspan="2"></td><td colspan="2"></td><td colspan="2"></td><td colspan="2">✓</td></tr>
<tr><td></td><td colspan="2">XOR</td><td colspan="2"></td><td colspan="2"></td><td colspan="2"></td><td colspan="2">✓</td></tr>
<tr><td>Shift</td><td colspan="2">Shift Left Logical</td><td colspan="2"></td><td colspan="2">✓</td><td colspan="2">✓</td><td colspan="2">✓</td></tr>
<tr><td></td><td colspan="2">Shift Right Logical</td><td colspan="2"></td><td colspan="2">✓</td><td colspan="2">✓</td><td colspan="2">✓</td></tr>
<tr><td></td><td colspan="2">Shift Right Arithmetic</td><td colspan="2"></td><td colspan="2">✓</td><td colspan="2">✓</td><td colspan="2"></td></tr>
<tr><td>Data Transfer</td><td colspan="2">Register to Register</td><td colspan="2"></td><td colspan="2"></td><td colspan="2">✓</td><td colspan="2">✓</td></tr>
<tr><td></td><td colspan="2">Load from Memory</td><td colspan="2"></td><td colspan="2"></td><td colspan="2">✓</td><td colspan="2">✓</td></tr>
<tr><td></td><td colspan="2">Store to Memory</td><td colspan="2"></td><td colspan="2"></td><td colspan="2">✓</td><td colspan="2">✓</td></tr>
<tr><td>Empty MMX</td><td colspan="10">✓</td></tr>
</table>

The data transfer operations can transfer 32-bit data from memory or general purpose registers. Furthermore, other operations perform 64-bit data transfers of packed data from memory or between the MMX registers. Arithmetic operations consist of addition and subtraction. Multiplication is only available for word operands with the ability to store either the high-order or low-order 16 bits into the destination operand. Multiply instructions require 3 cycle latency, however the unit is fully pipelined and a multiply instruction can start every cycle. The multiply and add instruction multiplies four 16-bit operands summing the four intermediate 32-bit values in pairs to produce two 32-bit results. The comparison operations generate a mask of ones or zeros which are written to the destination operand. Conversion instructions convert words into bytes and doublewords into words. Unpack instructions unpack bytes, words, doublewords from high or low order elements from the source operands and interleave them in the destination operand. Logical instructions perform a bitwise logical operation on quadword operands. Shift instructions shift each element by a specified number of bit positions. Finally, the EMMS instructions empties the MMX registers to prepare for x87 FPU instructions.

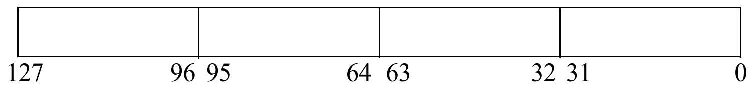
2.3.2 Streaming SIMD (SSE) Extensions

The new architectural enhancement of the PIII processor was to introduce single-precision floating point SIMD instructions(44,45) on packed 32-bit operands. The implementation of the Streaming SIMD extension unit consumed about 10% of the die area.

SSE instructions are for packed (32-bit) single precision floating point values. The eight XMM registers are 128 bits wide and the purpose of the SSE functional unit is to execute four 32-bit microoperations in a single clock cycle. However, Intel’s microprocessor designers achieve this not by implementing a 128 bit functional unit but by double-clocking a 64-bit execution unit and performing two microoperations per the shorter clock cycle to produce 4 microoperations in a single clock cycle of the processor.

The SSE adds a new state to the processor and has a control/state MXSCR register. The XMM register file is flat as opposed to the stacked x87 register file. The SSE unit is depipelined, has one multiplier and one adder, so it can only run an add and mult operation back to back.

The new data types are either packed four 32-bit single precision floating point operands or a scalar single 32-bit floating point operand. The new data types are shown in Figure 2.



Contains either a:

- packed 4 single precision floating point values

- or a single 32-bit floating point value in the lower order bits.

Figure 2 New data types introduced with the SSE instruction extensions.

The SSE instructions provide the ability to perform SIMD operations on packed single precision floating point values as well as scalar instructions. These instructions include, arithmetic instructions such as addition, multiplication, division, reciprocal, square root, reciprocal of square roots, max, and min operations. The logical instructions include AND, AND NOT, OR, XOR. The comparison instructions can compare operands using 12 comparison conditions. The shuffle and unpack operations shuffle or interleave the low or high pair elements from two source packed operands into one destination operand. The conversion operations support packed and scalar conversions from single precision floating point to doubleword integer formats. The data transfer instructions move single precision floating point data between the XMM registers and between an XMM register and memory. The memory address must be aligned to a 16-byte boundary, otherwise an exception is generated. To balance the memory bandwidth requirements and execution, new instructions are provided to allow the programmer to enhance concurrent execution by caching soon to be used data. Also, non-allocating (streaming) store instructions are available to limit cache pollution.

Furthermore, some new MMX instructions are also introduced with SSE extensions. They include computing the average of unsigned packed byte (8-bit) or packed word (16-bit) operands, return the min or max of unsigned packed byte operands, return min or max of signed packed word operands, multiply unsigned word operands and return high result, as well as other move, mask and shuffle instructions.

2.3.3 SSE2 Extensions

The P4(43) introduced 144 new instructions which include support for 128-bit integer arithmetic operations as well as 128-bit double-precision floating point operations and cache/memory management operations.

The SSE2 instructions(44,45) introduce six new data types. The include a 128-bit packed double-precision (64-bit) floating point values packed into a quad doubleword, 128-bit packed byte (8-bit) integers, 128-bit packed word (16-bit) integers, 128-bit packed doubleword (32-bit) integers, 128-bit packed quadword (64-bit) integers. These new types are shown in Figure 3.

The SSE2 extensions maintain the same state as the SSE extensions. The main additions are the packed double precision floating point operations and the 64-bit as well as 128-bit packed integer operations. The instruction categories closely match what is available in the MMX and SSE extensions. Many new conversion instructions are introduced to convert values between single, double precision floating point values in the XMM registers and quad double word, quadword integer values in the XMM registers as well as quadword values in the MMX registers and doubleword values in the general purpose integer registers.

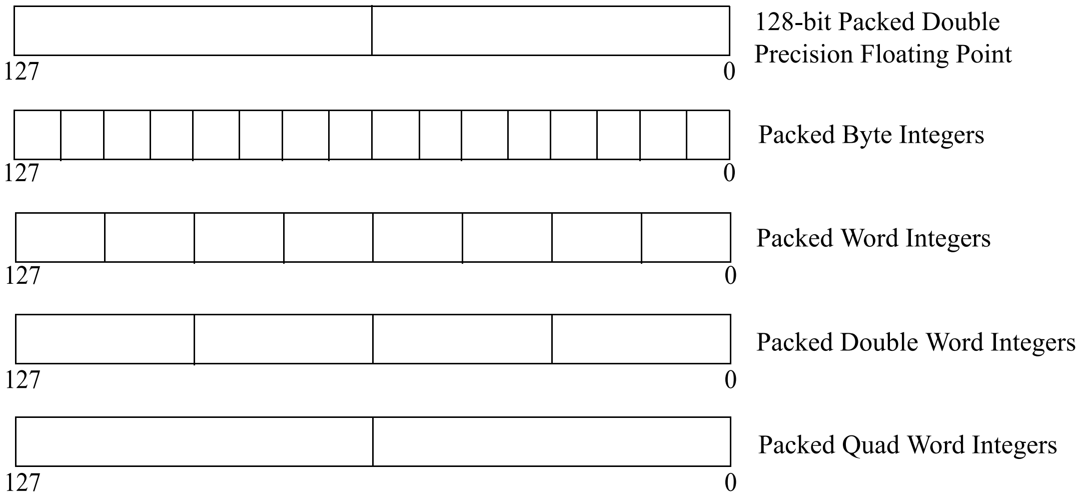
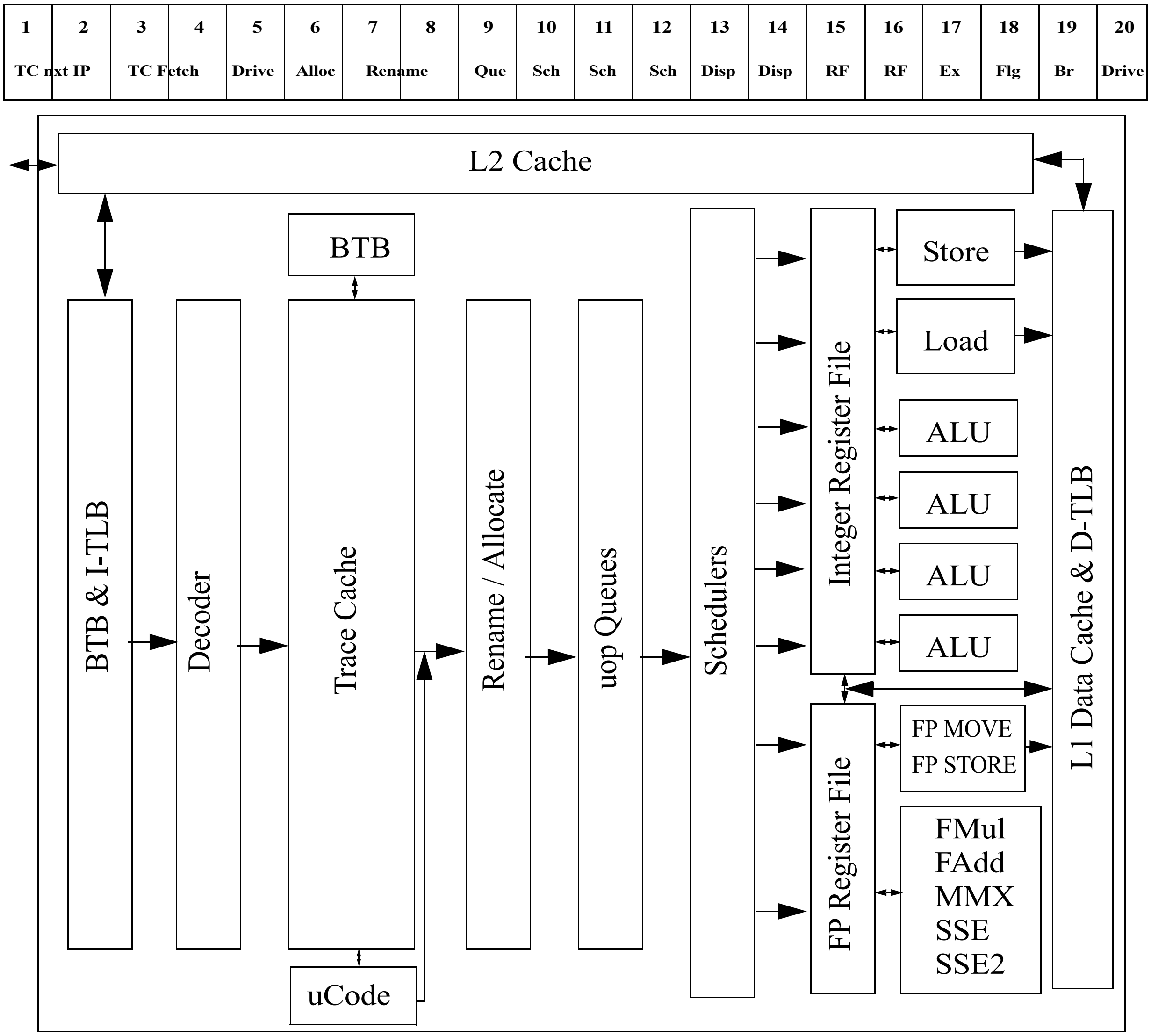

128-bit Packed Double Precision Floating Point
127
0
Packed Byte Integers
127
0
Packed Word Integers
127
0
Packed Double Word Integers
127
0
Packed Quad Word Integers
127
0

Figure 3 New data types introduced with the SSE2 instruction extensions.

The only arithmetic exceptions raised in the SSE and SSE2 unit are from the floating point operations. The SSE2 extensions maintain the capability of storing non-temporal (streaming) data by bypassing cache.

The SIMD execution units are employed within the Intel netburst architecture. The netburst architecture is a hyper pipelined architecture with sophisticated hardware based control to enhance the extraction of Instruction Level Parallelism (ILP) from the executing instruction stream. The 20 stage hyper pipeline and sophisticated control around the datapath are shown in Figure 4(43).

The hyper pipelined netburst architecture enables very high clock rate execution by reducing the amount of logic between pipeline stages. Instructions and data are read from main memory to L2 cache at a bandwidth of 3.2 GB/s. The trace cache is one of the novel features in the netburst architecture. Instructions are fetched decoded and stored in the trace cache in a manner that matches their execution sequence. Further, the trace cache eliminates the overhead of having to re-decode instructions accessed from the I-Cache as in previous designs. The pipelined execution shown in Figure 4 performs as follows: Fetch the pointer from the Branch Target Buffer (BTB)



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC nxt IP		TC Fetch		Drive	Alloc	Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flg	Br	Drive

Figure 4 The Intel NetBurst microarchitecture 20 stage execution hyper pipeline.

indicating the location of the next instruction in the trace cache; Read the decoded instruction from the trace cache; Wire delay; Allocate resources; Register renaming (from 8 logical registers to 128 physical registers); Place microoperation into the queue; Schedule by computing dependencies; Dispatch microoperation to appropriate execution unit; Read the register file; Execute; Compute flags; Branch check; and drive the result to the front end of the machine.

2.3.4 Limitations of Intel's Subword SIMD Instruction Extensions

There are several limitations to using subword SIMD instructions to achieve a significant speedup on multimedia applications. In this thesis we limit our comments to the MMX, SSE and SSE2 SIMD extensions employed within the P4 execution datapath.

The first limitation is due to the overhead paid in trying to extract instruction level parallelism (ILP) from the dynamic instruction stream. The SIMD instructions execute as part of the hyper pipelined P4 microprocessor. As illustrated in Figure 4 in the previous subsection, a large segment of the 20 stage pipeline is spent on dynamic techniques to extract higher ILP from the instruction streams. Since multimedia applications exhibit regular control structures as well as being inherently parallel, the complex dynamic ILP extraction techniques are unnecessary because the compiler should identify and extract the available parallelism statically. These dynamic hardware techniques offer high ILP for non-media centric integer-based applications which exhibit complex control structures and, hence, a lot of branching within the code. Therefore, a large overhead is paid in dynamic control techniques since the SIMD functional unit is embedded within the hyper pipelined datapath and every instruction targeting it must traverse the pipeline before executing. This control overhead is significant when executing multimedia applications since the compiler can analyze their control structure, extract the parallelism and schedule the instructions statically.

Second, the memory hierarchy is not the best solution for streaming data access. In the P4 microprocessor, the data is loaded into registers only through the L1 and L2 Caches, and the data can be stored directly to main memory by bypassing the caches. A 1.5GHz P4 has a 4-pumped 100MHz bus with an aggregate off-chip bandwidth of 3.2 GB/s and an on-chip bandwidth, between L1 and L2 caches, of 48GB/s. If most of the data being processed is streaming and exhibits reduced degrees of locality then the on-chip bandwidth is ineffectively utilized and the caches only increase the overhead of delivering the data to the execution units. Further, reading a 128-bit packed data type from L1 cache into the 128-bit register file requires a lengthy 6 clock cycles. Hence, the memory interface is designed to target applications that exhibit high degrees of locality. This interface is less effective at accessing memory in a streaming fashion to satisfy the data hungry multimedia applications.

Third, SIMD execution is restrictive, the parallelism is achieved only as a single operation on packed and aligned data. These restrictions limit the effectiveness of an automatic compiler to utilize these instructions. Further, to enjoy higher performance through concurrent SIMD execution, the programmer must insure that the overhead due to library function calls, data transport and manipulation, which is required to enable the SIMD execution, must not override the gains of the parallel execution.

Lack of language support to enable vectorization using an automatic compiler is a problem. Vector standards exist in a few high-level programming languages which further complicates the problem for automatic targeting of SIMD execution units. However, there is an effort to address this problem by introducing streaming-data based high-level representation(49).

A minor limitation of the SIMD instruction set extensions is that integer arithmetic operations do not raise any exceptions which further complicates the compiler’s job when automatically choosing to use a lower precision data type for a variable. Numeric exceptions allow the complier to add cleanup code in case of an overflow or underflow condition rather than producing imprecise results.

The data that is operated on in the SIMD unit requires several pack and unpack operations to get the data ready for the packed arithmetic, logical and shift operations. The instruction overhead of these pack/unpack operations can be up to 20% of total dynamic instruction count which is very significant. For example, for SIMD instructions to provide a significant speedup when performing 3D vertex computations, the data must be in a structure of arrays (SOA) format(40). However, sometimes 3D data is stored in an array of structures (AOS) format and, hence, must be transformed to SOA dynamically using data reorganization instructions which incur an instruction overhead of 20 to 25 percent leading to a dismal 10% speedup. This significant overhead is paid in order to match the data to SIMD execution and make use of concurrent execution using these operations which questions the efficiency of the SIMD extensions. A recent study(39) verifies that the efficiency of the SIMD unit in a processor is very low due to this overhead and introduces a hardware unit to perform many of the overhead instructions.

SIMD extensions in general purpose processors are not an advantageous solution due to the limitations listed above. Next, we illustrate the architectures of several embedded processors, some of them employing similar SIMD extensions as the ones discussed above, and discuss the difference in targeting multimedia applications in an embedded product versus a general purpose processor.

2.4 EMBEDDED MULTIMEDIA PROCESSORS

An emerging trend in processor design has been the transition of embedded DSP devices to employ VLIW processor cores such as the Philips Trimedia processors(46), Equator/Hitachi's MAP(47) and BSP(48) processors and Fujitsu's FR-V family of processors(49). Other novel approaches are streaming embedded processors such as the Stanford Imagine(50) and the Berkeley VIRAM(52) architectures which employ streaming register files or on-chip main memory modules to satisfy the high-bandwidth requirements of streaming media applications.

These devices target the consumer electronics, automotive and communications markets. The primary design constraints of such devices are low-power, small die area, reduced code size, and high performance for specific applications. These system-on-a-chip (SOC) devices can be used in several products and are required to have enough flexibility to be capable to perform different functionality depending on the product and its corresponding applications. This flexibility is achieved by including a programmable processor within the device which also allows for late changes in specifications or designs as well as software reusability. These SOCs include several application-specific hardware modules, coprocessors, that are employed to satisfy the high performance and low-power design constraints for certain applications such as video decoding, image manipulation, FIR filtering, and encryption. The functionality included in the application specific hardware coprocessors is based on a software/hardware codesign of the target applications. Also, these devices usually include certain analog input and output modules as well as standard memory and communication interfaces. We illustrate the different core processor designs within these embedded devices.

The Philips Trimedia CPU64 architecture(46) is an embedded processor that targets digital televisions and the set-top box market. The processor architecture is a 5-issue VLIW with a uniform 64-bit datapath and memory interface. The trimedia processor is capable of supporting SIMD subword processing by treating the 64-bit word as a vector of 8, 16, or 32 bit operands. The subword SIMD media-centric operations are similar to the ones described in the MMX/SSE instruction sets of the Pentium processors.

Hitachi and Equator's MAP DSP(47) and BSP processor(48) employ a 4-issue VLIW core. The Datapath consists of two 32-bit integer functional units and two variable width integer/graphical functional units that are capable of performing certain operations on 32-bit or 64-bit operands. Further, the integer/graphical units can achieve subword computation using a SIMD model on a vector of 8, 16, 32 and 64 bit operands.

Fujitsu's latest embedded processor design(47) employs an 8-way VLIW core. Four integer units and four floating/media units. This architecture is an extension of the FR-V embedded processor family which are a 4-issue VLIW core. The four integer units are 32-bit fixed width units. The floating/media units are also 32-bits wide, however, they can perform either four concurrent single precision floating point operations or two SIMD operations on vectors of four 16-bit operands.

The Imagine architecture[(51)], uses eight VLIW arithmetic clusters in a SIMD fashion. Each cluster consists of 8 ALUs, which can execute a sequence of VLIW instructions on 32, 16 and 8-bit operands. A 128KB streaming register file loads and stores data streams from off-chip memory modules. The Imagine architecture is deployed as a loosely coupled coprocessor to a host processor where the host processor sends stream instructions to the stream controller in Imagine. Applications targeting the Imagine processor have to be programmed using a special dialect of the C programming language based on the stream programming model[(50)].

The VIRAM architecture[(52)], is a single issue 64-bit MIPS core with a vector coprocessor that consists of 2 pipelined arithmetic units where each unit contains four 64-bit vector datapaths. The vector operations can be performed on 64, 32 or 16-bit operands. The VIRAM processor also has 13 MBytes of on-chip main memory. The on-chip memory is targeted at satisfying the high bandwidth requirements of streaming media applications.

The domain of embedded computation is very different than the general purpose processing domain. The target applications of the embedded processor domain are ones that are mature, well developed and well understood. Furthermore, the processor design constraints are a careful balance between die area size, performance requirements and power consumption. Subword media processing is limited to the SIMD model in order to conserve on the die area dedicated to functional units. Therefore, companies developing applications in this domain are required to manually analyze and optimize their applications to effectively use the SIMD units in order to meet the performance requirements given the hardware available and the power consumption constraints. The streaming processors suffer from similar limitations, the VIRAM vectorizing compiler can vectorize simple code structures but requires hand optimizations at the assembly level for other code structures. The Imagine has a complicated programming model which limits ease of implementation and software portability.

The hardware specific implementation of hand-optimization of applications is possible to do in this domain. However, in a general purpose processing domain, we are not granted the benefits of fixed application implementations targeting fixed architectures. Emerging general purpose applications are implemented in a general fashion to target a diverse set of processors. Further, several compiler technologies exist to target many diverse platforms and applications are not usually hand tuned to execute on one particular platform.

2.5 SUMMARY OF RELATED WORK

In summary, we have discussed the state of the art in general purpose processors and their SIMD extensions in order to target multimedia applications. We also identified the limitations of using SIMD instructions within the existing architectures. Also, we have listed the latest architectures in the embedded processor domain and discussed the implications of having a fixed set of mature applications targeting a fixed processor architecture, embedded into a product, versus emerging applications targeting general purpose platforms.

Our goal in this dissertation is to discuss how to overcome the limitations within general purpose processors by employing a MIMD subword-VLIW datapath that can offer more hardware flexibility to the compiler in order to better target multimedia applications and provide the speedups required to enable the desired audio/visual realism.

Before delving into the details of the proposed architecture, we must take a close look at the applications being evaluated in this domain. In the next section, we analyze the implementations of several emerging multimedia applications and evaluate their code structures, data types and computation requirements.

3.0 CHARACTERISTICS OF MULTIMEDIA APPLICATIONS

In this Chapter, we analyze and discuss the execution characteristics of several multimedia and streaming applications. Specifically, we analyze the kernel of each application and characterize the types of operations, operand data-types as well as control-flow and data-flow structures. We end this chapter with a general multimedia application characterization and highlight possible avenues to exploit the parallelism and streaming data features.

As discussed earlier in Section 2.1, typical multimedia and streaming workloads include applications such as video conferencing, video authoring, visualization, virtual reality modeling, 3D graphics, animation, realistic simulation, speech recognition, and broadband communication. The input data to such applications or the output data delivered by these applications is usually visual and auditory data, such as, images, frames of video, 3-D objects, animated graphics, and audio. Multimedia applications may also have a real-time constraint to satisfy, such as quality of service. At the core of these applications are compute intensive algorithms such as, encoding and decoding functions in compression algorithms for visualization and for lowering communication bandwidth requirements, data encryption and decryption algorithms to ensure security, translation of geometric models to realistic graphics, as well as analysis algorithms, such as detection and matching.

These algorithms in multimedia applications are inherently parallel, compute intensive, and perform a regular set of operations on large data sets. The input and output data sets have two distinct features, varying data precision requirements as well as being streaming in nature. In this chapter, we take a close look at the characteristics of several implementations of multimedia applications in order to understand the challenges and changes required in microprocessor and compiler designs in order to achieve better performance.

3.1 KERNELS OF MULTIMEDIA APPLICATIONS

Multimedia applications usually contain one or more code kernels which account for the majority of all dynamic instructions executed. The processing performed by these kernels can be characterized as inherently parallel and compute intensive, where the code has regular control structures, consisting of operations performed on large sets of contiguous, streaming, variable-precision data. Following is a more detailed description of these four general characteristics of multimedia kernels.

First, the parallelism stems from the fact that the applications perform the same set of operations on large independent data sets. Therefore, these operations can be performed in parallel. Second, typically, a large set of compute intensive operations are performed on the data and the control structure of the code is not complex. The control structure is usually regular with little branching as compared to typical integer applications. Third, the input and output data is characterized as streaming, for example streaming video frames or audio samples, which do not exhibit temporal locality. This is because, typically, streaming data is operated on and then discarded, or the result is stored straight back to memory or sent onto the network as an output data stream. Fourth, the precision requirements of the data elements varies for different types of data, such as 8-bits for image pixels and 16-bits for audio samples. In general, the data sizes vary from 8, 16, 32, and 64-bit integer elements to single (32-bit) and double precision (64-bit) floating point elements. In summary, these are highly parallel, compute intensive, bandwidth hungry kernels.

Following this general description, we analyze implementations of popular applications from the following representative domains, speech transcoding, data encryption audio coding/decoding (codec) and video coding/decoding using the multimedia applications shown in Table 4. These applications are from the MediaBench and MediaBenchII benchmark suites[70].

Table 4 The multimedia domains and nine applications that are analyzed.

Application Domain	Multimedia Applications
Speech	GSM Encoder GSM Decoder
Encryption	PEGWIT Encryption PEGWIT Decryption
Audio	ADPCM Encoder ADPCM Decoder
Video	MPEG-2 (DVD) Encoder MPEG-2 (DVD) Decoder MPEG4 (DivX) Encoder

First, we look at speech compression and decompression algorithms, specifically that of the Global System for Mobile (GSM) communication standard. Second, we analyze a general block data encryption and decryption method, the PEGWIT Algorithm. Third, we will discuss the characteristics of a DVD video encoding and decoding technique, based on the MPEG-2 codec. Finally, we analyze an implementation of the DivX encoder algorithm based on the MPEG-4 codec.

3.2 GSM - LOSSY SPEECH TRANSCODING ALGORITHM

GSM is an implementation of the final draft of the Global System for Mobile telecommunication, GSM 06.10[70], standard for full-rate speech transcoding. The GSM speech coding algorithm uses the Regular-Pulse Excitation Long-Term Predictor (RPE-LTP). This algorithm was chosen for this domain based on radio channel data bandwidth, subjective speech quality, algorithm complexity, cost, processing delay, and power consumption. Furthermore, this algorithm is a popular speech codec used in many real-time video conferencing applications.

The GSM algorithm compresses a digital speech signal that is generated as follows. An analog speech signal is digitized on a fixed phone network, such as the Integrated Services Digital Network (ISDN), using Pulse Coded Modulation (PCM). The important frequencies in a speech signal go up to 4kHz, and, hence, when a speech signal is digitized, it is sampled at 8kHz and quantized on a linear scale. The digitized speech signal that is used as input to this algorithm consists of 20ms long frames represented as 160 13-bit linear PCM values sampled at 8kHz. The 20ms period represents the typical time between the opening and closing of two vocal folds caused by air pushed from the lungs when humans speak. During this 20ms period, the speech signal does not change much, which enhances the opportunity for compression.

The basic idea of the coding algorithm is that information from previous speech samples, which does not change very quickly, is used to predict the current sample. The speech signal is represented by the coefficients of the linear combination of the previous samples, adding on to it an encoded form of the residual. The residual is the difference between the predicted and actual sample.

The GSM 06.10 algorithm, illustrated in Figure 5, models the speech signal in three phases. The first is the linear predictive short-term filter which divides the speech signal into short-term predictable parts. The second is the long-term predictive filter, which calculates the lag and scale parameters for the long-term predictable parts and finally, the last phase, encodes the remaining residual pulse. The decoder, synthesizes the speech by passing the residual pulse through the long-term filter and then the short-term filter.

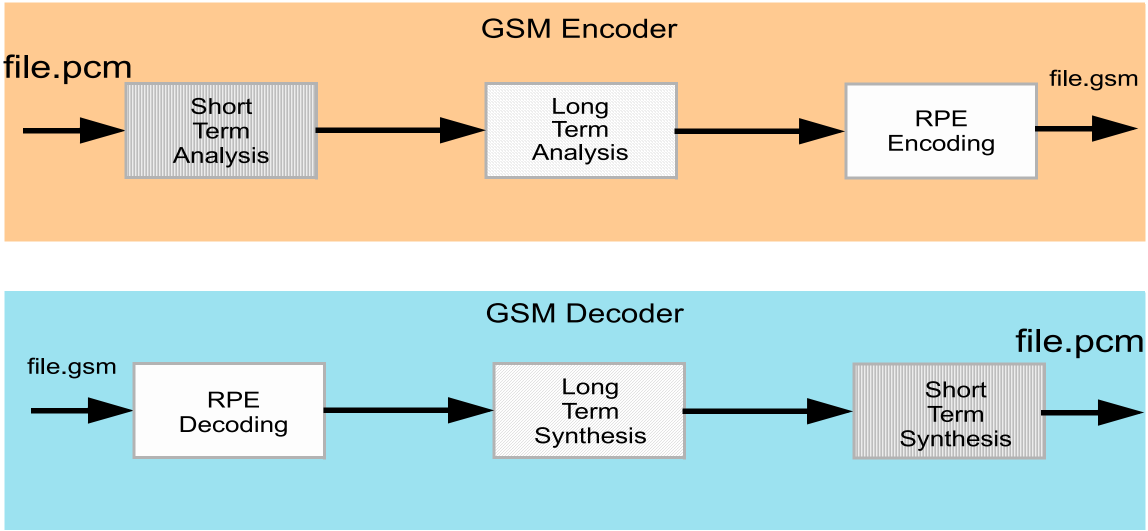


Figure 5 The block diagram of the GSM Encoder and Decoder Algorithm.

The GSM algorithm encodes the 20ms long frames of 160 13-bit PCM samples into 260-bit GSM frames, or decodes GSM frames into linear PCM frames. However, this implementation of the algorithm uses 16-bit values to store the PCM sample and generates 264-bit GSM frames to satisfy the “power of 2” byte value precision in the C programming language and target processors. The GSM algorithm compresses a 20ms PCM speech frame into 264 bits, resulting in a total bit rate of 13 kbps.

This is a lossy compression/decompression technique which stores the linear-predictor filter parameters as a compression technique and uses them to synthesize the speech signal when decoding. Running a good PCM speech signal through this encoder and decoder ten times repetitively completely degrades the quality of the speech.

In the next subsection, we identify the code kernels of the GSM encoder and decoder and analyze their code structure, control flow, data flow and instruction type breakdown. This analysis allows us to understand the computation requirement of the most significant portion of the GSM encoder and decoder so that we can identify a means with which we hope to speed up the execution time of these two applications.

3.2.1 Kernel of the GSM Encoder

The implementation of the GSM encoder consists of the three phases discussed in the previous section. In the first two phases, the speech samples are analyzed and the short-term and long-term filter parameters that can best predict the speech samples are computed. These filter parameters are used to synthesize the signal in the decompression stage. The final phase encodes the residual signal.

To identify the code kernel in this application, we execute the compression algorithm on a typical speech signal and record the number of clock cycles spent in each function of the application. During a typical execution of this speech compression algorithm, over 80% of the execution time is spent executing a single function that performs the calculation of the Long Term Predictor (LTP) parameters (Figure 6). The short-term predictor analysis function accounts for 10% of the total clock cycles. Both of these functions can be considered kernels for this application, however, we focus our study on the more significant kernel, the calculation of the LTP parameters (Calculation_of_the_LTP_parameters).

Function Calculation_of_the_LTP_parameters computes the gain and the lag for the long term analysis filter. This requires calculating a maximum of the cross-correlation function between the current sub-segment short term residual signal, vector (*v*), of 40, 16-bit elements and the residual signal, vector (*r*), of 120, 16-bit elements as shown in the following equation:

$$max\{A_j\}\text{, where } A_j = \left(\sum_{i=0}^{39}(v[i]\cdot r[i+j])\right) \text{ and } 0 < \mathrm{j} < 80$$

The computation required to get the maximum cross-correlation between these two vectors, is the maximum of the resulting 80 correlation computations.

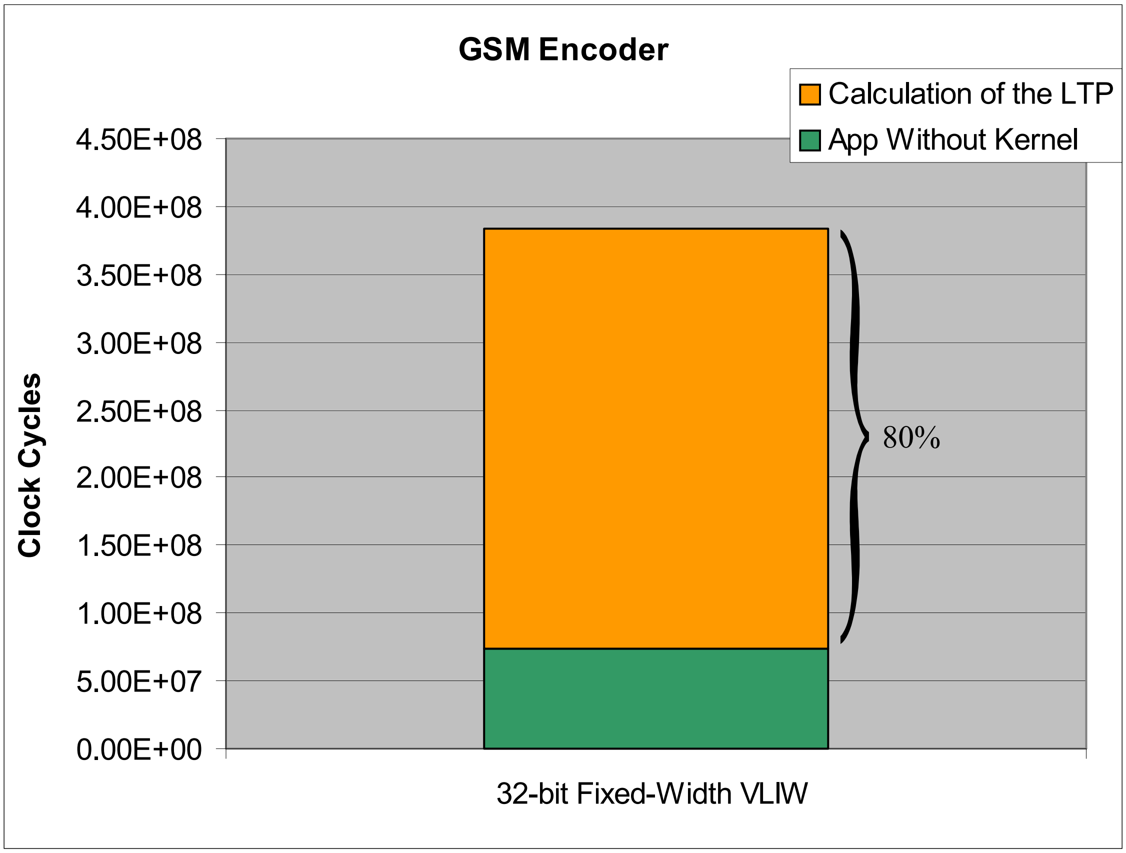


Figure 6 The kernel of the GSM Encoder accounts for 80% of clock cycles during a typical execution.

The code listing for this kernel (Calculation_of_the_LTP_parameters) is shown in Figure 7. The highlighted portion of the function is the *for loop* where the cross-correlation is calculated and is the most computationally intensive segment of the function which accounts for 79% of the total execution time required by the application to perform the speech encoding.

The control-flow of this kernel is a simple *for* loop, with a known upper bound, that performs the cross-correlation computation. The body of the loop consists of the 40 multiplications of two, 16-bit values and their summation. As for the data-flow, there exists true dependence (read after write) and an output dependence (write after write) between the accumulator (`L_result`) of all

79%
of the GSM
encoder's clock
cycles are spent
executing this
code segment

```
static void Calculation_of_the_LTP_parameters P4((d,dp,bc_out,Nc_out),
register word* d,       /* [0..39]IN   */
register word* dp,      /* [-120..-1]IN*/
word           * bc_out,/*           OUT     */
word           * Nc_out/*            OUT     */
)
{
register int  k, lambda;
word           Nc, bc;
word           wt[40];

longwordL_max, L_power;
word           R, S, dmax, scal;
register wordtemp;

[...] /* code deleted for bervity */

/* Search for the maximum cross-correlation and coding of the LTP lag
 */
L_max = 0;
Nc    = 40;/* index for the maximum cross-correlation */

for (lambda = 40; lambda <= 120; lambda++) {

#       define STEP(k) (wt[k] * dp[k - lambda])

        register longword L_result;

        L_result  = STEP(0)  ; L_result += STEP(1) ;
        L_result += STEP(2)  ; L_result += STEP(3) ;
        L_result += STEP(4)  ; L_result += STEP(5)  ;
        L_result += STEP(6)  ; L_result += STEP(7)  ;
        L_result += STEP(8)  ; L_result += STEP(9)  ;
        L_result += STEP(10) ; L_result += STEP(11) ;
        L_result += STEP(12) ; L_result += STEP(13) ;
        L_result += STEP(14) ; L_result += STEP(15) ;
        L_result += STEP(16) ; L_result += STEP(17) ;
        L_result += STEP(18) ; L_result += STEP(19) ;
        L_result += STEP(20) ; L_result += STEP(21) ;
        L_result += STEP(22) ; L_result += STEP(23) ;
        L_result += STEP(24) ; L_result += STEP(25) ;
        L_result += STEP(26) ; L_result += STEP(27) ;
        L_result += STEP(28) ; L_result += STEP(29) ;
        L_result += STEP(30) ; L_result += STEP(31) ;
        L_result += STEP(32) ; L_result += STEP(33) ;
        L_result += STEP(34) ; L_result += STEP(35) ;
        L_result += STEP(36) ; L_result += STEP(37) ;
        L_result += STEP(38) ; L_result += STEP(39) ;

        if (L_result > L_max) {

                Nc    = lambda;
                L_max = L_result;
        }

}

[...] /* code deleted for bervity */

}
```

Figure 7 The code of the kernel GSM Encoder which calculates the LTP parameters.

the multiplications that are performed within the highlighted loop body. Further, the execution of each iteration of the loop is completely independent of subsequent iterations, therefore, no inter-loop dependence.

The dynamic instruction distribution for this kernel executing on a fixed width (32-bit) VLIW processor consisting of a register file with 64 registers, 4 integer ALUs, and 2 memory units using a typical input data set is shown in Figure 8. As shown, almost 60% of all operations are *integer* *alu* operations, 24% are *load* operations and 16% are *store* operations. Only 1% of the operations are *branches* and *compares*. The integer operations are due to performing the address calculation of the two vectors, `wt[k]` and `dp[k - lambda]`, the multiplication of the two 16-bit values stored as 32-bit operands and finally their summation. The large number of loads is expected, however, the large number of stores must be due to register pressure and having to often store and then re-load temporary variables.

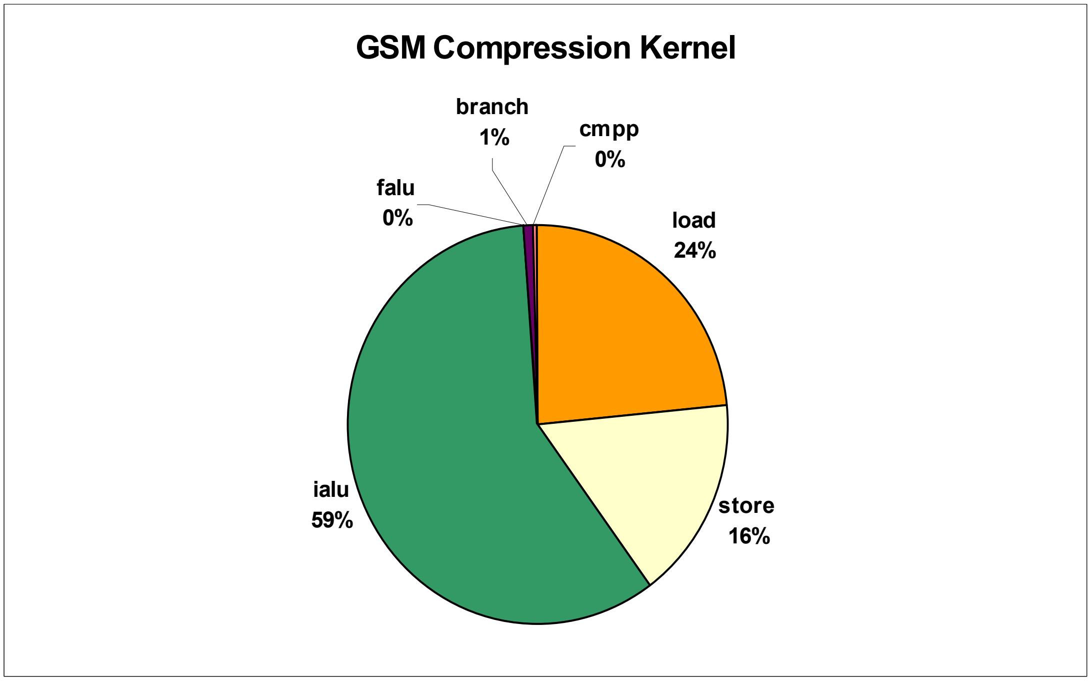


Figure 8 The dynamic instruction breakdown of the GSM Compression Kernel.

The kernel of the compression algorithm, GSM encoder, has a very simple control flow, the output data-dependence can be circumvented by using temporary variables and hence the majority of the integer operations have subword operands and can be performed in parallel.

3.3 SPEECH DECOMPRESSION USING THE GSM DECODER

The speech signals encoded using the GSM encoder in the previous section are then decoded back into PCM samples using the GSM decoder. The decoder synthesizes a 264bit GSM frame into a speech signal of 160 16-bit PCM samples. This procedure is performed by first decoding the residual signal into distinct samples, then the samples are fed through the long-term and short-term synthesizers using the filter parameters stored in the GSM frame to reproduce the PCM speech signal. Since this algorithm is lossy, the synthesized speech signal should sound rather like what was handed to the GSM encoder but is not identical to it.

