Program Generation for Intel AES New Instructions

Thesis submitted for the degree of Doctor in Philosophy

2011

Raymond Keith Scott Manley

Declaration

I declare that this thesis has not been submitted as an exercise for a degree at this or any other university and it is entirely my own work.

I agree to deposit this thesis in the University's open access institutional repository or allow the Library to do so on my behalf, subject to Irish Copyright Legislation and Trinity College Library conditions of use and acknowledgement.

Raymond Keith Scott Manley

Acknowledgements

The work presented in this dissertation could have not been completed without the knowledge, experience, and support of my supervisor, Dr. David Gregg. He ensured several years ago that I got the opportunity to study in Ireland. I am as equally thankful to him for that as I am for his influence on my academic pursuits.

I would also like to thank Jason McCandless, Kevin Williams, Paul Biggar, Nicholas Nash, and Bobb Crosbie. As far as research groups go, the members of our group were a social and enjoyable bunch to talk to, both inside and outside the office.

I could not have done any of this work without funding. I would like to thank IRCSET (The Irish Research Council for Science, Engineering and Technology) and Intel Ireland for providing me this unique opportunity.

I would like to thank all my friends on both sides of the pond and specifically to my proofreaders: Aoife Carolan, Marianne Cassidy, and Sophie Geary. Finally, I would like to thank my parents. I know they would have liked to see me more over this period, but they have been incredibly understanding and supporting of my endeavours throughout the years. This work is dedicated to them.

Abstract

High-performance primitive libraries are used to replace parts of sub-optimal code with optimized implementations. These libraries often come in the form of highly-optimized assembly routines, which raises several issues. Small changes to assembly routines can require significant rewrites. New versions of microarchitectures will often require changes in the assembly to keep code both efficient and functional. Maintaining multiple versions of the same basic piece of assembly code is a costly software engineering problem. One approach to solving this problem is using a program generator.

AES-NI is an instruction-set extension on Intel processors that implement a full round of AES encryption in a single instruction. Existing libraries use hand-tuned assembly language to overlap the execution of multiple AES instructions to extract maximum performance. In this dissertation, we argue that using a program generator is suitable substitute for writing highly-optimized assembly routines that use AES-NI. We present a program generation system that seamlessly integrates high-level algorithmic choices with scheduling strategies that exploit instruction-level parallelism.

This program generation system returns AES implementations that achieve near optimal performance. We also show that the generator is dynamic enough to take exploratory approaches when optimizing code. As a result, this dissertation also contributes two novel encryption modifications. For CTR mode, we present a "mixed-mode" operation that combines traditional lookup table optimizations with AES-NI instructions. In cyclic modes, such as CBC, we show how manipulating the `xor` instructions can shorten the chain of dependent operations. These optimized implementations are found using an adapted simulated annealing algorithm. We show these implementations can achieve similar or superior cycle per byte times compared to the high-performance library versions provided by Intel. The end result is a program generation technique that could potentially be adapted to optimize other algorithms that rely on instruction-set extensions.

Contents

Abstract **v**

List of Tables **xi**

List of Figures **xiii**

List of Code Listings **xv**

1 Motivation **1**

1.1 Thesis 3
1.2 Contributions 3
1.3 Relevant Academic Publications 5
1.4 Dissertation Outline 5

2 Background **7**

2.1 Architectural Features 7
2.1.1 Instruction Pipelining 7
2.1.2 Superscalar 9
2.1.3 Simultaneous Multithreading 10
2.1.4 Vector/SIMD 11
2.1.5 Autovectorization 12
2.2 Software ILP Optimizations 13
2.2.1 Instruction Scheduling 13
2.2.2 Loop Unrolling 13
2.2.3 Software Pipelining 14
2.2.4 Modulo Scheduling 15
2.3 Program Generation 17
2.3.1 Domain Specific Code Generators 18
2.3.2 Traversing the Search Space 19

2.3.3 Simulated Annealing 21
2.4 Streaming Languages 23

3 Vector Code Generation for GPPs 27
3.1 Introduction 28
3.2 Brook 29
3.3 Modifications to Brook 30
3.3.1 Modifying the Code Generator 31
3.3.2 Modifying the Runtime 31
3.4 Generating Vector Code 32
3.4.1 General operations 32
3.4.2 Arrays 33
3.4.3 Conditional Assignments 34
3.4.4 For loops inside kernels 35
3.4.5 Reduce Kernels 35
3.4.6 Selective Vectorization 36
3.5 Results 37
3.5.1 Analyzing Performance 38
3.6 Related Work 42
3.7 Conclusion 43

4 AES Encryption in Software and Hardware 47
4.1 AES 48
4.2 Block Cipher Modes 51
4.2.1 Parallel Modes 51
4.2.2 Cyclic Modes 52
4.2.3 Combined Encryption and Authentication 56
4.3 AES Software Acceleration 57
4.4 AES Hardware Acceleration 61
4.5 Intel AES-NI 64
4.6 Conclusion 66

5 CTR and CBC Program Generation 67
5.1 Introduction 67
5.2 CTR and CBC Code Generation 69
5.2.1 CTR Optimizations 70
5.2.2 Optimizations for Both Modes 73

5.2.3 CBC Optimizations . . . 73
5.2.4 Simulated Annealing . . . 75
5.3 Results . . . 76
5.3.1 Cycles per Byte . . . 78
5.3.2 Selective-Exhaustive Searches . . . 78
5.3.3 Guided Search using Simulated Annealing . . . 86
5.4 Conclusion . . . 89

6 Generalized AES Program Generation **91**
6.1 Introduction . . . 91
6.2 GEN1 vs. AES-GEN . . . 93
6.2.1 Function Stitching . . . 94
6.2.2 Cycles per Round . . . 94
6.3 The AES-GEN Program Generator . . . 95
6.3.1 Algorithmic Choices with Cheetah . . . 96
6.3.2 The ILP Optimizer . . . 98
6.3.3 Code Tuning . . . 101
6.4 Generator Flexibility with Parallel Algorithms . . . 102
6.4.1 Counter (CTR) . . . 102
6.4.2 Electronic Codebook (ECB) . . . 107
6.4.3 Performance Observations for CTR and ECB Modes . . . 109
6.5 Algorithmic Variations with Cyclic Algorithms . . . 110
6.5.1 Cipher-Block Chaining (CBC) . . . 110
6.5.2 PCBC, CFB, and OFB . . . 113
6.5.3 Applicability of XOR Optimizations to Other AES Modes . . . 115
6.6 Combining Algorithms via Function Stitching . . . 116
6.6.1 Galois/Counter Mode (GCM) . . . 116
6.6.2 Counter with CBC-MAC (CCM) . . . 118
6.7 Experimental Results . . . 119
6.7.1 Generated Code Performance . . . 120
6.7.2 AES-GEN vs Hand-tuned Assembly . . . 124
6.7.3 Why Not Optimize with a Standard Compiler? . . . 128
6.8 Related Work . . . 131
6.9 Conclusion . . . 132