3.3.1 Kernel of the GSM Decoder

Executing the GSM decoder on a typical input data set, we observed that the GSM decoder spends 70% of all dynamic execute cycles in a single function as shown in Figure 9. This function performs the short term filtering synthesis required to reproduce the original signal.

The code listing for this kernel (Short_term_synthesis_filtering) is shown in Figure 10. The inner *for loop* consumes 96% of the total clock cycles of the function. This translates to 66% of the total execution time required by the application to perform the GSM decoding.

The kernel synthesizes the 160, 16bit PCM samples for each GSM frame. The computation required to perform this synthesis is a sequence of saturating subtracts, multiplications and additions of two 16-bit values into a 16-bit result. The control-flow is a pair of nested *for* loops. The upper bound of the nested loop is known while that of the outer loop is input dependent. Further, there is are two if-statements within the nested loop. As for data-flow, there exists a true dependence between every instruction inside the nested loop as well as inter-loop iteration direct data dependence. The control flow and data dependence of this kernel is more complex than that of the GSM encoder.

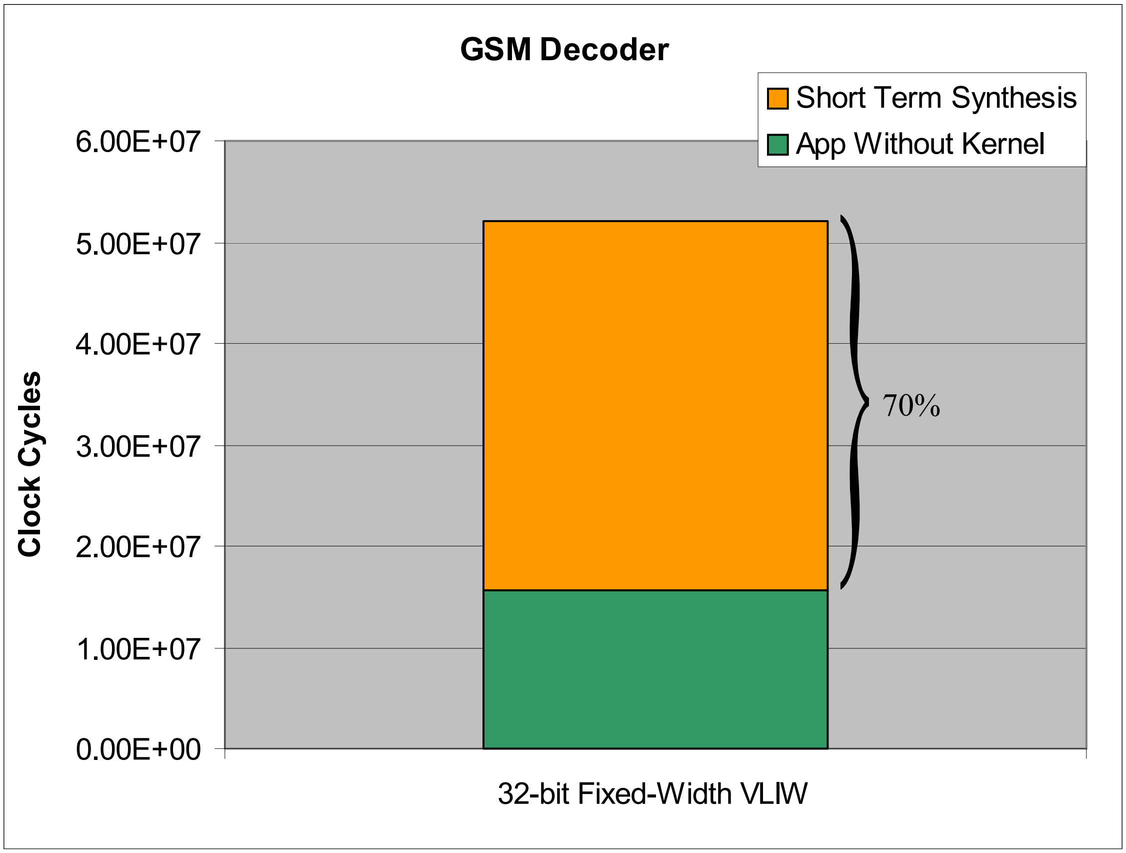


Figure 9 The kernel of the GSM Decoder accounts for 70% of clock cycles during a typical execution.

The instruction distribution for a typical execution of this kernel executing on a fixed width (32-bit) VLIW processor with register file of size 64, 4 integer ALUs, and 2 memory units is shown in Figure 11. The integer alu operations still dominate with 56% of the total number of operations. As expected from the complex control flow, and unlike the GSM encoder, the percent-

```
static void Short_term_synthesis_filtering P5((S,rrp,k,wt,sr),
struct gsm_state * S,
register word* rrp,/* [0..7]IN*/
register intk, /* k_end - k_start*/
register word* wt,/* [0..k-1]IN*/
register word* sr/* [0..k-1]OUT*/
)
{
register word  * v = S->v;
register int   i;
register word  sri, tmp1, tmp2;
register longwordltmp;/* for GSM_ADD  & GSM_SUB */

#define MIN_WORD        ((-32767)-1)
#define MAX_WORD        ( 32767)

/* >> is a signed arithmetic shift right */
#define SASR(x, by)     ((x) >> (by))

#define GSM_MULT_R(a, b) /* word a, word b, !(a == b == MIN_WORD) */
        (SASR( ((longword)(a) * (longword)(b) + 16384), 15 ))

#define GSM_ADD(a, b)   \
        ((ulongword)((ltmp = (longword)(a) + (longword)(b)) -
                MIN_WORD) > \
                MAX_WORD - MIN_WORD ? (ltmp > 0 ? MAX_WORD :
                MIN_WORD) : ltmp)

# define GSM_SUB(a, b)  \
        ((ltmp = (longword)(a) - (longword)(b)) >= MAX_WORD \
        ? MAX_WORD : ltmp <= MIN_WORD ? MIN_WORD : ltmp)
```

70%
of application
clock cycles

is spent in this
function

```
while (k--) {
        sri = *wt++;
        for (i = 8; i--;) {

                tmp1 = rrp[i];
                tmp2 = v[i];

                tmp2   = GSM_MULT_R(tmp1, tmp2);
                sri    = GSM_SUB( sri, tmp2 );
                tmp1   = GSM_MULT_R(tmp1, sri);
                v[i+1] = GSM_ADD( v[i], tmp1);
        }
        *sr++ = v[0] = sri;
}
```

Figure 10   The code of the GSM Decompression Kernel

age of branch operations is 23% and compare operations is 11%. The percentage of loads and stores are 6% and 4% respectively. There are no floating point operations in this kernel.

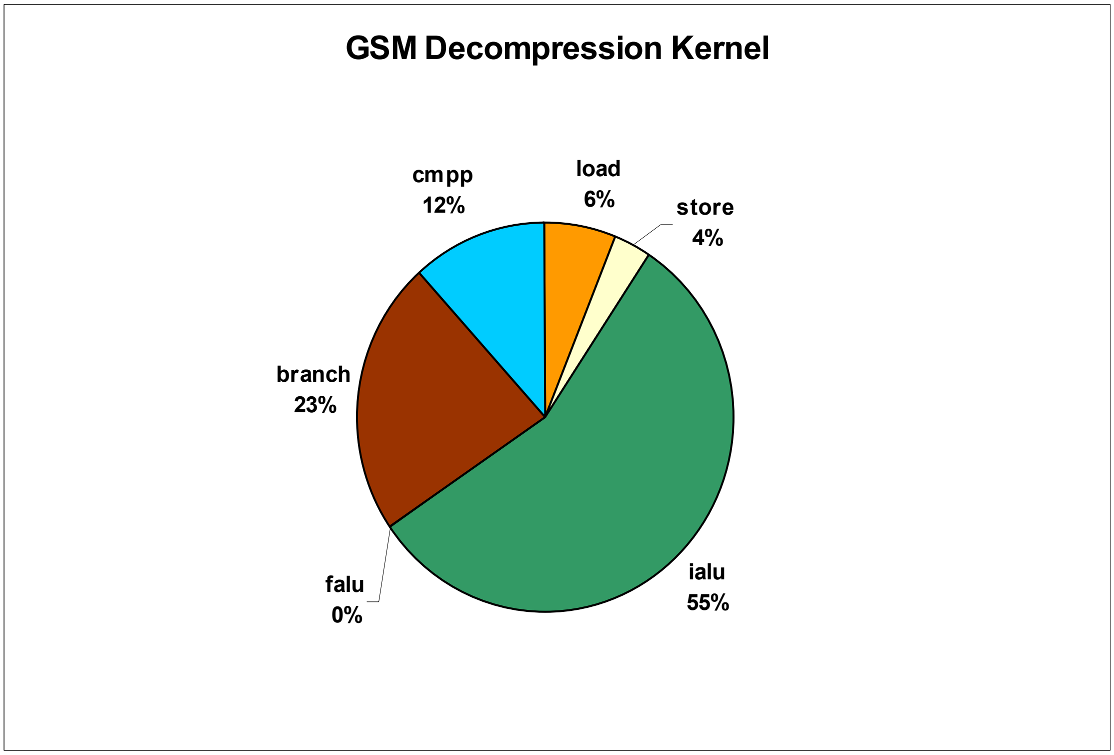


Figure 11 The dynamic instruction breakdown of the GSM Decompression Kernel

The kernel of the GSM decoder consists of two nested loops, a extensive direct data dependence in the operations of the nested loop as well as inter-loop data dependence between subsequent invocations of the nested loop which limits the opportunity for parallel execution of the operations. The integer operations are performed on subword 16-bit operands.

3.4 PEGWIT ENCRYPTION ALGORITHM

The PEGWIT algorithm(70) is utilized for performing public key encryption and authentication. It uses an elliptic curve over Galois Field GF(2^{225}), the SHA1 algorithm for hashing and the symmetric block cipher square for encryption. We look at the execution characteristics of both the encryption and decryption of an ASCII file, the text of the General Public License (GPL).

3.4.1 Kernel of the PEGWIT Encryption algorithm

After running the encryption algorithm on an ASCII file, we identified two subkernels. The first performs a Galois Field (GF) multiply and add operation and the second performs a GF element multiplication. The two subkernels combined account for 68% of the dynamic execution cycles when encrypting a typical ascii file as shown in Figure 12.

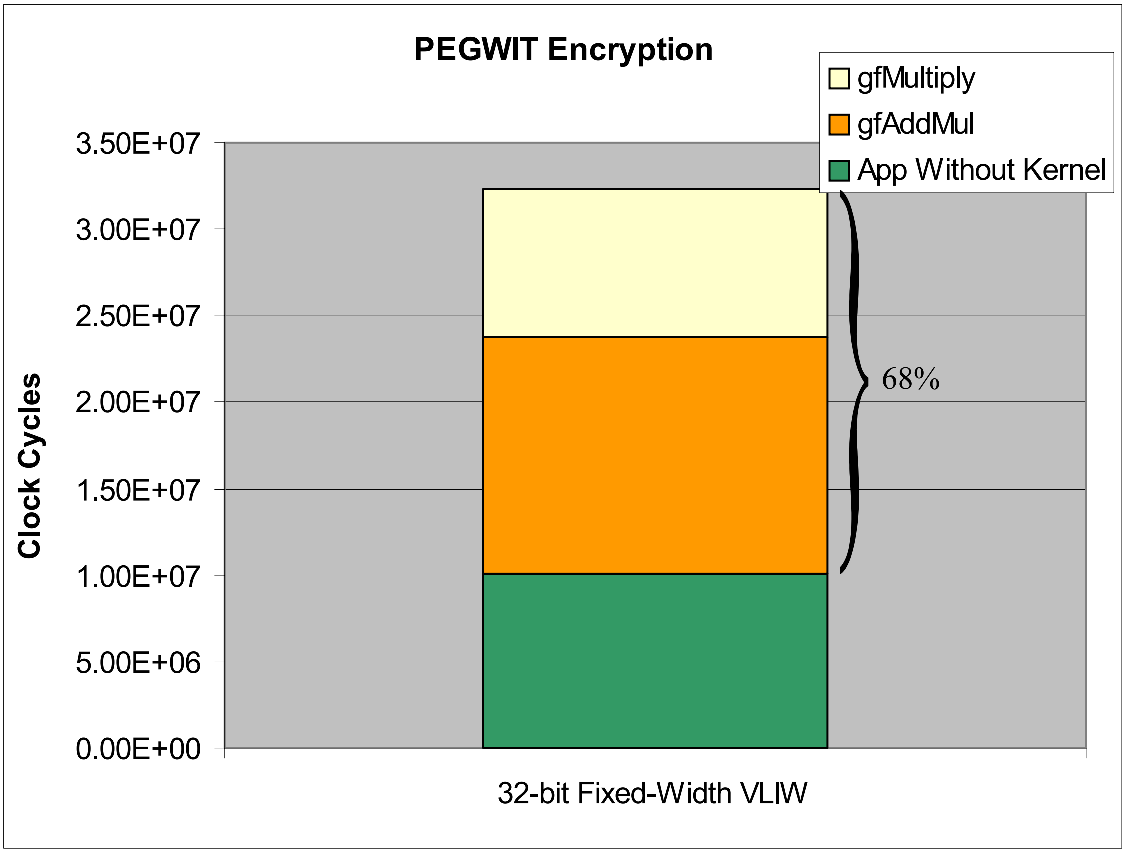


Figure 12 The kernel of the PEGWIT Encryption accounts for 68% of clock cycles during a typical execution.

The code of the first kernel, gfAddMul, is shown in Figure 13. It performs a GF add and multiply operation on two GF points, each represented as an array of 16-bit elements. The operation lies within a single-nested loop. The control-flow within the kernel consists of three main loops, two while loops and one for loop. The body of the for loop consists of a couple of if-statements. There exists a direct data dependence between the operations in both while-loops.

42%
of application
clock cycles

is spent in this
function

```
static void gfAddMul (gfPoint a, ltemp alpha, ltemp j, gfPoint b)
{
        ltemp i, x, la = logt[alpha];
        lunit *aj = &a[j];

        assert (logt != NULL && expt != NULL);
        while (a[0] < j + b[0]) {
                a[0]++; a[a[0]] = 0;
        }
        for (i = b[0]; i; i--) {
                if ((x = logt[b[i]]) != TOGGLE) { /* b[i] != 0 */
                  aj[i] ^= expt[(x += la) >= TOGGLE ? x - TOGGLE : x];
                }
        }
        while (a[0] && a[a[0]]==0) {
                a[0]--;
        }
} /* gfAddMul */
```

Figure 13 The code of the gfAddMul Kernel of PEGWIT.

The second kernel performs the GF element multiplication. The computation is within a doubly-nested for-loop as shown below. Further, the body of each loop has an if-statement.

26%
of function clock
cycles

is spent in
this region

```
     void gfMultiply (gfPoint r, const gfPoint p, const gfPoint q)
            /* sets r := p * q mod (x^GF_K + x^GF_T + 1) */
    {
            int i, j;
            ltemp x, log_pi, log_qj;
            lunit lg[GF_K + 2]; /* this table should be cleared after
use */

    [...] /* code portion deleted for brevity */

    /* perform multiplication: */
           gfClear (r);
           for (i = p[0]; i; i--) {
               if ((log_pi = logt[p[i]]) != TOGGLE) { /* p[i] != 0 */
                   for (j = q[0]; j; j--) {
                     if ((log_qj = lg[j]) != TOGGLE) { /* q[j] != 0 */
                          r[i+j-1] ^= expt[(x = log_pi + log_qj) >=
                            TOGGLE ? x - TOGGLE : x];
                     }
                    }
                }
            }
                r[0] = p[0] + q[0] - 1;

    [...] /* code portion deleted for brevity */


    x = log_pi = log_qj = 0;
        memset (lg, 0, sizeof (lg));
    } /* gfMultiply */
```

Figure 14 The code of the gfMultiply Kernel of PEGWIT

The dynamic instruction distribution for the two subkernels executing on a fixed width (32-bit) VLIW processor with register file of size 64, 4 integer ALUs, and 2 memory units is shown in the following pie chart

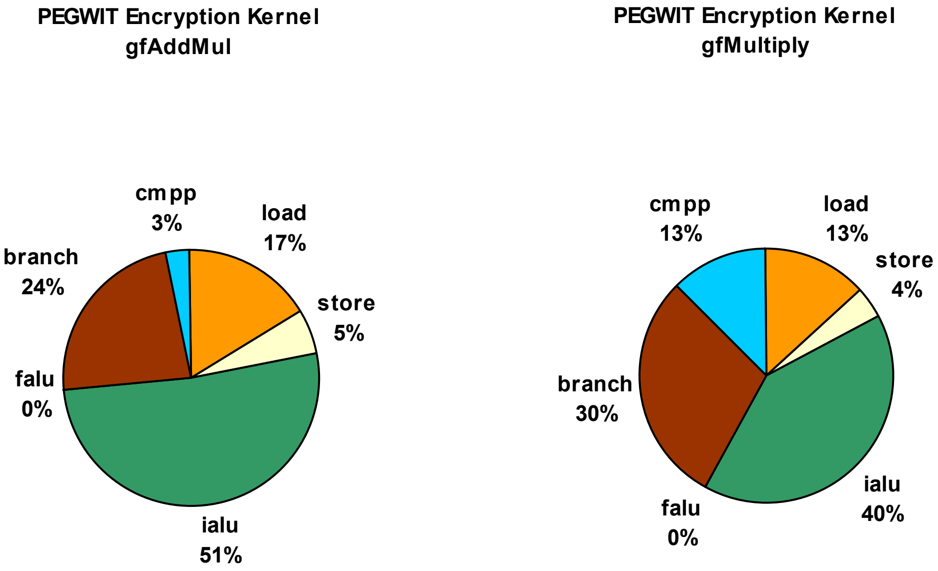


Figure 15 The dynamic instruction breakdown of the Kernels in the PEGWIT Encryption Algorithm.

The subkernels perform most computation on 16 bit operands. As shown, 51% of all operations in gfAddMul and 44% of all dynamic operations in gfMultiply are *integer alu* operations. The large percentage is due to extensive address calculation of the arrays used within the loop bodies. We notice a high percentage of *compare* and *branch* operations due to the *if* statements and *conditional expressions* within the bodies of the *for* & *while* loops.

3.5 THE PEGWIT DECRYPTION ALGORITHM

The PEGWIT algorithm[(70)] is for performing public key encryption, decryption and authentication. We invoke it to decrypt a previously encrypted ASCII file.

3.5.1 Kernel of the PEGWIT decryption algorithm

For this algorithm, the kernel consists of the same two subkernels shown in the encryption portion of this application. The first performs a GF multiply and add operation and the second performs a GF element multiplication. The two subkernels combined account for 62% of the dynamic execution cycles when decrypting a typical ascii file.

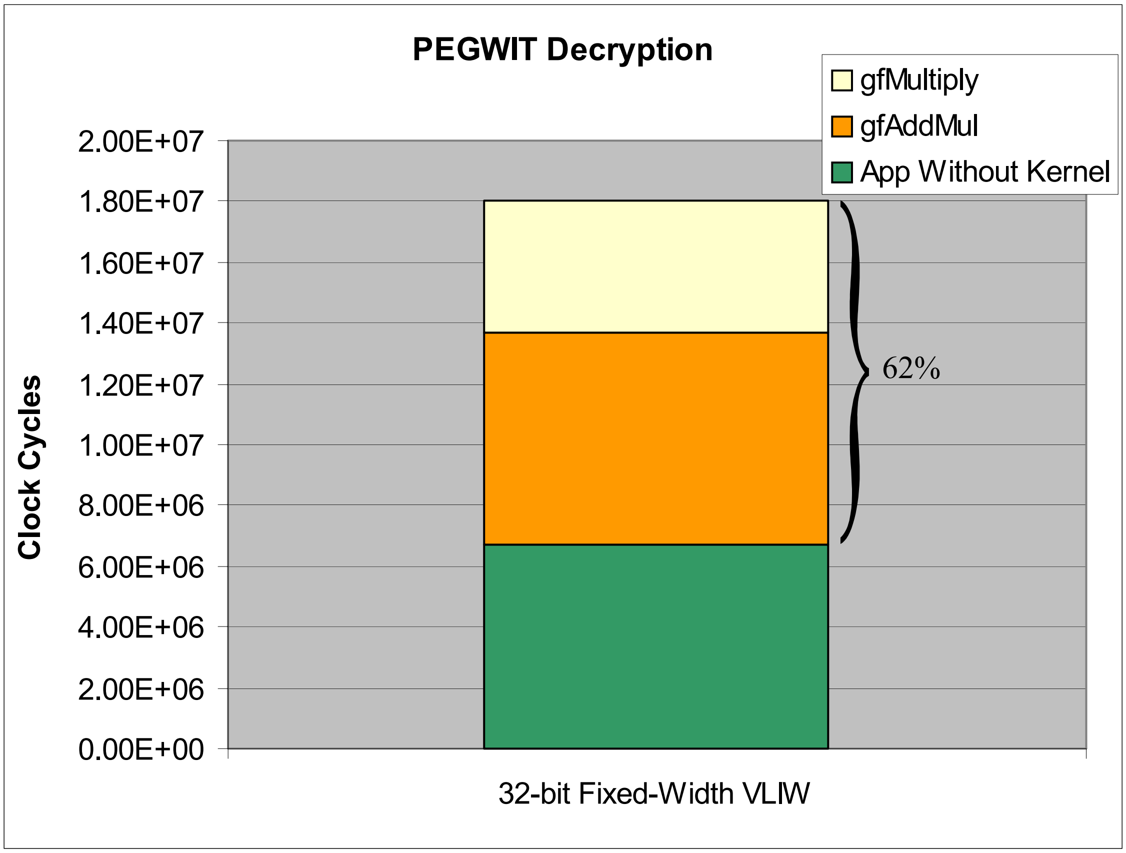


Figure 16 The kernel of the PEGWIT Decryption algorithm accounts for 62% of clock cycles during a typical execution.

The dynamic instruction distribution for the two subkernels executing on a fixed width (32-bit) VLIW processor with register file of size 64, 4 integer ALUs, and 2 memory units is shown in Figure 17:

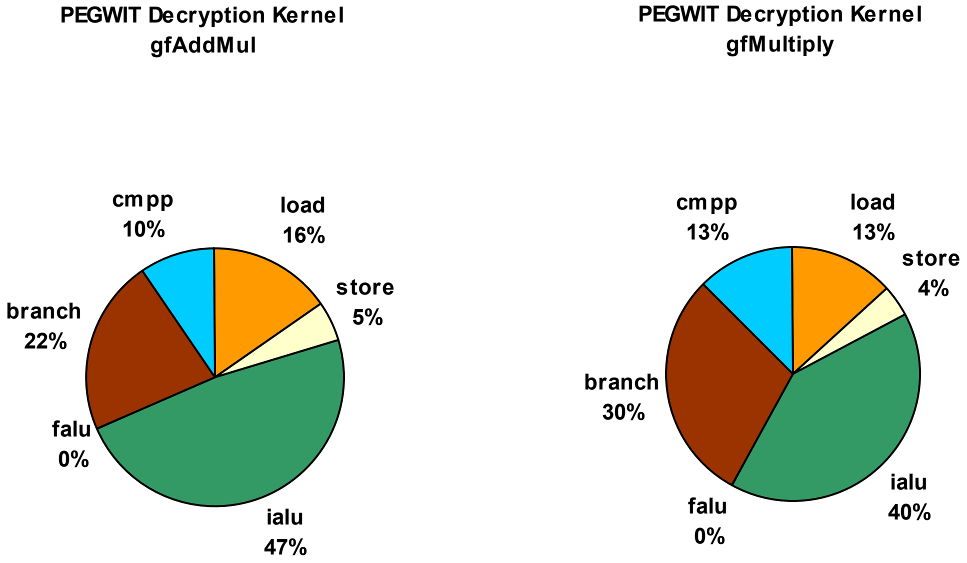


Figure 17 The dynamic instruction breakdown of the PEGWIT Decryption Kernels.

The instruction breakdown of the subkernels during decryption varies slightly from the encryption invocation. As shown, the majority of all dynamic operations in are *integer alu* operations due to address calculation and integer operations. Also, a high percentage of *compare* and *branch* operations due to the *if* statements and *conditional expressions* within the bodies of the *for* and *while* loops.

3.6 THE ADPCM ENCODER ALGORITHM

The ADPCM application[70], is an implementation of the Intel/DVI ADPCM adaptive pulse code modulation algorithm (Figure18). It is a simple adaptive differential pulse code modulation scheme utilized for compression of 16-bit PCM audio stream samples into 4-bit ADPCM samples.

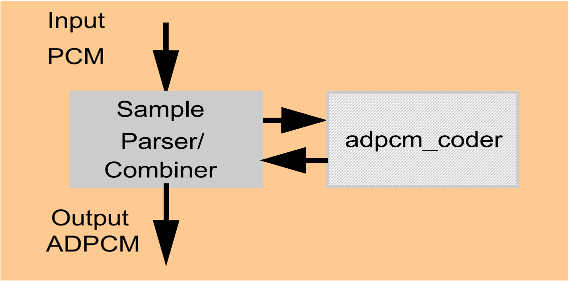


Figure 18   The block diagram of the ADPCM Encoder Algorithm.

To identify the kernels of the ADPCM Encoder, we execute the application on a typical input set and record the clock cycles of each function in the application. For this application, we identified one function that accounts for a large percentage of the total execution cycles as shown in Figure 19. This function, adpcm_coder, performs the adaptive compression of the PCM audio samples.

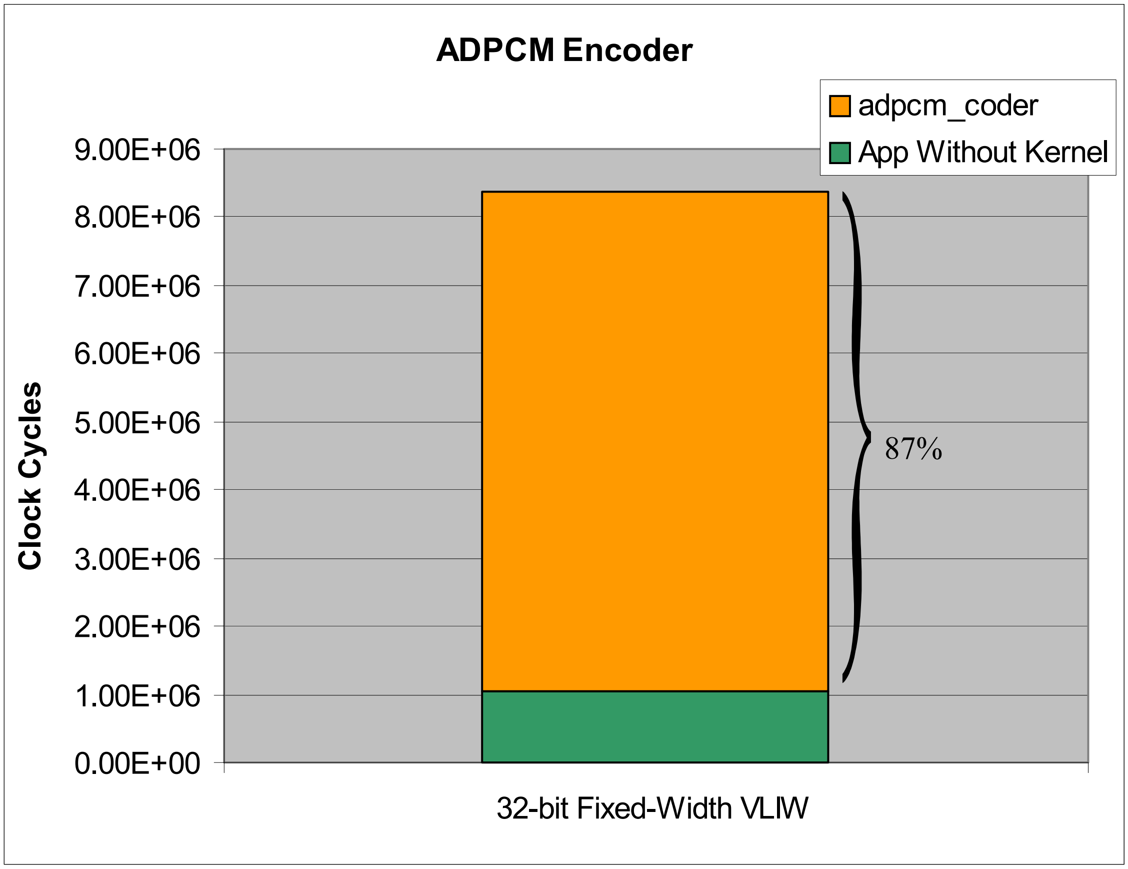


Figure 19 The adpcm_coder kernel of the ADPCM Encoder algorithm accounts for 87% of clock cycles during a typical execution.

The code of the adpcm_coder kernel is shown in Figure 20. The input data-types are 16-bit PCM samples and the output are 8-bit ADPCM samples. The control flow is a single for loop where the audio samples are compressed. The upper bound of the for loop is data-dependent and unknown. The body of the loop within this kernel consists primarily of nine if-statements with simple operations in between. Further, there exists inter-loop dependence of the previous predicted value. There is extensive data-dependence within the loop body, primarily true dependence.

```
void adpcm_coder(indata, outdata, len, state)
    short indata[]; char outdata[]; int len; struct adpcm_state *state;
{
    short *inp;                 /* Input buffer pointer */
    signed char *outp;          /* output buffer pointer */
    outp = (signed char *)outdata;
    inp = indata;
    [...] code deleted for brevity
    valpred = state->valprev;
    index = state->index;
    step = stepsizeTable[index];
    bufferstep = 1;
    for ( ; len > 0 ; len-- ) {
        val = *inp++;

        /* Step 1 - compute difference with previous value */
        diff = val - valpred;
        sign = (diff < 0) ? 8 : 0;
        if ( sign ) diff = (-diff);

        /* Step 2 - Divide and clamp */
        delta = 0;
        vpdiff = (step >> 3);

        if ( diff >= step ) {
            delta = 4;
            diff -= step;
            vpdiff += step;
        }
        step >>= 1;
        if ( diff >= step  ) {
            delta |= 2;
            diff -= step;
            vpdiff += step;
        }
        step >>= 1;
        if ( diff >= step ) {
            delta |= 1;
            vpdiff += step;
        }

        /* Step 3 - Update previous value */
        if ( sign )
          valpred -= vpdiff;
        else
          valpred += vpdiff;

        /* Step 4 - Clamp previous value to 16 bits */
        if ( valpred > 32767 )
          valpred = 32767;
        else if ( valpred < -32768 )
          valpred = -32768;

        /* Step 5 - Assemble value, update index and step values */
        delta |= sign;

        index += indexTable[delta];
        if ( index < 0 ) index = 0;
        if ( index > 88 ) index = 88;
        step = stepsizeTable[index];

        /* Step 6 - Output value */
        if ( bufferstep ) {
            outputbuffer = (delta << 4) & 0xf0;
        } else {
            *outp++ = (delta & 0x0f) | outputbuffer;
        }
        bufferstep = !bufferstep;
    }

    /* Output last step, if needed */
    if ( !bufferstep )
      *outp++ = outputbuffer;
    state->valprev = valpred;
    state->index = index;
}
```

Figure 20 The code of the adpcm_coder Kernel of the ADPCM Encoder

The dynamic instruction distribution for the adpcm_coder kernel executing on a fixed width (32-bit) VLIW processor with register file of size 64, 4 integer ALUs, and 2 memory units is shown in Figure 21.

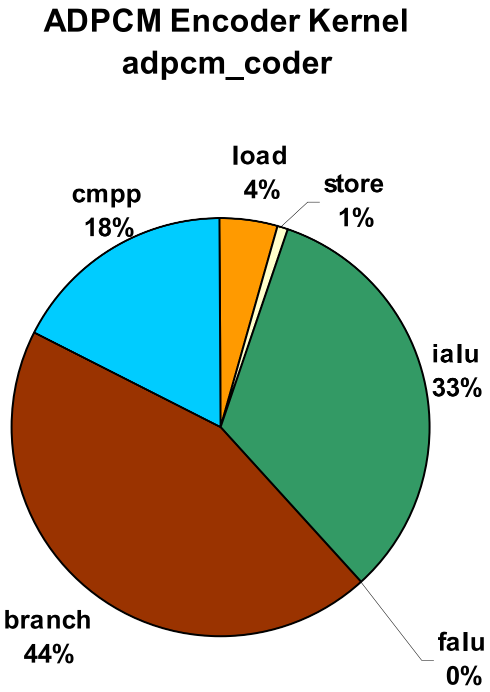


Figure 21 The dynamic instruction breakdown of adpcm_coder kernel of ADPCM

As expected, the majority of the code consists of branch operations due to all the if-statements within the loop body. The computation is performed on subword 8, 16 and 32-bit operands. As shown, 33% of all dynamic operations are *integer alu* operations.

3.7 THE ADPCM DECODER ALGORITHM

The ADPCM decoder algorithm(70), converts the 4-bit ADPCM samples back to the 16-bit PCM audio samples as shown in Figure 22.

To identify the kernels of the ADPCM decoder, we execute the application on a typical input set and record the clock cycles of each function in the application. For this application, we identi-

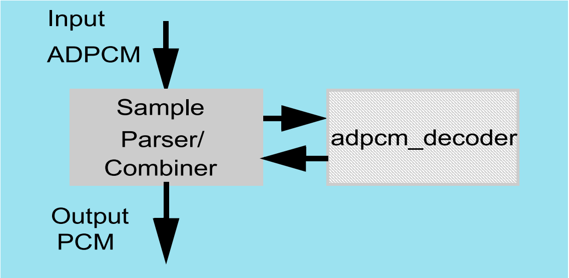


Figure 22 The block diagram of the ADPCM Decoder Algorithm.

fied one function, adpcm_decoder, that accounts for 84% of the total execution cycles as shown in Figure 23. This function, adpcm_coder, performs the adaptive compression of the PCM samples.

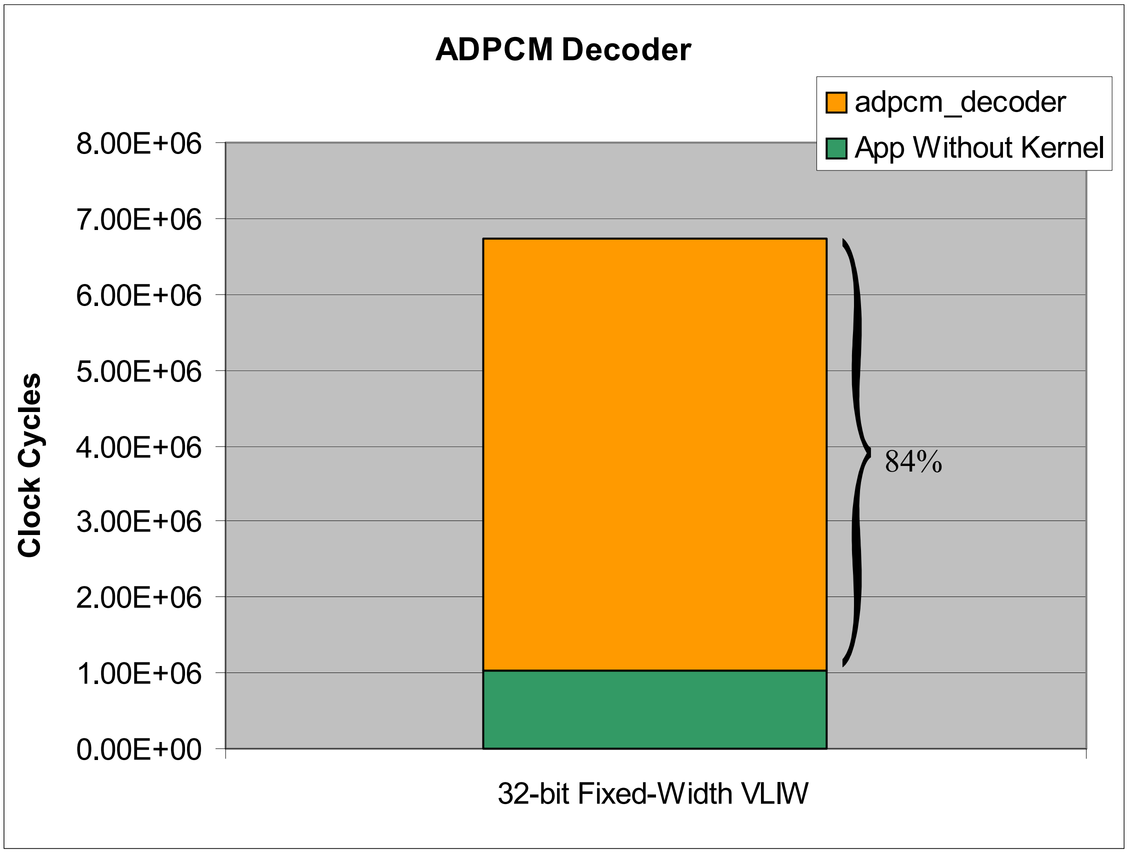


Figure 23 The adpcm_decoder kernel of the ADPCM Decoder algorithm accounts for 84% of clock cycles during a typical execution.

The code of the adpcm_coder kernel is shown in Figures 24. The input data-types are 8-bit ADPCM samples and the output are 16-bit PCM audio samples. The control flow is a single for loop where the audio samples are decompressed. The upper bound of the for loop is data-dependent and unknown. The body of the loop within this kernel consists primarily of eight if-statements with simple operations in between. Further there exists inter-loop dependence of the previous step value. There is extensive data-dependence within the loop body, primarily true dependence.

```
void adpcm_decoder(indata, outdata, len, state)
    char indata[]; short outdata[]; int len; struct adpcm_state *state;
{
    signed char *inp;           /* Input buffer pointer */
    short *outp;                /* output buffer pointer */
    [...] code deleted for brevity
    outp = outdata;
    inp = (signed char *)indata;

    valpred = state->valprev;
    index = state->index;
    step = stepsizeTable[index];
    bufferstep = 0;

    for ( ; len > 0 ; len-- ) {

        /* Step 1 - get the delta value */
        if ( bufferstep ) {
            delta = inputbuffer & 0xf;
        } else {
            inputbuffer = *inp++;
            delta = (inputbuffer >> 4) & 0xf;
        }
        bufferstep = !bufferstep;

        /* Step 2 - Find new index value (for later) */
        index += indexTable[delta];
        if ( index < 0 ) index = 0;
        if ( index > 88 ) index = 88;

        /* Step 3 - Separate sign and magnitude */
        sign = delta & 8;
        delta = delta & 7;

        /* Step 4 - Compute difference and new predicted value */
        /*
        ** Computes 'vpdiff = (delta+0.5)*step/4', but see comment
        ** in adpcm_coder.
        */
        vpdiff = step >> 3;
        if ( delta & 4 ) vpdiff += step;
        if ( delta & 2 ) vpdiff += step>>1;
        if ( delta & 1 ) vpdiff += step>>2;

        if ( sign )
          valpred -= vpdiff;
        else
          valpred += vpdiff;

        /* Step 5 - clamp output value */
        if ( valpred > 32767 )
          valpred = 32767;
        else if ( valpred < -32768 )
          valpred = -32768;

        /* Step 6 - Update step value */
        step = stepsizeTable[index];

        /* Step 7 - Output value */
        *outp++ = valpred;
    }

    state->valprev = valpred;
    state->index = index;
}
```

Figure 24 The code of the adpcm_decoder Kernel of the ADPCM Decoder.

The dynamic instruction distribution for the adpcm_coder kernel executing on a fixed width (32-bit) VLIW processor with register file of size 64, 4 integer ALUs, and 2 memory units is shown in Figure 25.

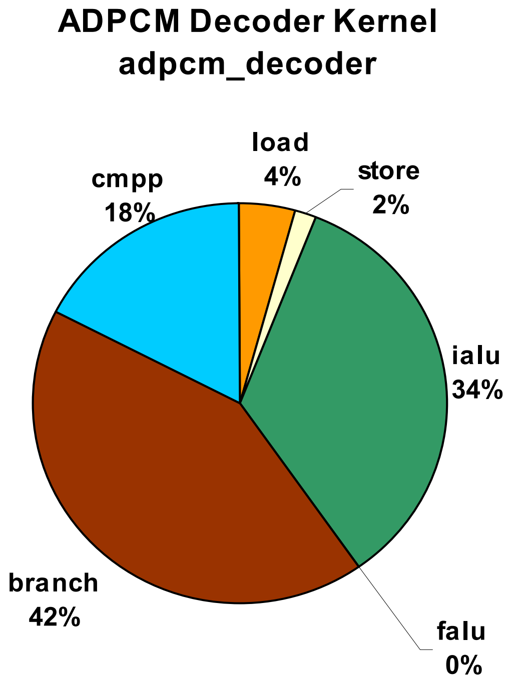


Figure 25 The dynamic instruction breakdown of adpcm_coder kernel of ADPCM

As expected, the majority, 42%, of the dynamic operations consist of branch operations due to all the if-statements within the loop body. The computation is performed on subword 8, 16 and 32-bit operands. As shown, 34% of all dynamic operations are *integer alu* operations.

3.8 THE MPEG-2 ENCODING ALGORITHM

The MPEG-2 algorithm, is based on the mpeg2encode algorithm$^{(70)}$ by the MPEG Software Simulation Group. MPEG-2 is a video sequence coding and decoding method.

The MPEG-2 compression (Figure 26) is performed using the following prediction techniques, motion estimation in the encoder and motion compensation in the decoder. The residual is calculated by subtracting the predicted video frame from the actual frame. The residual is encoded using the spatial 2 dimensional discrete cosine transform (DCT) which is performed on 8x8 blocks of pixels followed by the quantization of DCT coefficients. Finally, the quantized DCT coefficients are coded using Huffman and run/level encoding which is combined with the motion estimation vector to produce the encoded output. A stream of video frames is encoded into three types of frames, I, P and B frames. The I frames are encoded without any prediction. The P frames are encoded using prediction from previous frames. While the B frames are encoded using prediction from both previous frames and subsequent frames.

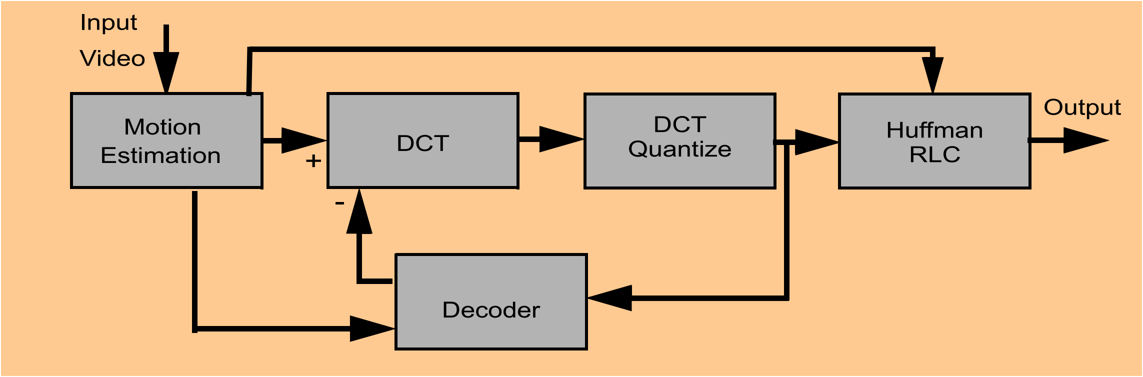


Figure 26 The block diagram of the MPEG-2 Encoder Algorithm.

To identify the kernels of the MPEG-2 Encoder, we execute the application on a typical input set and record the clock cycles of each function in the application. For this application, we identified one function that accounts for a large percentage of the total execution cycles as shown in Figure 27. This function, dist1, performs the calculation of the motion vector of a pixel block between two video frames.

The code of the dist1 kernel is shown in Figures 28 and 29. Depending on the input values to the function dist1, one of four possible for loops are invoked to calculate the difference between two pixel blocks. All four loops have a nested for-loop where the upper bound of the outer loop is data dependent while the upper bound of the inner loop is known. Also, the inner loop of the first for-loop is unrolled.

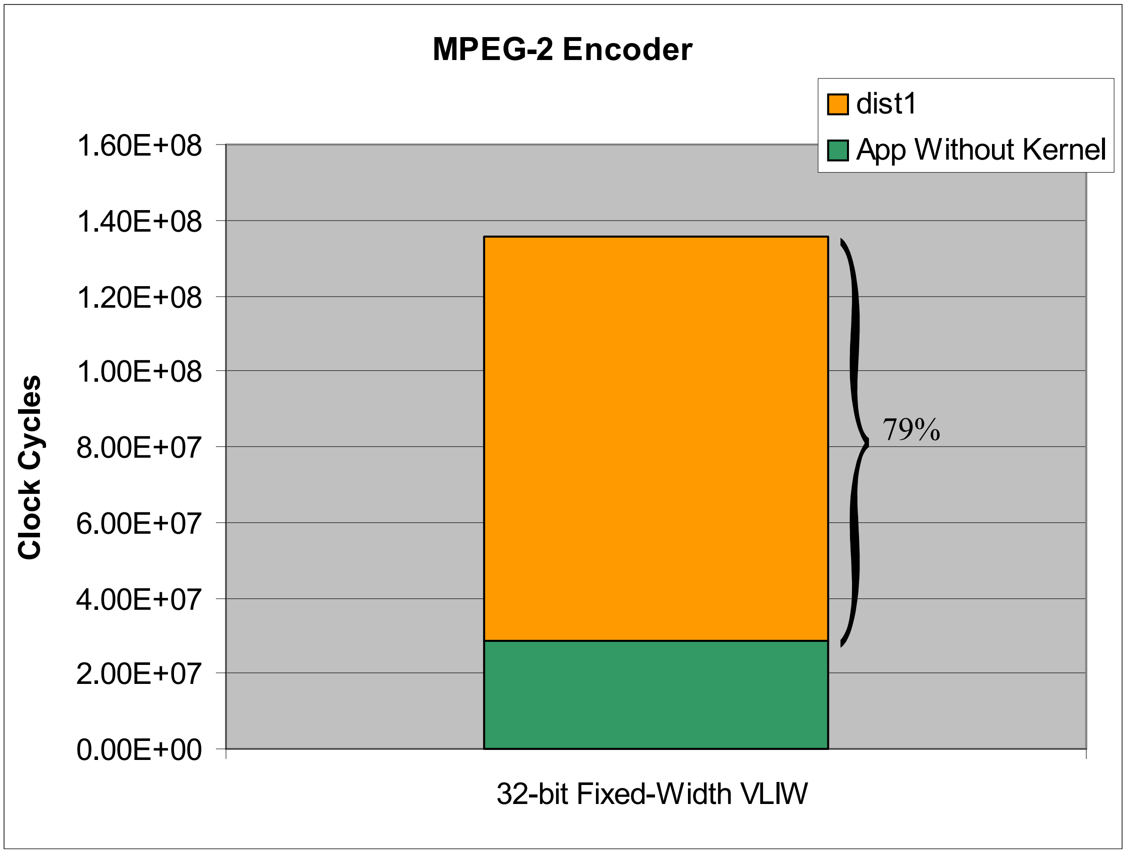


Figure 27 The dist1 kernel of the MPEG-2 Encoder algorithm accounts for 79% of clock cycles during a typical execution.

The calculation within the first inner loop is a sequence of if-statements performing the sum of absolute difference between two pixel blocks. The subtraction is between two 8-bit variables while the addition goes into a 32-bit accumulator. Besides the output dependence (write after write) between every statement in the inner loop, the code is completely independent.

```
static int dist1(blk1,blk2,lx,hx,hy,h,distlim)
unsigned char *blk1,*blk2;
int lx,hx,hy,h;
int distlim;
{
  unsigned char *p1,*p1a,*p2;
  int i,j;
  int s,v;

  s = 0;
  p1 = blk1;
  p2 = blk2;

  if (!hx && !hy)
    for (j=0; j<h; j++)
    {
      if ((v = p1[0]  - p2[0])<0)  v = -v; s+= v;
      if ((v = p1[1]  - p2[1])<0)  v = -v; s+= v;
      if ((v = p1[2]  - p2[2])<0)  v = -v; s+= v;
      if ((v = p1[3]  - p2[3])<0)  v = -v; s+= v;
      if ((v = p1[4]  - p2[4])<0)  v = -v; s+= v;
      if ((v = p1[5]  - p2[5])<0)  v = -v; s+= v;
      if ((v = p1[6]  - p2[6])<0)  v = -v; s+= v;
      if ((v = p1[7]  - p2[7])<0)  v = -v; s+= v;
      if ((v = p1[8]  - p2[8])<0)  v = -v; s+= v;
      if ((v = p1[9]  - p2[9])<0)  v = -v; s+= v;
      if ((v = p1[10] - p2[10])<0) v = -v; s+= v;
      if ((v = p1[11] - p2[11])<0) v = -v; s+= v;
      if ((v = p1[12] - p2[12])<0) v = -v; s+= v;
      if ((v = p1[13] - p2[13])<0) v = -v; s+= v;
      if ((v = p1[14] - p2[14])<0) v = -v; s+= v;
      if ((v = p1[15] - p2[15])<0) v = -v; s+= v;

      if (s >= distlim)
        break;

      p1+= lx;
      p2+= lx;
    }
[...] /* continued in the next figure */
```

Figure 28 The code of the dist1 Kernel of the MPEG-2 Encoder

The rest of the loops are not unrolled and they perform the same operation discussed above, however, using a for-loop structure. There is no inter-loop dependence besides the accumulator *s*.

```
 [...] /* continued from the previous figure */
 else if (hx && !hy)
    for (j=0; j<h; j++)
    {
      for (i=0; i<16; i++)
      {
        v = ((unsigned int)(p1[i]+p1[i+1]+1)>>1) - p2[i];
        if (v>=0)
          s+= v;
        else
          s-= v;
      }
      p1+= lx;
      p2+= lx;
    }
  else if (!hx && hy)
  {
    p1a = p1 + lx;
    for (j=0; j<h; j++)
    {
      for (i=0; i<16; i++)
      {
        v = ((unsigned int)(p1[i]+p1a[i]+1)>>1) - p2[i];
        if (v>=0)
          s+= v;
        else
          s-= v;
      }
      p1 = p1a;
      p1a+= lx;
      p2+= lx;
    }
  }
  else /* if (hx && hy) */
  {
    p1a = p1 + lx;
    for (j=0; j<h; j++)
    {
      for (i=0; i<16; i++)
      {
        v = ((unsigned int)(p1[i]+p1[i+1]+p1a[i]+p1a[i+1]+2)>>2) - p2[i];
        if (v>=0)
          s+= v;
        else
          s-= v;
      }
      p1 = p1a;
      p1a+= lx;
      p2+= lx;
    }
  }

  return s;
}
```

Figure 29 The continuation of the dist1 Kernel of the MPEG-2 Encoder

The dynamic instruction distribution for the dist1 kernel executing on a fixed width (32-bit) VLIW processor with register file of size 64, 4 integer ALUs, and 2 memory units is shown in Figure 30.

As expected, the majority of the code consists of integer alu operations. The computation is performed on subword 8 bit operands. As shown, 46% of all dynamic operations are *integer alu* operations. The *compare* and *branch* operations are due to the *if* statements within the nested *for* loops.

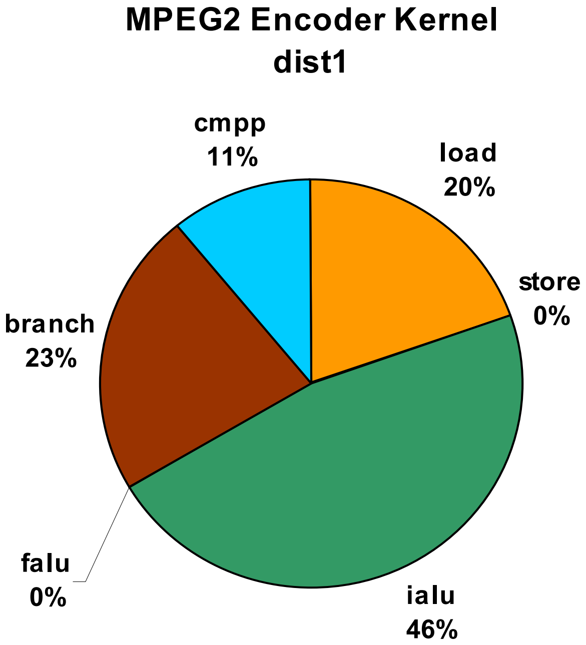


Figure 30 The dynamic instruction breakdown of dist1, the MPEG Encoder Kerne

3.9 THE MPEG-2 DECODING ALGORITHM

The MPEG-2 algorithm, is based on the mpeg2decode algorithm by the MPEG Software Simulation Group. The MPEG-2 decoder algorithm (Figure 31) receives the MPEG-2 coded data and decodes it using the Huffman decoder. The first output, the motion vectors, are sent to the motion compensation predictor. The quantized DCT coefficients are sent through the inverse quantizer

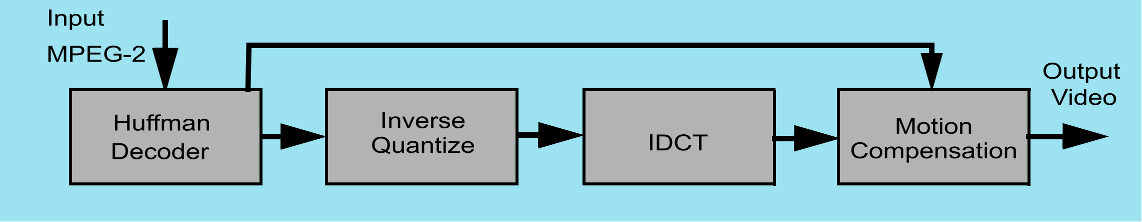


Figure 31 The block diagram of the MPEG-2 Decoder Algorithm.

and then the IDCT, the inverse discrete cosine transform, to get the residual signal which is added onto the predicted signal to produce the video frames.

There are two versions of the inverse discrete cosine transform that can be used, the high precision floating point reference version and the fast, integer only computation, lower precision version. We invoke the integer-only version of the transform.

3.9.1 Kernel of the MPEG-2 decoding algorithm

To identify the kernels of the MPEG-2 Decoder(70), we execute the application on a typical input set and record the clock cycles of each function in the application. For this application, we identified two functions that account for large percentage of the total execution cycles as shown in Figure 32.

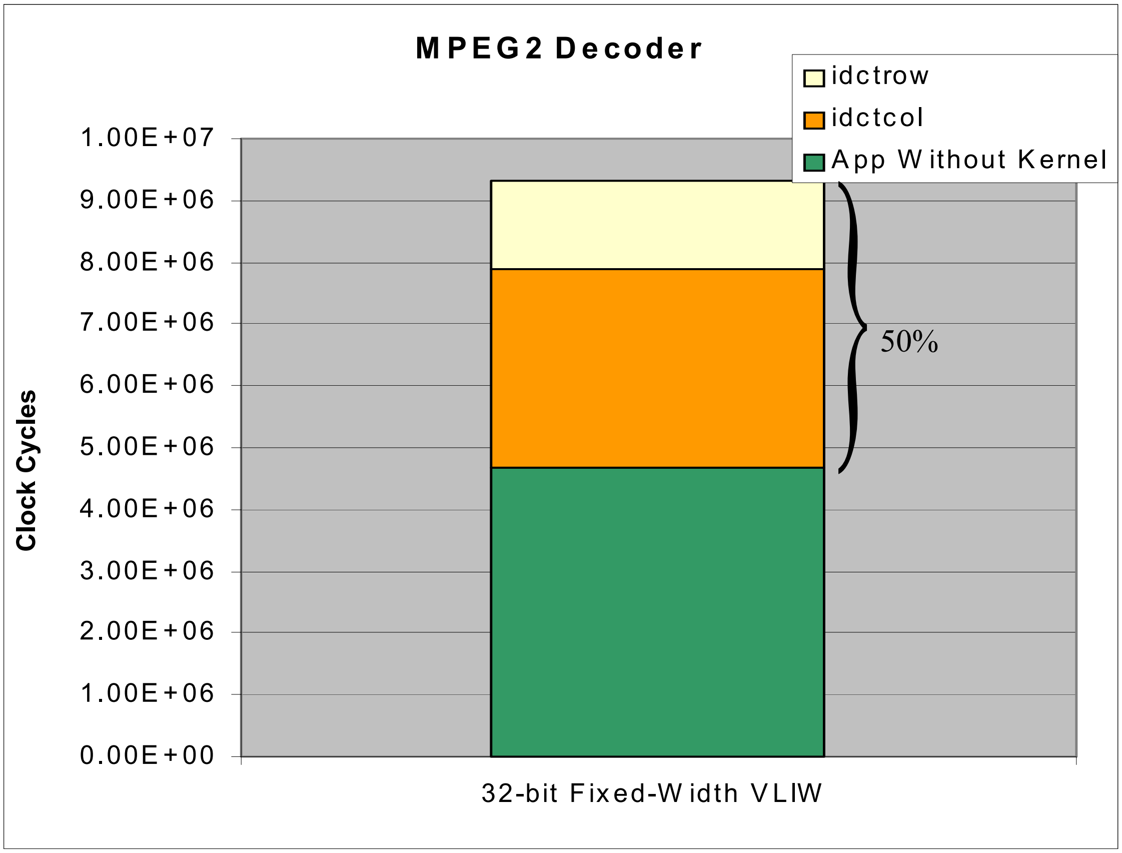


Figure 32 The kernel of the MPEG-2 Decoder algorithm accounts for 50% of clock cycles during a typical execution.

The two code kernels that perform the fast IDCT algorithm, a two dimensional inverse discrete cosine transform, are the idctrow and the idctcol. The IDCT algorithm invokes these two functions eight times in the body of a single *for* loop.

The code of the idctcol is shown in Figure 33. It consists of operations on 8-bit and 32-bit operands. The control flow within the function is an if-statement.

```
static void idctcol(blk)
short *blk;
{
  int x0, x1, x2, x3, x4, x5, x6, x7, x8;

  /* shortcut */
  if (!((x1 = (blk[8*4]<<8)) | (x2 = blk[8*6]) | (x3 = blk[8*2]) |
        (x4 = blk[8*1]) | (x5 = blk[8*7]) | (x6 = blk[8*5]) | (x7 =
         blk[8*3])))
  {
blk[8*0]=blk[8*1]=blk[8*2]=blk[8*3]=blk[8*4]=blk[8*5]=blk[8*6]=blk[8*7]=
      iclp[(blk[8*0]+32)>>6];
    return;
  }

  x0 = (blk[8*0]<<8) + 8192;

  /* first stage */
  x8 = W7*(x4+x5) + 4;
  x4 = (x8+(W1-W7)*x4)>>3;
  x5 = (x8-(W1+W7)*x5)>>3;
  x8 = W3*(x6+x7) + 4;
  x6 = (x8-(W3-W5)*x6)>>3;
  x7 = (x8-(W3+W5)*x7)>>3;

  /* second stage */
  x8 = x0 + x1;
  x0 -= x1;
  x1 = W6*(x3+x2) + 4;
  x2 = (x1-(W2+W6)*x2)>>3;
  x3 = (x1+(W2-W6)*x3)>>3;
  x1 = x4 + x6;
  x4 -= x6;
  x6 = x5 + x7;
  x5 -= x7;

  /* third stage */
  x7 = x8 + x3;
  x8 -= x3;
  x3 = x0 + x2;
  x0 -= x2;
  x2 = (181*(x4+x5)+128)>>8;
  x4 = (181*(x4-x5)+128)>>8;

  /* fourth stage */
  blk[8*0] = iclp[(x7+x1)>>14];
  blk[8*1] = iclp[(x3+x2)>>14];
  blk[8*2] = iclp[(x0+x4)>>14];
  blk[8*3] = iclp[(x8+x6)>>14];
  blk[8*4] = iclp[(x8-x6)>>14];
  blk[8*5] = iclp[(x0-x4)>>14];
  blk[8*6] = iclp[(x3-x2)>>14];
  blk[8*7] = iclp[(x7-x1)>>14];
}
```

34%
of application
clock cycles

are spent in this
function

Figure 33   The code of the idctcol Kernel of the MPEG-2 Decoder

The code structure of the idctrow kernel is shown in Figure 34.

```
static void idctrow(blk)
short *blk;
{
  int x0, x1, x2, x3, x4, x5, x6, x7, x8;

  /* shortcut */
  if (!((x1 = blk[4]<<11) | (x2 = blk[6]) | (x3 = blk[2]) |
        (x4 = blk[1]) | (x5 = blk[7]) | (x6 = blk[5]) | (x7 = blk[3])))
  {
    blk[0]=blk[1]=blk[2]=blk[3]=blk[4]=blk[5]=blk[6]=blk[7]=blk[0]<<3;
    return;
  }

  x0 = (blk[0]<<11) + 128; /* for proper rounding in the fourth stage */

  /* first stage */
  x8 = W7*(x4+x5);
  x4 = x8 + (W1-W7)*x4;
  x5 = x8 - (W1+W7)*x5;
  x8 = W3*(x6+x7);
  x6 = x8 - (W3-W5)*x6;
  x7 = x8 - (W3+W5)*x7;

  /* second stage */
  x8 = x0 + x1;
  x0 -= x1;
  x1 = W6*(x3+x2);
  x2 = x1 - (W2+W6)*x2;
  x3 = x1 + (W2-W6)*x3;
  x1 = x4 + x6;
  x4 -= x6;
  x6 = x5 + x7;
  x5 -= x7;

  /* third stage */
  x7 = x8 + x3;
  x8 -= x3;
  x3 = x0 + x2;
  x0 -= x2;
  x2 = (181*(x4+x5)+128)>>8;
  x4 = (181*(x4-x5)+128)>>8;
/* fourth stage */
  blk[0] = (x7+x1)>>8;
  blk[1] = (x3+x2)>>8;
  blk[2] = (x0+x4)>>8;
  blk[3] = (x8+x6)>>8;
  blk[4] = (x8-x6)>>8;
  blk[5] = (x0-x4)>>8;
  blk[6] = (x3-x2)>>8;
  blk[7] = (x7-x1)>>8;
}
```

16%
of application
clock cycles

are spent in this
function

Figure 34 The code of the idctrow Kernel of the MPEG-2 Decoder

Although the code of idctrow and idctcol is similar, idctrow requires fewer clock cycles because the probability of going though the shortcut code is much higher, and hence, less code is executed.

The dynamic instruction distribution for the fast idct executing on a fixed width (32-bit) VLIW processor with register file of size 64, 4 integer ALUs, and 2 memory units is shown in Figure 35.

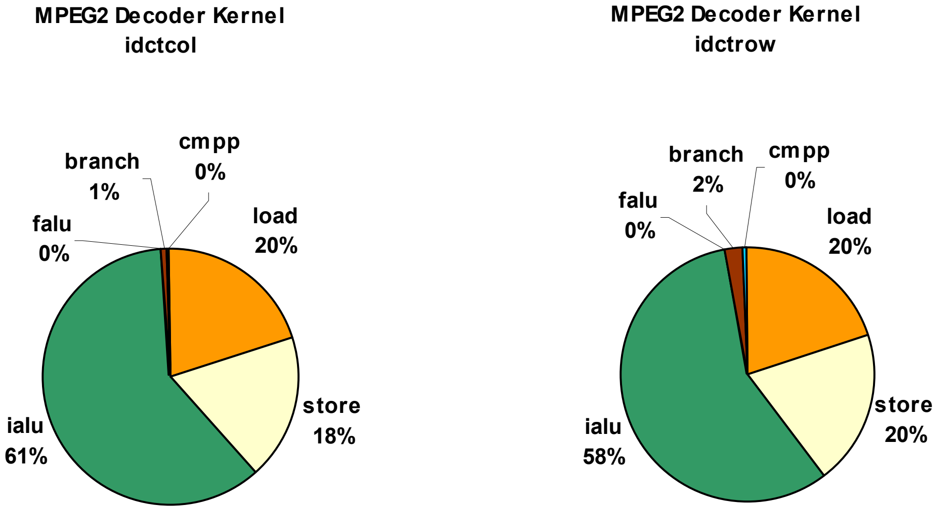


Figure 35 The dynamic instruction breakdown of the MPEG Decoding Kernel

As expected, the majority of the code consists of integer alu operations. The computation is performed on 8 bit operands as well as 32-bit operands. As shown, 61% of all dynamic operations of idctcol and 58% of idctrow are *integer alu* operations. We notice a low percentage of *compare* and *branch* operations due to the single *if* statement within both kernels.

3.10 THE MPEG-4 (DIVX) ENCODER ALGORITHM

MPEG-4 has become one of the dominant standards in multimedia applications. The MPEG-4 video standard is an object based hybrid natural/synthetic coding standard that enables efficient compression and content-based interactivity, such as object manipulation, scaling and bitstream editing. A block diagram depicting the algorithm of the MPEG-4 encoder is shown in Figure 36.

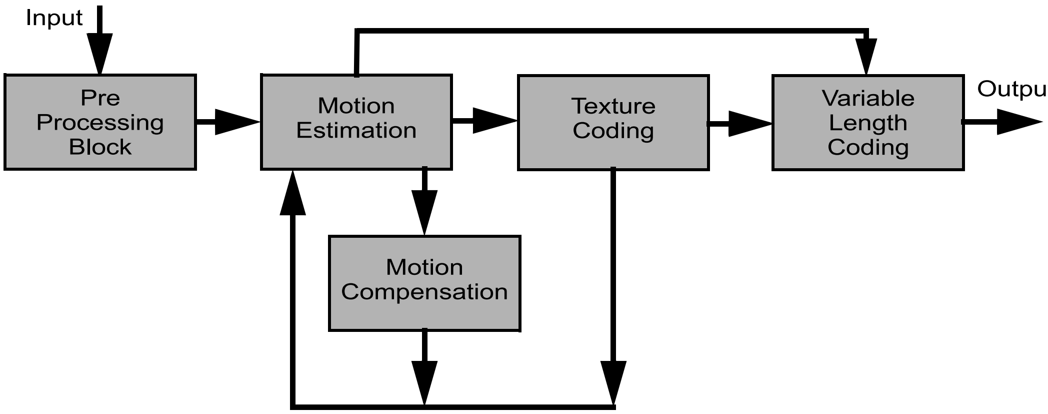


Figure 36 The block diagram of the MPEG-4 Encoder Algorithm.

In almost all video compression standards, including the MPEG-4 visual part, the block-matching motion estimation algorithm is the dominant part of the computation load. In a typical video system, motion estimation requires 80% of the computation for the entire encoding application. Motion estimation is performed as a search to find the best matching position of a pixel block in a video frame with minimum distortion by evaluating the distortion measure between non-overlapping 16x16 pixel blocks, called macroblocks, across frames. For each macroblock, the motion vector is generated by finding the macroblock with minimum distortion between the current and previous frames.

Distortion between two macroblocks is calculated using the sum of absolute differences (SAD):

$$SAD_N = \sum_{i,j}^{N} |original - previous| ; \quad N = 16$$

where N is the dimension of the macroblock and in our case is equal to 16 pixels. In the implementation of this algorithm used for this study, there are three slight variants of the distortion evaluation in addition to the SAD function depicted above. These functions use the pixel average of several macroblocks in the *previous* frame. Since the code and control structure of these functions are similar, in the next subsection, we illustrate and discuss the SAD function described above in detail.

3.10.1 Kernel of motion estimation in the MPEG-4 encoding algorithm

The code kernel for this application is the motion estimation algorithm. There have been many motion estimation algorithm implementations, which usually trade-off precision for computation load. A full search algorithm for the motion vector exhibiting the least distortion between the current object frame and reference frame is the most computationally expensive, however, it produces the most accurate motion vector and, hence, the most efficient compression ratios. The dynamic clock cycle breakdown for this application is shown in Figure 37.

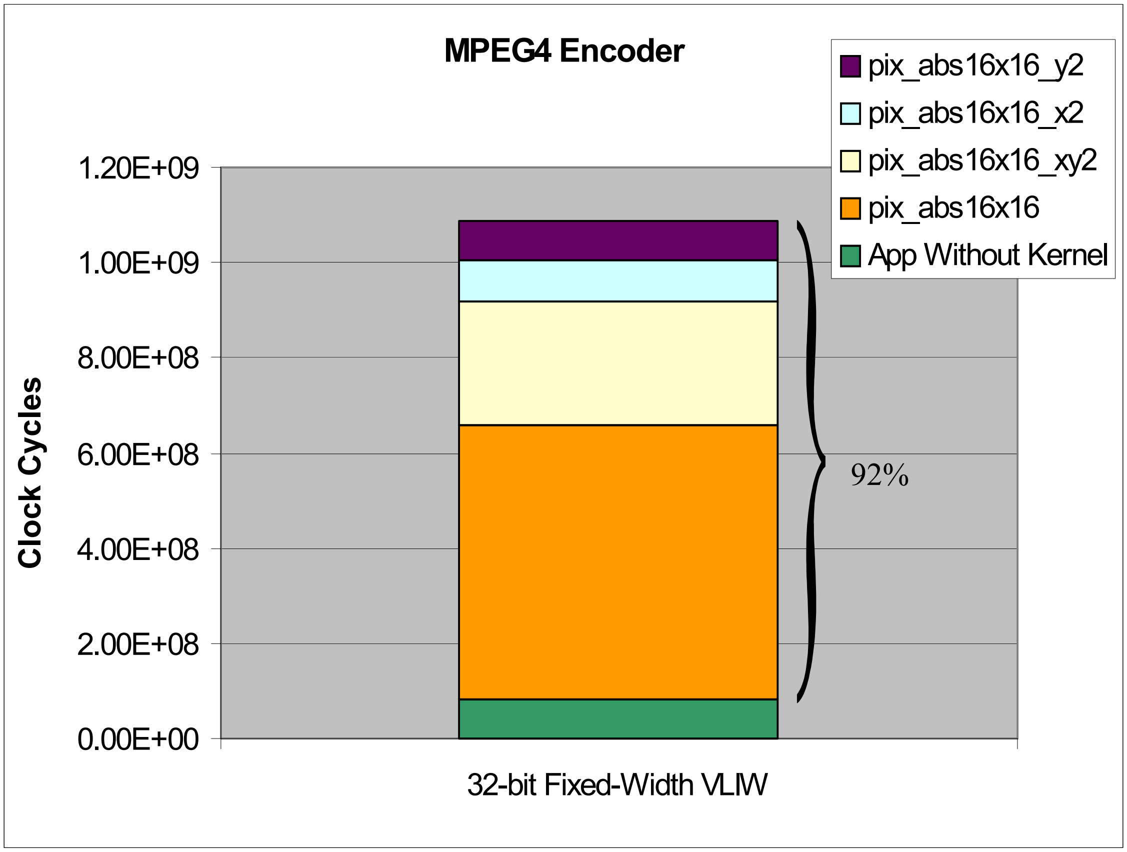


Figure 37 The kernel of the MPEG4 Encoder algorithm accounts for 92% of clock cycles during a typical execution.

The implementation of the MPEG-4 decoder used in this study uses the log-search algorithm and the distortion function shown below. There are four distinct searches that are used in the log-search implementation.

The code structure of the first kernel, which is the most significant during execution, is shown in Figure 38. The operation lies within a single *for* loop body. The operations performed are the sum of absolute differences, which requires subtracting two 8-bit values, finding the absolute value of the result and adding it to a 32-bit accumulator. The upper bound of the for-loop is data dependent and unknown. The instructions within the body of the loop have true dependence (read after write) and an output dependence (write after write) between the variable *s* and itself in subsequent instructions. The subtraction operations are parallel and independent.

```
int pix_abs16x16_c(UINT8 *pix1, UINT8 *pix2, int line_size, int h)
{
    int s, i;

    s = 0;
    for(i=0;i<h;i++) {
        s += abs(pix1[0] - pix2[0]);
        s += abs(pix1[1] - pix2[1]);
        s += abs(pix1[2] - pix2[2]);
        s += abs(pix1[3] - pix2[3]);
        s += abs(pix1[4] - pix2[4]);
        s += abs(pix1[5] - pix2[5]);
        s += abs(pix1[6] - pix2[6]);
        s += abs(pix1[7] - pix2[7]);
        s += abs(pix1[8] - pix2[8]);
        s += abs(pix1[9] - pix2[9]);
        s += abs(pix1[10] - pix2[10]);
        s += abs(pix1[11] - pix2[11]);
        s += abs(pix1[12] - pix2[12]);
        s += abs(pix1[13] - pix2[13]);
        s += abs(pix1[14] - pix2[14]);
        s += abs(pix1[15] - pix2[15]);
        pix1 += line_size;
        pix2 += line_size;
    }
    return s;
}
```

53%
of application
clock cycles

is spent in this
function

Figure 38 The code of the pix_abs16x16 kernel of the MPEG-4 Encoder Algorithm.

The code for the second kernel, pix_abs16x16_xy2 is shown below in Figure 39. It is very similar to the code of the first kernel, however, before the subtraction operation, the average of four 8-bit values is calculated.

```
int pix_abs16x16_xy2_c(UINT8 *pix1, UINT8 *pix2, int line_size, int h)
{
    int s, i;
    UINT8 *pix3 = pix2 + line_size;

    s = 0;
    for(i=0;i<h;i++) {
        s += abs(pix1[0] - avg4(pix2[0], pix2[1], pix3[0], pix3[1]));
        s += abs(pix1[1] - avg4(pix2[1], pix2[2], pix3[1], pix3[2]));
        s += abs(pix1[2] - avg4(pix2[2], pix2[3], pix3[2], pix3[3]));
        s += abs(pix1[3] - avg4(pix2[3], pix2[4], pix3[3], pix3[4]));
        s += abs(pix1[4] - avg4(pix2[4], pix2[5], pix3[4], pix3[5]));
        s += abs(pix1[5] - avg4(pix2[5], pix2[6], pix3[5], pix3[6]));
        s += abs(pix1[6] - avg4(pix2[6], pix2[7], pix3[6], pix3[7]));
        s += abs(pix1[7] - avg4(pix2[7], pix2[8], pix3[7], pix3[8]));
        s += abs(pix1[8] - avg4(pix2[8], pix2[9], pix3[8], pix3[9]));
        s += abs(pix1[9] - avg4(pix2[9], pix2[10], pix3[9], pix3[10]));
        s += abs(pix1[10] - avg4(pix2[10], pix2[11], pix3[10], pix3[11]));
        s += abs(pix1[11] - avg4(pix2[11], pix2[12], pix3[11], pix3[12]));
        s += abs(pix1[12] - avg4(pix2[12], pix2[13], pix3[12], pix3[13]));
        s += abs(pix1[13] - avg4(pix2[13], pix2[14], pix3[13], pix3[14]));
        s += abs(pix1[14] - avg4(pix2[14], pix2[15], pix3[14], pix3[15]));
        s += abs(pix1[15] - avg4(pix2[15], pix2[16], pix3[15], pix3[16]));
        pix1 += line_size;
        pix2 += line_size;
        pix3 += line_size;
    }
    return s;
}
```

24%
of application
clock cycles

is spent in this
function

Figure 39 The code of the pix_abs16x16_xy2 kernel of the MPEG-4 Encoder.

The code for the third kernel, pix_abs16x16_x2 is shown below in Figure 40. It is similar to the code in the first kernel. However, before the subtraction, the average of two 8-bit values is performed.

```
int pix_abs16x16_x2_c(UINT8 *pix1, UINT8 *pix2, int line_size, int h)
{
    int s, i;

    s = 0;
    for(i=0;i<h;i++) {
        s += abs(pix1[0] - avg2(pix2[0], pix2[1]));
        s += abs(pix1[1] - avg2(pix2[1], pix2[2]));
        s += abs(pix1[2] - avg2(pix2[2], pix2[3]));
        s += abs(pix1[3] - avg2(pix2[3], pix2[4]));
        s += abs(pix1[4] - avg2(pix2[4], pix2[5]));
        s += abs(pix1[5] - avg2(pix2[5], pix2[6]));
        s += abs(pix1[6] - avg2(pix2[6], pix2[7]));
        s += abs(pix1[7] - avg2(pix2[7], pix2[8]));
        s += abs(pix1[8] - avg2(pix2[8], pix2[9]));
        s += abs(pix1[9] - avg2(pix2[9], pix2[10]));
        s += abs(pix1[10] - avg2(pix2[10], pix2[11]));
        s += abs(pix1[11] - avg2(pix2[11], pix2[12]));
        s += abs(pix1[12] - avg2(pix2[12], pix2[13]));
        s += abs(pix1[13] - avg2(pix2[13], pix2[14]));
        s += abs(pix1[14] - avg2(pix2[14], pix2[15]));
        s += abs(pix1[15] - avg2(pix2[15], pix2[16]));
        pix1 += line_size;
        pix2 += line_size;
    }
    return s;
}
```

8%
of application
clock cycles

is spent in this
function

Figure 40 The code of the pix_abs16x16_x2 kernel of the MPEG-4 Encoder.

Finally, the code for the fourth kernel, pix_abs16x16_y2 is shown below in Figure 41. The computation is the sum of absolute difference between two 8-bit values which is added into a 32-bit accumulator.

```
int pix_abs16x16_y2_c(UINT8 *pix1, UINT8 *pix2, int line_size, int h)
{
    int s, i;
    UINT8 *pix3 = pix2 + line_size;

    s = 0;
    for(i=0;i<h;i++) {
        s += abs(pix1[0] - avg2(pix2[0], pix3[0]));
        s += abs(pix1[1] - avg2(pix2[1], pix3[1]));
        s += abs(pix1[2] - avg2(pix2[2], pix3[2]));
        s += abs(pix1[3] - avg2(pix2[3], pix3[3]));
        s += abs(pix1[4] - avg2(pix2[4], pix3[4]));
        s += abs(pix1[5] - avg2(pix2[5], pix3[5]));
        s += abs(pix1[6] - avg2(pix2[6], pix3[6]));
        s += abs(pix1[7] - avg2(pix2[7], pix3[7]));
        s += abs(pix1[8] - avg2(pix2[8], pix3[8]));
        s += abs(pix1[9] - avg2(pix2[9], pix3[9]));
        s += abs(pix1[10] - avg2(pix2[10], pix3[10]));
        s += abs(pix1[11] - avg2(pix2[11], pix3[11]));
        s += abs(pix1[12] - avg2(pix2[12], pix3[12]));
        s += abs(pix1[13] - avg2(pix2[13], pix3[13]));
        s += abs(pix1[14] - avg2(pix2[14], pix3[14]));
        s += abs(pix1[15] - avg2(pix2[15], pix3[15]));
        pix1 += line_size;
        pix2 += line_size;
        pix3 += line_size;
    }
    return s;
}
```

8%
of application
clock cycles

is spent in this
function

Figure 41 The code of the pix_abs16x16_y2 kernel of the MPEG-4 Encoder.

As shown, these kernels exhibit a lot of parallelism. All the average operations are independent, as are all the subtraction operations. There is a true dependence between each average operation and each subtraction. There is a true dependence and an output dependence between the accumulator *s* and itself in subsequent instructions. The upper bound of the loop is data dependent and unknown.

The dynamic instruction distribution for the SAD distortion computation between two 16x16 pixel macroblocks executing on a fixed width (32-bit) VLIW processor with register file of size 64, 4 integer ALUs, and 2 memory units is shown in the Figure 42.

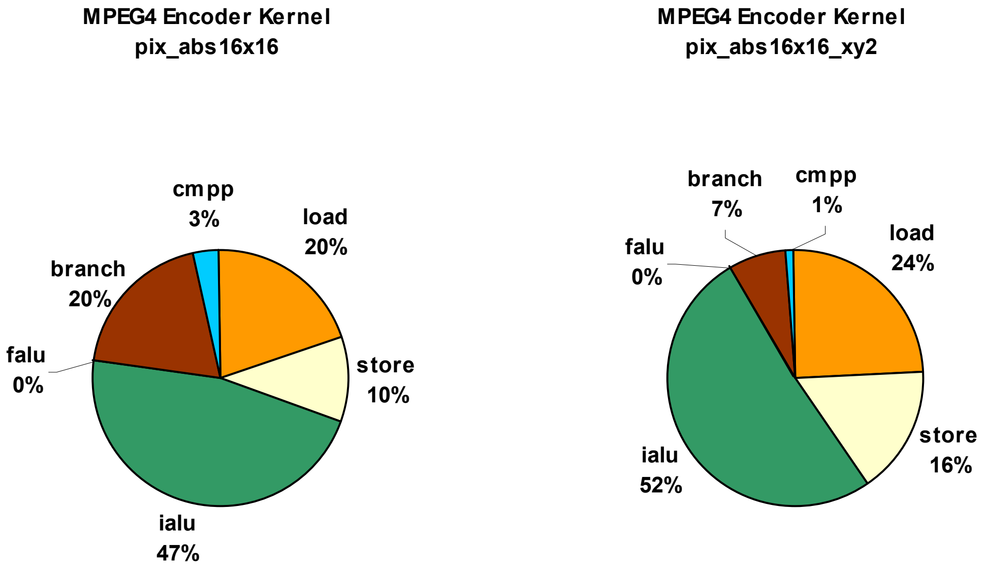


Figure 42 The dynamic instruction breakdown of functions pix_abs16x16 and pix_abs16x16_xy2 of the MPEG-4 Encoder Kernel

The dynamic instruction distribution for the SAD distortion computation between two 16x16 pixel macroblocks executing on a fixed width (32-bit) VLIW processor with register file of size 64, 4 integer ALUs, and 2 memory units is shown in the Figure 43.

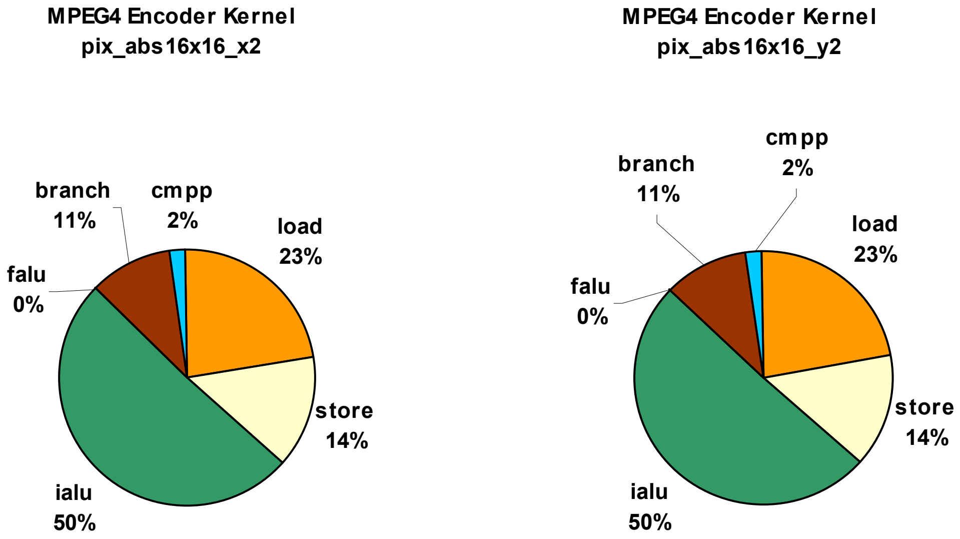


Figure 43 The dynamic instruction breakdown of functions pix_abs16x16_x2 and pix_abs16x16_y2 of the MPEG-4 Encoder Kernel

The instruction breakdown of all four kernels are very similar to one another. There is a large percentage of loads/stores, however, the kernels do load the pixel values but do not store anything but the overall result. Hence, we hypothesize that due to register pressure on the 64 registers in the register file, many registers need to be reused causing a large number of stores. Furthermore, the significant percentage of branch instructions are due to the if-statements within and *abs* operation. Finally, close to 50% of all operations are integer ALU for all four kernels.