7 Final Thoughts **133**
7.1 Security 133
7.2 Future Work 134
7.2.1 High-level Choices 135
7.2.2 ILP Optimizations 135
7.2.3 Traversing the Search Space 137
7.3 Applicability to Other Applications 138
7.3.1 Generality 138
7.3.2 Instruction Scheduling for Out-of-Order Architectures 140
7.4 Assessment of Contributions 141
7.5 Conclusion 143

Bibliography **145**

Appendix A Commonly Used Acronyms **157**

Appendix B Additional Tables **159**

Appendix C Additional Source Code Listings **161**

Appendix D "Sandy Bridge" Results **177**

List of Tables

3.1 Level 1 cache misses incurred during kernel execution 39
3.2 Level 2 cache misses incurred during kernel execution 40

4.1 AES-NI Instruction Set 65

5.1 GEN1 Simulated Annealing Results 87
5.2 GEN1 Results vs. Intel HPL 87
5.3 GEN1 vs. Baseline Results 87

6.1 AES-GEN Results, CTR mode (cycles/round) 103
6.2 AES-GEN Results, CTR mode (cycles/byte) 106
6.3 AES-GEN Results, CBC mode (cycles/byte) 112
6.4 AES-GEN Results, GCM mode (cycles/byte) 117
6.5 AES-GEN Results, CCM mode 119
6.6 AES-GEN Results for all modes 121
6.7 AES-GEN SMT Results for all modes 123
6.8 AES-GEN vs. Standard Compiler Results 129

B.1 ECB finite results for various modes 159
B.2 ECB-128 finite results with various input sizes 159
B.3 AES-GEN Results, GCM mode using SMT (cycles/byte) 159
B.4 CTR finite results, AES-GEN vs. Intel ASM 160
B.5 CBC finite results, AES-GEN vs. Intel ASM 160

D.1 AES-GEN Results for all modes (Sandy Bridge) 177
D.2 AES-GEN SMT Results for all modes (Sandy Bridge) 178

List of Figures

2.1 Generic five-stage processor pipeline . . . 8
2.2 Generic superscalar pipeline. . . . 9
2.3 SIMD processing . . . 11
2.4 Software pipelining example . . . 15
2.5 Modulo scheduling example . . . 16
2.6 Compiling a Streaming Program . . . 24

3.1 Compiling a Streaming Program in Brook . . . 30
3.2 Data swizzling . . . 31
3.3 Conditional assignments in vector mode. . . . 35
3.4 Autovectorization Experimental Results . . . 38

4.1 AES round encryption process . . . 50
4.2 Electronic Code Book (ECB) mode . . . 53
4.3 Counter (CTR) mode . . . 53
4.4 Cipher-block Chaining (CBC) mode . . . 53
4.5 Propagating Cipher-block Chaining (PCBC) mode . . . 55
4.6 Cipher Feedback (CFB) mode . . . 55
4.7 Output Feedback (OFB) mode . . . 55
4.8 Galois/Counter Mode (GCM) authentication . . . 57

5.1 XOR Tree for CBC Mode . . . 76
5.2 Simulated CTR results with 2 cycle latency on Core 2 . . . 80
5.3 Simulated CTR results with 5 cycle latency on Core 2 . . . 80
5.4 Simulated CBC4 results with 2 cycle latency on Core 2 . . . 82
5.5 Simulated CBC4 results with 5 cycle latency on Core 2 . . . 82
5.6 CTR results on Core i5 . . . 84
5.7 CBC4 results on Core i5 . . . 84
5.8 Results, streams vs. interleaving distance in CBC (Core i5) . . . 86

6.1 Structure of the AES-GEN System. 95
6.2 Structure of the ILP Optimizer. 98
6.3 CTR R1 128, Initiation Interval vs. Latency Set. 101
6.4 ECB Results with 1K buffer 108
6.5 ECB 128 results with various input buffer sizes 108
6.6 CBC algorithm implementations 111
6.7 PCBC algorithm implementations 113
6.8 CFB algorithm implementations 114
6.9 OFB algorithm implementations 115
6.10 GCM 128 Results using SMT 118
6.11 Counter Results, AES-GEN vs. Intel Assembly 126
6.12 CBC Results, AES-GEN vs. Intel Assembly 127

List of Code Listings

2.1 Simulated annealing pseudocode. 22

3.1 Stream source code, general operations. 32
3.2 Generated vector code, general operations. 33
3.3 Stream source code, array lookups. 33
3.4 Generated vector code, array lookups. 34
3.5 Stream source code, for-loops 35
3.6 Generated vector code, for-loops 36

4.1 Pseudo-AES code using ECB block cipher mode 48
4.2 Using AES-NI in AES-128 CTR mode. 65

5.1 Using AES-NI in AES-128 CBC mode. 69
5.2 Interleaving three iterations in CTR mode. 70
5.3 Software pipelining with an initiation interval of 2 72
5.4 Interleaving three encryption streams in CBC mode. 75
5.5 Psuedocode of our implementation of simulated annealing. 77

6.1 Cheetah templated AES-128 Counter code. 97
6.2 Modulo scheduling example, standard CTR. 99
6.3 AES CTR (round 1) encryption in C using AES-NI instructions. 104
6.4 AES CTR (round 2) encryption in C using AES-NI instructions. 105
6.5 AES-128 ECB Encryption loop 107
6.6 Strict interleaving four streams of CFB mode 130

C.1 Fastest scheduled CTR (round 1) code found by AES-GEN. 161
C.2 Fastest scheduled CTR (round 2) code found by AES-GEN. 167

Chapter 1

Motivation

There is no shortage of sensitive information that is transmitted on a daily basis. Medical records, financial transactions, and military communications[1] are all examples of sensitive information that we do not want to freely share with others. To prevent unintended parties from accessing this type of information, encryption is the only answer. Encryption is a vital and necessary component of communicating through several media.

Encryption (or more generally, cryptography) often conjures an image of mathematicians trying to detect patterns in some sort of encrypted text vital to national security. However, modern cryptography actually encompasses several disciplines of study. Mathematical, electrical engineering, and computer science concepts must be combined in order for modern encryption processes to work effectively. As such, the study of modern cryptology surrounding these topics is massive and often appeals to specific fields. Mathematicians are generally concerned with the security and complexity of an encryption algorithm which is referred to as cryptanalysis. Engineers and computer scientists find themselves with the task of implementing these algorithms in hardware and software.

Implementing encryption algorithms on modern architectures has its downside. It can be costly in both time and power requirements. Crucial financial transactions for both investment firms and personal credit card based purchases alike depend on data being encrypted and decrypted as fast as possible for optimal usability. Implementing fast encryption software can be problematic for developers.