3.11 CHARACTERIZATION SUMMARY OF MULTIMEDIA KERNELS

The characteristics of the kernels of the nine representative applications chosen and analyzed in the previous sections are summarized in Table 5. As was discussed above, many of these kernels consist of a single or nested loops, where some have known bounds.

Table 5 Characteristics of the multimedia kernels

Application	Kernels	Data Type	Control Flow	Data Dependence
GSM Encoder	Calculation of LTP	16-bit 32-bit	Simple (single for loop)	Simple (output & true dependences)
GSM Decoder	Short Term Synthesis	16-bit 32-bit	Complex (nested loops)	Complex (true dependence)
PEGWIT Encryption	gfAddMul gfMultiply	16-bit 32-bit	very complex (nested loops with unknown upper bounds)	true dependence inter-loop dependence
PEGWIT Encryption	gfAddMul gfMultiply	16-bit 32-bit	very complex (nested loops with unknown upper bounds)	true dependence inter-loop dependence
ADPCM Encoder	adpcm_coder	8-bit 16-bit 32-bit	complex (single for loop with unknown upper bound and many if-statements)	true dependence inter-loop dependence
ADPCM Decoder	adpcm_decoder	8-bit 16-bit 32-bit	complex (single for loop with unknown upper bound and many if-statements)	true dependence inter-loop dependence
MPEG-2 Encoder	dist1	8-bit 32-bit	simple (single for loop with known upper bound)	true dependence output dependence
MPEG-2 Decoder	idctcol idctrow	8-bit 32-bit	simple (single for loop with known upper bound)	true dependence
MPEG-4 Encoder	pix_abs16x16 pix_abs16x16_xy2 pix_abs16x16_x2 pix_abs16x16_y2	8-bit 32-bit	simple (single for loop with known upper bound and many if-statements)	Complex (unknown loop bounds output dependence, true dependence)

The body of the loops are either free of control-flow, or have some if-statements. Some loop bodies, such as the PEGWIT kernels, have very complex control-flow. The compiler is capable of optimizing the loops with apparent parallelism. The complex loops pose a challenge.

The majority of the data dependence between the operations in the loop bodies is of the true dependence (read after write) and output dependence type (write after write). Where the result of one operation is a source operand in a subsequent instruction.

The data-types of operands within the loop bodies are mostly a mix between 8-bit, 16-bit and 32-bit operands. Where 32-bit operands account for the accumulators used in most loops. This mix of operands strengthens the need for the ability to perform different operations on different operand sizes concurrently.

Finally, the instruction breakdown is quite similar between most kernels except for ADPCM where there are more branch instructions than integer alu instructions.

In the next chapter, we specify architectural designs of the datapath based on these characteristics in order to construct a datapath that can target these applications effectively.

4.0 PROPOSED ARCHITECTURE

In this chapter we present the reasoning justifying a subword VLIW datapath as an adequate solution for executing the multimedia applications examined in the previous chapter.

As discussed in Chapter 2.0, to carefully design a processing system that targets media applications, the complete chain, from the application implementation to the capability of the compiler technology and the underlying processor architecture, should be examined. General purpose applications are typically implemented using high-level programming languages which are platform independent. Therefore, to gain better performance, we can attempt to design new compiler techniques and more effective processor architectures. However, these designs cannot depend entirely on hardware specific alterations that have to be made to the high-level implementation of general purpose applications.

With that in mind, and since the majority of the dynamic cycles of general purpose processors are spent on media applications, we take the results of the analysis performed in Chapter 3.0 to design a new and effective solution for general purpose processing.

We commence with evaluating the architectural requirements of media applications. Then we discuss the general VLIW architecture by presenting a classical fixed-width VLIW datapath and present its benefits as well as limitations when targeting general purpose applications and examine the important role of the compiler in VLIW architectures. Finally, we present our extensions to the classical VLIW architecture to achieve a MIMD subword VLIW datapath and discuss how these extensions match the requirements of media applications.

4.1 ARCHITECTURAL REQUIREMENTS OF MULTIMEDIA APPLICATIONS

To enable the effective execution of real-time multimedia applications, given the characteristics summarized in Section 3.11, we itemize the architectural requirements that closely satisfy the specified characteristics as follows:

First, the code kernels of multimedia applications consist of regular control structures that can be statically analyzed and scheduled by a compiler. An example from the GSM compression kernel, is shown below in Figure 44.

```
for (lambda = 40; lambda <= 120; lambda++) {

#       define STEP(k) (wt[k] * dp[k - lambda])

        L_result  = STEP(0)  ; L_result += STEP(1) ;
        L_result += STEP(2)  ; L_result += STEP(3) ;
        [...] code deleted for brevity
        L_result += STEP(38) ; L_result += STEP(39) ;

        if (L_result > L_max) {

                Nc    = lambda;
                L_max = L_result;
        }
}
```

Figure 44 The control flow from the GSM kernel.

The for loop has known bounds which the compiler can normalize and then optimize statically. Therefore, since the control structure of these code segments is visible to the compiler, a simple control mechanism around the datapath will suffice. This eliminates the overhead paid in complex control techniques that analyze the code dynamically.

Second, the computation performed in the code kernels is intensive, as seen in the kernel of the MPEG-2 decoder (Figures 33 and 34) and, hence, a powerful mix of operations is required to allow effective execution of these kernels. Furthermore, we can achieve substantial performance gains through media specific instructions. The primary operations in the motion estimation kernel of the MPEG-4 decoder consists of finding the average of 2 or 4 operands as well as the sum of absolute differences (SAD) between 2 operands (Figures 39 and 40).

Third, the input and output data sets are large and streaming in nature. A data element is read once, processed and stored. Streaming data exhibits very little temporal locality as shown in the GSM decode kernel below in Figure 45.

```
register word* wt,/* [0..k-1]IN*/
register word* sr/* [0..k-1]OUT*/

while (k--) {
        sri = *wt++;
        for (i = 8; i--;) {

                tmp1 = rrp[i];
                tmp2 = v[i];

                tmp2   = GSM_MULT_R(tmp1, tmp2);
                sri    = GSM_SUB( sri, tmp2 );
                tmp1   = GSM_MULT_R(tmp1, sri);
                v[i+1] = GSM_ADD( v[i], tmp1);
        }
        *sr++ = v[0] = sri;
}
```

Figure 45 The streaming nature of the GSM Decompression Kernel.

Therefore, a high bandwidth streaming memory interface that can bypass the cache memories is desirable. Further, special on-chip stream-based buffer structures, similar to the streaming register buffers in the Imagine processor(51), that hold data before and after processing is required.

Fourth, the code kernels are inherently parallel. An example from the motion estimation kernel is shown in Figure 46 below. As seen, the instructions in the for loop have an output dependence (write after write). However, the compiler can easily transform this code and yield independent instructions. Hence, the data path should have the capability of processing the data in a parallel fashion to allow concurrent execution on independent functional units.

Fifth, the execution is performed on variable precision, or subword, data. An example is shown in Figure 46. Both pix1 and pix2 arrays are unsigned 8-bit integers. Therefore, the processor must be capable of subword data transfer, storage and processing. This enables efficient use of memory bandwidth and functional units, the latter also decreases the amount of power dissipated by the processor.

```
int pix_abs16x16_c(UINT8 *pix1, UINT8 *pix2, int line_size, int h)
{
    int s, i;

    s = 0;
    for(i=0;i<h;i++) {
        s += abs(pix1[0] - pix2[0]);
        s += abs(pix1[1] - pix2[1]);
        s += abs(pix1[2] - pix2[2]);
        s += abs(pix1[3] - pix2[3]);
        s += abs(pix1[4] - pix2[4]);
        s += abs(pix1[5] - pix2[5]);
        s += abs(pix1[6] - pix2[6]);
        s += abs(pix1[7] - pix2[7]);
        s += abs(pix1[8] - pix2[8]);
        s += abs(pix1[9] - pix2[9]);
        s += abs(pix1[10] - pix2[10]);
        s += abs(pix1[11] - pix2[11]);
        s += abs(pix1[12] - pix2[12]);
        s += abs(pix1[13] - pix2[13]);
        s += abs(pix1[14] - pix2[14]);
        s += abs(pix1[15] - pix2[15]);
        pix1 += line_size;
        pix2 += line_size;
    }
    return s;
}
```

Figure 46 In the motion estimation kernel of the MPEG-4 encoder, the operations are independent, the operands are 8-bits.

These general architectural characteristics are necessary in order to achieve large data throughput by the processor to effectively target multimedia applications. The proposed subword VLIW architecture takes into consideration the architectural requirements presented above.

Before we present the proposed architecture, we first present the characteristics of a classical VLIW architecture and discuss its benefits as well as its drawbacks. We then argue that a VLIW architecture is a good match to the architectural requirements listed above.

4.2 VLIW ARCHITECTURE

The VLIW architecture(30,31,32,53) offers concurrent execution through a simple and flexible execution model. The premise behind this architecture is that the compiler is capable of effectively analyzing an application statically in order to extract the available parallelism and target the available hardware resources. Hence, such an architecture does not require complex hardware techniques to dynamically perform this task. A typical VLIW architecture is shown in Figure 47. It consists of a memory interface, a multiported register file and several independent pipelined functional units. All operations to be simultaneously executed by the functional units are synchronized in a VLIW instruction. The instruction parallelism and data transfer is completely specified statically, at compile time.

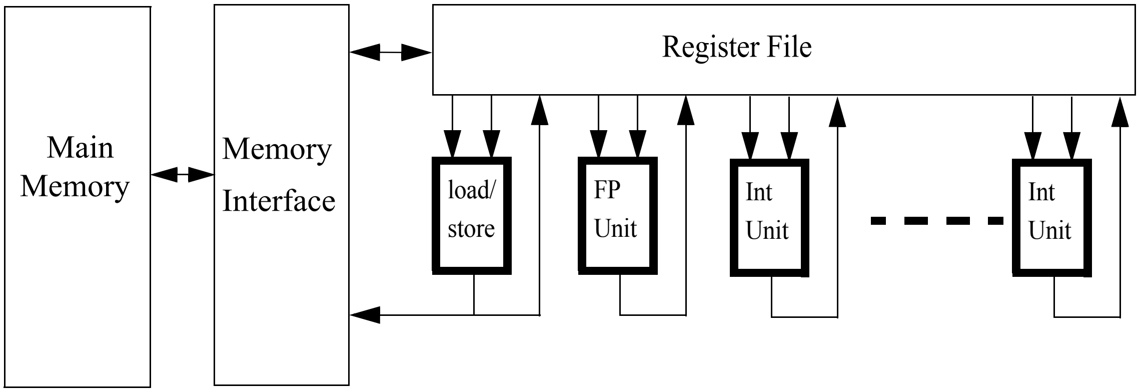


Figure 47 A typical VLIW architecture.

In a VLIW architecture, the flexibility in concurrent execution extends the types of parallelism to be identified and exploited by the compiler. The extent of exploitable parallelism is dependent on the mix of operations available in the hardware resources to the compiler while building the VLIW instruction. Given an application, the compiler analyzes it through control-flow and data-flow analysis. The compiler then transforms the application in order to highlight possibilities for parallel execution. Finally, given the underlying hardware, the compiler maps the sequential instructions onto the datapath (Figure 48), scheduling as many instructions in parallel as resources and data-dependence allows. Therefore, the success of this architecture depends primarily on the effectiveness of the compiler in extracting the available parallelism in the application and scheduling the instruction execution in a fashion that makes use of all the parallelism available in hardware. This is achieved by profiling the execution of the application to reveal the dynamic behavior.

The compiler explicitly encodes the parallelism in long instructions. Therefore, a VLIW processor does not incur the overhead of hardware support required to detect parallelism dynamically. However, due to the lack of such dynamic techniques, VLIW architectures do not promise

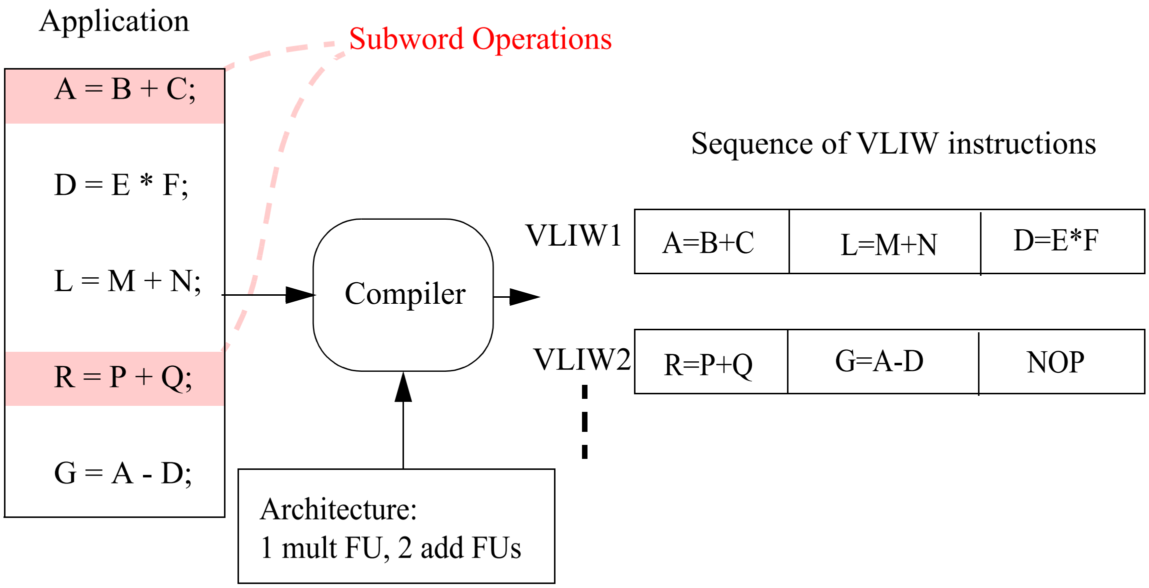


Figure 48 An example of constructing a sequence of VLIW instructions.

high performance on applications with a complex control structure that is data (value) dependent. In other words, if static compiler analysis cannot extract parallelism from the application, then the performance gains are limited. Further, VLIW architectures suffer from limited binary compatibility across processor generations, although some techniques overcome this limitation, full performance gains require recompilation of the code to better target the new processor generation.

The VLIW architecture is a close match to several of the execution characteristics of multimedia applications. Since these applications have regular control structures, the compiler can effectively analyze them and achieve fast execution through the low overhead execution datapath. Further, the pipelined functional units, equipped with a powerful instruction set, can satisfy the compute intensive computation required by multimedia applications. Finally, the limited execution restrictions allows a greater flexibility in the types of parallelism that a VLIW can target. The flexibility in the architecture is extended to the compiler and eases the path to more powerful exploitation of parallelism and good targetability.

An example of a fixed-width VLIW datapath is illustrated in Figure 49. The datapath is 128 bits wide, consisting of four 32-bit fixed width registers and functional units. This MIMD datapath can be described as 128-bits wide; has an instruction mix of 4, meaning it can perform four different instruction concurrently; and has a throughput of 4, meaning it produces 4 distinct results. The throughput measure is an important measure of performance and instruction mix is an important attribute This description is used in the next subsection to compare this datapath to a subword VLIW datapath.

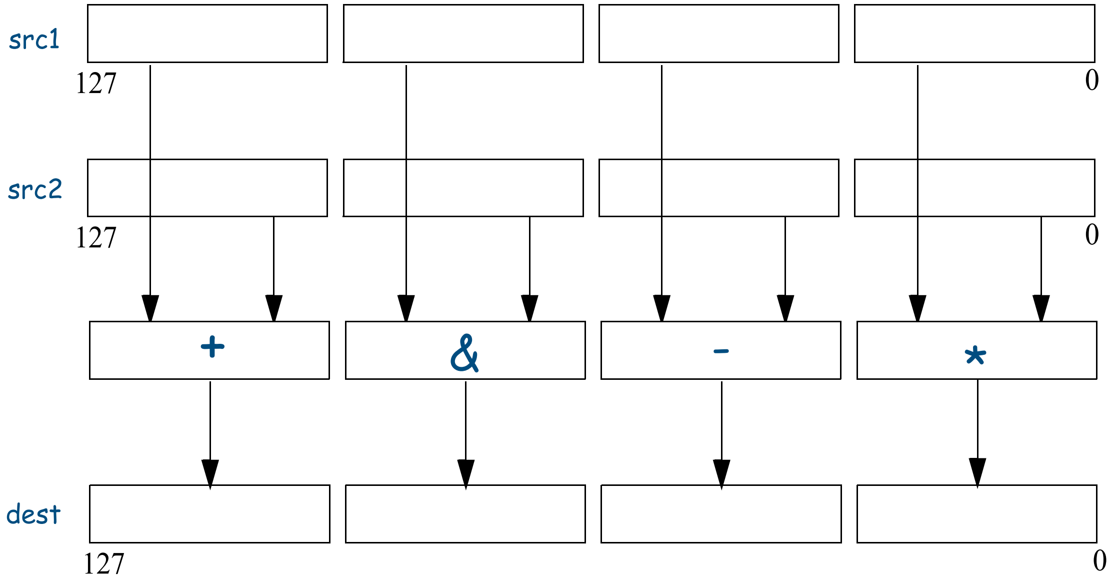


Figure 49 An example of the datapath of a general purpose VLIW microprocessor.

Although a classical VLIW architecture satisfies some of the media requirements, it does not fully address the architectural requirements of multimedia applications, particularly because it does not incorporate subword execution and streaming memory access. In the next section, we discuss how to extend the above VLIW architecture in order to achieve an execution engine that closely matches all the multimedia execution requirements.

4.3 A SUBWORD MIMD VLIW DATAPATH

A VLIW architecture offers many desirable features, however, to better target the architectural requirements specified in Section 4.1, the following enhancements are required.

Since we are targeting streaming applications, it is essential that the memory interface effectively deliver streaming data to the processor. Specifically, we need a high bandwidth interface that transfers streaming data from and to memory. Furthermore, stream-based on-chip data buffers that store and deliver the data to the functional units are required.

As shown in Chapter 3.0 and discussed in section 4.1, media applications perform many operations on subword operands. Subword operands are 8-bits (char) and 16-bits (short). Hence, in order to use the processor real estate more effectively, subword compatibility in the datapath should be enabled in all functional units, register files, streaming buffers and the memory interface. Furthermore, compared to fixed-width datapaths, subword datapaths increase the ability to perform concurrent execution. This flexibility enables the compiler to achieve shorter schedules which translate into better performance.

A rich and general subword instruction set is required. Many general operations are already available in most fixed-width VLIW architectures and must be extended to target subword operations. Furthermore, introducing several specialized media centric operations, that replace a sequence of several instructions, within the datapath is beneficial.

In this thesis, we do evaluate the benefits of enabling subword computation in the datapath. However, we do not study the effects of employing a high bandwidth streaming memory interface nor do we evaluate the performance benefits of including media-centric operations. We leave this study to a future time as discussed in the Future Work section.

An example of a general purpose processor employing a subword MIMD VLIW datapath is shown in Figure 50. The subword extensions are limited only to the integer units and register file. We evaluate the efficacy of this datapath at targeting multimedia applications.

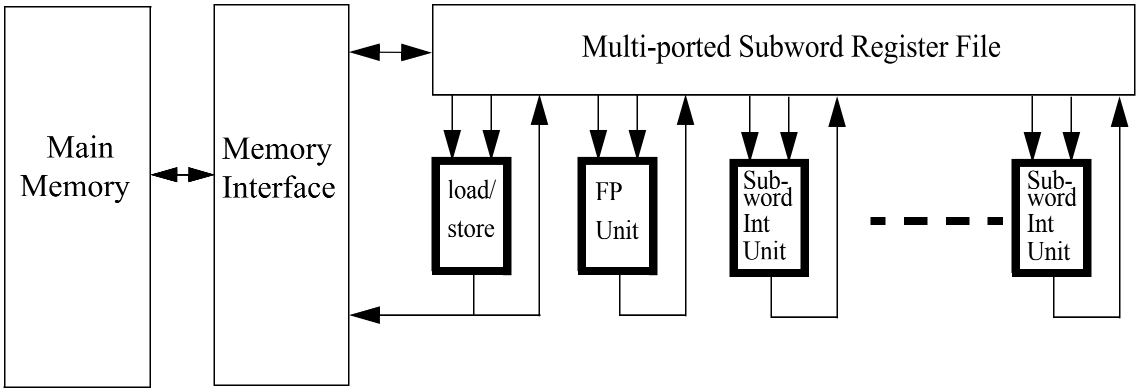


Figure 50 A VLIW processor with support for subword execution in the datapath.

With a subword datapath, the compiler has access to more hardware resources when scheduling the execution of a sequence of instructions. As illustrated in Figure 48, when the compiler targets a fixed-width datapath, it must schedule the execution of a subword operation on a wider register and functional unit. However, we can see in Figure 51 that the compiler can schedule more concurrent execution now that more hardware resources are available in a subword datapath.

An example block diagram depicting an instance of the variable-width subword MIMD VLIW datapath is illustrated in Figure 52. The datapath is 128 bits wide, consisting of four 8, 16, or 32-bit variable width registers and functional units. This subword MIMD datapath can be described as 128-bits wide; has a maximum instruction mix of 16, meaning it can perform up to

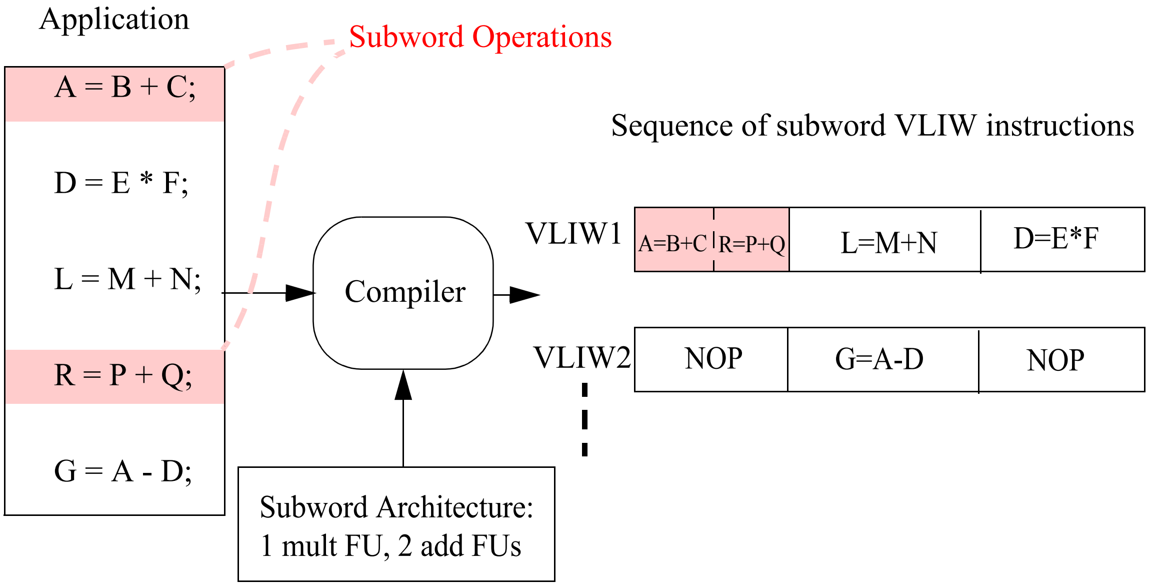


Figure 51 An example of constructing a sequence of subword VLIW instructions.

16 different instructions concurrently when performing 8-bit operations; and has a maximum throughput of 16, meaning it is capable of producing 16 distinct results.

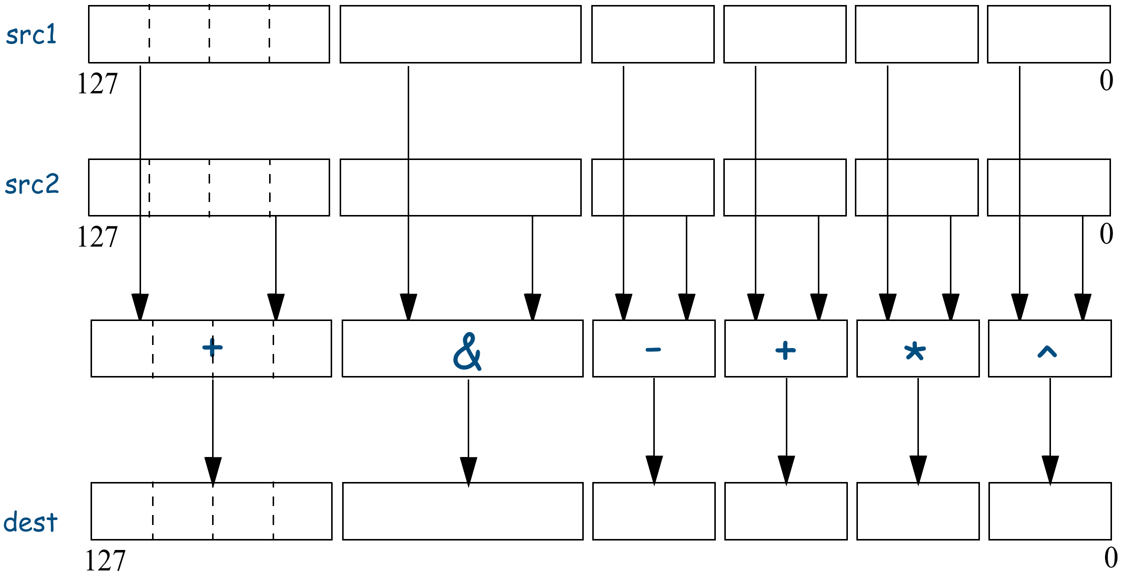


Figure 52 An example of a Subword MIMD VLIW datapath which provides increased execution flexibility to the compiler.

Hence, our hypothesis is that an augmented subword VLIW processor coupled with a subword targeting VLIW compiler is a suitable match to executing multimedia application kernels and can achieve high performance through shorter schedules or increased parallel execution on available data while incurring low overhead.

Next, we compare the throughput and instruction mix measures with a fixed-width VLIW datapath as well as a variable-width subword SIMD datapath.

4.3.1 Datapath Comparison

A subword SIMD datapath issues a single instruction that is executed on several data operands. An example block diagram depicting an instance of the variable-width subword SIMD datapath is illustrated in Figure 53. The datapath is 128 bits wide, consisting of four 8, 16, or 32-bit variable width registers and functional units. This subword SIMD datapath can be described as 128-bits wide; has a maximum instruction mix of 1, meaning it can only perform a single instruction at once; and has a maximum throughput of 16, meaning it is capable of producing 16 distinct results.

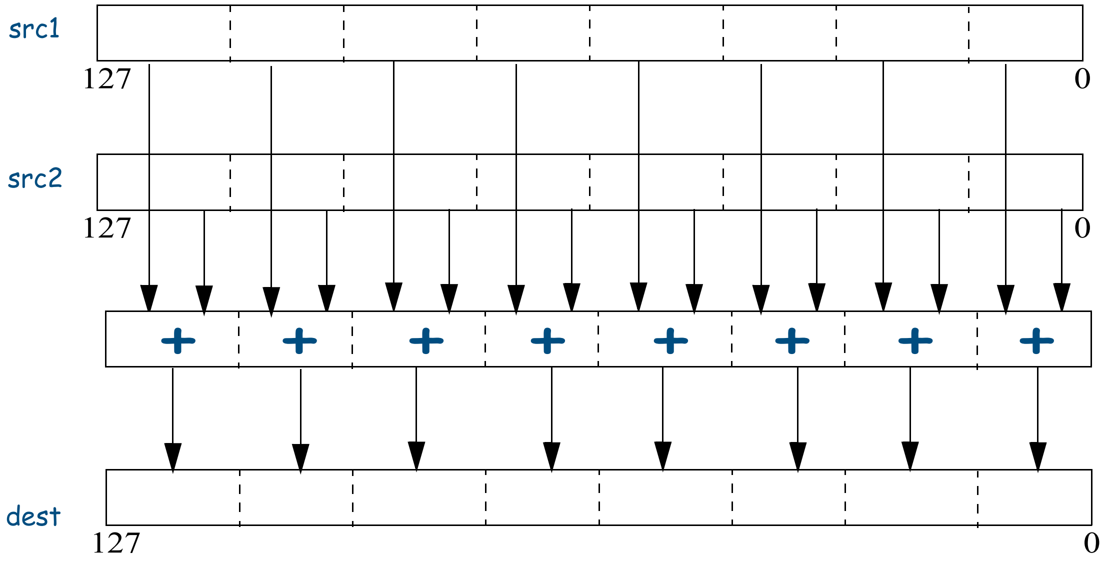


Figure 53 A n example configuration of Subword SIMD datapath which presents limited execution flexibility to the compiler.

The subword SIMD datapath provides parallel execution of subword data, however, it can only perform a single operation on all the subword data in parallel. The benefits of such an architecture are limited control overhead as well as small die area usage. The drawbacks, however, are that the compiler cannot automatically schedule non-SIMD implementations of media applications onto a SIMD datapath.

In Figure 54, we compare the flexibility and throughput of each of the three datapath architectures discussed. All three datapaths are 128-bits wide, the fixed-width MIMD datapath, offers limited instruction flexibility and limited throughput. The MIMD subword datapath offers a lot of flexibility and a lot of throughput, when performing operations on 8-bit operands. The SIMD subword datapath offers a lot of throughput but with extremely limited flexibility.

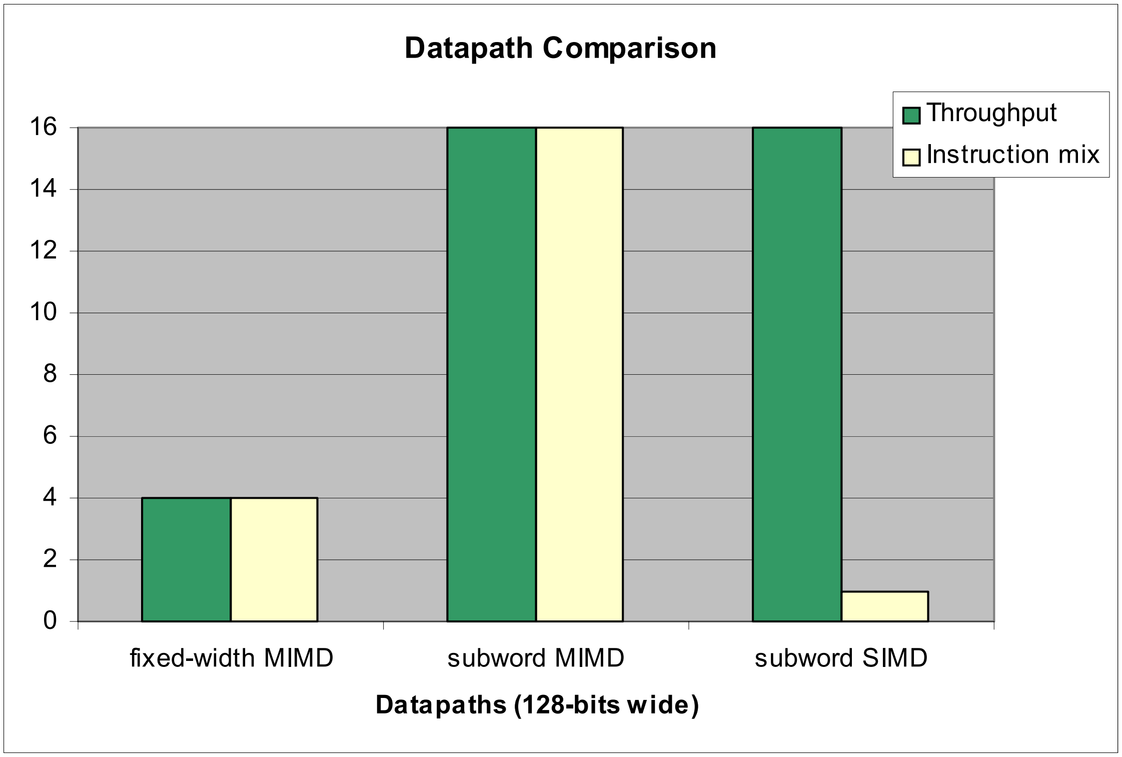
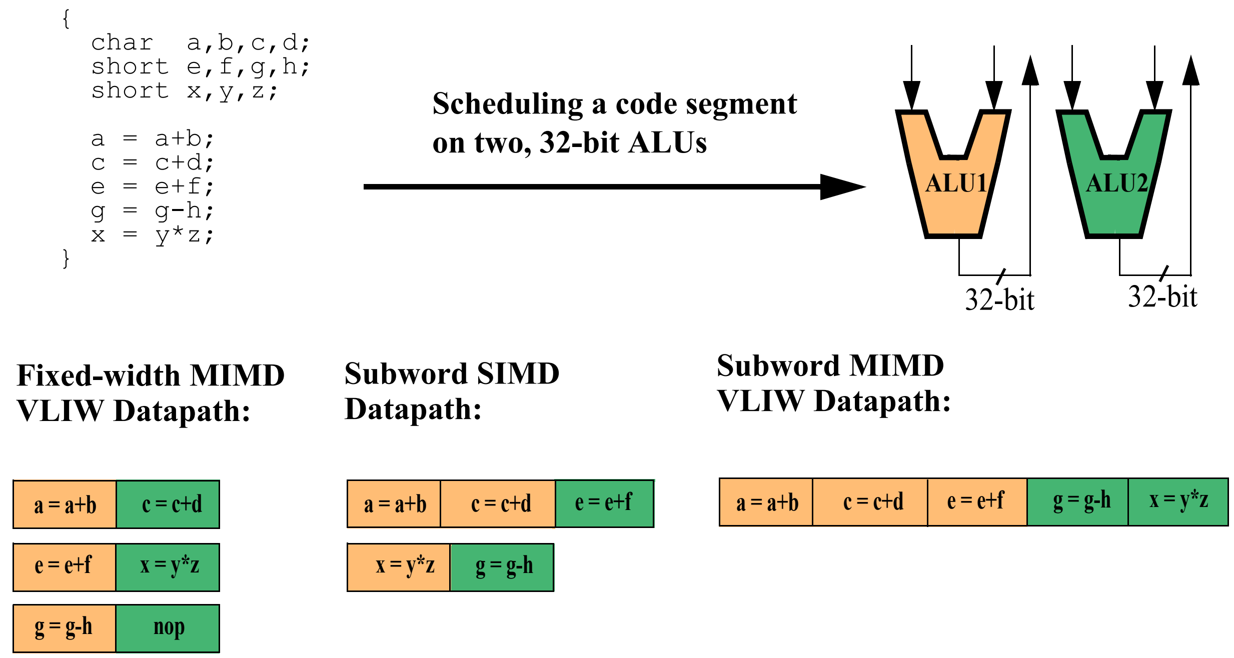


Figure 54 Comparing the maximum throughput and instruction mix in three 128-bit datapaths.

The SIMD subword datapath conserves die-area by executing a single operation on several operands in order to offer parallel execution. This trade-off seriously limits the ability of the compiler to achieve efficient schedules when targeting a SIMD datapath.

An example of scheduling a code segment on the three discussed architectures, fixed-width MIMD VLIW, subword-MIMD VLIW and subword-SIMD is presented in Figure 55.



```
{
  char  a,b,c,d;
  short e,f,g,h;
  short x,y,z;

  a = a+b;
  c = c+d;
  e = e+f;
  g = g-h;
  x = y*z;
}
```

Fixed-width MIMD VLIW Datapath:

a = a+b	c = c+d
e = e+f	x = y*z
g = g-h	nop

Subword SIMD Datapath:

a = a+b	c = c+d	e = e+f
x = y*z	g = g-h	

Subword MIMD VLIW Datapath:

a = a+b	c = c+d	e = e+f	g = g-h	x = y*z

Figure 55 Scheduling a code segment on three datapaths, a fixed-width MIMD VLIW, a subword SIMD and subword MIMD VLIW.

This scheduling example highlights the characteristics of the three datapath architectures. The code segment includes four independent operations, two additions of 8-bit variables, an addition of 16-bit variables, a subtraction of 16-bit variables and a multiplication of 16-bit variables. Only two operations can be scheduled concurrently on the fixed-width datapath. The two 8-bit addition operations can be scheduled on one of the subword-SIMD ALUs since SIMD requires operations to be of the same precision in order to execute concurrently on an ALU. Hence, the 16-bit addition operation can be scheduled to execute on the other ALU in parallel. This requires the 16-bit multiplication operation and the 16-bit subtraction operation to be scheduled subsequent to the execution of the addition operations. However, all the operations can be scheduled concurrently on the subword-MIMD VLIW datapath, since it is capable of concurrently executing operations of different precision on the same ALU as well as different types of operations on the same ALU.

The subword MIMD datapath offers a lot of flexibility as well as a lot of parallelism, however, there is a high die-area cost associated with this design. In the next section we discuss the architectural support required to achieve a subword MIMD datapath.

4.3.2 Architectural Parameters of a Subword VLIW Datapath

The architectural support required to extend a fixed-width MIMD VLIW datapath into a subword MIMD VLIW datapath are the following:

- **Variable width Functional Units**; Modular 8-bit functional units that can be programmed to perform computation on variable width (8,16 and 32-bit) operands. The 8-bit operations within modular units can be combined to perform a 32-bit operation. A carry skip adder can be used using four 8-bit adder modules. A 32-bit*32-bit Wallace Tree multiplier can be extended to support four 8-bit*8-bit multiply operations and two 16-bit*16-bit multiply operations with a 10% increase in die area cost(55).

- **Subword register files**; Modular multiported 8-bit register file, with enough ports to match the number of functional units.

These general architectural characteristics are necessary in order to achieve large data throughput through the processor to effectively target multimedia applications. The die area cost associated with the above design is important and is discussed next.

We evaluate the die-area cost of a 32-bit fixed-width MIMD VLIW datapath and compare it to the cost of extending this datapath to a subword MIMD datapath that is capable of performing operations on 8, 16 and 32-bit operands. The design of the fixed-width datapath is shown in Figure 56.

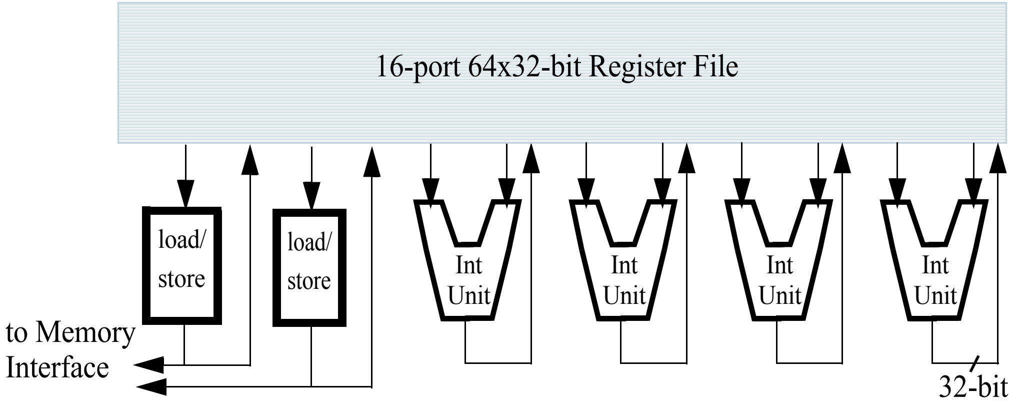


Figure 56 A fixed-width VLIW datapath.

The fixed-width 32-bit datapath consists of four 32-bit functional units, two 32-bit load/store units, and a 16-port 64x32-bit registers. Based on a study that performs VLSI modeling of designs of different processor components at 0.25um technology[56], we deduce that the area required by a multiported register file is substantial. Based on designs of ALUs and multipliers[56,57,58], using the same technology, we list the die-area cost of the functional units in Table 6.

Table 6 Die area for the fixed-width functional units

Module	Width	Area (mm^2)
ALU	32-bit	0.6 (x4)
Multiplier	32-bit*32-bit	9 (x4)
Total		38.4

The subword datapath, shown in Figure 57, consists of sixteen 8-bit functional units, two 32-bit load/store units, and a 64-port 256x8-bit register file.

Based on the literature, the 64-port 256 8-bit register file will be significantly larger than that of the fixed-width, on the order of 5 to 10 times its size. However, there are several approaches that resolve this problem be employing multiple register files with reduced number of ports. Further, several effective scheduling techniques have been developed to target this architecture.

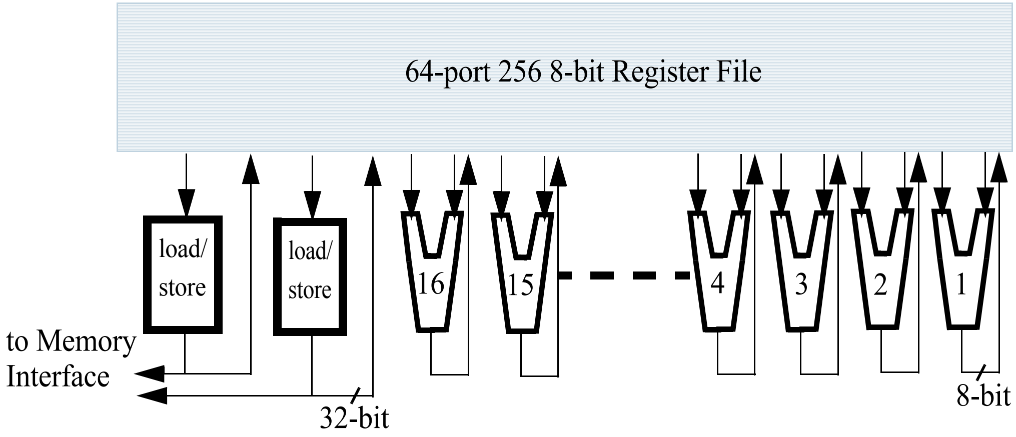


Figure 57 A fixed-width VLIW datapath.

Based on designs of ALUs and multipliers(56,57,58), and the fact that a Wallace Tree multiplier extended to support subword multiplication grows in size by 10%(55), we illustrate the die-area cost of the functional units in Table 7.

Table 7 Die area for the subword functional units

Module	Width	Area (mm^2)
ALU	8-bit	0.26 (x16)
Multiplier	four 8-bit*8-bit two 16-bit*16-bit one 32-bit*32-bit	9.9 (x4)
Total		43.76

The subword functional units require 14% more die area than the fixed-width datapath. Needless to say, there are other factors that might contribute to increased die area such as busses and control lines.

In the following chapter, we present the compiler and architecture simulator infrastructure. Also, we present the methodology used to evaluate the effectiveness of a subword MIMD VLIW at executing multimedia applications.

5.0 COMPILER FRAMEWORK AND EXPERIMENTAL METHODOLOGY

In order to perform our analysis, we need a compiler that can extract the parallelism within the multimedia applications and target the architectures defined above. Also, a simulation engine that is capable of simulating the execution of the compiled applications onto the architectures specified in the previous section are required. The Trimaran(59) compiler infrastructure enables us to perform the required analysis and evaluation.

5.1 THE TRIMARAN FRAMEWORK

Trimaran is a compilation, simulation and monitoring infrastructure. It was developed by a consortium from the Impact Research Group at University of Illinois at Urbana Champaign, the Compiler and Architecture Research Group at Hewlett Packard Laboratories and Center for Research on Embedded Systems and Technology (CREST) at the Georgia Institute of Technology. (CREST was the ReaCT-ILP Laboratory at New York University). It consists of three parts, the front-end machine independent compiler, Impact(60), the back-end machine targeting compiler, Elcor(61) and the system simulation engine(59). We first discuss the compiler specifications, capabilities, and limitations and then move on to describe the simulation engine.

5.1.1 Compiler Support

The front-end compiler, Impact, performs program analysis and transformation that requires high level program structure information. The Impact compiler performs all the classical program optimizations. It also has the capability of performing more aggressive transformations, such as loop unrolling, predicated execution[62], basic block, superblock[63] and hyperblock[64] formation. The following optimization paths can be invoked:

- **Basic block**: The Impact front-end compiler performs the classical optimizations. Then, Elcor, the back-end compiler, performs Loop Region Formation, Modulo Scheduling, Rotating register allocation, Pre-pass scheduling, Register allocation and Post-pass scheduling. Trimaran offers documentation on each segment of the infrastructure at [59].

- **Superblock**: The Impact front-end performs superblock formation using profile information and classical optimizations. The back-end performs Superblock scheduling, Loop region formation, Modulo Scheduling, Rotating register allocation, Pre-pass scheduling, Register allocation and Post-pass scheduling.

- **Hyperblock**: The Impact front-end performs hyperblock formation, predicated execution, loop unrolling and classical optimizations. The back-end performs Hyperblock scheduling, Loop region formation, Modulo Scheduling, Rotating register allocation, Pre-pass scheduling, Register allocation, and Post-pass scheduling.

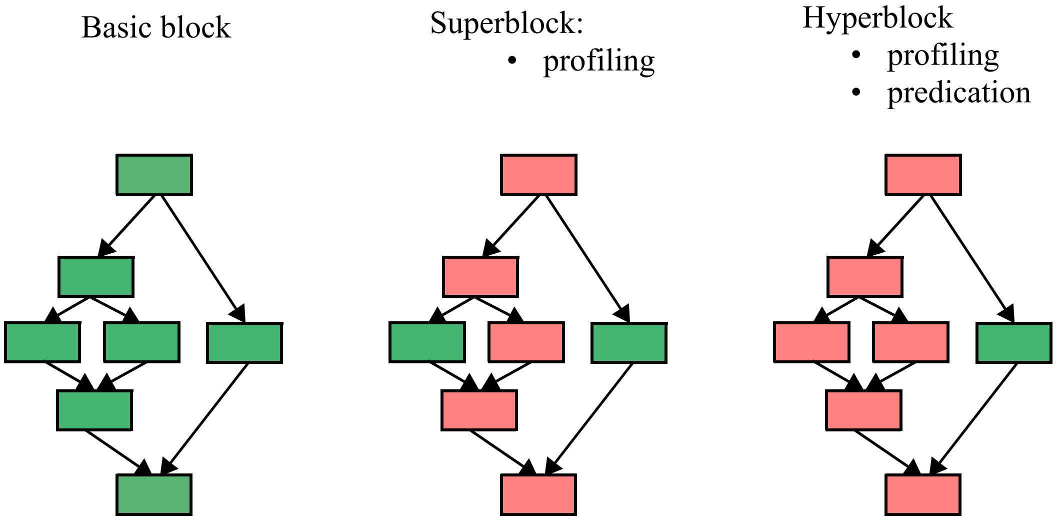


Figure 58 Machine independent code transformations in Trimaran.

Identifying basic blocks is the result of conventional control-flow analysis to generate code blocks containing a sequence of instructions that are surrounded by branches. Superblock formation uses profiling information to transform a frequently taken path through the code into a contiguous portion of code called a superblock. An example is shown in Figure 58, the frequently executed red basic blocks are transformed into a single superblock. In case the predicted path in the superblock is not taken, the compiler generates clean up code to recover from that situation. Hyperblock formation is more aggressive, it uses profiling information as well as if-conversion, which is enabled by predicated execution, to remove all the control-flow of a frequently executed segment of code and generates a single hyperblock encompassing all the instructions. Predicated execution allows the results of the valid operations to be committed, while those of the speculative operations are discarded.

These techniques perform program transformations based on profile information that aim at increasing the amount of parallelism visible to the back end compiler, Elcor.

Elcor, is a retargetable compiler back-end that uses a machine-description language, Play-Doh(67,68,69) (HPL-PD), to customize the optimizations, transformations and code generation to the specific processor specified by its description. This compiler makes queries to a machine-description database (mdes), which provides the processor information needed by the compiler. The compiler can be retargeted to different processors by changing the contents of this database. These processors may vary widely, by changing the:

- **Functional Units**: number of functional units, their pipeline structure and latencies;
- **Operations**: the set of opcodes that each functional unit can execute;
- **Data Types**: the set of data types required;
- **Storage**: the number of register files, the number of registers in each register file;
- **Interconnect**: register file accessibility from the various functional units and the busing structure between the register files and the functional units; and

- **I/O**: the number of channels to external memory.

Although these processor descriptions could be varied in mdes, Elcor, in its current state, does not allow the user to vary all of the above variables. For example, Elcor hardcodes the set of opcodes that it can target, hence, even if a new set of opcodes can be specified within mdes, Elcor does not make use of them. However, even with Elcor's current limitations, we are able to study and target the architecture specified in Chapter 4.0.

In order to perform this study, we need a powerful compiler that can exploit the parallelism provided by the subword datapath. Fortunately, Elcor is equipped with very aggressive optimization, scheduling and register allocation techniques which are suitable for the analysis we perform.

Besides program transformation, machine dependent optimizations and register allocation, the biggest responsibility of a back-end compiler is to produce the best schedule possible. Elcor achieves that by performing specific superblock and hyperblock scheduling as well as Modulo Scheduling(65,66) on counted loops. Modulo scheduling is a software pipelining technique which moves operations from future loop iterations, and, hence, overlaps the execution of different iterations of a loop.

5.1.2 Simulation Engine

The simulator uses the HPL-PD processor description, it transforms the Elcor intermediate representation of a function into low-level C code. This code is then linked with a library that consists of the HPL-PD virtual machine. It is basically an interpreter and a set of emulation routines from the HPL-PD virtual machine. On every procedure entry, the interpreter is invoked and it emulates the instruction stream until the procedure returns. There is one emulation function for each HPL-PD operation.

Therefore, for each application, code is generated as an executable that emulates the execution of the machine specified using HPL-PD. Furthermore, the executable is instrumented to generate several execution statistics of interest. We primarily collect the execution time, in clock cycles, that each function consumed.

5.2 METHODOLOGY

Execution performance of an application depends on several factors: first, algorithm design and implementation; second, the effectiveness of the compiler optimizations and code generation; and third, the underlying processor and system architecture. Since we are performing this datapath performance evaluation for general purpose applications, we need to ignore the impact on performance that major algorithmic changes can have. We assume a single design and implementation of an algorithm.

Our goal is to evaluate the performance of a subword MIMD VLIW datapath at executing an application's time-dominant kernels and loop structures that exhibit large degrees of parallelism, as presented in Chapter 3.0. We compare this performance to the performance of a fixed-width VLIW datapath. We employ a wide VLIW datapath as the subword MIMD datapath and execute only the kernel code, which includes most of the subword operations, on this datapath. This is performed to limit the effect of employing a wide VLIW datapath on the remaining portion of the application.

Further, we examine the performance benefits gained from using aggressive code transformation techniques to extract parallelism hindered by control-flow. Specifically, we evaluate the effectiveness of hyperblock formation.

Finally, we perform simple high-level code transformations to the kernels in order to reveal more parallelism that the compiler is unable to extract at its current state. We argue that the code transformations performed are very simple and that they do not constitute major algorithmic changes and can easily be automated by a compiler.

5.2.1 Experimental Setup

For each application discussed in Chapter 3.0, we perform four experiments. First, we execute the application on a fixed-width datapath to identify the code kernels of the multimedia applications. The architecture is a basic VLIW processor. The application is compiled and executed on the fixed-width processor using a fixed input data set. Using the execution statistics gathered by the simulator, the time-dominating functions or kernels of the application are identified and the overall clock cycle count is recorded. This experiment is considered the base case, the performance results of all subsequent experiments are compared to it.

Second, these kernels are then compiled to target the subword-VLIW datapath being evaluated. We choose two specific sets of optimization paths through the front-end and back-end compilers. The significant difference in code transformations take place at the front-end of the compiler, where the high-level program structure is very visible and hence can be exploited. Hence, we evaluate the efficacy of a subword VLIW datapath at executing the kernel portion of the application compiled using basic block formation. This result is compared against the base case discussed above. Third. aggressive compiler techniques are enabled and evaluated. Namely, hyperblock formation. The result is also compared to the base case.

Fourth, A series of simple code transformations are performed on the loops inside the kernels. Transformations such as, loop unrolling, embedding temporaries, simple code motion. These are performed only when we observe that the compiler should have taken advantage of parallel execution within the loop, however, it can not due to compiler constraint parameters. These four experiments are listed in Table 8.

Table 8 The experiments performed.

Experiments	Datapath	Compiler
Base Case, 32-bit fixed-width Datapath	64 Registers 4 Functional Units 2 Load/Store Units	Basic Block
Performance evaluation of 8-bit subword Datapath	256 Registers 16 Functional Units 2 Load/Store Units	Basic Block
Performance Impact when utilizing a subword datapath and aggressive compiler techniques	256 Registers 16 Functional Units 2 Load/Store Units	Hyperblock
Performance Impact when utilizing a subword datapath and aggressive compiler techniques and simple high-level code transformations.	256 Registers 16 Functional Units 2 Load/Store Units	Hyperblock + code transformations

In the next chapter, we carry out the four experiments specified above using the applications presented in Chapter 3.0.

6.0 EXPERIMENTAL ANALYSIS

In this chapter we discuss the performance analysis of the execution of the multimedia kernels discussed in Chapter 3.0 on the variable width VLIW datapath. We also compare the performance of this datapath to a classical VLIW datapath. Our hypothesis is that a subword VLIW processor is able to achieve high throughput when targeting multimedia applications compared to a fixed-width VLIW processor.

In this chapter we present the experiments performed, report the results of our simulations and discuss their outcome. We evaluate the efficacy of the architectures presented in Chapter 3.0 at executing nine multimedia based applications, encoding and decoding a stream of audio using the GSM(70); encryption and decryption using the PEGWIT(70) algorithm; encoding and decoding an audio stream using ADPCM(70); encoding and decoding a video stream using the MPEG-2(70) decompression algorithm; and finally, encoding a video stream using MPEG-4. All these algorithms are part of the MediaBench(70) and MediaBenchII benchmark suites.

We chose these algorithms in order to capture a wide range of multimedia applications by evaluating speech, encryption, audio and video based applications.

6.1 KERNEL ANALYSIS AND TRANSFORMATIONS

6.1.1 Kernel of the GSM Encoder

The GSM Encoder has a highly parallel kernel, however, it does include true data dependences and output dependences in the loop body. We anticipate the compiler to overcome the limitations due to these dependences and exploit most of the available parallelism.

We execute this application on the fixed-width 32-bit VLIW datapath, and on a subword VLIW datapath. For this initial test, we do not utilize any aggressive compiler techniques to extract parallelism by performing code transformations. This allows us to evaluate the ability of classical compiler techniques to take advantage of more hardware resources when targeting the subword datapath. The results are shown in Figure 59.

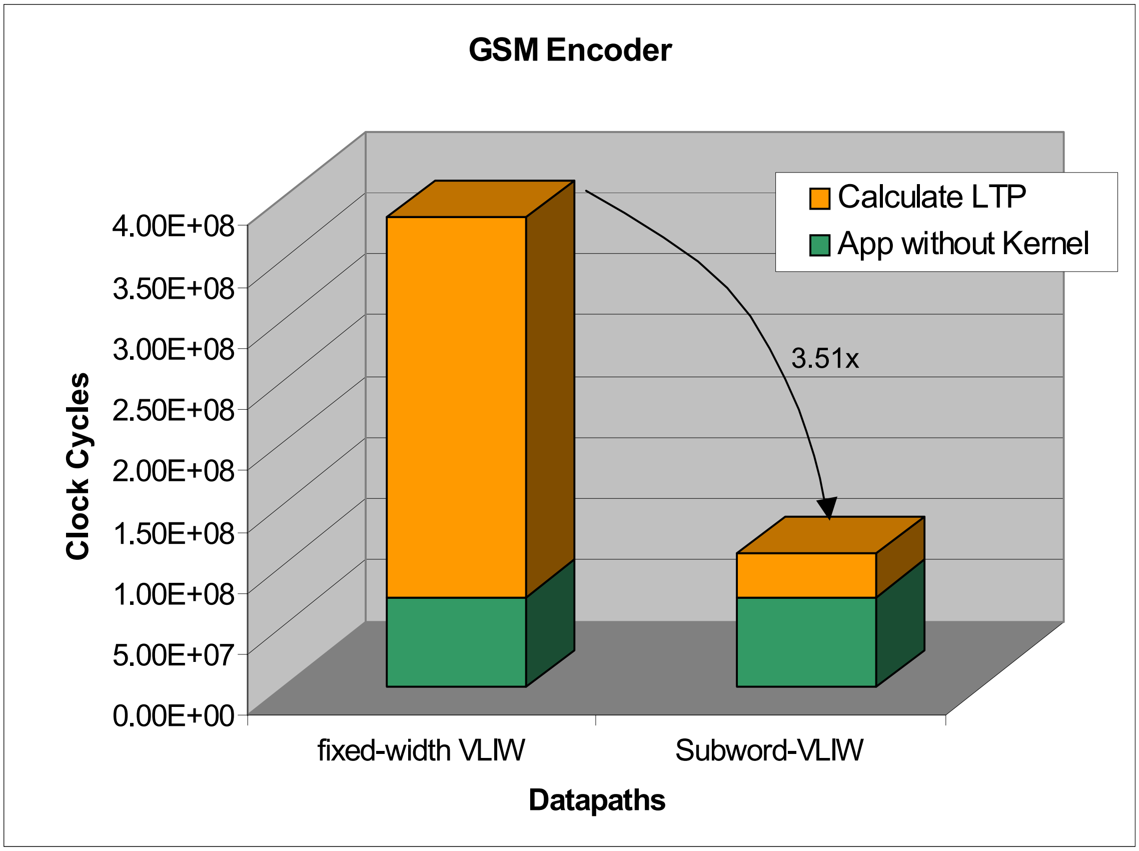


Figure 59 The performance comparison of fixed-width VLIW vs. subword VLIW for the GSM Encoder kernel.

As expected, the compiler removed most of the data dependences in this kernel by using temporary variables to store the results of the multiplications which are then summed with all the other 40 multiplications. This compiler technique increased the amount of parallelism in the kernel. Executing the kernel on a subword VLIW datapath, the GSM Encoder application was speed up by a factor of 3.51.

Next, we evaluate the ability of aggressive compiler code transformations to highlight more parallelism in the kernel. We employ hyperblock formation in the compiler and compare the result to the base case as shown in Figure 60.

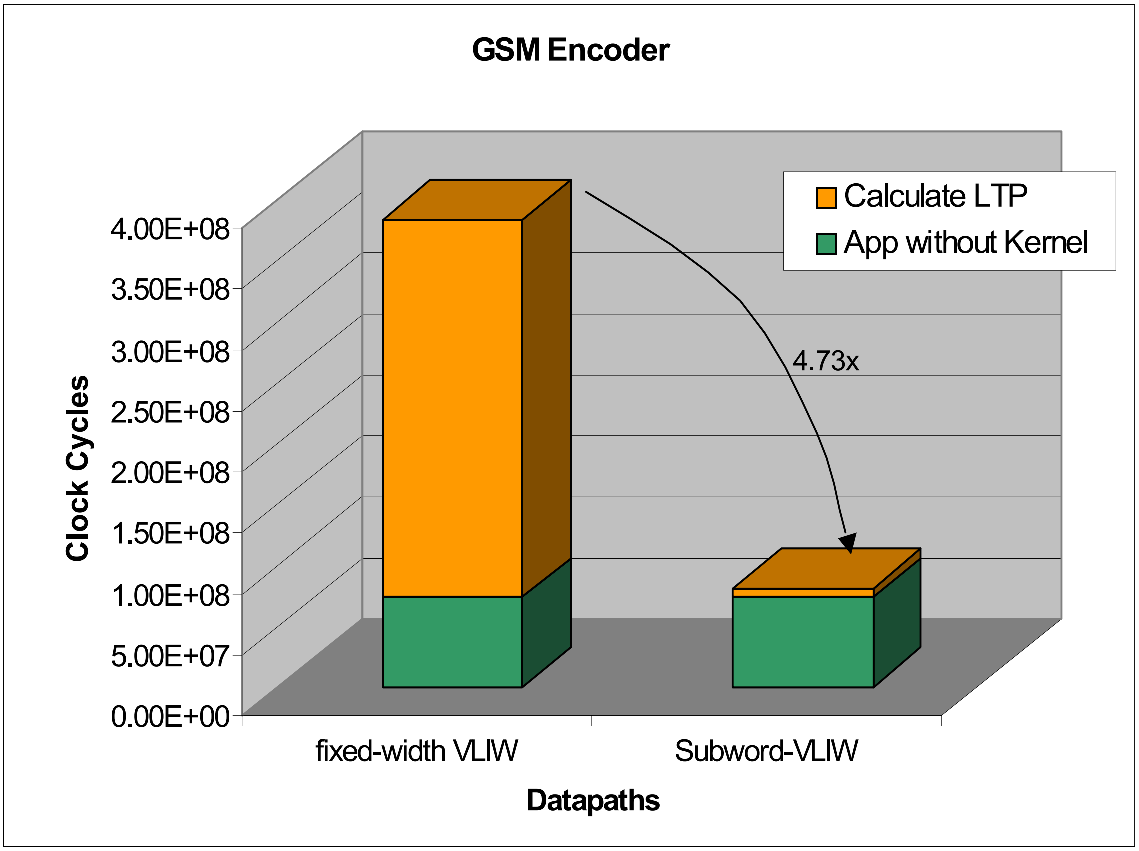


Figure 60 The performance impact on the GSM Encoder after enabling aggressive compiler techniques.

As can be observed, hyperblock formation further improved the performance and increased the speedup to 4.73. For this application, the compiler was capable of exploiting most of the available parallelism. When we implemented simple code transformations on the kernel we did not record any further speedups when executing this application on a subword-VLIW datapath.

6.1.2 Kernel of the GSM Decoder

The kernel for the GSM decoder algorithm was identified in section 3.3. As observed, the GSM decoder spends 70% of all dynamic execute cycles in a single function, which is the short term filtering synthesis required to reproduce the original speech signal.

The code structure of this kernel was discussed in detail in section 3.3. In summary, the control flow is a pair of nested *for* loops where the upper bound of the nested loop is known while that of the outer loop is input dependent. There exists a true dependence between every instruction inside the nested loop as well as inter-loop iteration direct data dependence. Hence, this kernel does not exhibit a lot of parallelism which limits the opportunity for parallel execution.

We perform the base case simulation on a fixed-width datapath. Also, we execute this kernel on the subword datapath with minimal compiler optimizations and observe the results in Figure 61.

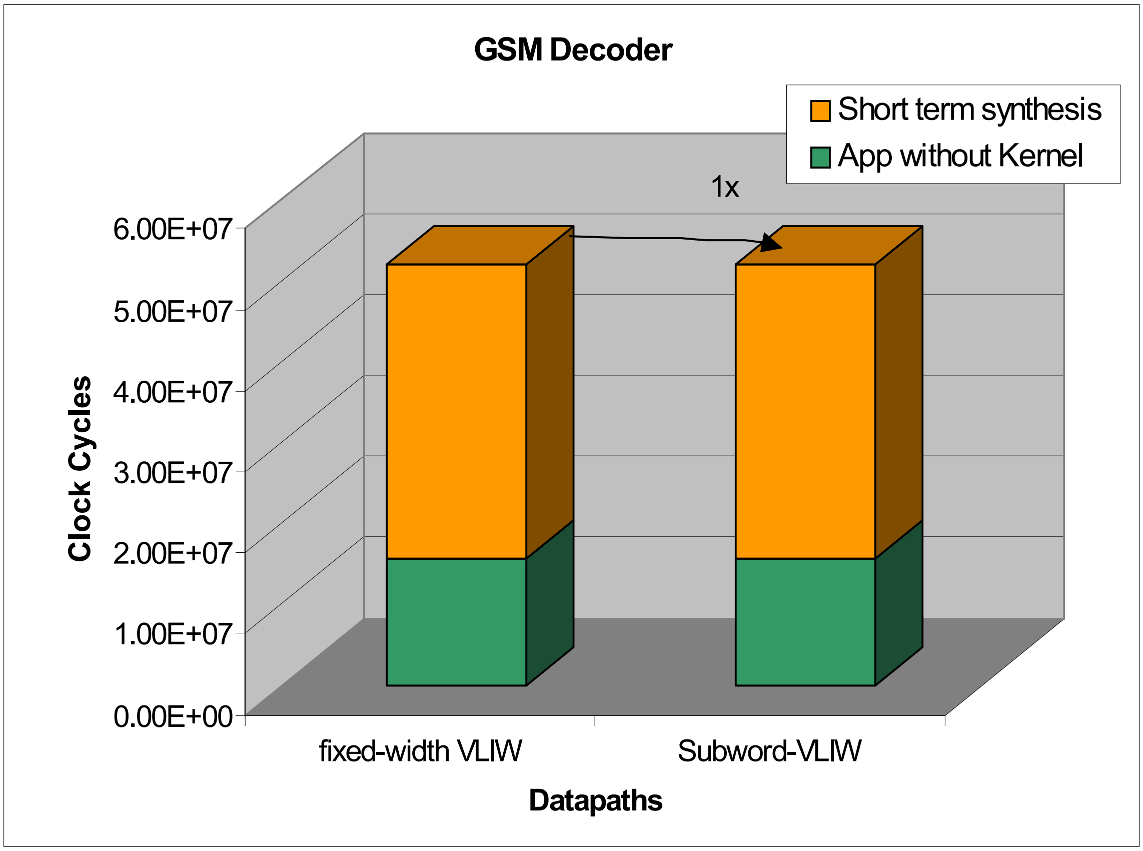


Figure 61 The performance comparison of using a fixed-width datapath to a subword datapath.

As expected, the simple compiler techniques are not able to extract parallelism from the kernel due to the nested loop structure and the extensive dependence in the loop body. Therefore, in the next simulation, we compile the application and enable hyperblock code transformations which overcomes control-flow limitations by performing code speculation using predication. The result compared to the base case is shown in Figure 62.

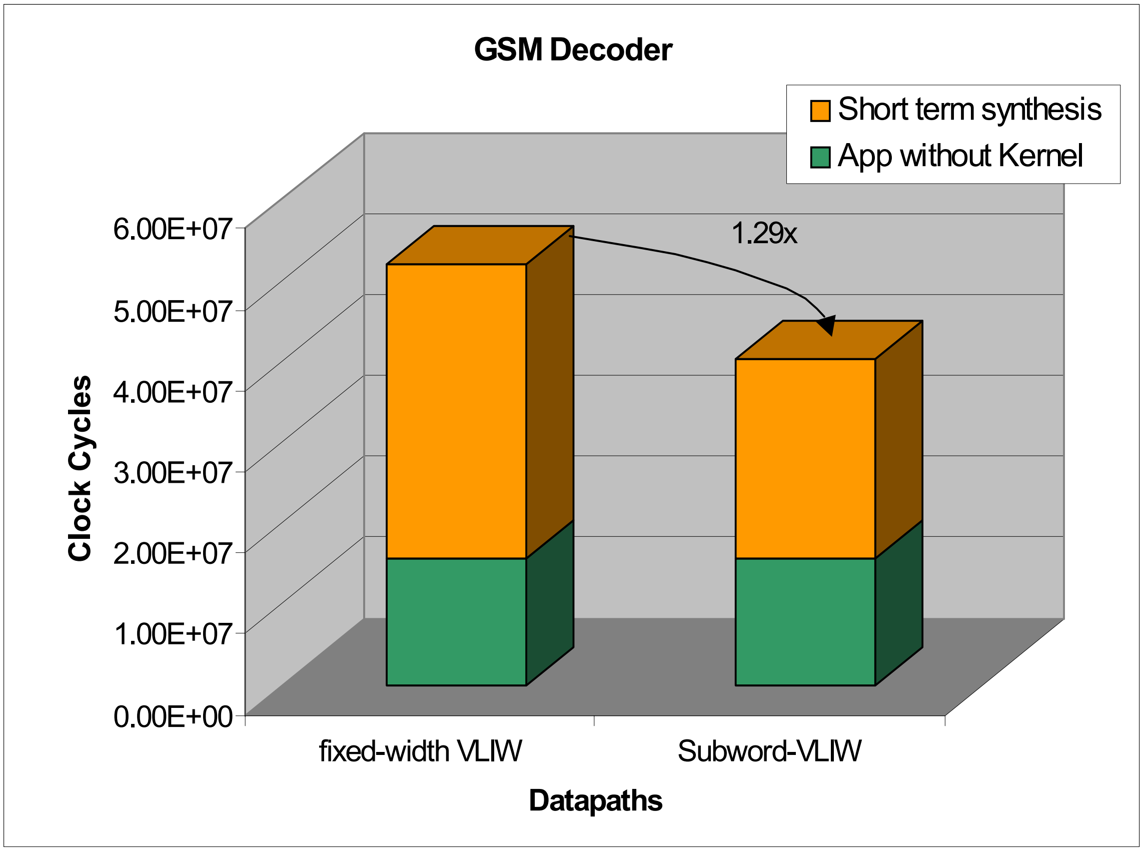


Figure 62 The performance impact on enabling aggressive compiler techniques.

The compiler is capable of extracting some parallelism and achieving a speedup of 1.29.

In order to take advantage of the parallel execution modules available within the subword VLIW datapath, we must first highlight and extract some parallelism from the kernel. In order to accomplish this task, we employ simple code transformations of the kernel and evaluate their impact on performance.

Since the upper bound of the nested loop is known, we simply unroll the inner loop in order to highlight any resulting independent inter-loop operations. The resulting body of the kernel is shown below:

```
while (k--) {
               sri = *wt++;

               /* iteration #1 */
               tmp1_7 = rrp[7];
               tmp2_7 = v[7];
               tmp2_7 = GSM_MULT_R(tmp1_7, tmp2_7);
               sri    = GSM_SUB( sri, tmp2_7 );
               tmp1_7 = GSM_MULT_R(tmp1_7, sri);
               v[8]   = GSM_ADD( v[7], tmp1_7);

               /* iteration #2 */
               tmp1_6 = rrp[6];
               tmp2_6 = v[6];
               tmp2_6 = GSM_MULT_R(tmp1_6, tmp2_6);
               sri    = GSM_SUB( sri, tmp2_6 );
               tmp1_6 = GSM_MULT_R(tmp1_6, sri);
               v[7]   = GSM_ADD( v[6], tmp1_6);

               /* iteration #3 */
               tmp1_5 = rrp[5];
               tmp2_5 = v[5];
               tmp2_5 = GSM_MULT_R(tmp1_5, tmp2_5);
               sri    = GSM_SUB( sri, tmp2_5 );
               tmp1_5 = GSM_MULT_R(tmp1_5, sri);
               v[6]   = GSM_ADD( v[5], tmp1_5);

               /* iteration #4 */
               tmp1_4 = rrp[4];
               tmp2_4 = v[4];
               tmp2_4 = GSM_MULT_R(tmp1_4, tmp2_4);
               sri    = GSM_SUB( sri, tmp2_4 );
               tmp1_4 = GSM_MULT_R(tmp1_4, sri);
               v[5]   = GSM_ADD( v[4], tmp1_4);

               /* iteration #5 */
               tmp1_3 = rrp[3];
               tmp2_3 = v[3];
               tmp2_3 = GSM_MULT_R(tmp1_3, tmp2_3);
               sri    = GSM_SUB( sri, tmp2_3 );
               tmp1_3 = GSM_MULT_R(tmp1_3, sri);
               v[4]   = GSM_ADD( v[3], tmp1_3);

               /* iteration #6 */
               tmp1_2 = rrp[2];
               tmp2_2 = v[2];
               tmp2_2 = GSM_MULT_R(tmp1_2, tmp2_2);
               sri    = GSM_SUB( sri, tmp2_2 );
               tmp1_2 = GSM_MULT_R(tmp1_2, sri);
               v[3]   = GSM_ADD( v[2], tmp1_2);

               /* iteration #7 */
               tmp1_1 = rrp[1];
               tmp2_1 = v[1];
               tmp2_1 = GSM_MULT_R(tmp1_1, tmp2_1);
               sri    = GSM_SUB( sri, tmp2_1 );
               tmp1_1 = GSM_MULT_R(tmp1_1, sri);
               v[2]   = GSM_ADD( v[1], tmp1_1);

              /* iteration #8 */
               tmp1_0 = rrp[0];
               tmp2_0 = v[0];
               tmp2_0 = GSM_MULT_R(tmp1_0, tmp2_0);
               sri    = GSM_SUB( sri, tmp2_0 );
               tmp1_0 = GSM_MULT_R(tmp1_0, sri);
               v[1]   = GSM_ADD( v[0], tmp1_0);

               *sr++ = v[0] = sri;
        }
```

Figure 63   The body of the loop after performing a simple unroll of the inner loop.

The performance results of performing loop unrolling is shown in Figure 64. As expected, the aggressive compiler transformations already unrolled the loop and took advantage of the parallelism that became apparent. That is why we do not gain any additional speedups due to this code transformation.

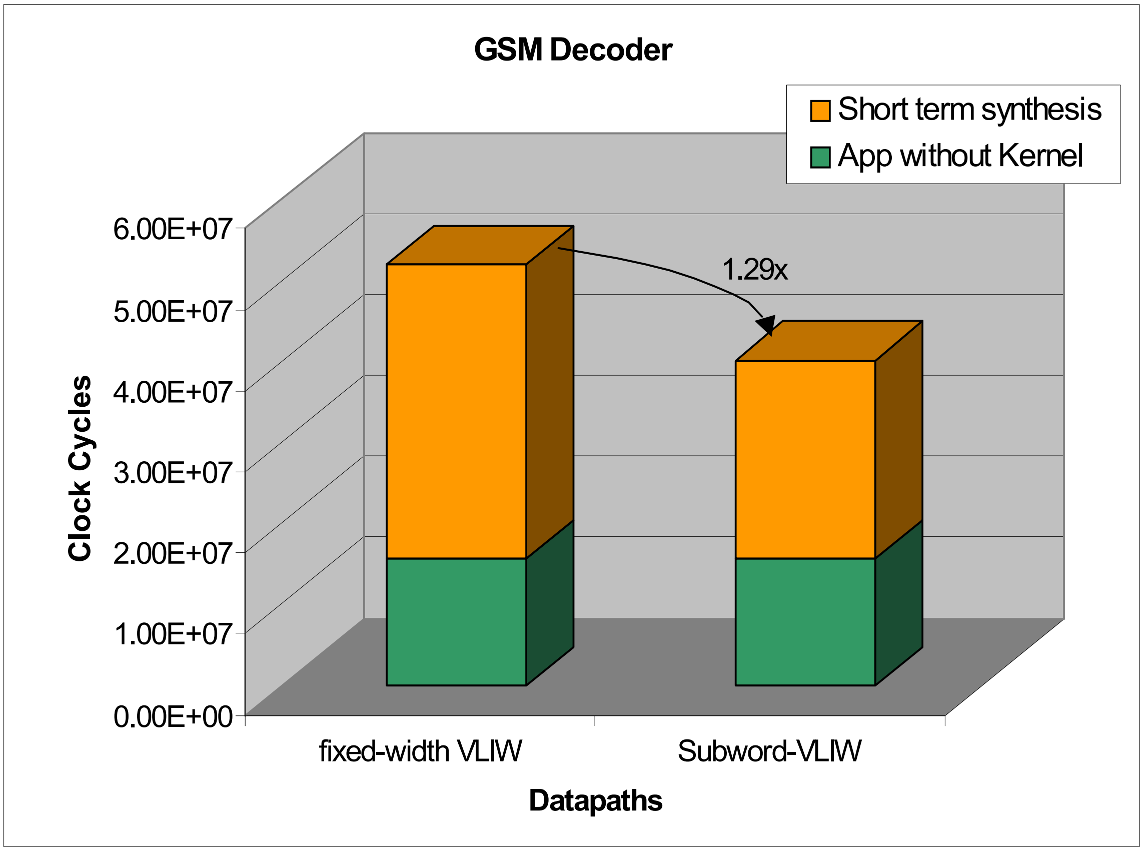


Figure 64 The performance impact on performing loop unrolling on the inner loop.

In order to extract even more parallelism, we introduce temporary variables to hold intermediate results and then perform data dependence analysis to identify more independent operations within the kernel. The resulting body of the pipelined kernel is shown below:

```
tmp1_7 = rrp[7];
tmp1_6 = rrp[6];
tmp1_5 = rrp[5];
tmp1_4 = rrp[4];
tmp1_3 = rrp[3];
tmp1_2 = rrp[2];
tmp1_1 = rrp[1];
tmp1_0 = rrp[0];
while (k--) {
         sri = *wt++;

         tmp2_7 = v[7];tmp2_6 = v[6];tmp2_5 = v[5];tmp2_4 = v[4];tmp2_3 = v[3];
         tmp2_2 = v[2];tmp2_1 = v[1];tmp2_0 = v[0];

         tmp2_7 = GSM_MULT_R(tmp1_7,tmp2_7); tmp2_6 = GSM_MULT_R(tmp1_6,tmp2_6);
         tmp2_5 = GSM_MULT_R(tmp1_5, tmp2_5);
         tmp2_4 = GSM_MULT_R(tmp1_4,tmp2_4); tmp2_3 = GSM_MULT_R(tmp1_3,tmp2_3);
         tmp2_2 = GSM_MULT_R(tmp1_2, tmp2_2);
         tmp2_1 = GSM_MULT_R(tmp1_1, tmp2_1); tmp2_0 = GSM_MULT_R(tmp1_0, tmp2_0);

         sri_7    = GSM_SUB( sri, tmp2_7 );            /* iteration #1 */

         tmp1_7 = GSM_MULT_R(tmp1_7, sri_7);           /* iteration #1 */
         sri_6   = GSM_SUB( sri_7, tmp2_6 );           /* iteration #2 */

         v[8]   = GSM_ADD( v[7], tmp1_7);              /* iteration #1 */
         tmp1_6 = GSM_MULT_R(tmp1_6, sri_6);           /* iteration #2 */
         sri_5  = GSM_SUB( sri_6, tmp2_5 );            /* iteration #3 */

         v[7]   = GSM_ADD( v[6], tmp1_6);              /* iteration #2 */
         tmp1_5 = GSM_MULT_R(tmp1_5, sri_5);           /* iteration #3 */
         sri_4  = GSM_SUB( sri_5, tmp2_4 );            /* iteration #4 */

         v[6]   = GSM_ADD( v[5], tmp1_5);              /* iteration #3 */
         tmp1_4 = GSM_MULT_R(tmp1_4, sri_4);           /* iteration #4 */
         sri_3  = GSM_SUB( sri_4, tmp2_3 );            /* iteration #5 */

         v[5]   = GSM_ADD( v[4], tmp1_4);              /* iteration #4 */
         tmp1_3 = GSM_MULT_R(tmp1_3, sri_3);           /* iteration #5 */
         sri_2  = GSM_SUB( sri_3, tmp2_2 );            /* iteration #6 */

         v[4]   = GSM_ADD( v[3], tmp1_3);              /* iteration #5 */
         tmp1_2 = GSM_MULT_R(tmp1_2, sri_2);           /* iteration #6 */
         sri_1  = GSM_SUB( sri_2, tmp2_1 );            /* iteration #7 */

         v[3]   = GSM_ADD( v[2], tmp1_2);              /* iteration #6 */
         tmp1_1 = GSM_MULT_R(tmp1_1, sri_1);           /* iteration #7 */
         sri_0  = GSM_SUB( sri_1, tmp2_0 );            /* iteration #8 */

         v[2]   = GSM_ADD( v[1], tmp1_1);              /* iteration #7 */
         tmp1_0 = GSM_MULT_R(tmp1_0, sri_0);           /* iteration #8 */

         v[1]   = GSM_ADD( v[0], tmp1_0);              /* iteration #8 */

         *sr++ = v[0] = sri_0;
        }
```

Figure 65 The body of the loop, after unrolling, moving loop invariant code and pipelining the unrolled loop.

In order to compare the performance of the base version of the kernel to the unrolled and the pipelined versions of the kernel, we compile the application using different optimization paths, basic block, and hyper block. Further we simulate the execution of the resulting kernel on a subword VLIW machine with register file of size 256, 16 integer ALUs, and 2 memory units. The performance results of the above transformation is shown in Figure 66.

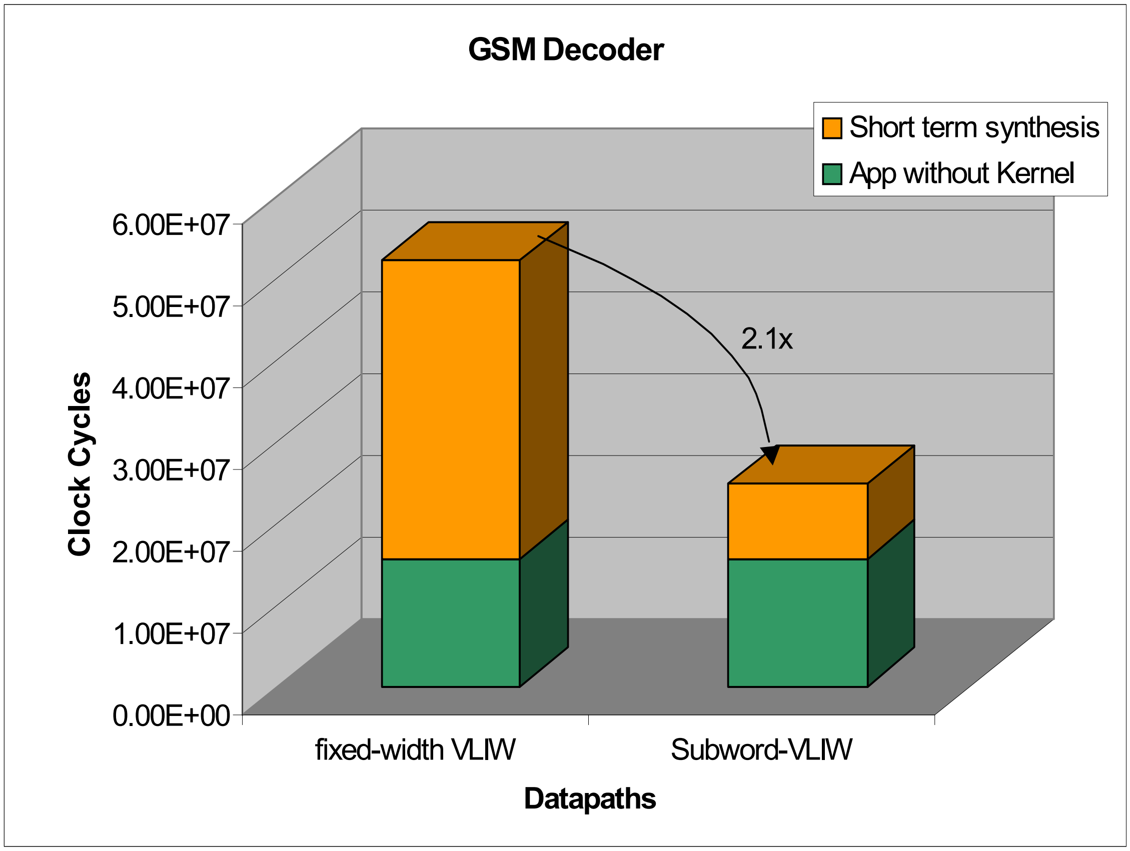


Figure 66 The performance impact of pipelining on the inner loop.

The code transformations proved extremely useful at easing the level of data-dependence. Due to these transformations, the performance speedup increased to a factor of 2.1 over the base case.

6.1.3 Performance Analysis of the PEGWIT Encryption

The kernel of the PEGWIT algorithm consists of two functions. The first performs a Galois Field (GF) element multiplication and the second performs a GF multiply and add operation. These kernels have been analyzed in section 3.4. The kernels have a complex control-flow structure consisting of a series of nested *for-loops* and *if-statements*. Further, the bodies of these nested structures contain a single operation. This limits the chances for parallel execution.

For the PEGWIT encryption algorithm, we anticipate little speedup due to the complex control structure in the nested loops as well as the data dependences. We execute this application on the fixed-width datapath and compare the performance to executing on the subword datapath. The results are shown in Figure 67.