The Advanced Encryption Standard (AES) [Daemen and Rijmen, 2002; The National Institute of Standards and Technology (NIST), 2001] is a common and widely used method of encrypting data on computer networks. AES was developed as a suc-

[1] or, perhaps more topical: US embassy cables

cessor to the Data Encryption Standard (DES), which for many years was the most analyzed, important, and widely used cryptoalgorithm [Smid and Branstad, 1988]. Sequential software implementations of the AES algorithm take large amounts of CPU clock time. The AES algorithm uses repeated byte substitutions, shifts, and bitwise xor operations to encrypt the data. Generalized byte substitutions are difficult to implement efficiently in software. Optimized software implementations combine multiple operations with table lookups. These optimized solutions still require 10 to 14 memory operations to encrypt every byte of data. Due to the importance of using AES, faster implementations are required.

In January 2010, in order implement these costly AES instructions on-chip, Intel released the Westmere microarchitecture that dedicated an instruction-set extension called AES New Instructions (AES-NI) [Gueron, 2010] to its x86_64 architecture. These instructions have dramatically increased the performance of traditional AES implementations by an order of magnitude. The instructions have high throughput rates, but long latencies. Achieving good encryption performance using these instructions requires the use of hand-tuned assembly libraries. However, relying on assembly libraries has its problems. Assembly implementations are not necessarily optimal for all systems. As new hardware emerges, existing assembly libraries need to be re-evaluated and modified to reflect architectural features. Modifying an algorithm even slightly can require massive changes at the assembly-level and in turn limits exploratory approaches to solving problems. Making both architectural and algorithmic changes to assembly code is incredibly expensive and time-consuming.

With AES-NI being included on new architectures from the world's market share leader of computer microprocessors, opportunities to use local encryption[2] are rapidly increasing. Using AES-NI effectively is rapidly becoming an important problem to solve. Different processor architectural properties greatly affect the performance of AES-NI code. This dissertation presents a program generation system that can build and search many possible AES implementations in a short time-span to find a near optimal solution. Based on the impressive results of this system, it is reasonable to suggest that the scope of these techniques could be widened by further research; the generator could possibly be adapted to optimize other algorithms that use instruction-set extensions.

[2] Such as encrypting entire hard-disks, subsections of a hard-disk for private user account files, or compressed file archives.

1.1 Thesis

This dissertation argues that using AES-NI instructions effectively is not a trivial process. Current sequentially executing AES-NI based implementations rely too much on the out-of-order architecture to exploit instruction-level parallelism. The thesis of this work is that using a generalized program generator is an effective solution for easily finding near optimal AES implementations that use AES-NI instructions and it serves as a suitable alternative to generating assembly code. We argue that a program generator can accomplish this with three important techniques: (1) making high-level algorithmic changes automatically allows significant opportunities for optimization. Combining these changes seamlessly with traditional optimizations, such as (2) scheduling the AES code to exploit instruction-level parallelism, is a valuable strategy. The program generator also (3) finely tunes arguments that affect (1) and (2) and searches this space using an adapted simulated annealing algorithm that finds a solution quickly. This system allows near optimal AES code to be generated for any base architecture supporting AES-NI in a variety of execution environments[3].

1.2 Contributions

This dissertation makes the following contributions:

- **Vector Code Generation** — In Chapter 3, we find generating vector code is an effective strategy to map streaming languages to general purpose processors. Using a streaming language as a framework to generate vector instructions is a minor, but important contribution. We found that this strategy worked, but allowed us to investigate different systems for code generation.

- **AES Program Generation** — In Chapter 5, we present a program generator that builds CTR and CBC AES implementations. We show this generator can build AES code that performs similarly to hand-coded assembly language libraries. The generator automatically tunes implementations to the underlying architecture.

- **Flexible Program Generation** — In Chapter 6, we present a generalized program generation system that can easily schedule optimized AES code from high-level source files. We present evidence that using software pipelining and modulo

[3]Execution environments such as sequential execution, encryption using simultaneous multi-threading, or encrypting multiple streams simultaneously

scheduling can lead to very good schedules and superior performance for parallelizable AES modes.

- **Algorithmic Choices** — We explore the use of automating algorithmic choices through the use of optimizations such as: assigning round keys to registers, setting restrict pointers, adding software prefetch directives, and `xor` optimizations. These choices make small but crucial changes in encryption runtime. Making these changes manually is difficult if one has to completely reschedule the code. We find applying these algorithmic changes seamlessly with other optimizations is important in achieving best performance.
- **Exploiting `xor` for Cyclic Modes** — With block ciphers, the `plaintext` of each block must be added into the key stream at some point. This is done with an `xor` operator. `Xor` is both an associative and a commutative logical operation. We propose a novel way of generating AES code that exploits its mathematical properties. Doing this results in a reduction of the dependency chain and allows additional parallelism within the loop body. We also exploit this property at different "levels" within the generation system.
- **Function Stitching** — For function stitched code (like combined encryption/authentication GCM mode), our program generation system can still schedule code effectively. In assembly, function stitching requires merging two assembly functions. Rescheduling two merged functions in assembly code is difficult. We show the ease of finding an optimized schedule in our system by using loop fusion to merge encryption and authentication functions.
- **Simulated Annealing** — Our program generation systems can effectively produce an infinite number of possible AES solutions. With an adapted simulated annealing algorithm, we show that it is a useful code tuning tool to traverse the search space quickly while finding a good solution.
- **Assembly Code Alternative** — The work in Chapters 5 and 6 present two different program generators. In both cases, we find that using a program generator to build AES code is a suitable alternative to writing assembly. This is shown in two ways. First, we are able to generate AES implementations that achieve similar or superior performance compared to reported cycles/byte figures documented by Intel's hand-tuned assembly listings. Secondly, the importance of this contribution is compounded by our inability to reproduce the results reported by Intel, using their assembly listings.

1.3 Relevant Academic Publications

In chronological order, from earliest:

(1) Mapping Streaming Languages to General Purpose Processors through Vectorization — Raymond Manley and David Gregg — *The 22nd International Workshop on Languages and Compilers for Parallel Computing 2009* — Newark, DE, USA — Volume 5898 of *Lecture Notes in Computer Science*, pp. 95–110. Springer Berlin / Heidelberg.

(2) Code Generation for Hardware Accelerated AES — Raymond Manley, Paul Magrath, and David Gregg — *21st IEEE International Conference on Application-specific Systems Architectures and Processors(ASAP), 2010* — Rennes, France — pp. 345 –348.

(3) A Program Generator for Intel AES-NI Instructions — Raymond Manley and David Gregg — *11th International Conference on Cryptology in India, Indocrypt 2010* — Hyderabad, India — Volume 6498 of *Lecture Notes in Computer Science*, pp. 311–327. Springer.

The work in publications listed above was primarily conducted by the author of this dissertation under the supervision of Dr. David Gregg. Additional background research was provided by Paul Magrath in (2). Mike O'Hanlon at Intel Shannon, provided feedback and suggestions for our work in addition to facilitating early access to AES-NI hardware for work done in both (2) and (3). Vinodh Gopal at Intel provided additional feedback on our work in (3).