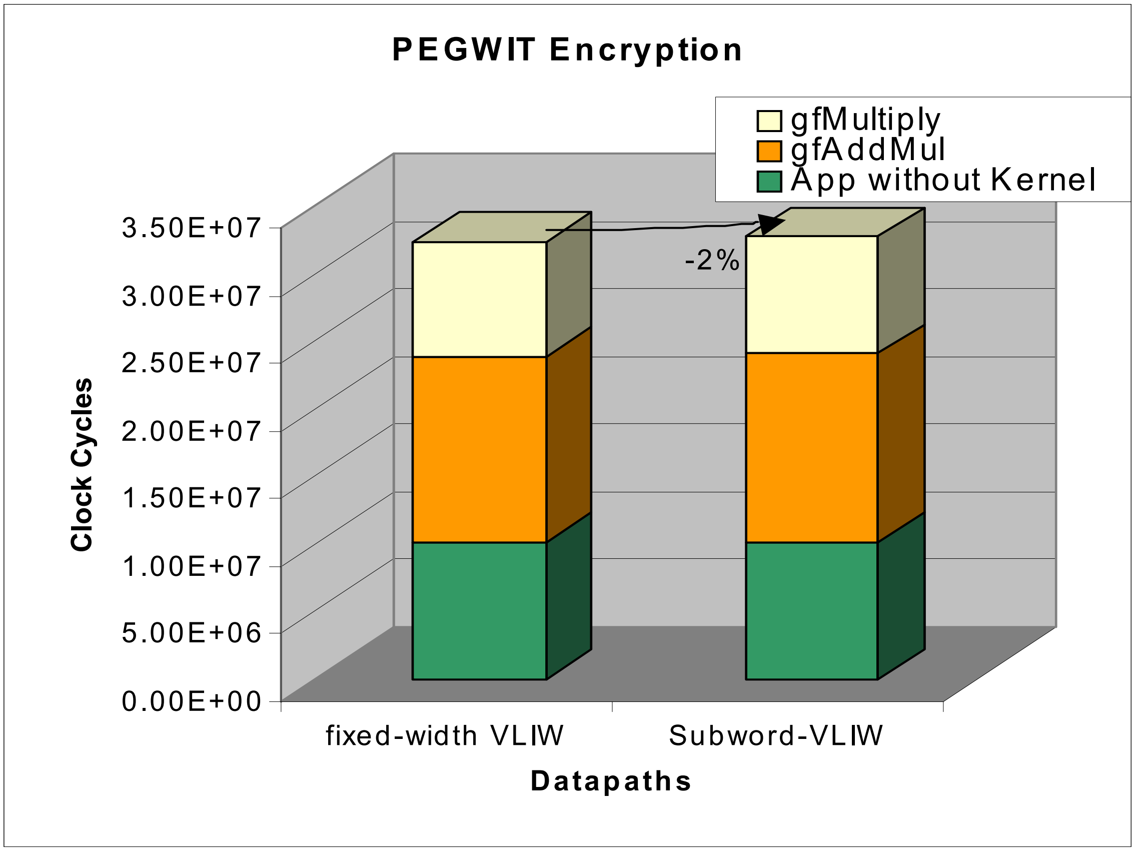


Figure 67 The performance comparison of fixed-width datapath to subword datapath.

As anticipated, the application incurred a slight slowdown using the subword datapath. Next, we enable the complex compiler code transformations and perform the same experiment on the subword datapath. The results are shown in Figure 68.

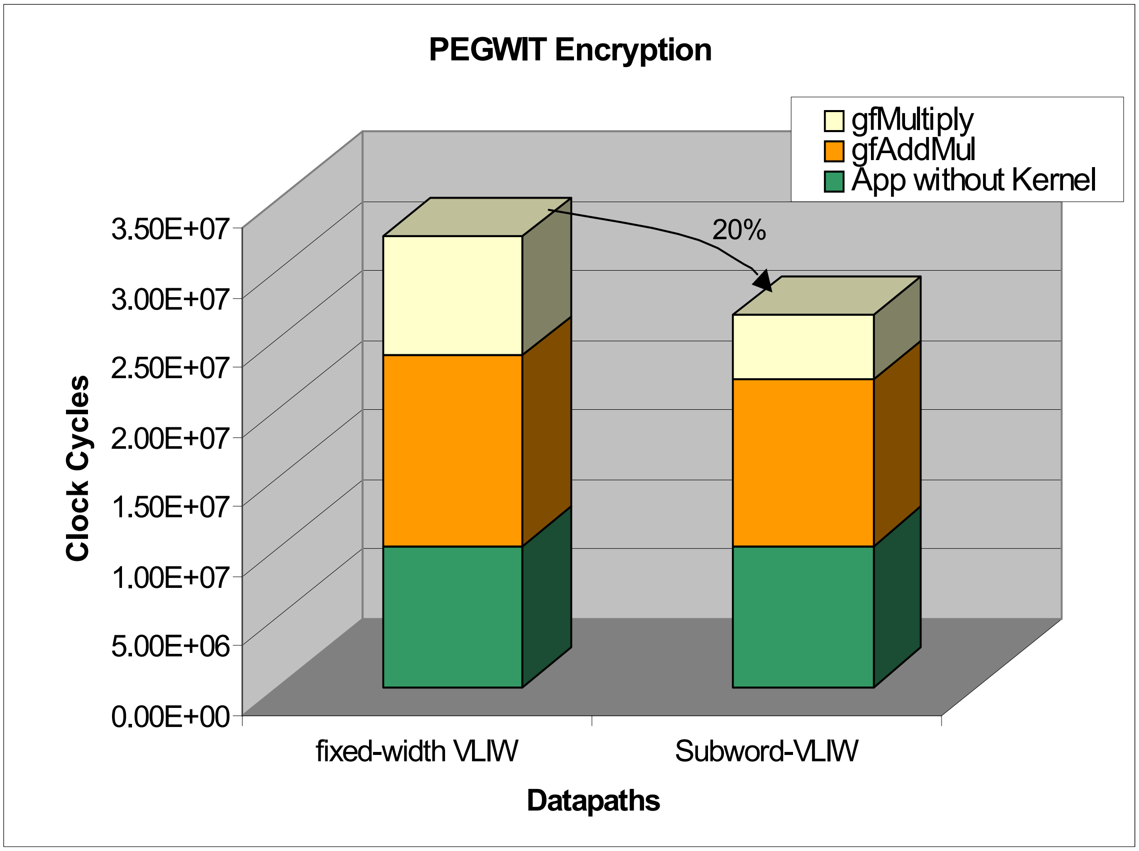


Figure 68 The performance impact on enabling aggressive compiler techniques.

The hyperblock formation optimization managed to extract some parallelism from the application in spite of the complex control-structure. The compiler took advantage of the subword datapath and achieved a speedup of 20% over the base case.

Finally, code transformations that did not include a redesign of the loop structure code in the kernel did not result in any further speedups. We do not evaluate code transformations that require altering the implementation of the algorithm under the constraint that we cannot require the developers to re-implement general purpose applications with intimate knowledge of the underlying architecture in order to achieve significant speedups.

6.1.4 Performance Analysis of the PEGWIT Decryption

For decryption algorithm, the two kernels are the same ones as the encryption algorithm. The first performs a Galois Field (GF) element multiplication and the second performs a GF multiply and add operation.

We perform the base case execution of the application on the fixed-width VLIW datapath and compare the performance to the subword datapath. The comparison is depicted in Figure 69.

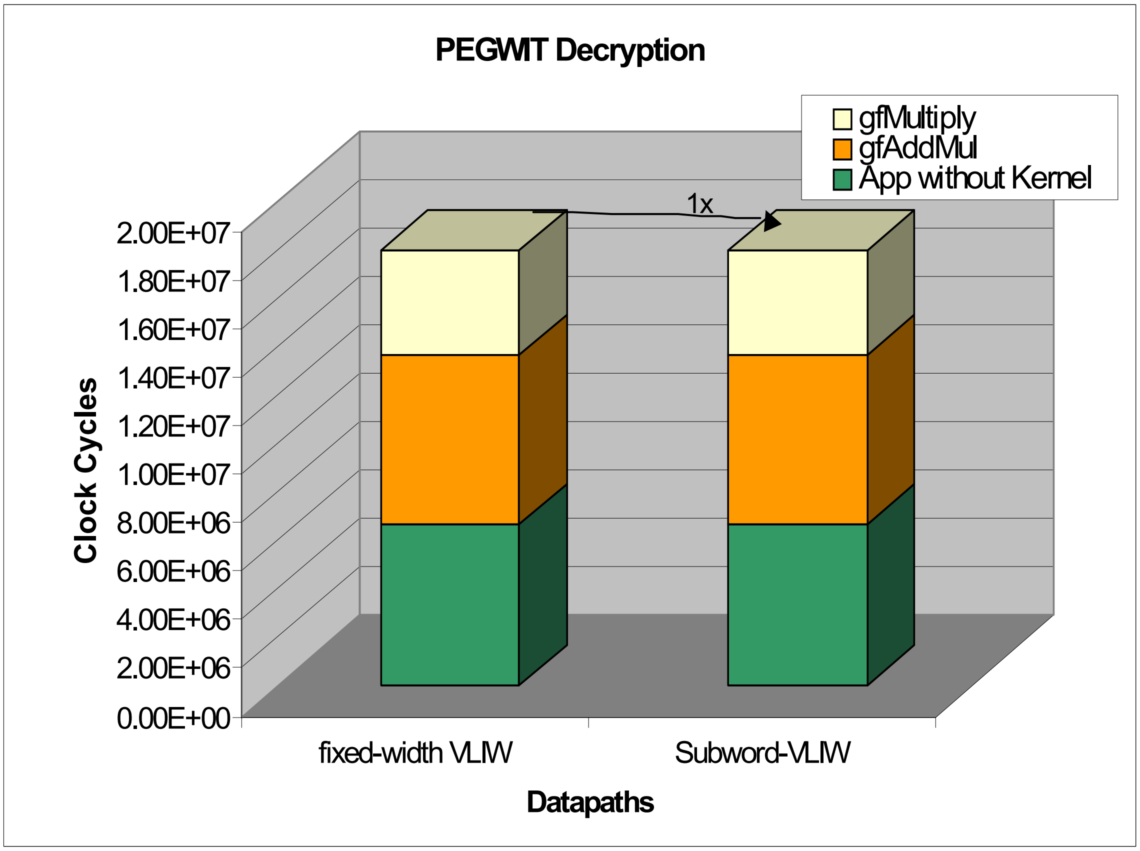


Figure 69 The performance comparison of executing the decryption algorithm on a fixed-width datapath and a subword datapath.

As expected, due to the complex structure in the kernel code, the compiler without aggressive code transformations cannot achieve any speedup.

We enable complex compiler transformations and measure the clock cycle count for executing the pegwit decryption algorithm on the subword datapath. The comparison to the base case is depicted in Figure 70.

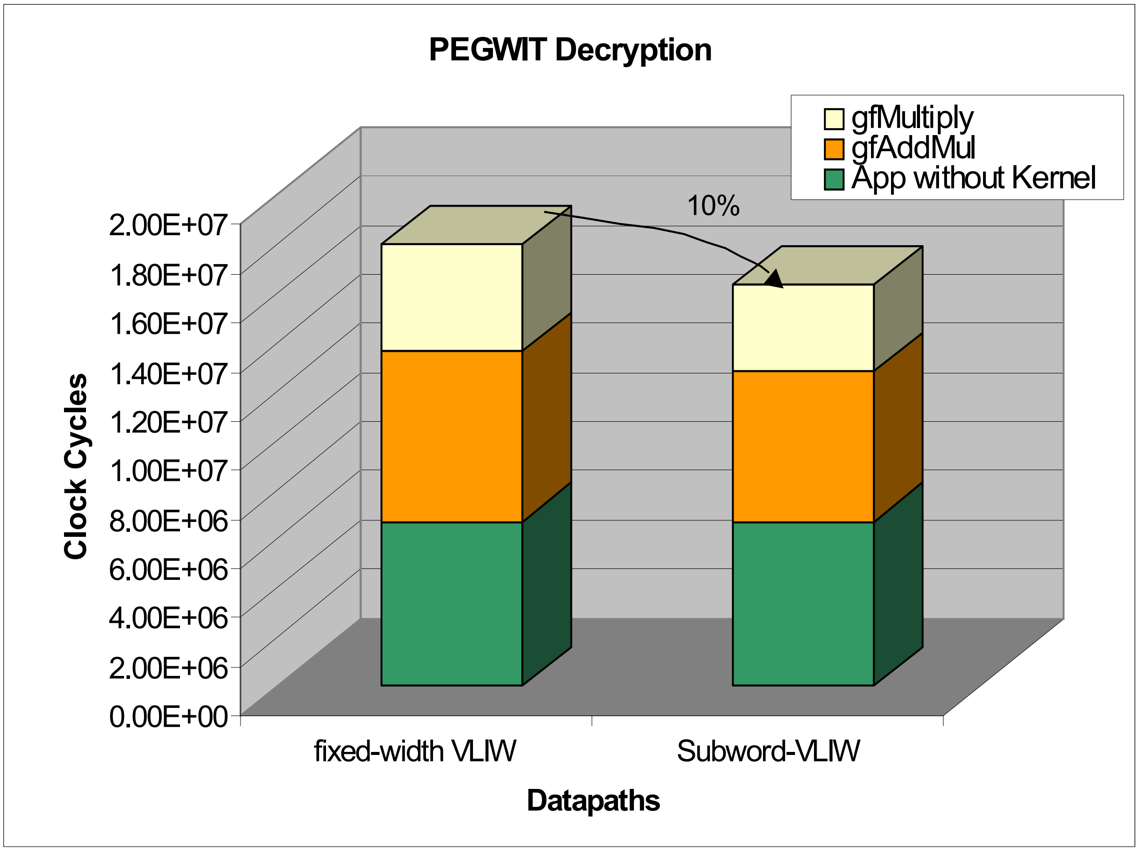


Figure 70 Performance due to employing hyperblock formation in the compiler.

The compiler is capable of extracting some parallelism from the application and better scheduling it on the subword datapath. A speedup of 10% is achieved. The speedup is not equivalent to that of the encryption algorithm since the control flow is data dependent and hence executing different portions of the kernels where little parallelism exists.

6.1.5 Performance Analysis of the ADPCM Encoder

The code structure of the ADPCM kernel is a single loop. The upper loop bound is data dependent and hence unknown at compile time. We compile and execute this application on the fixed-width datapath and compare the performance to that of the subword datapath using simple compiler techniques. The results are depicted in Figure 71.

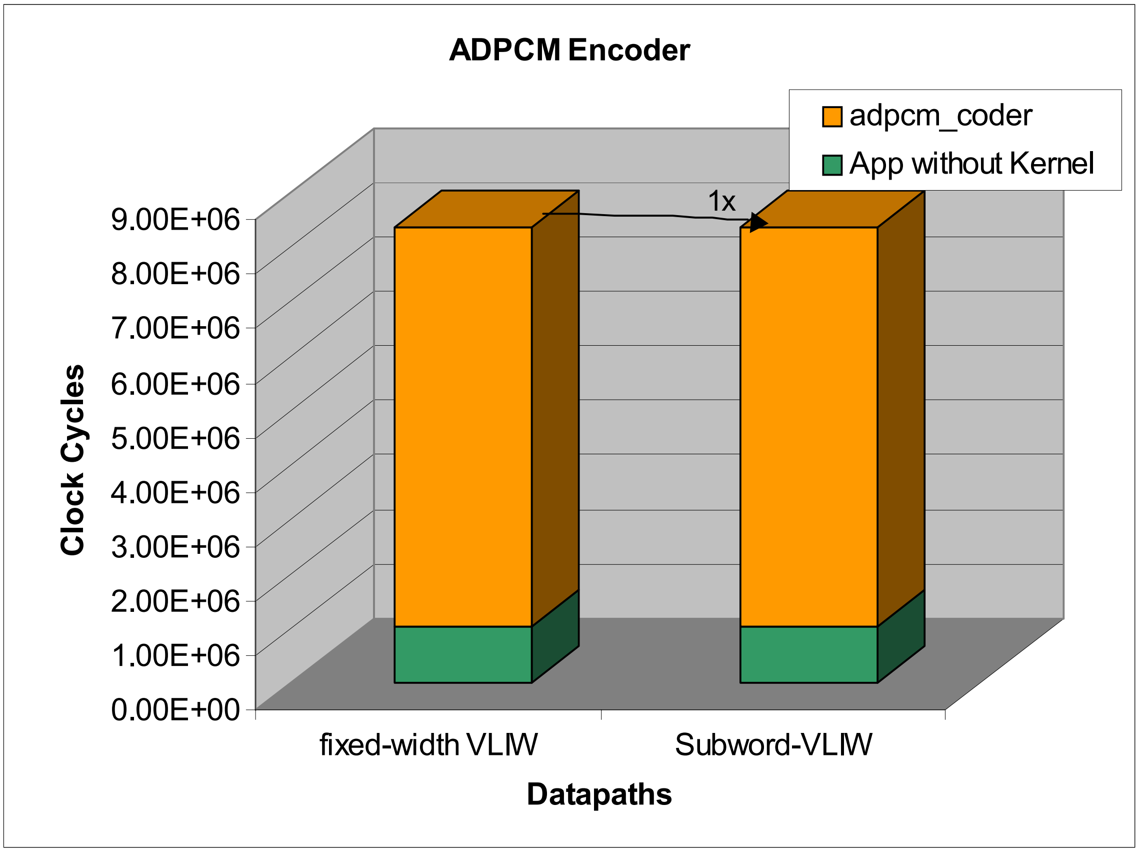


Figure 71 The performance impact of executing the adpcm application on both the fixed-width and subword datapaths.

The compiler is not capable of extracting any parallelism from the loop of the adpcm_coder kernel when targeting the subword datapath. This resulted in no performance speedup.

However, if we enable more aggressive compiler algorithms, such as hyperblock formation, the compiler is capable of achieving a better schedule and a speedup of 1.5 as shown in Figure 72.

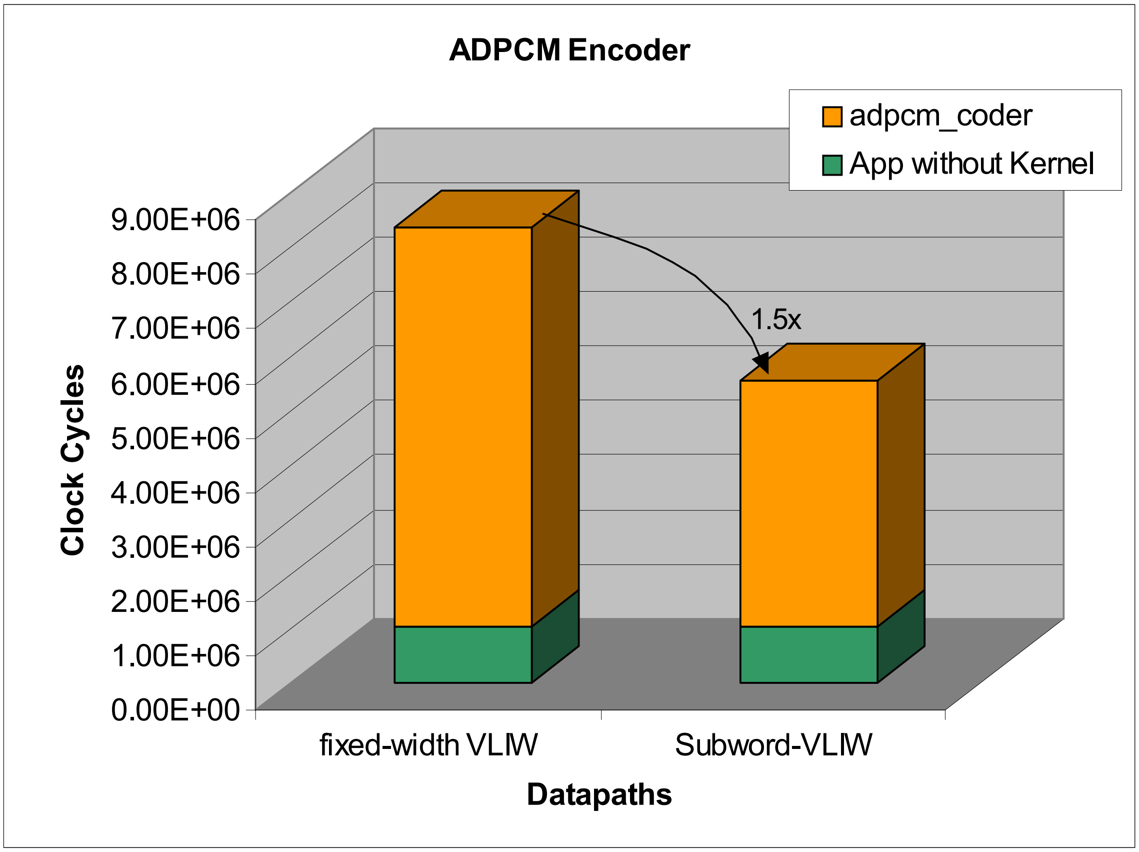


Figure 72 The performance impact of performing hyperblock formation on the kernel.

If we perform code transformations and unroll the inner loop once, the compiler is capable of achieving an even better schedule and a speedup of 1.85 as shown in Figure 73.

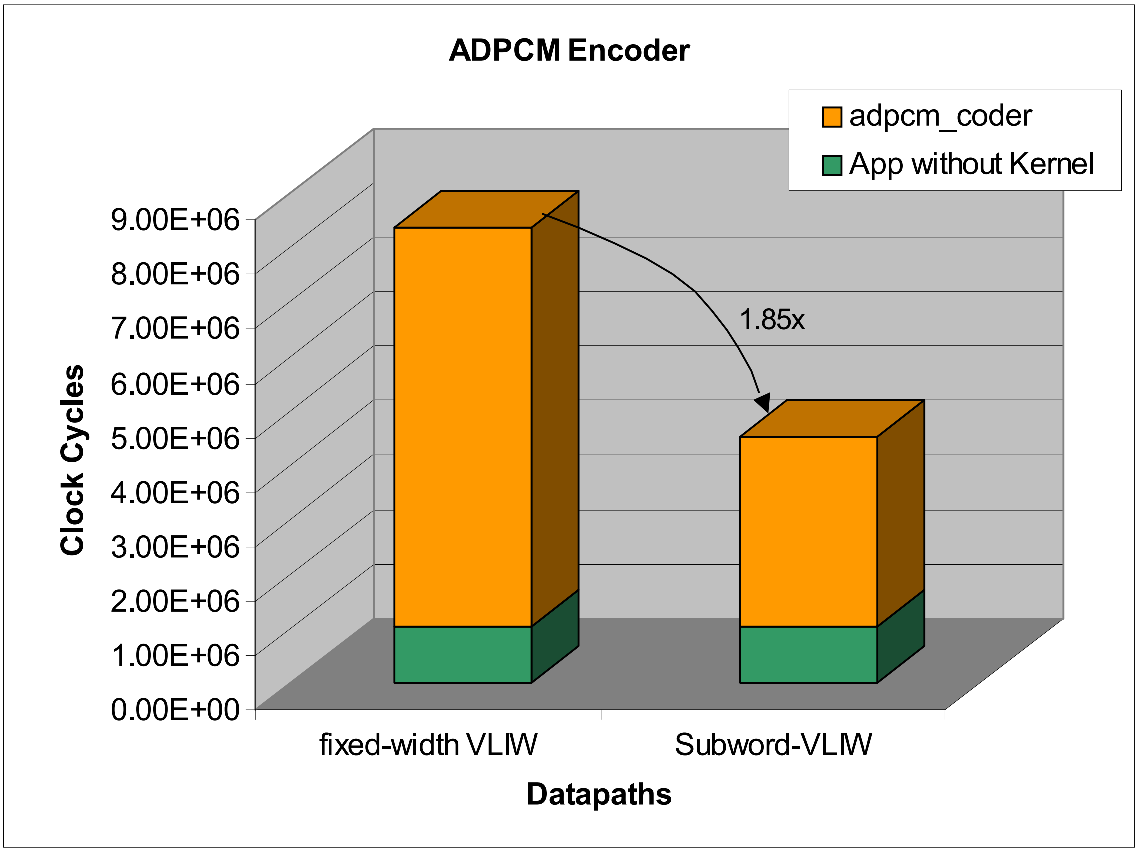


Figure 73 The performance impact of unrolling the inner loop and performing hyperblock formation on the kernels.

6.1.6 Performance Analysis of the ADPCM Decoder

The loop body for the adpcm_decoder kernel is slightly different than the encoder (section 3.7), however, the complexity is the same. When we compare the performance of compiling and executing this application on the fixed-width datapath and the subword datapath using simple compiler transformations and optimization techniques, we do not observe any speedup (Figure 74).

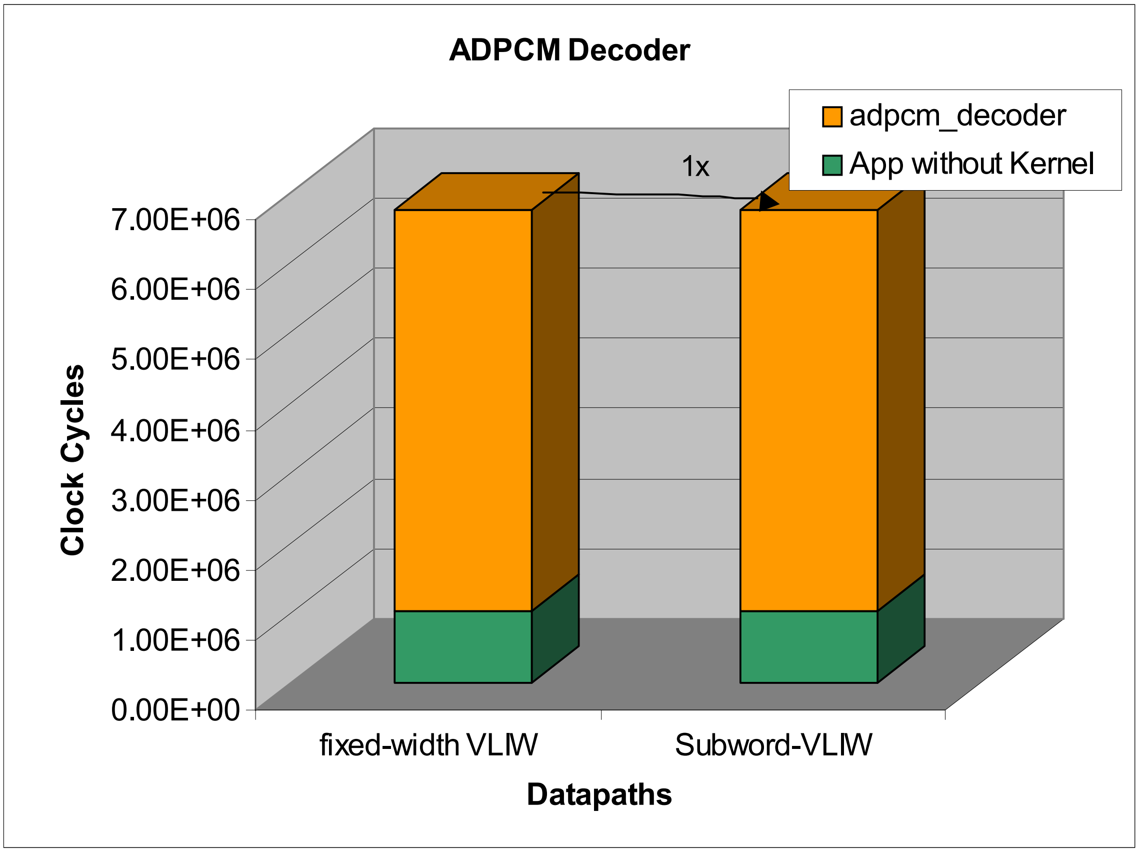


Figure 74 The performance comparison of targeting a fixed-width datapath and a subword datapath.

However, when we allow the compiler to perform aggressive code transformations, hyperblock formation, it is able to extract enough parallelism and a better schedule to achieve a speedup of 1.95 compared to the base case. This comparison is depicted in Figure 75.

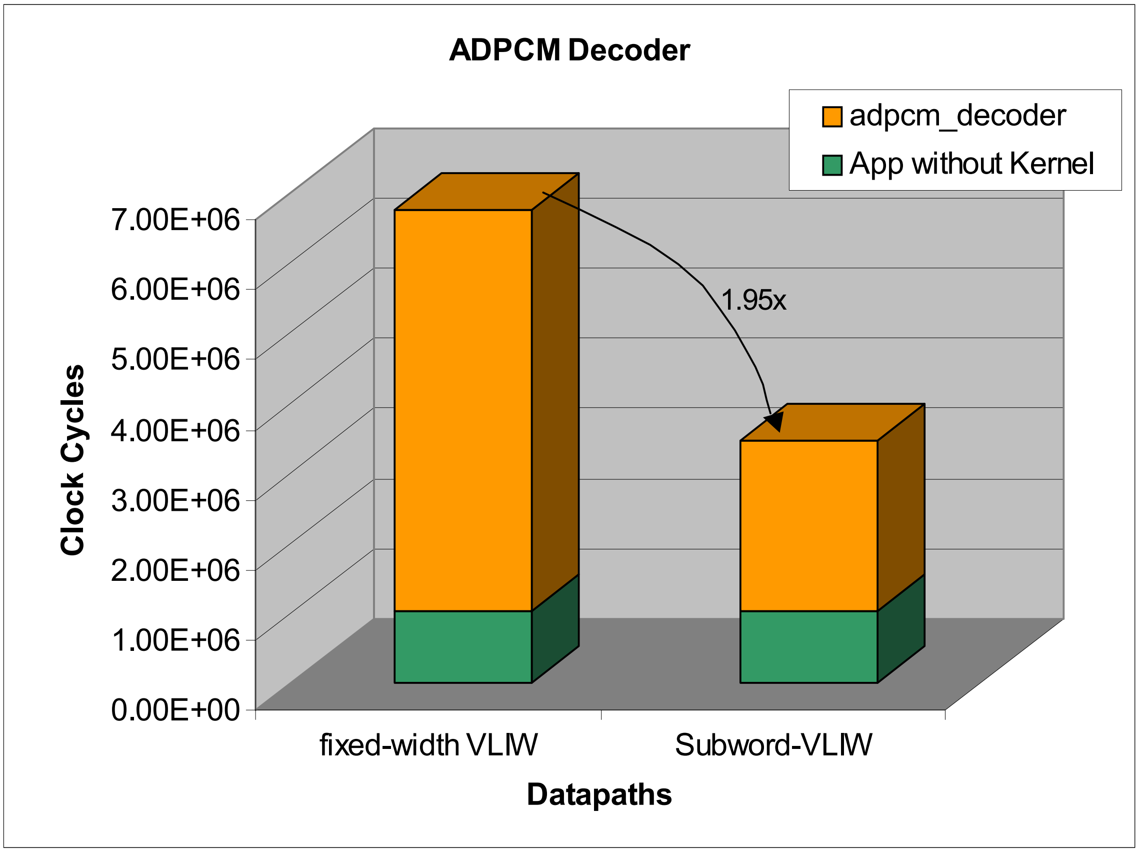


Figure 75 The performance comparison of targeting a fixed-width datapath and a subword datapath.

If we perform simple code transformations, such as unrolling the loop of the adpcm decoder kernel 4 times, we get an even greater speedup of 2.46 as shown in Figure 76.

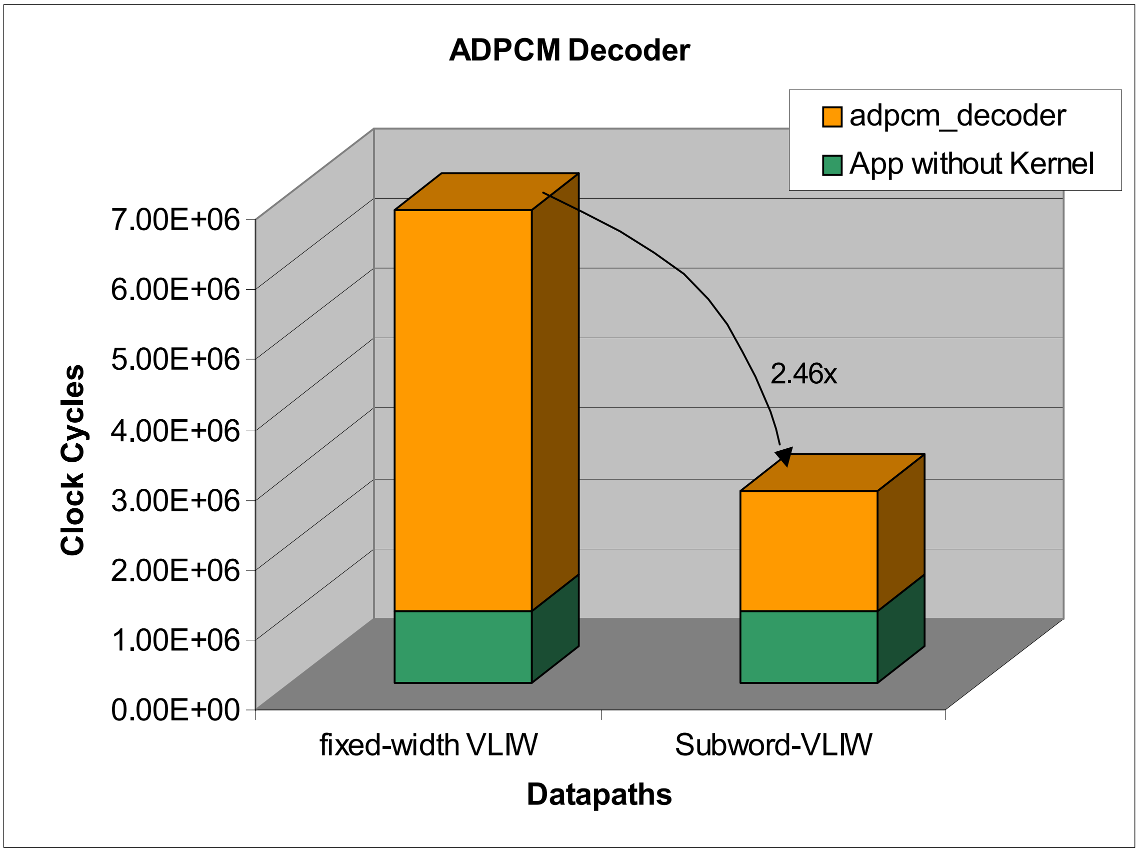


Figure 76 The performance impact of unrolling the inner loop four times and performing hyperblock formation on the kernels.

Although the loop has inter-loop data dependence, unrolling the loop exposes more opportunity for parallel execution.

6.1.7 Performance Analysis of the MPEG-2 Encoder

The kernel for the MPEG-2 encoder is the motion estimation vector computation. The kernel contains a sequence of four nested loops, where the top loop has been unrolled (section 3.8). We compile and execute the application on both the fixed-width VLIW datapath and the subword datapath. The results are shown below in Figure 77.

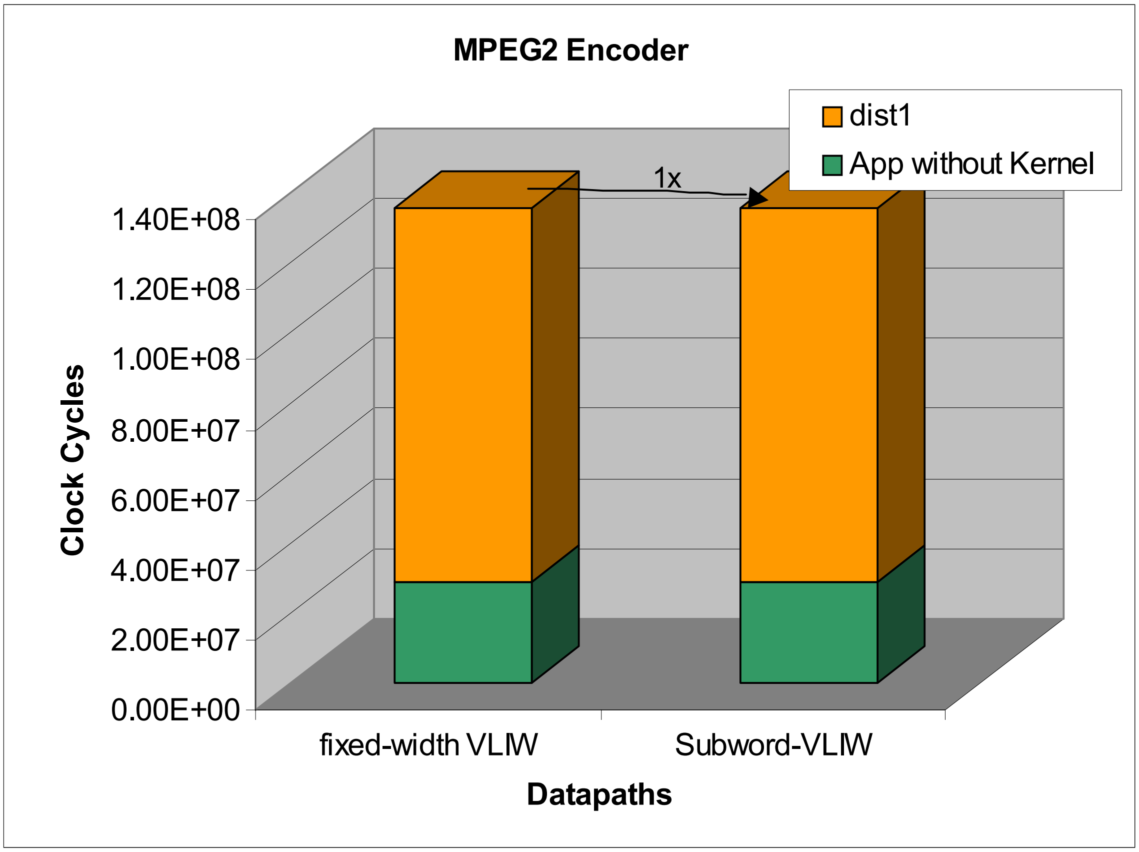


Figure 77 The performance comparison of executing the mpeg2 kernel on the fixed-width datapath and subword datapath.

The simple compiler techniques are not capable of extracting any parallelism in order to exploit the parallel subword hardware resources in the subword-VLIW datapath.

When we enable the hyperblock formation algorithm in the compiler, it is then capable of exploiting more parallelism and achieving shorter schedules. This is interpreted through the speedup of 1.38 shown in Figure 78.

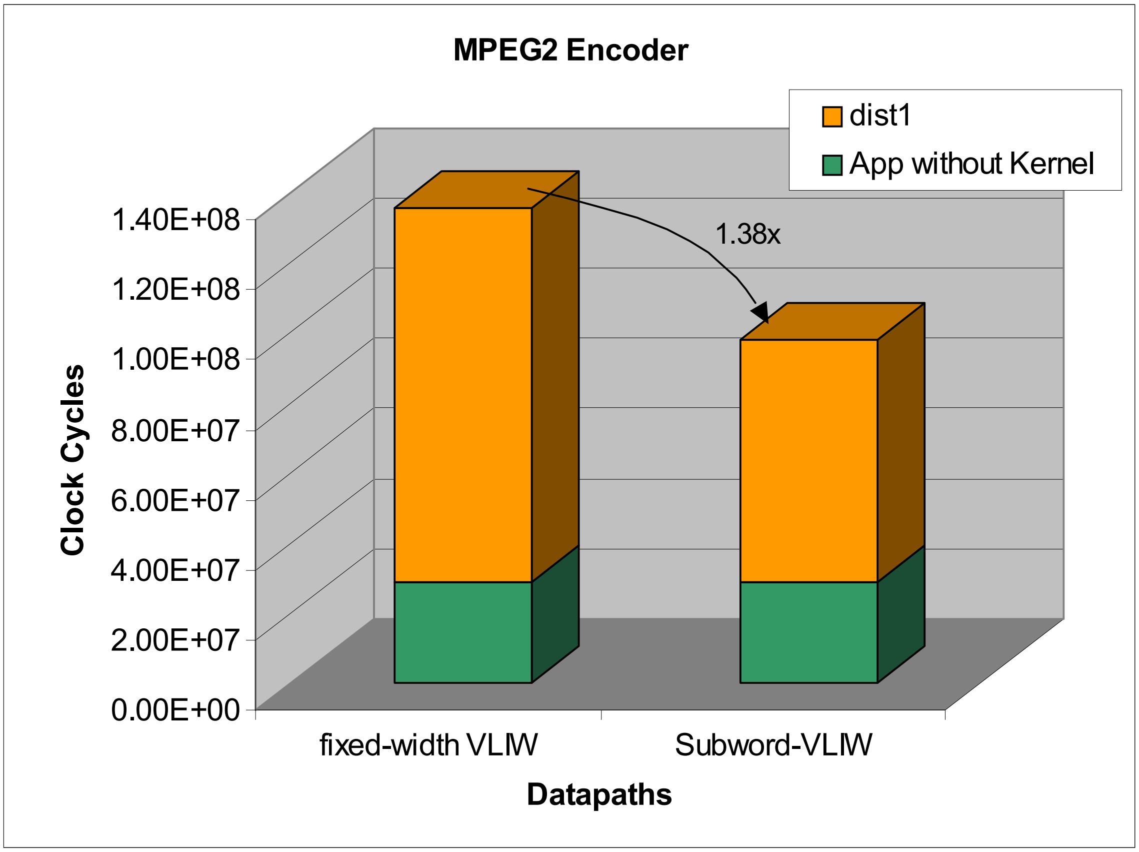


Figure 78 The performance impact due to enabling hyperblock formation when targeting the subword datapath.

When we perform code transformations by unrolling the three inner loops of motion estimation we observe (Figure 79) a slight increase in speedup to 1.41. This is due to the fact that the aggressive compiler techniques are already performing similar transformations and hence the minimal increase in speedup.