1.4 Dissertation Outline

The title of this dissertation is *Program Generation for Intel AES New Instructions.* The work presented in this document covers topics that relate to the generation of code using instruction-set extensions. Following this introductory chapter, the dissertation continues with a background of architectural features, software ILP optimizations, code generation and streaming languages. Chapter 3 presents work on the automatic code generation for vector instructions for stream applications on general purpose processors. We narrow down the scope of code generation to a single and important problem and an in-depth background and literary survey on AES encryption, block cipher modes, and both software and hardware optimizations for AES is provided in Chapter 4. Chapter 5 presents our first system to generate AES code for both CTR and CBC modes,

using a “static” program generator that both simulates and fully supports AES-NI instructions. Expanding code generation for other AES modes, Chapter 6 presents the generalized AES program generator system that argues our thesis statement. Finally, this dissertation concludes with our final thoughts in Chapter 7. This final chapter contains a critique of our work, future work ideas and an assessment of our contributions. Appendix A contains a glossary of acronyms commonly used throughout this dissertation. Appendix B is attached for “finite” values of graphs found in Chapter 6. Appendix C contains full optimized and scheduled code listings. Appendix D includes AES-GEN performance results when running on a different microarchitecture that supports AES-NI—Sandy Bridge.

Chapter 2

Background

This chapter provides background material and literature surveys on the topics necessary to the discussion of the work presented in this dissertation. Namely: Architecture in Section 2.1, Software ILP Optimizations in Section 2.2, Program Generation in Section 2.3, and Streaming Languages in Section 2.4.

2.1 Architectural Features

As this dissertation deals with exploiting instruction-level parallelism (ILP) when using high latency instructions through code generation, this section provides background on the architectural features that enable these optimizations. The following techniques that improve ILP deal with an idea that *latency*, or the number of clock cycles an instruction takes to complete, is not uniform for all instructions. In addition, the number of clock-cycles that must elapse before a new instruction can be issued (or the *throughput*) will also affect the amount of ILP that can be exploited on multi-issue architectures.

2.1.1 Instruction Pipelining

On multiple-issue architectures, pipelining is a technique that allows machine level instructions to be completed in stages. The number of stages that exist in the pipeline and how the stages are grouped varies for each processor. An instruction will "occupy" a stage in the pipeline, allowing multiple instructions to be executed in parallel.

A new instruction is issued when the instruction currently in the first stage moves on to the second stage and so on. The pipeline stages are designed to be as balanced as possible, or each stage should complete in a unified number of clock cycles. Stages

Instruction	Cycle								
	0	1	2	3	4	5	6	7	8
1	IF	ID	EX	MEM	WB				
2		IF	ID	EX	MEM	WB			
3			IF	ID	EX	MEM	WB		
4				IF	ID	EX	MEM	WB	
5					IF	ID	EX	MEM	WB

Figure 2.1: Generic five-stage instruction pipeline.

are often broken down into several additional stages for instructions that take longer to complete within the pipeline. The number of stages implemented on a processor is influenced by the latency and throughput of its instruction set. While the length of pipelines has gradually increased since their widespread adoption in the 1980s, more stages does not necessarily result in reduction of run-time. The Pentium 4 has a 31 stage pipeline [Hinton et al., 2001], but the Core i7 has reduced this number to 14 [Intel Corp., 2011]. The classic five-stage instruction pipeline [Hennessy and Patterson, 1992] is shown in Figure 2.1 with the following stages:

1. **(IF) Instruction Fetch** Instructions are fetched from the cache.
2. **(ID) Instruction Decode** Instructions are decoded in divided into parts needed for execution.
3. **(EX) Execute** Now decoded, instructions execute and take a variable number of cycles to complete, depending on type of instruction.
4. **(MEM) Memory Operations** This stage accommodates long-latency memory operations if the executing instruction needs to fetch data.
5. **(WB) Write-back** Write results of the instruction back to memory.

More complex pipelines are sometimes designed with additional stages to accommodate instruction-set extensions to the architecture. For example, this dissertation mentions the `aesenc` instruction multiple times which has a 2 cycle throughput and a 6 cycle latency on the Intel Westmere processor [Akdemir et al., 2010]. For this to work, multiple stages must exist to allow for possibly three of these instructions to execute in parallel. While a pipeline has the ability to execute multiple instructions in parallel, data dependency between instructions will cause the pipeline to *stall*.

On in-order processors, if an instruction is held up in a particular stage, the processor must wait to push instructions down the pipeline. This stall wastes clock cycles while the processor waits to complete the instruction. Stalls are caused by structural stalls, control stalls, or data stalls. Structural stalls happen when a multiple instructions require the same processor resource. Control stalls occur with conditional statements that could change the target of the following instruction. Branch predictors try to prevent these stalls. Data stalls occur when an instruction has a data dependency on another instruction already in the pipeline. The instruction cannot continue until its predecessor completes executing. There are a number of possible solutions to prevent and/or limit these pipeline stalls. However, limiting data stalls are of particular interest to us[1].

	Cycle								
Instruction	0	1	2	3	4	5	6	7	8
1	IF	ID	EX	MEM	WB				
2	IF	ID	EX	MEM	WB				
3		IF	ID	EX	MEM	WB			
4		IF	ID	EX	MEM	WB			
5			IF	ID	EX	MEM	WB		
6			IF	ID	EX	MEM	WB		
7				IF	ID	EX	MEM	WB	
8				IF	ID	EX	MEM	WB	
9					IF	ID	EX	MEM	WB
10					IF	ID	EX	MEM	WB

Figure 2.2: Generic superscalar pipeline.

2.1.2 Superscalar

Instructions on a single-issue scalar processor execute sequentially and process one or two operands. Functional units that are not needed while the current instruction is being executed remain idle when they could be doing something useful. Superscalar architectures were designed to improve instruction scheduling by dispatching instructions to idle functional units. A superscalar architecture is a multiple-issue processor

[1] As we will see in Chapter 4, the AES algorithm is a block-cipher that has a chain of dependent encryption rounds and executing these rounds will cause data stalls.

which implements a form of instruction-level parallelism by executing more than one instruction at a time on different functional units. Figure 2.2 shows a superscalar pipeline that fetches and executes two instructions at a time.

Using a superscalar pipeline allows the CPU to have greater throughput than its clock speed would suggest and would normally process more than one instruction per clock cycle. The ability to parallelize at this level depends on several factors. One such factor is the number and types of functional units. Different functional units can process different types of instructions. For example, given a processor has a functional unit that can process integer and float operations and a functional unit that can process integer and memory operations, two integer operations could be completed in parallel, whereas two sequential float instructions would cause a stall. The data dependency between instructions is also checked by the hardware at run time to ensure that programs execute properly. This is a important limitation of superscalar hardware to note, as other optimization techniques must be exploited in these instances.