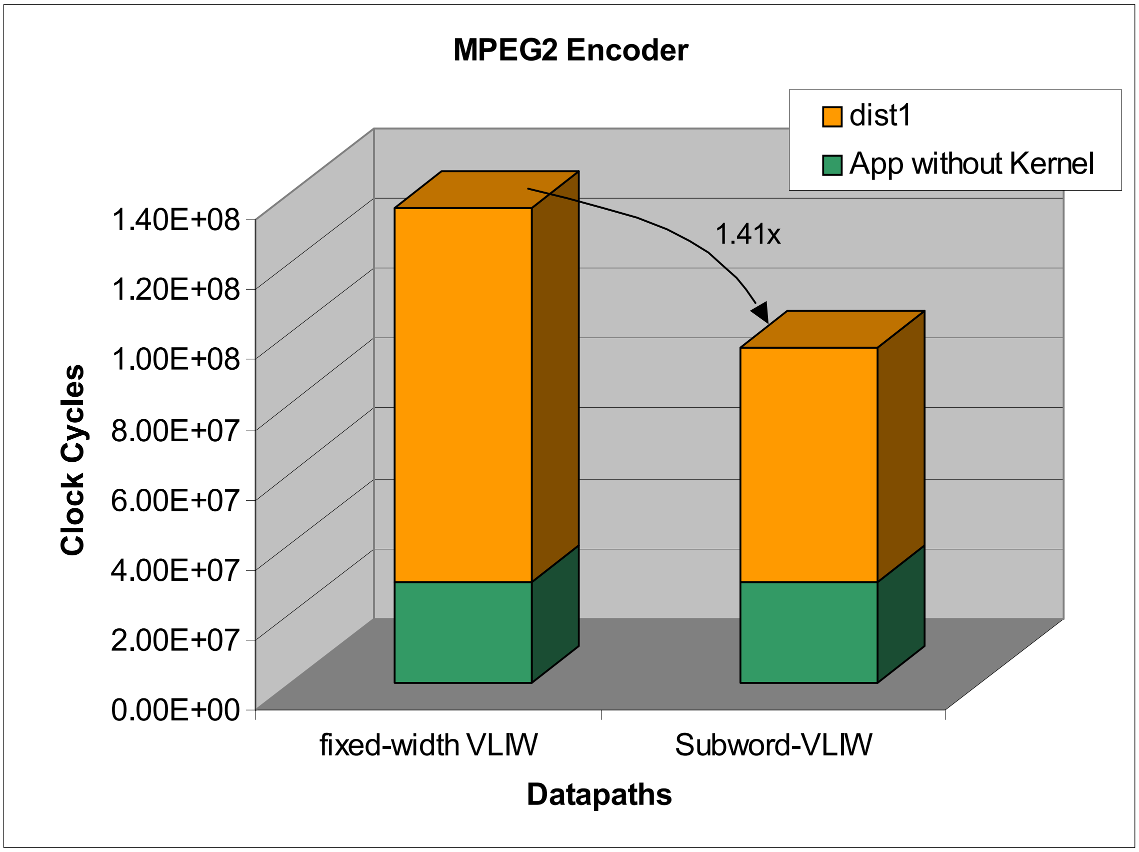


Figure 79 The performance benefit due to loop unrolling.

6.1.8 Performance Analysis of the MPEG-2 Decoder

The code kernel is the fast IDCT algorithm, a two dimensional inverse discrete cosine transform (section 3.9). When we compile and execute the kernel of this application using standard optimization methods, on both the fixed-width datapath and subword datapath, we observe a speedup of 1.68 (Figure 80).

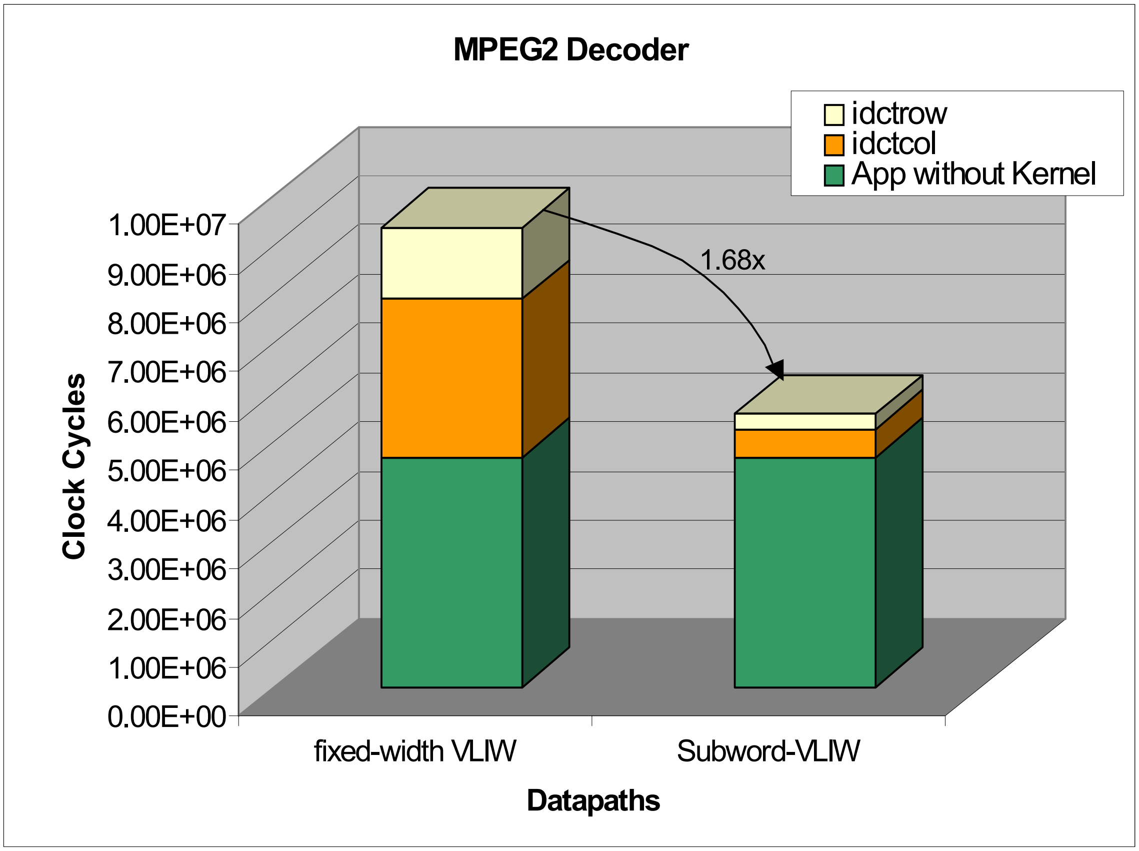


Figure 80 The performance impact on compiling and executing the application on the fixed-width datapath and the subword datapath.

These two kernels do not require advanced compilation techniques for the compiler to extract some parallelism and exploit it while scheduling onto the subword-VLIW datapath.

When we enable advanced compiler techniques, a slight speedup increase to 1.7 is achieved as shown in Figure 81.

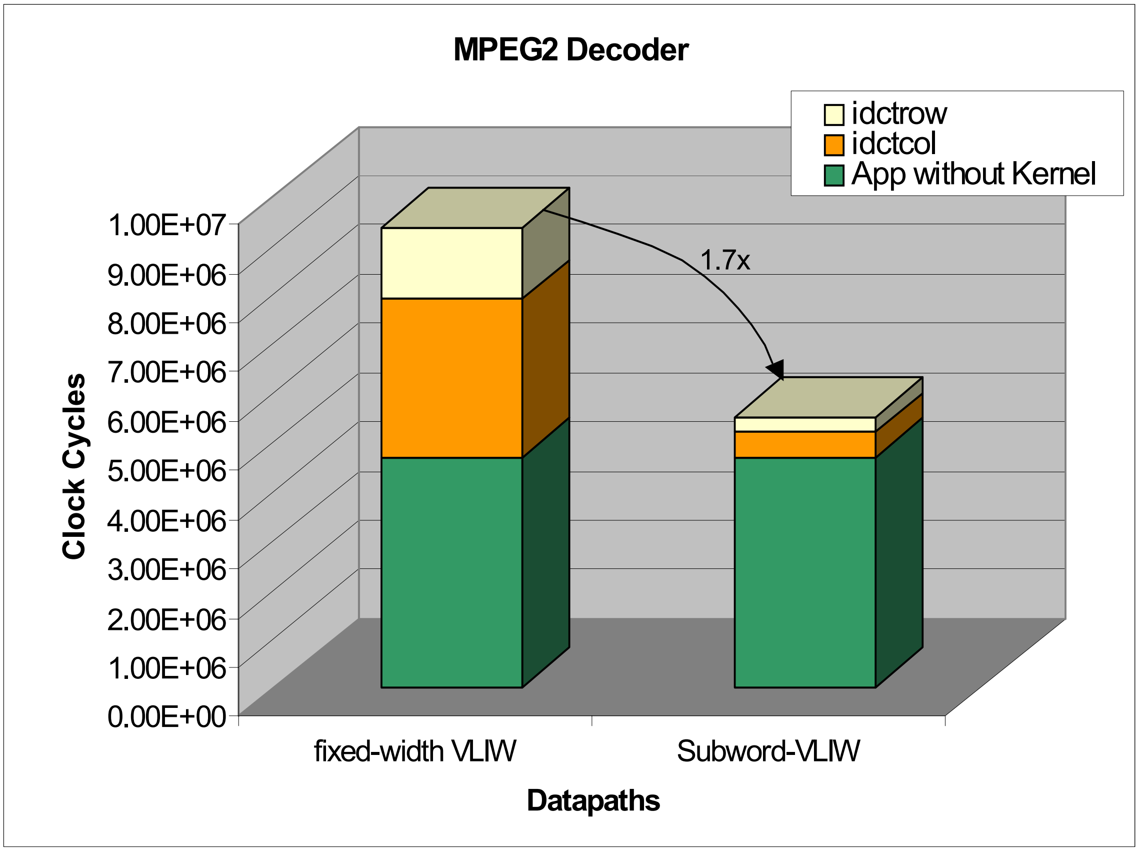


Figure 81 The performance impact on performing hyperblock formation on the kernels.

Performing simple code transformations did not result in any further speedups for this kernel.

6.1.9 Performance Analysis of the DIVX Encoder

The kernel for the DIVX encoder algorithm was identified in Section 3.10. The DIVX encoder spends 88% of all dynamic cycles in 4 small functions, which perform the motion estimation analysis. As discussed earlier, these four functions constitute the kernel of the DIVX encoder algorithm. Compiling and executing this application on a fixed width and a subword-VLIW datapath using simple compiler transformations has shown a speedup of 1.48 (Figure 82).

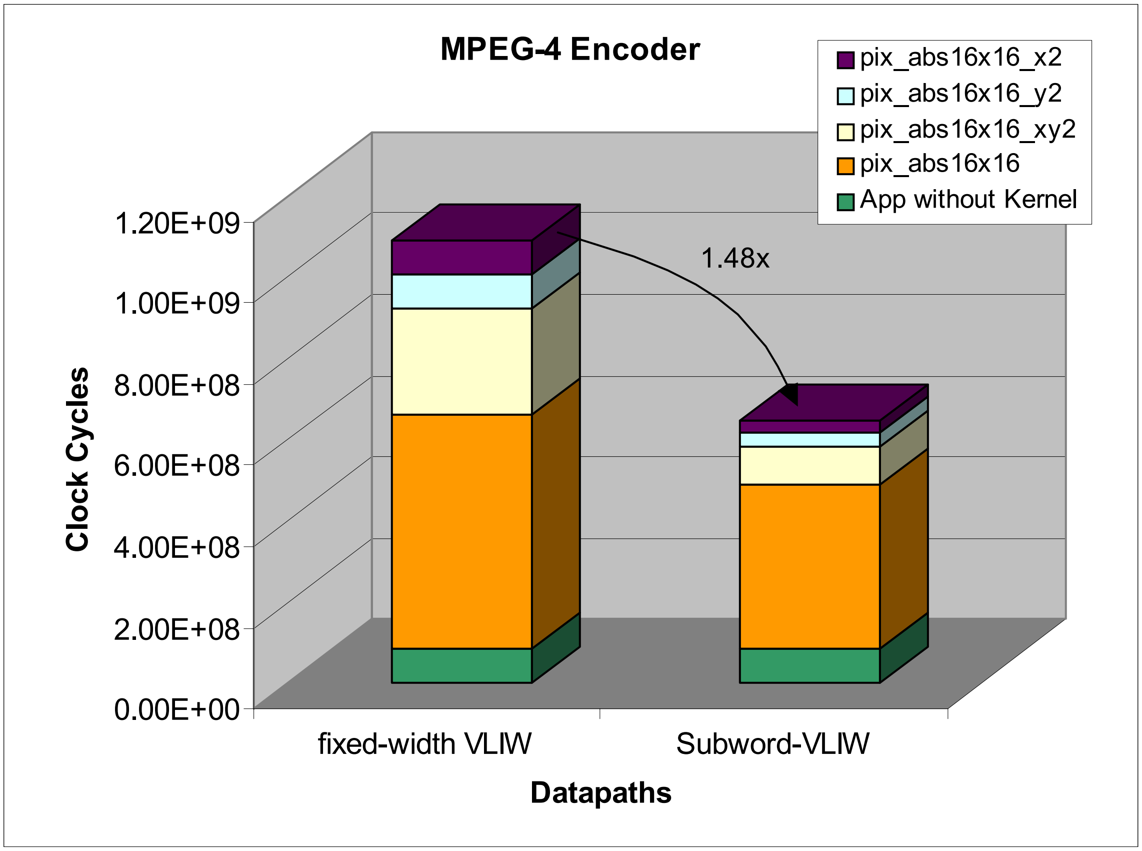


Figure 82 The performance impact due to compiling and executing the motion estimation kernels on the fixed-width datapath and the subword datapath.

Next, we enable hyperblock formation for these kernels and target the subword datapath. We observe a speedup of 1.61 as shown in Figure 83.

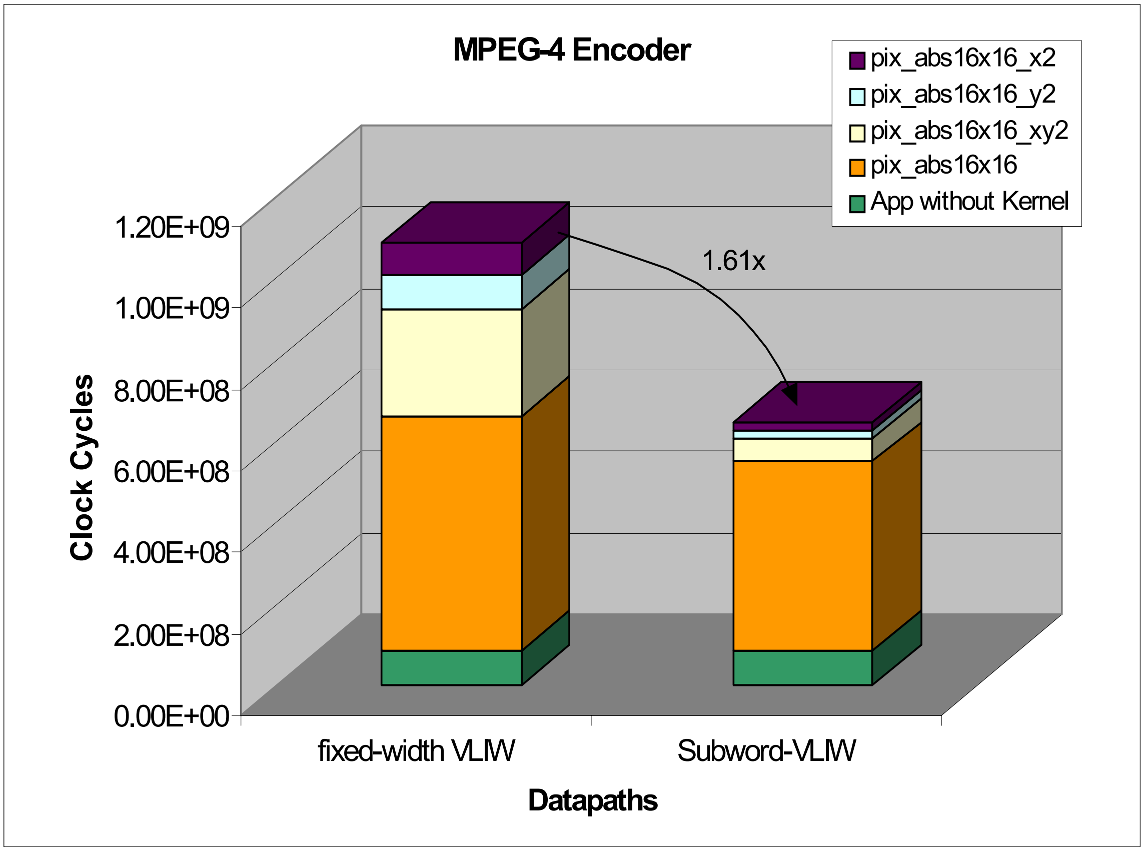


Figure 83 The performance impact due to compiling and executing the motion estimation kernels using hyperblock formation on the subword datapath.

Finally, we perform code transformations, where we transform all the abs operations to if-statements and add a few temporary variables in the loop bodies. We compile this kernel using hyperblock formation and compare the execution results to the base case in Figure 84.

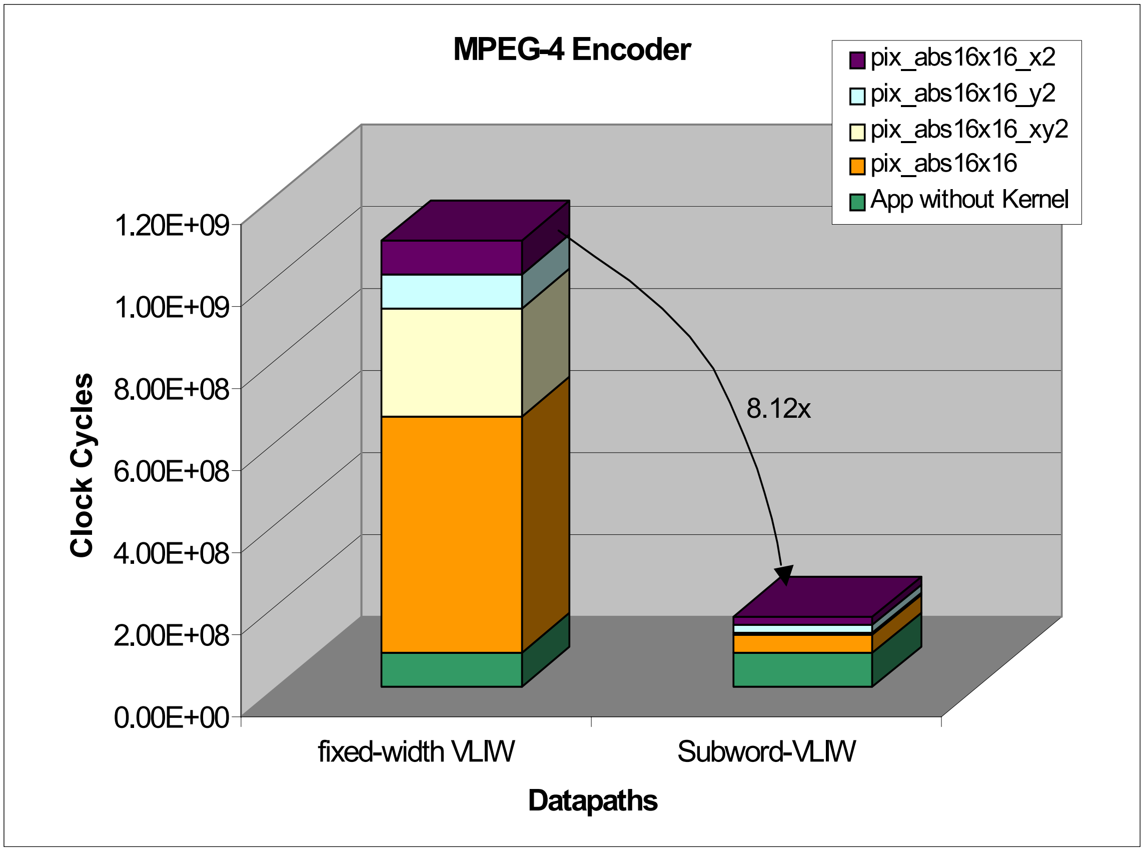


Figure 84 The performance after simple code transformations and after using hyperblock formation when targeting the subword datapath.

Now that the loop bodies include only if-statements surrounded by other operations, the compiler can transform the loop into a single hyperblock by using if-conversion on the if-statements as well as other optimizations. This speedup of 8.12 is very significant.

6.2 ANALYSIS OF EXPERIMENTAL RESULTS

In this section, we plot the performance results of the four experiments performed for each application.

For the GSM Encoder, the results of all the experiments performed are summarized in Figure 86. This application exhibits a lot of parallelism and, hence, simple compiler techniques are enough to extract a significant amount of parallelism. Subsequently, this parallelism is then exploited by targeting the subword VLIW datapath. More complex code transformations lead to more parallelism and higher speedups. Since the parallelism is explicit in this applications kernel

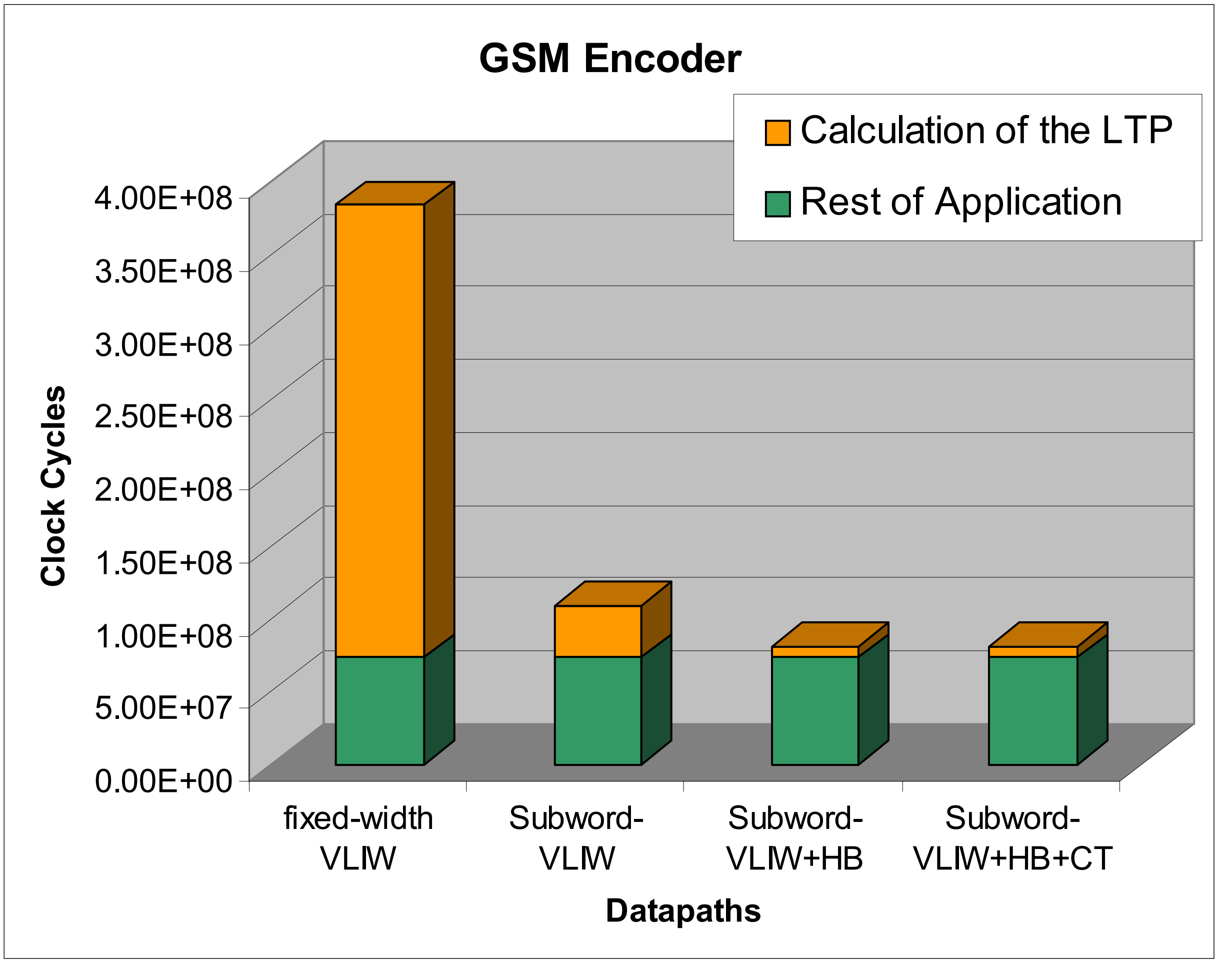


Figure 85 The relative execution times for the GSM Encoder application.

and since there are no control-flow structures within the body of the loop, we conjecture that a subword-SIMD datapath will perform equally well for this application.

The performance results of the GSM decoder are depicted in Figure 86. This application exhibits little parallelism and requires an aggressive compiler in order to extract some parallelism using speculative execution enabled by predicated execution in VLIW processors. Further, performance speedup due to simple code transformations indicates that more parallelism exists within the current implementation of the application and, hence, an opportunity for compiler optimizations to exploit.

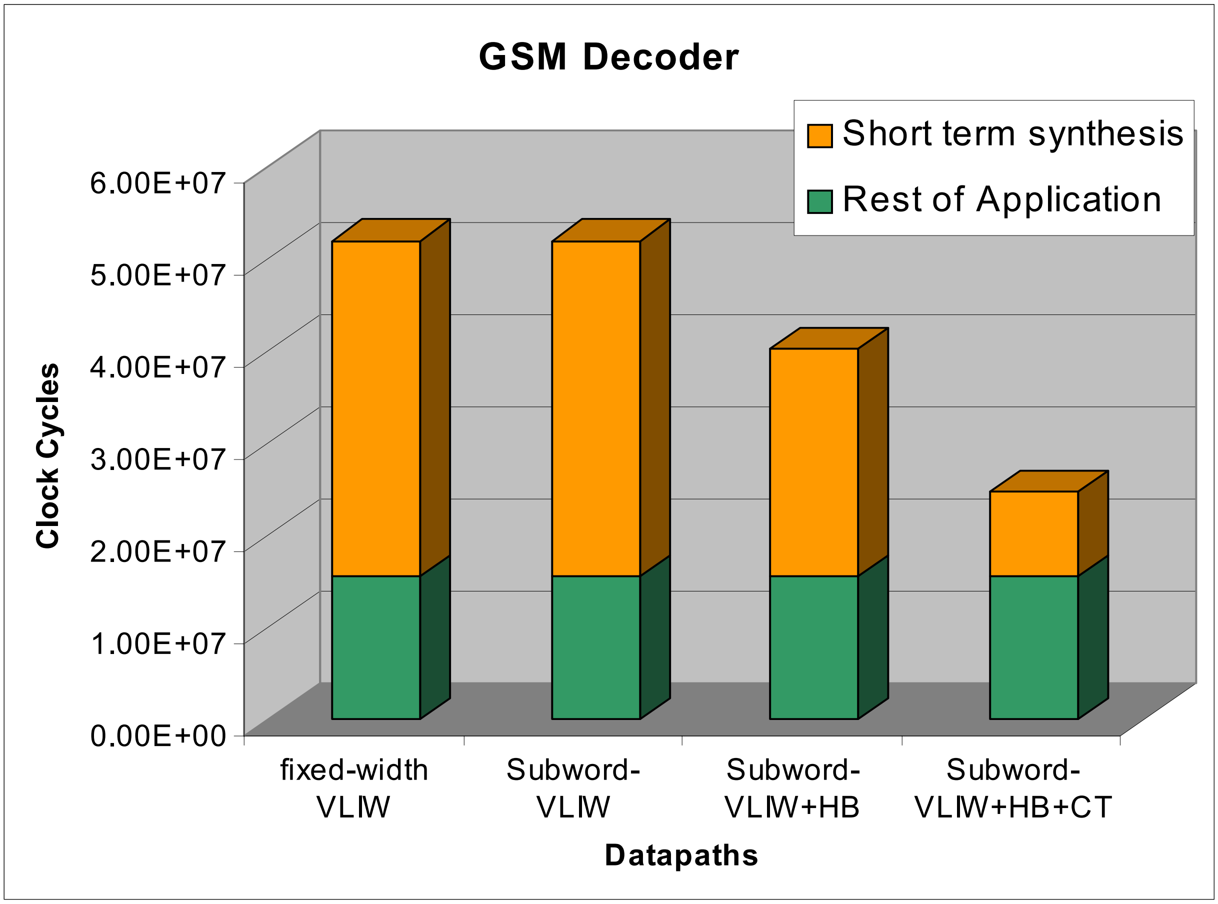


Figure 86 The overall performance speedups for the GSM Decoder application.

It is important to point out that the extraction of parallelism through aggressive compiler transformations can lead to performance slowdowns if the hardware resources cannot satisfy the overhead of executing speculative instructions. This behavior is shown in Figure 87, where performing hyperblock formation and code transformations will lead to a significant slowdown when targeting a fixed-width VLIW datapath.

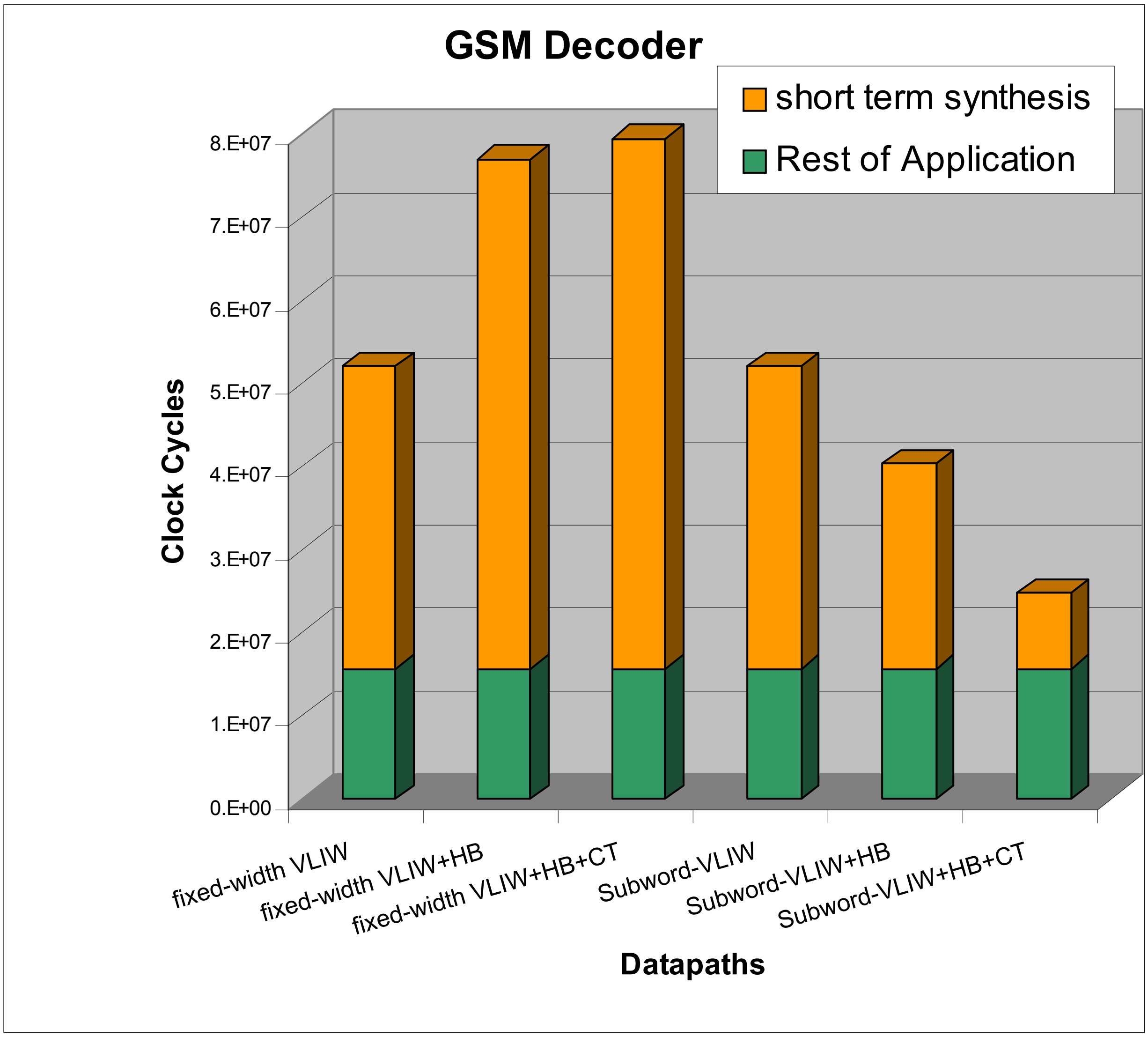


Figure 87 The performance impact of employing aggressive compiler transformations when targeting the fixed-width VLIW datapath for the GSM Decoder application.

The kernels of the PEGWIT encryption algorithm proved to be problematic for the compiler since it was not able to extract a lot of parallelism and achieve significant speedups when targeting the subword-VLIW datapath. The relative speedup results are depicted in Figure 88.

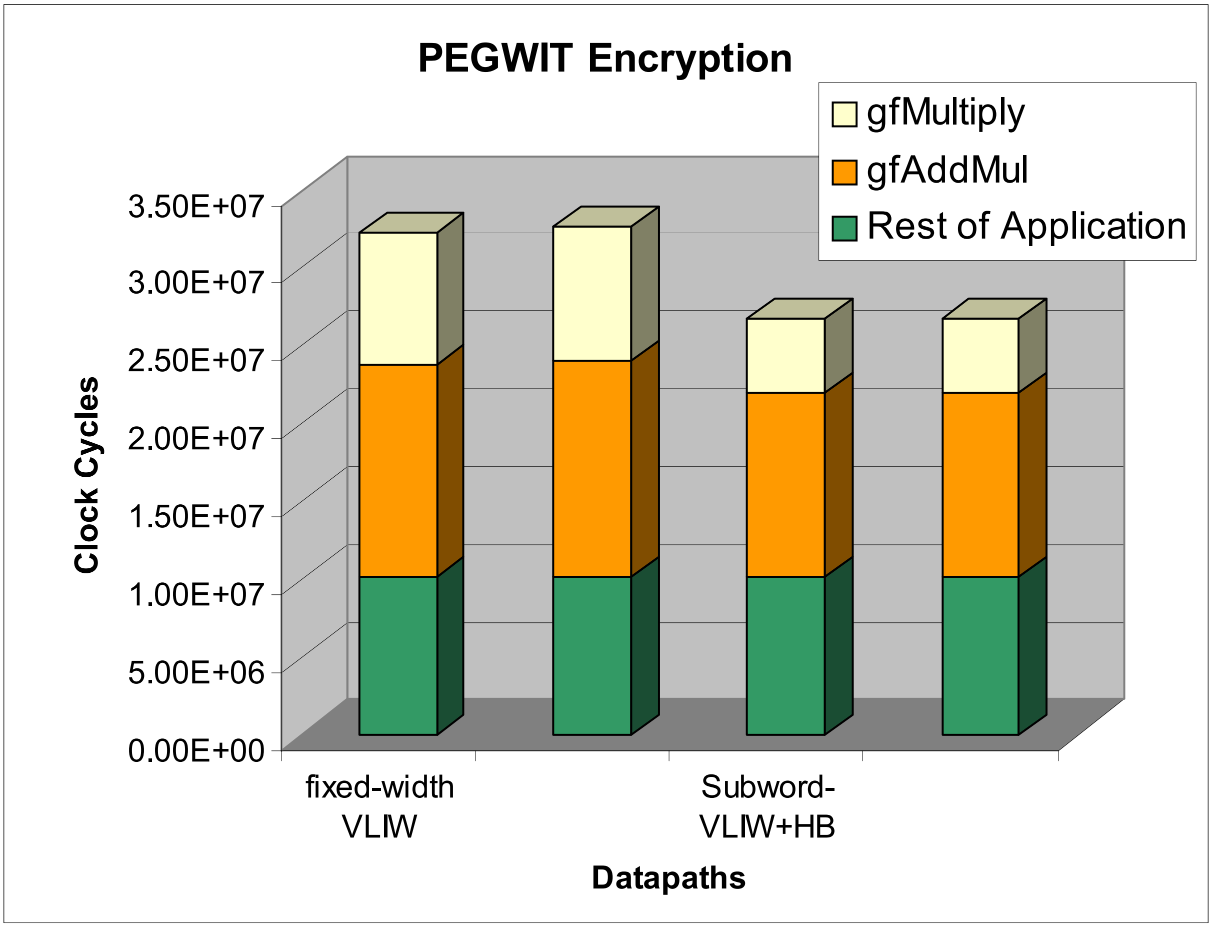


Figure 88 The relative performance speedups for the PEGWIT Encryption application.

The conclusion of the experiments using the PEGWIT decryption application, are similar to that of the encryption application since the code kernels are the same functions. The functions include complex control structures with limited opportunities for parallelism for the compiler to extract and exploit. We achieved our lowest speedup result of 10% for this application, the results of all the experiments performed are summarized in Figure 89.

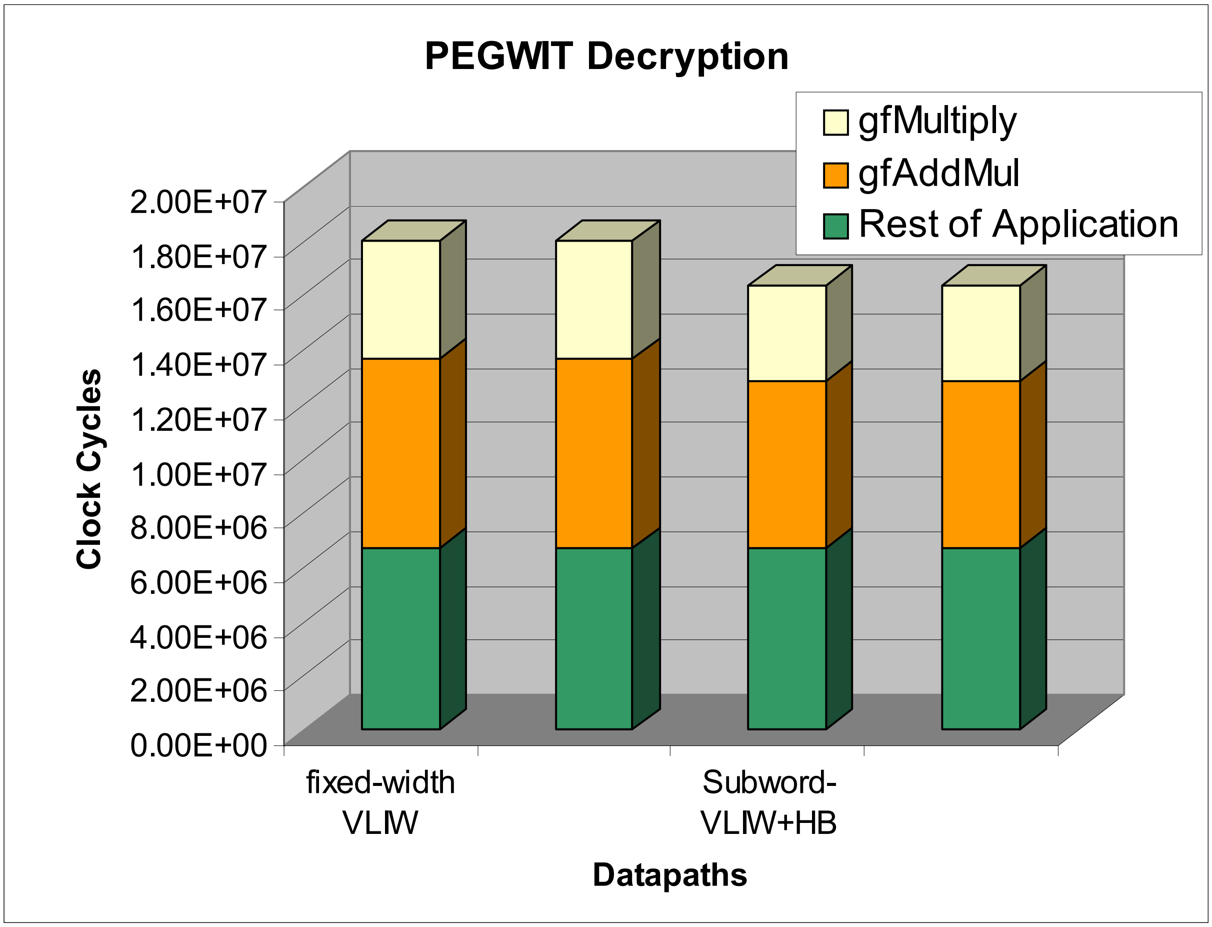


Figure 89 The relative performance speedups for the PEGWIT Decryption application.

The results of all the experiments performed for the ADPCM encoder are shown in Figure 90. This kernel contains some complex control-flow and data dependence within the loop body, however, aggressive compiler techniques and code transformations are able to extract some parallelism and exploit the flexibility underlying subword MIMD datapath. This combination of aggressive compiler transformations and flexible datapath is capable of achieving significant speedups for the adpcm_coder kernel.