2.1.3 Simultaneous Multithreading

Research has shown that superscalar architectures are limited at saturating the available resources (functional units) with multi-issue processors [Eggers et al., 1997]. Simultaneous multithreading (SMT) is a modern architectural feature on superscalar CPUs that implements hardware multithreading by permitting independent threads of execution in an effort to better utilize processor resources. To do this, instructions are issued from independent threads to multiple functional units. While increased resources are required to fetch instructions from multiple threads and keep thread information, the use of SMT will theoretically allow instructions from different threads to be used by idle parts of the processor. This is why SMT is implemented on a superscalar processor. This technique is very similar to superscalar pipelines but with multiple instruction queues to fetch from.

Work by Tullsen et al. [1995] showed that SMT has potential to increase resource usage on single-chip multiprocessors. They compared SMT to single-chip multiprocessing. While the two systems have similar organization layouts—both have their own set of registers, multiple functional units, and high issue-bandwidth—they have a very different strategy for partitioning and scheduling resources. Multiprocessing issues instructions to a fixed number of functional units to complete each thread. SMT can change this assignment every cycle. This allows for more efficient use of functional units.

On Intel processors since the Pentium 4 (and including the ones used for experiments

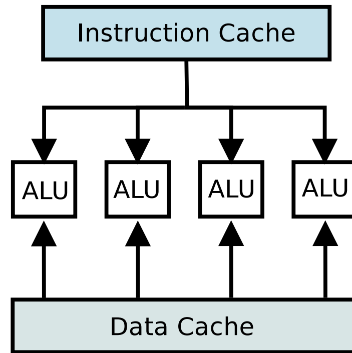


Figure 2.3: SIMD example, a single instruction is applied to multiple data.

found in this thesis), *hyper-threading* is the term used for Intel's SMT implementation [Intel Corp., 2011]. Intel implements SMT by addressing two virtual or logical processors for every one physical processor. It is possible to pin a process to one of these virtual processors, but this remains transparent to the operating system. The hardware does this so the operating system can think it is scheduling multiple processes simultaneously on "different processors", allowing the processor to send instructions down the pipeline from multiple threads. This limits structural stalls.

2.1.4 Vector/SIMD

SIMD (Single Instruction, Multiple Data) is a technique employed to achieve data level parallelism. On modern architectures, SIMD serves as a vector processor. In SIMD computer architecture [Flynn, 1972], the computer exploits multiple data streams against a single instruction stream in order to perform operations that may be easily parallelized. This is shown in Figure 2.3. By processing multiple data elements in parallel, SIMD processors provide a way to utilize data parallelism in applications that apply a single operation to all elements in a vector.

The idea behind SIMD programming is that one can perform the same action on several elements in parallel. On general purpose processors, the "vector processor" comes in the form of extensions to the instruction set architecture (ISA). For example, an instruction-set extension (ISE) on the x86 architecture is Streaming SIMD Extensions (SSE) [Intel Corp., 2011]. Other processors also have vector processing ISEs, such as: AltiVec for the PowerPC architecture [Freescale Semiconductor, 1999] and VIS on UltraSparc [Kohn et al., 1995]. SSE is a SIMD instruction set extension to the x86 architecture. SSE has access to eight 128-bit registers. The 64-bit extension to the x86 architecture (x86_64), adds an additional eight vector registers to use. XMM0 through XMM15 can be accessed in 64-bit operating mode, while only XMM0 through

XMM7 can be accessed in 32-bit operating mode. Other generations of SSE have been added to current x86 architectures. The latest iteration of SSE is version 4.2 which was released in 2009.

Intrinsics

Compilers are limited in applying vectorization from high-level code without additional annotations. While writing assembly-level vector instructions can be done, a set of macros called *intrinsics* exist to allow high-level language programmers to make use of SIMD instructions. Intrinsics are written as high-level language functions that often have a 1:1 correspondence to their assembly level counterparts. Code that uses intrinsics can also benefit from further low-level optimizations applied by a high-level language compiler as he compiler will manage register allocation and instruction scheduling. Instrinsics are also used in source-to-source translators and code generators that attempt to automatically vectorize code.

2.1.5 Autovectorization

While it is relatively easy to incorporate the advantage of using vectors while writing code on the assembly level, automatically generating vector code from high-level languages remains a serious challenge. Data dependency and loop analysis are key research points in vectorization [Wolfe, 1990; Allen and Kennedy, 1987]. Current autovectorization techniques are generally limited to transforming loops when a number of requirements are known and/or met. For example, gcc requires several conditions to be met in order to vectorize loops [Naishlos, 2004; Nuzman and Zaks, 2006]. If values of successive iterations of a loop are dependent on its predecessor, data dependency prevents the vectorization of loops. Data alignment becomes a problem as vector registers generally require data to be aligned to a specific boundary (most commonly quad-word). Interleaved data (found in user defined structures) also causes data-alignment issues. Control-flow structures, such as `if` statements, are difficult to vectorize as conditionals may affect only some elements currently loaded into a vector. Many autovectorization techniques are not implemented by stand-alone compilers. Autovectorization is often more effective when tuning for specific algorithms where memory boundaries and conditional flow in loops are known. Autovectorization is a combination of both hardware and software ILP optimization techniques.

2.2 Software ILP Optimizations

Hardware mechanisms to improve instruction-level parallelism can only go so far. For in-order architectures, exploiting ILP depends entirely on the compiler. Preventing stalls in the pipelines on superscalar machines depend on the sequence of instructions and which resources they wish to use. Both in-order and out-of-order execution architectures can benefit from high-level code that encourages standard compilers to exploit ILP in software using different *instruction scheduling* techniques.

2.2.1 Instruction Scheduling

Compilers use instruction scheduling to expose more instruction-level parallelism for superscalar architectures. On in-order architectures, the compiler uses instruction scheduling to re-arrange the operations into groups that can be independently executed. A simple but effective *list scheduling* algorithm can be applied to a basic block to create the independent groups and this can result in a near optimal scheduling solution [Lawler et al., 1987]. Groups are created by building a data dependence graph (DDG), in which operations are represented by nodes with directed edges that show dependencies between the operations. Exploiting ILP when scheduling basic blocks which execute repeatedly is an important problem.

2.2.2 Loop Unrolling

The majority of running time in most programs is spent iterating over a loop or several loops. Until recently, with the advent of streaming languages for example, the basic blocks of loops are usually not written to be executed in parallel in a high-level language. Usually, the iterations of the loop can be executed in parallel, and loop unrolling is a common way to exploit ILP. This allows out-of-order architectures to execute instructions from several iterations of the loop body that is likely to be subject to flow-dependency in single iterations. Unrolling is an easy optimization to implement in compilers and existing compilers apply it to loops to great effect [Lowney et al., 1993]. However, unrolling loops do have some drawbacks. Unrolling means more code and code growth can be a sensitive issue in things like embedded processors. Unrolling also suffers based on the degree of unrolling. If loops are unrolled by a degree that is not evenly divisible by the number of iterations, a scalar “clean-up” loop is necessary to execute remaining iterations. With small loops, this can seriously affect any speedup potential. With these drawbacks in mind, there is a technique that can solve both

these problems while continuing to exploit ILP.