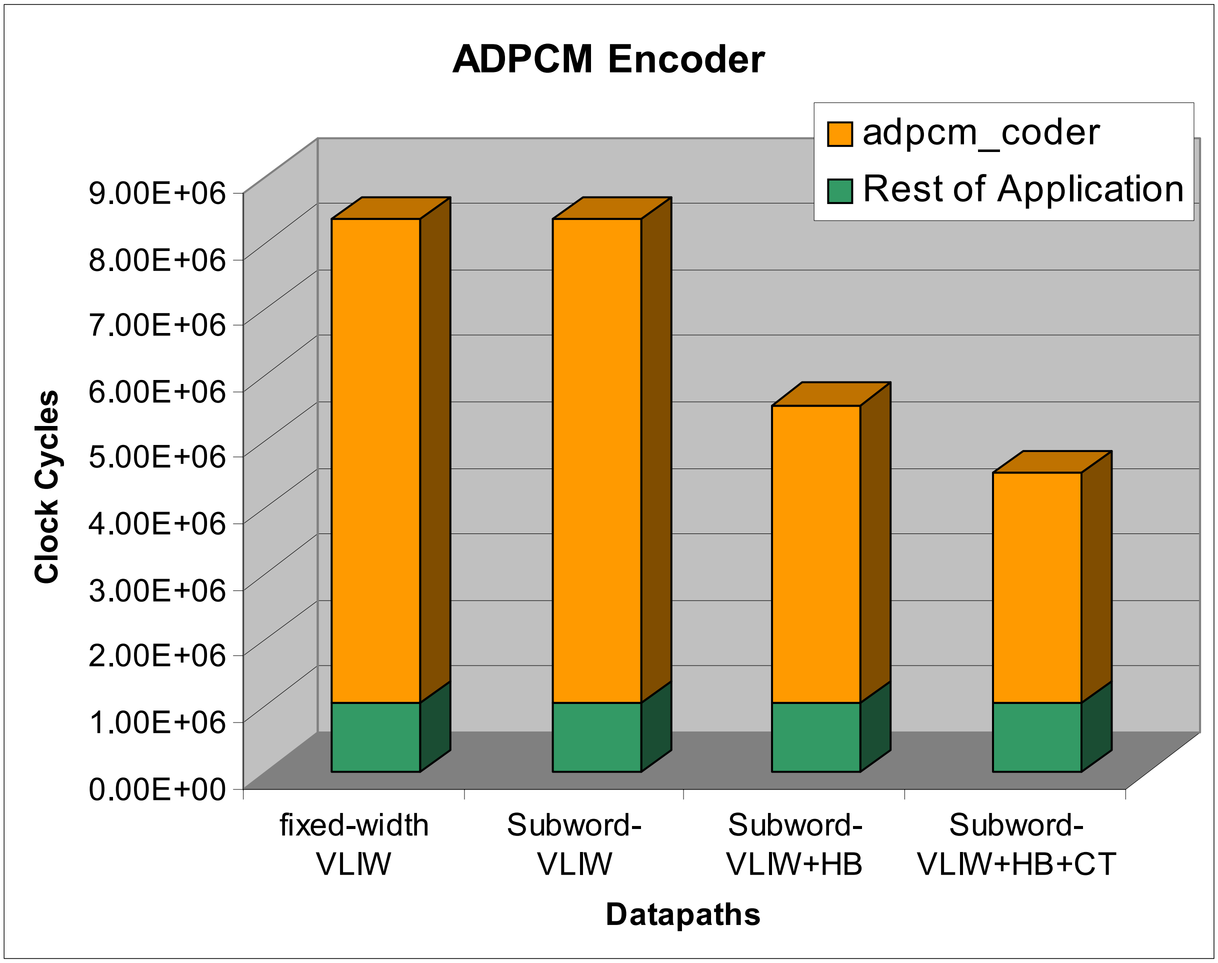


Figure 90 The relative performance speedups for the ADPCM Encoder application.

For the ADPCM decoder, the conclusion of the results is similar to that of the encoder. The performance for all the experiments done for this application are summarized in Figure 91.

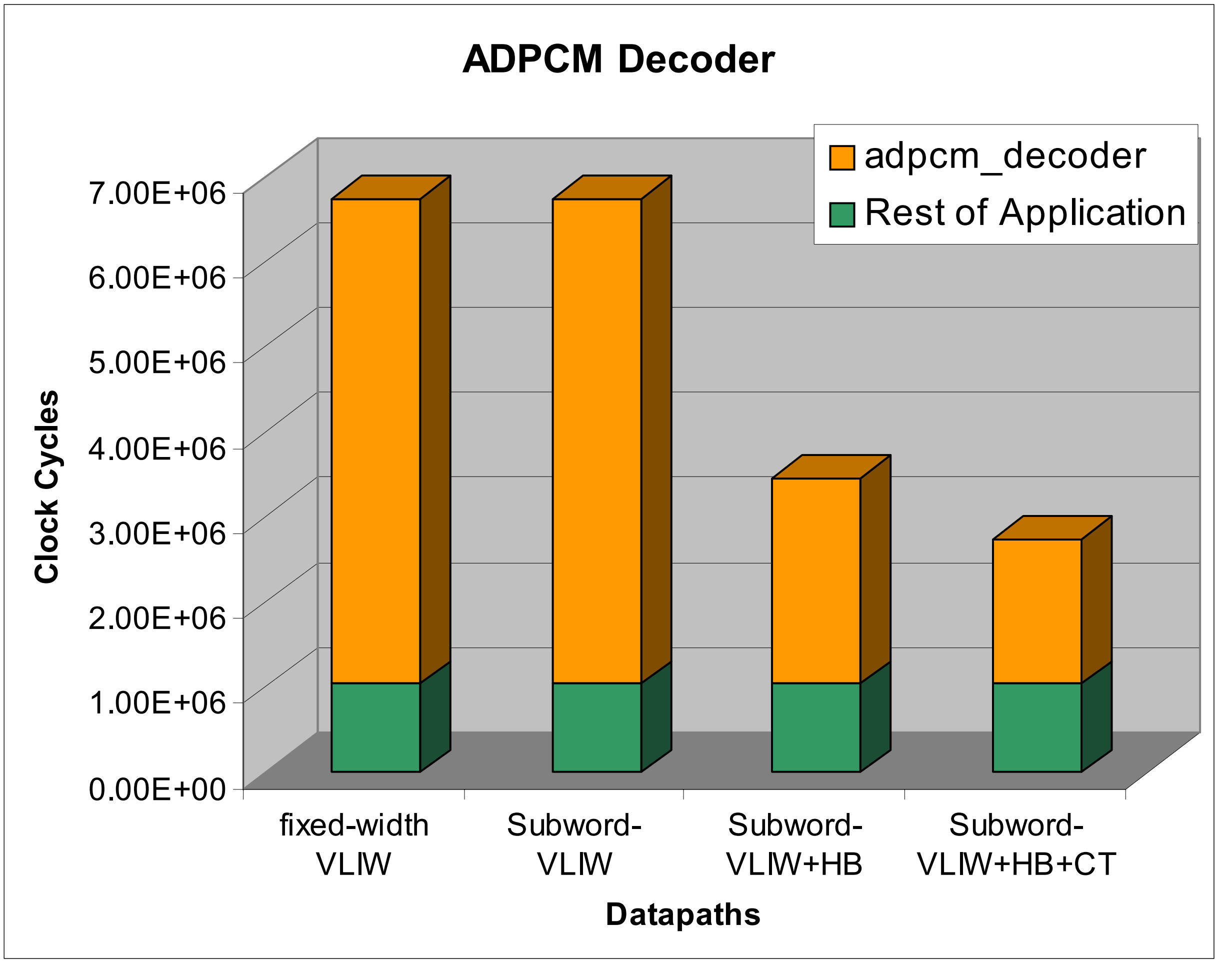


Figure 91 The performance speedups for the ADPCM Decoder application.

The results of all the experiments performed on the MPEG2 encoder are summarized in Figure 92. Similar to other applications, this kernel also requires the compiler to perform complex code transformations using speculative execution in order to achieve good speedups.

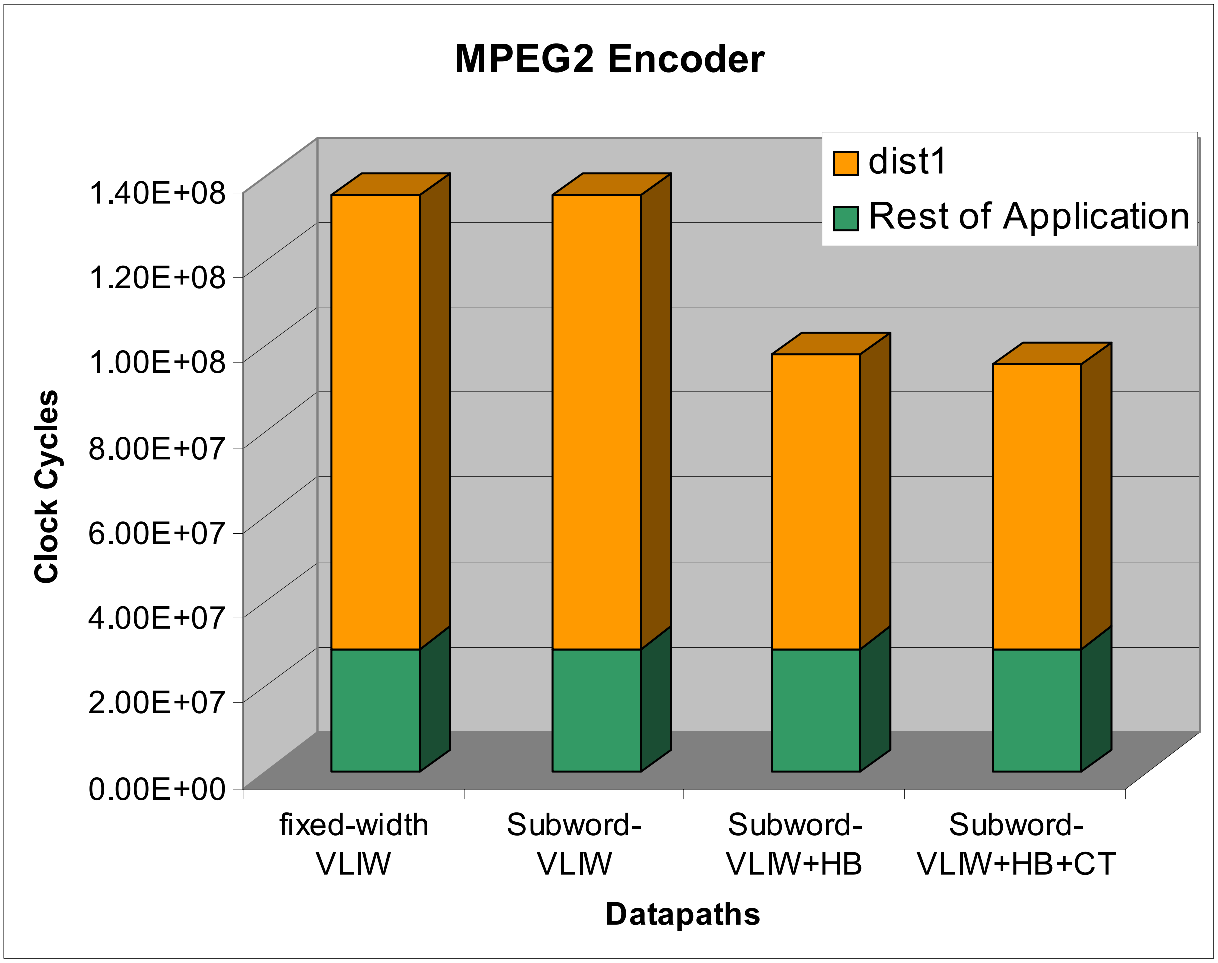


Figure 92 The performance speedups for the MPEG2 Encoder application.

The results for the MPEG2 decoder are summarized in Figure 93. This kernel does not require complex compiler transformations to take advantage of the parallel subword resources in the subword VLIW datapath..

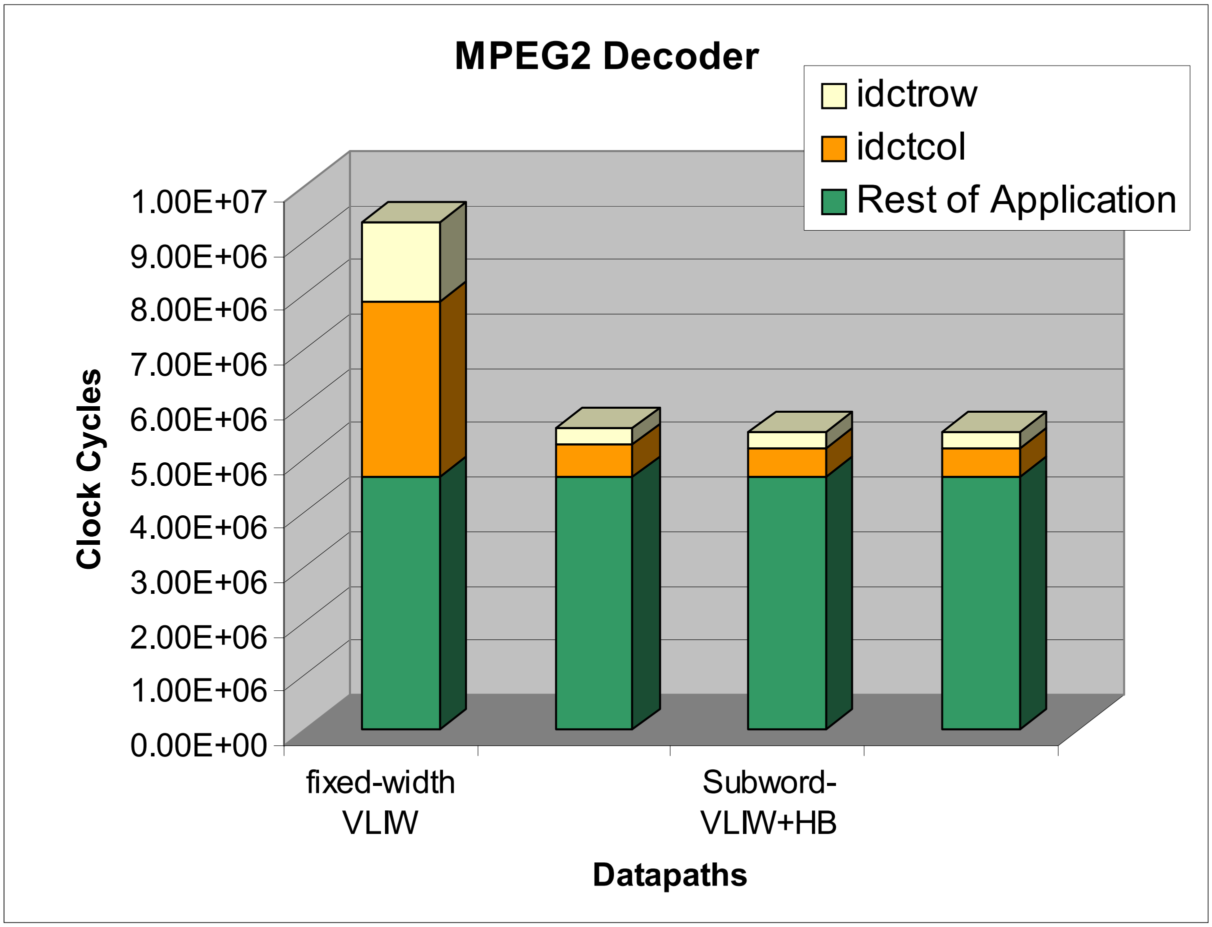


Figure 93 The performance speedups for the MPEG2 Decoder application.

Finally, the results for the four experiments performed using the MPEG4 Decoder application are shown in 94. Simple code transformations achieve significant speedups, however the anomalous result is that if-conversion within hyperblock formation is not yielding any improvements in performance. This is due to the fact that the *abs* function calls are system library calls and hence the high level code is not available for the compiler to perform code transforms in order to remove the restrictions of the control-flow within the body of the loop. Once we overcome this restriction by exchanging the *abs* calls with *if-statements* within the loop body, the hyperblock code transformations yielded our best result, a speedup by a factor of 8.12.

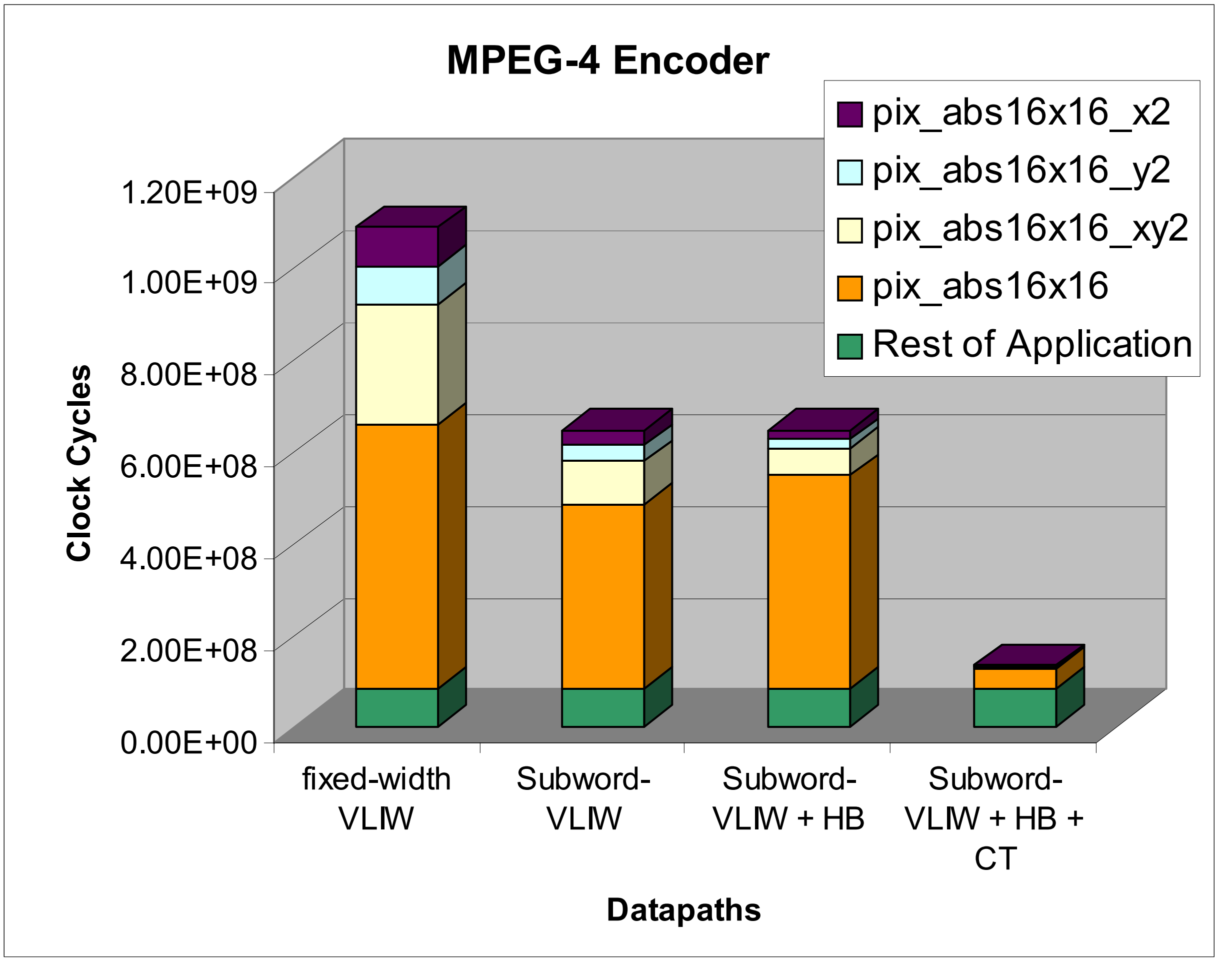


Figure 94 The performance speedups for the MPEG4 Decoder application.

6.3 SUMMARY OF EXPERIMENTAL RESULTS

For all the kernels examined, the subword VLIW datapath alone is not enough to achieve significant speedups over fixed-width datapath. Aggressive compiler techniques are required to extract the parallelism in these applications and then schedule them effectively onto the datapath. However, the MPEG-2 decoder (IDCT) and the GSM encoder did not need any advanced compiler techniques to achieve a high speedup.

In some situations, such as the GSM decoder and the MPEG-4 decoder, the simple code transformations led to a very significant increase in performance.

Table 9 Performance Speedup Summary of the Kernels and Applications

Application	Kernels	Kernel Speedups (by a factor of)	Application Speedups (by a factor of)
GSM Encoder	Calculation of LTP	42.13	4.73
GSM Decoder	Short Term Synthesis	4.72	2.23
PEGWIT Encryption	gfAddMul gfMultiply	1.33 (combined kernels)	1.2
PEGWIT Encryption	gfAddMul gfMultiply	1.17 (combined kernels)	1.1
ADPCM Encoder	adpcm_coder	2.1	1.85
ADPCM Decoder	adpcm_decoder	3.35	2.46
MPEG-2 Encoder	dist1	1.59	1.41
MPEG-2 Decoder	idctcol idctrow	5.79 (combined kernels)	1.7
MPEG-4 Encoder	pix_abs16x16 pix_abs16x16_xy2 pix_abs16x16_x2 pix_abs16x16_y2	19.65 (combined kernels)	8.12

7.0 SUMMARY AND CONCLUSIONS

In this chapter we present a summary of work performed in this dissertation. Following the summary, we present our conclusions based on the studies performed in the preceding chapters.

7.1 SUMMARY

In this dissertation we have presented the need for an improvement over current solutions at targeting the effective execution of emerging multimedia applications in the domain general purpose domain. In chapter 2.0, we discussed the benefits and limitations of several approaches, particularly, the SIMD instruction set extensions in general purpose processors. These solutions, although cost effective, introduce significant implementation restrictions which have limited the wide spread use of these SIMD instructions.

In Chapter 3.0, we performed a rigorous analysis of multimedia kernels and evaluated their characteristics, code structures and data-types. These findings are summarized in Table 5.

In Chapter 4.0, we presented a classical VLIW architecture and discussed the manner in which to extend it to achieve a subword VLIW datapath.

In Chapter 5.0, we presented our compilation and simulation infrastructure that was used to perform kernel analysis, evaluation of the proposed architecture as well as compilation techniques and code transformations.

The experimental evaluation and analysis was presented in Chapter 6.0. We evaluated the proposed architecture on nine multimedia applications. Further, we assessed the impact of predicated execution on performance. Finally, we evaluated the performance benefits of simple code transformations which attempt to highlight more parallelism within the code kernels. The results, summarized in Table 9 and shown in Figure 95, present significant speedups for the majority of the multimedia applications tested.

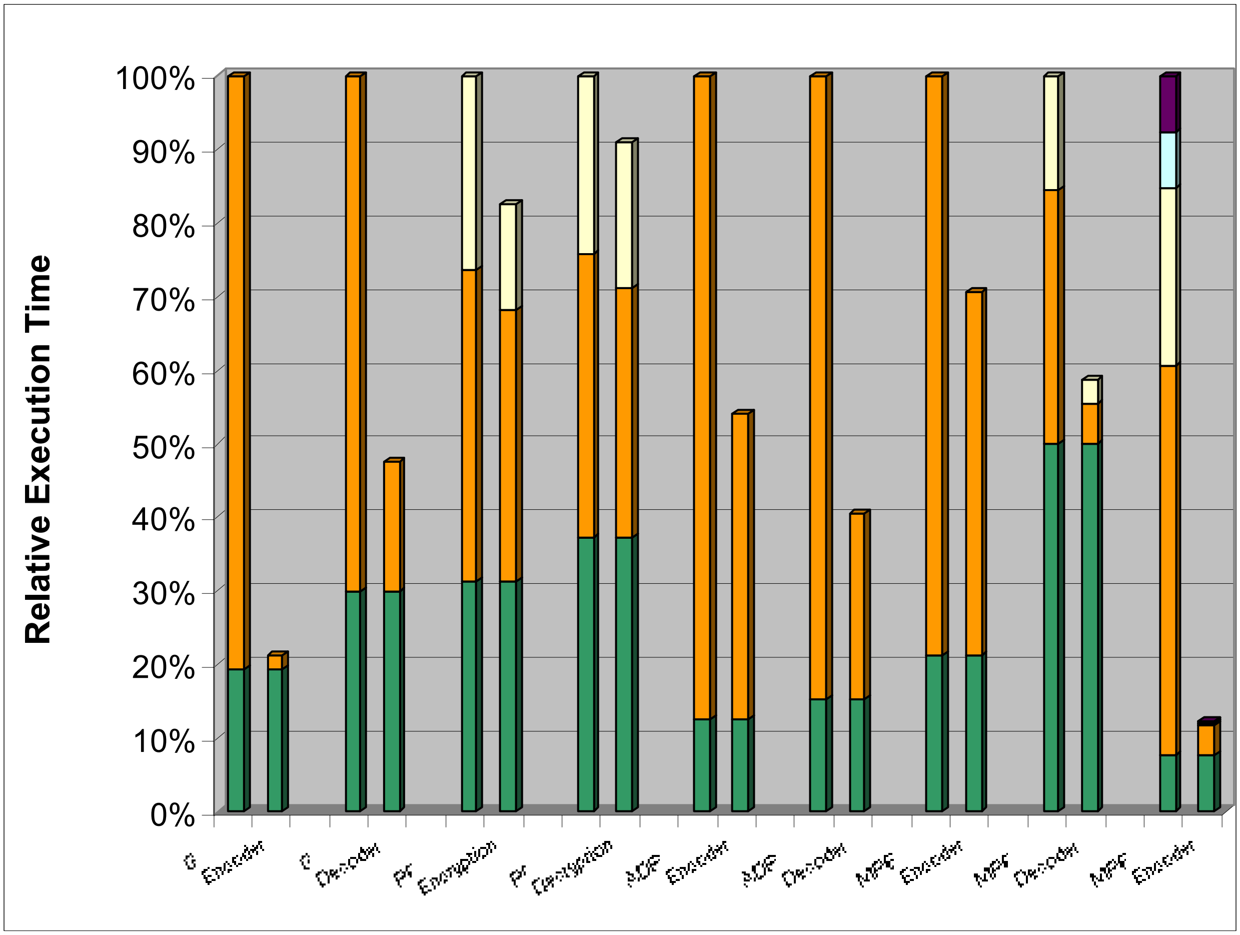


Figure 95 The performance speedups for all the applications examined.

The arithmetic mean of the speedups achieved over all nine applications is 2.75. The more conservative geometric mean of the performance speedups across all nine applications is 2.22.

7.2 CONCLUSIONS

A subword VLIW MIMD datapath is a viable solution as an extension of the VLIW programming paradigm when targeting general purpose multimedia applications. Significant speedups were achieved due to the flexibility in the datapath which allowed the compiler to better schedule the optimized kernels. However, a flexible parallel architecture is not enough to achieve good performance, a powerful compiler which is capable of performing code transformations to tease apart the control-flow within the loop bodies is also essential.

Furthermore, we conclude that predicated execution is a very important tool in order to allow the compiler to extract more parallelism from multimedia kernels. Hyperblock formation removed many of the control-flow in the loop bodies that was hindering performance. Therefore, if-conversion enabled by hardware predication is a powerful technique to reveal parallelism in media applications by performing speculative execution and discarding incorrect results.

Simple code transformations, such as loop unrolling of critical loops, including temporaries to ease data dependence, substitution of simple functions and code pipelining have lead to increased parallelism and a boost in performance.

We have shown that subword multimedia kernels can be statically analyzed by a compiler yielding a large amount of extracted parallelism. However, in order to exploit this extracted parallelism, flexible hardware resources must be available to achieve significant performance gains. A low overhead MIMD VLIW datapath that supports subword operations is an effective solution.

The number of clock cycles spent executing multimedia applications on general purpose processors continues to grow. Dynamic techniques to extract instruction level parallelism (ILP) are not needed for these applications since enough parallelism can be extracted statically. The overhead in die area, power dissipation and clock cycles spent performing these dynamic analyses will prove unwarranted once the percentage of time spent on media applications grows beyond a sensitive threshold.

Using SIMD functional units to target multimedia applications in general purpose processors is not a favorable solution. These designs require low die-area cost and low power dissipation, however, they suffer from inefficiency due to a restrictive programming paradigm, high data reorganization overhead and lack of automatic techniques to target these units. Further, we conjecture that if the same experimental procedure was performed targeting a subword-SIMD datapath, significant speedups would have been had only for the GSM Encoder algorithm.

A less restrictive MIMD programming model is more suited for media applications and it capitalizes on the well developed set of code transformation and optimization to target MIMD systems. A subword capable MIMD datapath enables effective and efficient execution of multimedia applications.

8.0 FUTURE WORK

There are several interesting studies that can be pursued. First, a cost/performance analysis that evaluates this paradigm and these applications using performance cost constraints.

Another study involves evaluating the impact on performance from using multiple multiported register files. Large multiported register files require a lot of die area and increase the access delay.

A direct performance comparison with SIMD architectures is important but it is not straight forward since there are many parameters that could offset performance. One such parameter is the compiler tool set used to optimize and then target each datapath type. Second is the simulation environment that allows precise performance comparison.

After evaluating the characteristics of media applications, we observed that media centric operations could highly impact performance. An architectural enhancement would utilize new media centric instructions that collapse several conventional instructions into one and reduce the number of clock cycles required to perform the complex operation. Examples are instructions such as sum of absolute differences (SAD) and tree adder to satisfy accumulator arithmetic. This task requires extending the compiler to understand the new set of special complex operations in order to utilize them and include them in the scheduling analysis. Adding new operations to the compiler requires extending all six intermediate representations of the compiler to support the operations, which is a substantial effort.

The design, evaluation and implementation of a high bandwidth memory interface is worthwhile. An interface that satisfies streaming accesses at minimum address calculation overhead.

Finally, extending the profiling idea into a feedback analysis, where several iterations of compilation and execution are performed to better understand program behavior and hence develop better and shorter schedules.

BIBLIOGRAPHY

1. Kozyrakis, C.E., Patterson, D.A., “A new direction for computer architecture research,” *IEEE Computer*, pp. 24-32, November 1998.

2. Diefendorff, K., Dubey, P.K., “How multimedia workloads will change processor design, “*IEEE Computer*, pp. 43 - 45, September 1997.

3. R.B., Lee, “Multimedia Extensions for General-Purpose Processors,” *Proceedings of the IEEE SIPS*, November 1997.

4. R., Leupers, “Code Selection for Media Processors with SIMD Instructions,” *In the Proceedings of the Conference on Design, Automation and Test in Europe*, March 2000.

5. Conte, T.M., Dubey, P.K., Jennings, M.D., Lee, R.B., Peleg, A., Rathnam, S., Schlansker, M., Song, P., Wolfe, A., “Challenges to combining general-purpose and multimedia processors,” *IEEE Computer*, pp. 33-37, December 1997.

6. Tremblay, M., Grohoski, G., Burgess, B., Killian, E., Colwell, R., Rubinfeld, P.I., “Challenges and Trends in Processor Design,” *IEEE Computer*, pp. 39-50, January 1998.

7. Deepu Talla and Lizy John, “Execution Characteristics of Multimedia Applications on a Pentium II Processor,” *In Proceedings of the IEEE International Performance, Computing and Communications Conference*, pages 516-524, 2000.

8. Oberman, S., Favor, G., and Weber, F., “AMD 3DNow! Technology: Architecture and Implementations,” *IEEE Micro*, pp. 37-48, March-April 1999.

9. Gwennap, L., “AltiVec Vectorizes PowerPC,” Microprocessor Report, pp. 1,6-9, May 1998.

10. Peleg, A., Weiser, U., “MMX Technology Extensions to the Intel Architecture,” *IEEE Micro*, pp. 42-50, August 1996.

11. Thakkar, S., Huff, T., “Internet Streaming SIMD Extensions,” *IEEE Computer*, November 1999.

12. Lee, R.B., A.M. Fiskiran and A. Bubshait, “Multimedia Instructions in IA-64,” *Proceedings of ICME 2001 IEEE International Conference on Multimedia and Expo*, August 2001.

13. Carson, D.A., “Multimedia Extensions for a 550MHz RISC Microprocessor,” *IEEE Journal of Solid-State Circuits*, 1997.

14. Tremblay, M., O’Connor, M., Narayan, V., He, L., “VIS Speeds New Media Processing,” *IEEE Micro*, pp. 10-20, August 1996.

15. Diefendorff, K., “Sony’s Emotionally Charged Chip,” *Microprocessor Report*, pp. 1,6-11, April 1999.

16. Oka, M. and Suzukoi, M, “Designing and Programming The Emotion Engine,” IEEE Micro, pp. 20-28, November-December 1999.

17. Hansen, C., “MicroUnity’s MediaProcessor Architecture,” *IEEE Micro*, pp. 34-51, August 1996.

18. Lindholm, E., Kilgard, M.J., Moreton, H., “A User-Programmable Vertex Engine,” *In Proceedings of ACM SIGGRAPH,* 12-17 August 2001.

19. Glaskowsky, P.N., “Transmeta Tips the TM6000,” *Microprocessor Report*, October 2001.

20. F. Jesus Sanchez, Antonio Gonzalez, and Mateo Valero, “Software management of selective and dual data caches,” *IEEE Technical Committee on Computer Architecture Newsletter*, March 1997.

21. M. Tomasko, S. Hadjiyiannis, and W. Najjar, “Experimental Evaluation of Array Caches,” *In IEEE Computer Society Technical Committee on Computer Architecture: Special Issue on Distributed Shared Memory and Related Issues*, March 1997.

22. G. Faanes, “A CMOS Vector Processor with a Custom Streaming Cache,” *In Hot Chips 10*, August 1998.

23. P. Ranganathan, S. Adve, and N. P. Jouppi, “Reconfigurable caches and their application to media processing,” *In Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

24. D. Chiou, P. Jain, L. Rudolph and S. Devadas, "Application-Specific Memory Management for Embedded Systems Using Software-Controlled Caches," *Proceedings of the 37th Design Automation Conference (DAC'00)*, June 2000.

25. P. Y. T. Hsu, “Design of the R8000 microprocessor. IEEE Micro,” April 1994.

26. Intel Corporation, “The Intel Itanium 2 Processor, Hardware Developer’s Manual,” *Document Number 251109-001*, July 2002.

27. R. Cucchiara, M. Piccardi, and A. Prati, "Exploiting cache in multimedia," *In the Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, June 1999.

28. Wawrzynek J., Asanovic K., Kingsbury B., Beck J., Johnson D., and Morgan N., "SPERT-II: A Vector Microprocessor System," *IEEE Computer*, pp. 79-86, March 1996.

29. Asanovic K., Kingsbury B., Irissou B., Beck J., and Wawrzynek J., "T0: A Single-Chip Vector Microprocessor with Reconfigurable Pipelines," *In Proceedings 22nd European Solid-State Circuits Conference*, September, 1996.

30. Fisher, J.A., “A very long instruction word architecture and the ELI-512,” *Proceedings of the 10th Annual International Symposium on Computer Architecture (ISCA)* , Stockholm, 1983.

31. Hwang, K., *Advanced Computer Architecture,* McGraw-Hill, 1993.

32. W G Rudd , Duncan A Buell , Donald M Chiarulli, “A High Performance Factoring Machine”, *Proceedings of the 11th Annual International Symposium on Computer Architecture* (ISCA), p.297-300, January 1984.

33. Conte, G., Tommisani, S., and Zanichelli, F., “The Long and Winding Road to High-Performance Image Processing with MMX/SSE,“ *In Proceedings of the Fifth IEEE International Workshop on Computer Architectures for Machine Perception (CAMP'00)*, pp. 302-310, Padova, Italy, September, 2000.

34. Talla, D., and Kurian John, L., “Execution Characteristics of Multimedia Applications on a Pentium II Processor,” *In Proceedings of the IEEE International Performance, Computing and Communications Conference*, pp. 516-524, 2000.

35. Peleg, A., and Weiser, U., “MMX Technology Extension to the Intel Architecture,” *IEEE Micro*, Vol.16, No. 4, pp. 42-50, August 1996.

36. Gwennap, L., “Intel’s MMX Speeds Multimedia,” *Microprocessor Report*, Vol. 10, No. 3, March 1996.

37. Lempel, O., Peleg, A., and Weiser, U., “Intel’s MMX Technology - A New Instruction Set Extension,” *In Proceedings IEEE COMPCON 97*, San Jose, CA, USA, February, 1997.

38. Diefendorff, K., “Katmai Enhances MMX,” *Microprocessor Report*, Vol. 12, No. 13, October 1998.

39. D. Talla, L. John, and D. Burger, “Bottlenecks in multimedia processing with SIMD style extensions and architectural enhancements,” *IEEE Transactions on Computers*, pages 35--46, August 2003.

40. Thakkar, S., and Huff, T., “Internet Streaming SIMD Extensions,” *IEEE Computer*, Vol. 32, No. 12, December 1999.

41. Diefendorff, K., “Pentium III = Pentium II + SSE,” *Microprocessor Report*, Vol. 13, No. 3, March 1999.

42. Bik, A. Girkar, M., Grey, P., and Tian, X., “Effecient Exploitation of Parallelism on Pentium III and Pentium 4 Processor-Based Systems,” *Intel Technology Journal*, Q1, 2001.

43. Hinton, G., Sager, D., Upton, M., Boggs, D., Carmean, D., Kyker, A., and Roussel, P., “The Microarchitecture of the Pentium 4 Processor,” *Intel Technology Journal*, Q1, 2001.

44. Intel Corporation, “IA-32 Intel Architecture Software Developer’s Manual - Volume 1: Basic Architecture,” *Order Number 245470*, 2001.

45. Intel Corporation, “IA-32 Intel Architecture Software Developer’s Manual - Volume 2: Instruction Set Reference,” *Order Number 245471*, 2001.

46. van Eijndhoven, J.T.J., F. W. Sijstermans, K. A. Vissers, E.- J. D. Pol, M. J. A. Tromp, P. Struik, R. H. J. Bloks, P. van der Wolf, A. D. Pimentel, and H. P. E. Vranken, "TriMedia CPU64 Architecture," in *International Conference on Computer Design*, Austin, Texas, 1999, pp. 586--592.

47. Hitachi, Ltd, Equator Technologies, Inc. “MAP-CA DSP Datasheet,” *Document Number HWR.CA.DS.2001.06.20*, June 2001.

48. Hitachi, Ltd, Equator Technologies, Inc. “BSP-15 Processor Datasheet,” *Document Number HWR.BSP15.DS.REV.H*, September 2002.

49. Hayakawa, F.; Okano, H.; Suga, A., “An 8-way VLIW embedded multimedia processor with advanced cache mechanism,“ *Proceedings of the 2002 IEEE Asia-Pacific Conference ASIC*, August 2002.

50. B. Khailany, W. Dally, U. Kapasi, P. Mattson, J. Namkoong, J. Owens, B. Towles, A. Chang, and S. Rixner, “Imagine: Media Processing With Streams,” *IEEE Micro*, 21(2):35--47, 2001.

51. Ujval J. Kapasi, William J. Dally, Brucek Khailany, John D. Owens, and Scott Rixner, “The Imagine Stream Processor,” *In Proceedings of the IEEE International Conference on Computer Design*, September 2002.

52. Kozyrakis, C.; Patterson, D, “Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks,“ *Proceedings. 35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)*, November 2002.

53. Thies, B., Karczmarek, M., and Amarasinghe, M., “StreaMIT: A Language for Streaming Applications, “ *MIT/LCS Technical Memo MIT-LCS-TM-620*, August 2001.

54. Hennessy, J.L., and Patterson, D.A., *Computer Architecture A Quantitative Approach*, Morgan Kaufmann, 1996.

55. S., Vagnier, H., Essafi, A., Merigot, “Impact of Configurable Processor in Parallel Architecture for Document Management,“ *The 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)*, June 26 - 29, 2000

56. S. Dutta, K.J., O’Connor, W., Wolfe, A., Wolfe, “A Design Study of a .25um Video Signal Processor,” *IEEE Transactions on circuits and sustems for video technoogy*, August 1998.

57. N. Ohkubo, et. al., “A 4.4ns CMOS 54x54-b Multiplier Using Pass-transistor Multiplexer,” *In Proc. of 1994 IEEE Custom Integrated Circuits Conf*, 1994.

58. M. Suzuki, et al., “A 1.5ns, 32b CMOS ALU in Double Pass Transistor Logic,” *In Proc. of International Solid State Circuits Conf.*, 1993.

59. The Trimaran Consortium, http://www.trimaran.org/.

60. August, D.I., Connors, D.A., Mahlke, S.A., Sias, J.W., Crozier, K.M., Cheng, B.C., Eaton, P.R., Olaniran, Q.B., and Hwu, W.W., “Integrated Predicated and Speculative Execution in the IMPACT EPIC Archtecture,” *Proceedings of the 25th International Symposium on Computer Architecture*, July 1998.

61. Rau, B.R., Kathail, V., Aditya, S., “Machine-Description Driven Compilers for EPIC Processors,” *HPL Technical Report, HPL-98-40, Hewlett Packard Laboratories*, 1998.

62. Mahlke, S., Hank, R., McCormick, J., August, D., Hwu, W., "A Comparison of Full and Partial Predicated Execution Support for ILP Processors," *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 138-149, June 1995.

63. Richard E. Hank, Scott A. Mahlke, Roger A. Bringmann, John C. Gyllenhaal, and Wen-mei W. Hwu, "Superblock Formation Using Static Program Analysis," *Proceedings of the 26th Annual ACM/IEEE Int'l Symposium on Microarchitecture*, pp. 247-256, December 1993.

64. Mahlke, S., Lin, D., Chen, W., Hank. R., Bringmann, R., "Effective Compiler Support for Predicated Execution Using the Hyperblock," *Proceedings of the 25th International Symposium on Microarchitecture*, pp. 45-54, December 1992.

65. Rau, B.R., "Iterative Modulo Scheduling: An algorithm for software pipelined loops," *Proceedings of the 27th international Symposium on Microarchitecture*, pp. 63-74, December 1994.

66. Rau, B.R. "Iterative Modulo Scheduling," *International Journal of Parallel Processing*, pp. 3-64, February 1996.

67. Kathail, V., Schlansker, M., and Rau, B.R., "HPL-PD Architecture Specification Version 1.1," *Technical Report HPL-93-80 (R.1). Hewlett-Packard Laboratories*, February 1994 (revised July 1999).

68. Gyllenhaal, J.C., Hwu, W.-m.W., and Rau, B.R., "HMDES Version 2.0 Specification," *Technical Report IMPACT-96-3. University of Illinois at Urbana-Champaign*, 1996.

69. S. Aditya, V. Kathail and B. R. Rau., "Elcor's Machine Description System: Version 3.0," *HPL Technical Report HPL-98-128*, Hewlett-Packard Laboratories, July 1998.

70. Chunho Lee; Potkonjak, M.; Mangione-Smith, W.H., "MediaBench: a tool for evaluating and synthesizing multimedia and communications systems," *Proceed-*

ings of Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture, pp. 330-335, 1997.

REFERENCES NOT CITED

Rastislav Bodik, “Personal Communication”.

Seth Goldstein, “Personal Communication”.

Randolph Harr, “Personal Communication”.