2.2.3 Software Pipelining

Software pipelining is an instruction scheduling technique that takes instructions from several loop iterations and combines them to build a new, parallelized basic block [Lam, 1988]. Figure 2.4 shows how code from a simple loop is scheduled using software pipelining. In Figure 2.4a, the control flow graph representation shows a loop with four statements. Using an acyclic schedule (shown in Figure 2.4b), the statements are executed sequentially. In Figure 2.4c, software pipelining is applied to the loop and the four statements are scheduled to execute simultaneously, exploiting ILP. This is achieved by breaking down software pipelining into three stages:

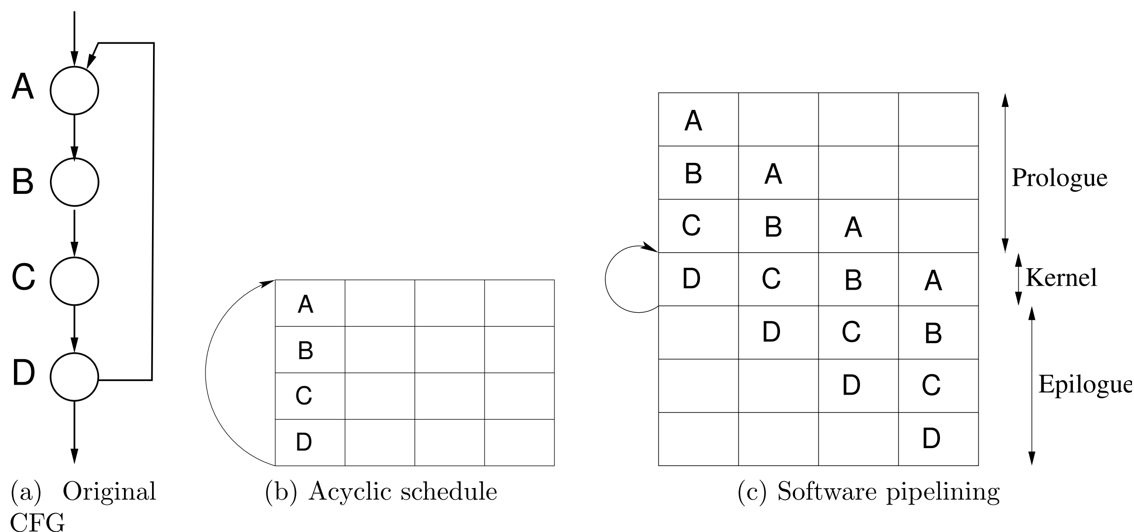
1. The *prologue* contains all instructions that are required to be executed before the instructions of the current loop that will be included in the loop body.

2. The loop body is referred to as the kernel or *steady state*, as once the program reaches this point, this body of code will repeat until it must exit. The kernel contains all instructions needed to complete one iteration of the loop, but as instructions will be from different iterations, they can be scheduled in parallel.

3. When the kernel finishes executing, some loop iterations are still live. The *epilogue* consists of code required to finish these iterations correctly.

While Figure 2.4 shows a simple four statement loop body, the number of statements from each iteration included inside the steady state needs to be flexible. It might be pertinent to allow long-latency instructions to be scheduled further down the pipeline, or across several kernel stages. Low-latency instructions may be grouped together and executed in a single pipeline stage. The "length" of the kernel is measured in machine cycles and is known as the *initiation interval* (*ii*). This value can be adjusted to exploit different levels of ILP. The *ii* sets the throughput of the pipeline.

The use of the three stages creates tighter ILP schedules while solving the problem of code growth and the need for a clean up loop. While software pipelining expands code, the growth is quite small in comparison to unrolling and affects performance less. Stifling code growth within the loop is the most important [Rau, 1994] and the software pipelining kernel is the same size as the original loop body. However, for multiple iterations to be executed the same loop body, variable renaming and copies must be included [Lam, 1988] to send data from one loop iteration to the next. This



(a) Original CFG (b) Acyclic schedule (c) Software pipelining

Figure 2.4: Software pipelining example. Images taken from [Gregg, 2001]

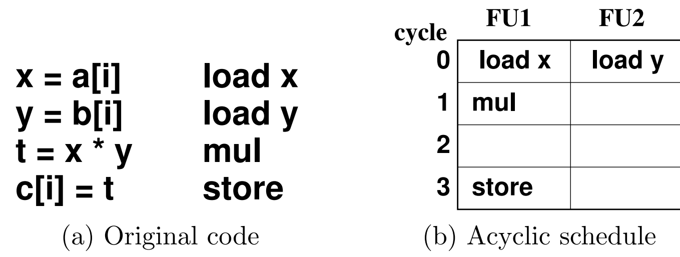
can increase register pressure, but the variables created by the renaming and copy instructions can be optimized out at compile time.

The prologue and epilogue also contribute to code growth, but the amount is dictated by how many loop iterations exist in the kernel. Inherently, the instructions in both stages can also be out-of-order which benefits during global instruction scheduling. Compilers can unroll loops to high degrees, causing an inflated number of instructions scheduled and this increases compensation code which is needed to support the overlapped iterations. Software pipelining also does not require an expensive clean up loop, as the epilogue finishes the loop cleanly.

2.2.4 Modulo Scheduling

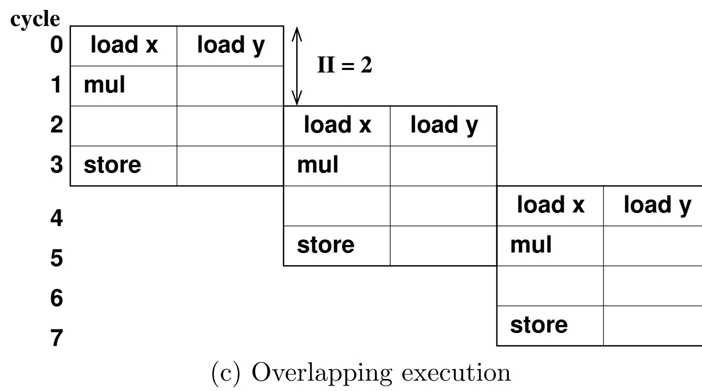
The initiation interval determines how a software pipelining kernel is built. From our experience, we used ii slightly differently in Chapters 5 and 6. In Chapter 5, the throughput is measured in number statement lines without any regard to what their corresponding throughput and latency would be at the machine level. In Chapter 6, we can assign specific values to each statement in the graph and ii value will schedule them accordingly. The approach we used to build our software pipelined loops in Chapter 6 is called *modulo scheduling* [Rau and Glaeser, 1981]. Modulo scheduling is a method of implementing software pipelining in loops that do not contain branches that produces good code while minimizing code growth.

Figure 2.5 shows how code from a simple loop would be pipelined using modulo scheduling. In Figure 2.5a, the original source code shows a loop body that loads two

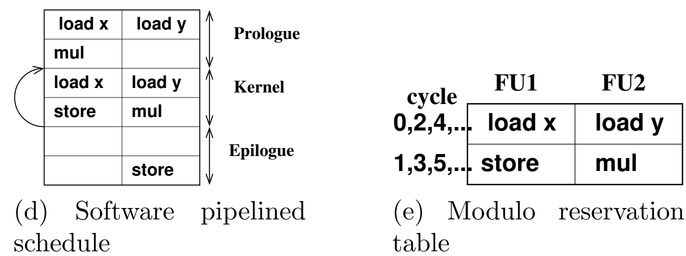


(a) Original code

(b) Acyclic schedule



(c) Overlapping execution



(d) Software pipelined schedule

(e) Modulo reservation table

Figure 2.5: Modulo scheduling example. Images taken from [Gregg, 2001]

values from separate arrays, multiplies these values together, and stores the result into a third array. Given that the load and store instructions take one cycle each and the multiply takes two cycles, the acyclic schedule for this loop on a processor with two functional units is scheduled as seen in Figure 2.5b. There is no cyclic dependency from one iteration to the next; each iteration completes in four cycles and successive iterations can be overlapped and execute in parallel as shown in Figure 2.5c using an *ii* value of 2. Figure 2.5d, shows the software pipelined schedule of the loop. The prologue contains the first three instructions. The kernel then contains the loads of `x[i+1]` and `y[i+1]` executes `mul` on those values while the previous iteration's `mul` instruction finishes and stores the value in `t[i]` before the loop continues. The epilogue then contains the clean up code, which in this case is a store. The order in which these instructions execute is listed in Figure 2.5e. While software pipelining and modulo scheduling are very effective techniques for instruction scheduling, they are many other methods to optimize entire algorithms or programs and is sometimes done with a *program generator* (or *code generator*).

2.3 Program Generation

The origin of early code generators were "superoptimizers" and "peephole optimizers". The scope of these optimizers were limited to small programs or sections of a program to reduce the number of instructions in a program through instruction selection, constant folding, and removing redundant code. Massalin [1987] created a "superoptimizer" to find the fewest instructions required to compute a given function. However, the author states that often this technique is only practical for programs that have 13 machine instructions or less, and the effectiveness is largely dependent on the target architecture (and which instructions are available). Peephole optimizers were often integrated into code generators. Davidson and Fraser [1984] described techniques on how to automatically generate peephole optimizations using a description of the underlying architecture. Fraser and Wendt [1988] extended this work to integrate a set of peephole optimizations with a naïve code generator to create an automatic code generator and optimizer that could be retargetted to different architectures with minimal effort.

The scheduling optimizations listed in the previous section are solutions that, when applied, are optimized for a given architecture. As the architectural features change for new processors, the scheduling optimizations will often need to be recalculated for optimal (or near-optimal) performance. Scheduling is also not the only optimization that must be considered when compiling for a different platform. When presented

with the challenge of optimizing an algorithm, many optimization techniques need to be considered. A system that considers these architectural or algorithmic changes, and generates an optimized version of the code is called a program generator.

Iterative program generators take a piece of code and try to find the fastest version by trying combinations of different techniques to optimize the code, just like a programmer would by hand. The difference is that a code generator does not have to make guesses like a programmer would. An iterative program generator can try all the conceivable combinations of the optimization techniques to find the best solution. The lack of user intervention is convenient and cost saving. The work involved in optimizing even a small piece of code can take weeks by hand. A code generator can generate all the combinations possible almost instantly. While it completes a time-consuming search for a solution, it requires no attention. In exchange for this one expensive search, the time saved by running a faster version of the code on a continual basis is desirable.

2.3.1 Domain Specific Code Generators

There are several well known code generators (also referred to as library generators) that focus on a particular subset of problems. One of the most successful code generators is Fastest Fourier Transform in the West (FFTW) [Frigo et al., 2005; Frigo and Johnson, 1998]. It is a library for computing the discrete Fourier transform (DFT) and is designed to adapt to hardware features to maximize performance. It is able to tune code for machine-specific instruction-set extensions, like SIMD, and uses a specialized compiler to generate optimized implementation of discrete cosine and sine transforms from the parent DFT algorithm. The DFT algorithm can be of real- or complex-type valued arrays of varied size and dimensions. Depending on these defined algorithmic properties and the underlying architecture, the derived DFT algorithm could work very differently from machine to machine.

FFTW generates this code in two parts. The FFTW planner (the code generator) is invoked and the user must specify the shape of the input data (type values and array dimension sizes) of the problem to be solved. The planner measures the actual run time of many different plans and returns the fastest. This is often a time-consuming process but as DFTs are computationally expensive, it is often worth while for "performance-critical" applications. Without getting into the mathematics behind DFTs, FFTW solves a problem first by reducing the defined vector rank of a DFT to a single (possibly multi-dimensional) DFT. This single DFT must then be reduced to a sequence of one-dimensional DFTs. These one-dimensional DFTs are then solved by a known DFT algorithm. Since these steps can be executed out of order and/or interleaves, searching a

series of plan-permutations eventually returns a “correct” plan. These transformations yield various solutions that will nest loops differently with varying vector sizes and so on. The cost to compute the DCT given these high-level code constructs is calculated given the architectural properties. This is how a “correct” plan is determined.

Whaley et al. [2000] presents a code generator called ATLAS, which generates linear algebra routines and focuses the optimization process on the implementation of matrix multiplication. Like the FFTW planner, the parameter values of a matrix multiplication implementation influence how the code is generated. The generator tries multiple values for the tile size and loop unrolling degrees and evaluates the costs of the various configurations. The almost exhaustive search proceeds by generating different versions (different sizes) of matrix multiplication that are dependent on the input values.

SPIRAL, or *Signal Processing Implementation Research for Adaptable Libraries*, is another code generation library. SPIRAL can generate code for several subsets and formulae associated with digital signal processing (DSP), such as linear transformations [Püschel et al., 2005]. Through SPIRAL, researchers have investigated the use of automatic vectorization of both the Discrete Fourier Transform (DFT) [Franchetti and Püschel, 2003] and the FFT [Franchetti and Püschel, 2007].

In those works, Franchetti and Püschel discuss the difficult process of exploiting SSE instructions on scalar code. In addition, they also mention the problem of finding an optimal solution when generating code for a target architecture. They offer a specific dynamic programming search to find a “good match” between the formula space and architecture it is being mapped to. Of more direct relevance is their work on the SPL compiler [Xiong et al., 2001]. The SPL compiler, developed for SPIRAL is a system that uses formula transformations and intelligent search strategies to create optimized DSP libraries.

2.3.2 Traversing the Search Space

As mentioned earlier, early code generators had a limited scope for optimization. Instruction sets were smaller and “useful programs” were about a dozen machine instructions long. With Massalin [1987]’s superoptimizer, he considered that it was feasible to exhaustively search every possible program. However, he concedes that the search grows exponentially with each additional instruction.

Frigo et al. [2005] states that finding a correct plan can be a very time-consuming process for FFTW. FFTW and ATLAS can both use exhaustive searches as their search spaces are finite, but code generators often have to traverse a search space which rapidly

expands with small options that can be turned on and off that may or may not improve the code depending on another option's state. Consider the optimization flags that could be passed to gcc alone. The resulting possibilities of a complex program could be in the billions. There needs to be a way to search the search space more efficiently because exhaustively trying all combinations is rarely computationally feasible. Several techniques have been developed to traverse these vast search spaces.

PEAK is a system introduced by Pan and Eigenmann [2006b] which considers a feedback-directed approach to making the most use of 38 of gcc's on/off optimization flags[2]. They find solutions quickly (in comparison to other iterative compilation techniques) by creating program "sections". PEAK detects which parts of the program could benefit from optimization and places them into a "section". The system then profiles each applies compiler optimizations independently to these sections. Once each section is tuned, the program is compiled and given to the user. To traverse the exponential search space, they apply an algorithm called combined elimination [Pan and Eigenmann, 2006a]. This algorithm combines several ideas to reduce the search space. It first identifies which optimizations increase runtime. Then it iteratively tries to eliminate each negative-optimization one at a time. The authors claim similar performance gains can be achieved from a search that takes several minutes, as opposed to a few hours.

Related work by Hoste and Eeckhout [2008] attempts to find the best gcc optimizations again amongst an even greater search space[3]. Their system, Compiler Optimization Level Exploration (COLE), uses a multi-objective search based on an evolutionary algorithm to adjust for code size, code quality, and/or compilation time. Acovea is a tool developed by Ladd [2009] that uses genetic algorithms to find the best options when compiling a program with gcc. Similar ideas have been proposed for code generators.

Work on using machine-learning algorithms to find the best general compiler options when tuning code for a particular target has also been investigated. Li et al. [2005] used SPIRAL to demonstrate the power of using machine learning techniques in automatic algorithm selection and optimization. Li et al. argue that while generators like SPIRAL and FFTW can find near-optimal performance for a particular machine just by using empirical searches, these algorithms have only been shown to work without considering the characteristics of input data. By using genetic algorithms and a classifier system that consider and adapt specifically to input data, Li et al. are able

[2]These options create 2^{38} combinations, and are used in their experiments.
[3]The authors state a search space of greater than 2^{60}.

to increase performance of classic sorting algorithms. They argue that using a genetic algorithm is necessary because the deep cache hierarchy and complex architectural features in modern systems are not considered when studying the complexity of sorting algorithms.

The MILEPOST GCC project [Fursin et al., 2008] also uses machine learning during the gcc compilation process for a broader range of inputs and hardware targets. What is unique about their approach is that gcc may be used to compile any program that could target a number of different platforms. As such, the machine learning compiler they describe can automatically adjust its optimization heuristics to make applications faster, reduce the code size, or even the compilation time. To train the machine-learning algorithm, they use a set of 500 random sequences that each applied a range of compiler flags turned on and off to a benchmark. They found that while being a time-consuming process, this training pass worked effectively to build other code when run on a particular architecture.

Expanding on the MILEPOST project, Leather et al. [2009] presented work that automatically finds "features" of a given program that are likely to be improved by the machine-learning techniques. It sends feature data to the machine learning tool and in return, influences the genetic algorithm. Tournavitis et al. [2009] extended work on MILEPOST as well by adapting machine-learning to generating auto-parallelized code. The authors argued that weaknesses exist in traditional parallelizing compilers (such as autovectorizing compilers—see Section 2.1.5). They found that adapting the learning techniques to this problem found more code that could be parallelized compared to that found by other automatic tools. The work far exceeds other tools and comes close to performance of manually parallelized code, such as manually tuned OpenMP.

The heuristic methods mentioned this section are based on genetic programming approaches, decision trees, and the *k-nearest neighbours* classifier to perform machine-learning. There are other methods of traversing a search space without narrowing the scope. One such algorithm is simulated annealing.

2.3.3 Simulated Annealing

Simulated annealing is a heuristic search algorithm that employs probabilistic reasoning to increase the search space [Aarts and Korst, 1988; Skiena, 1998]. It increases the search space by allowing for occasional steps in the wrong direction by jumping to inferior solutions. This jump is random based on the current "temperature". The transition probability from $solution_i$ to $solution_j$ (where s_i and s_j are defined as the cost of each solution) at temperature T is defined as: $P(s_i, s_j, T) = e^{(s_i - s_j)/(k_B T)}$ where

Listing 2.1: Simulated annealing pseudocode.

```
1  Simulated Annealing()
2      Create initial solution S
3      Initialize temperature t
4      repeat
5       for i = 1 to iteration-length do
6           Generate a random transition from S to S[i]
7           if (C(S) >= C(S[i])) then S = S[i]
8           else if (e^(C(S) - C(S[i]))/(k*t)) > random[0,1)) then S = S[i]
9       Reduce temperature t
10     until (no change in C(S))
11     Return S
```

k_B is Blotzmann's constant. Using simulated annealing has been found to be an effective solution for graph partitioning and traveling salesman problems[4][Kirkpatrick, 1984].

This equation is used in the simulating annealing algorithm to accept or reject the current solution. The solutions are evaluated by a cost function. In the beginning, the search can and should be quite erratic in direction, but always testing a neighbouring solution. One of the ideas behind simulated annealing is that bad solutions are accepted earlier in an effort to prevent the search from being stuck in local minima. As the search goes on, the probability of accepting bad solutions reduces. This is done through a temperature variable that "cools down" when progress is made. Once the temperature is fairly low, the algorithm suggests that it has found a good solution and is more likely to test solutions closer to the "good" solution.

Finding a practical cooling solution requires a trial-and-error approach and needs to be tailored for the cost function used (function $C(S)$ in Listing 2.1). This function evaluates the solution and returns a value or cost. The temperature variable itself cools down exponentially after every set number of iterations. The temperature rapidly cools because good solutions are starting to become evident. Solutions will always be accepted if the cost of the current solution is lower than the cost of the previous solution. In addition, the probability function noted above is compared against a random value between 0 and 1. If it the probability function is greater than the random number, then the solution is accepted regardless of cost. A general outline of the simulated annealing algorithm is shown with pseudocode in Listing 2.1.

[4]In Chapters 5 and 6, we use simulated annealing to find solutions among a search space that is computationally infeasible to search exhaustively.

2.4 Streaming Languages

As discussed in the previous section, using a program generator can be a way to expose parallelism in a program. However, some algorithms have properties that are simple and straightforward enough to benefit from a set of optimizations geared for parallel processing. The essential idea of stream processing is that large quantities of data are brought into local memory, manipulated with a single or very few instructions, stored back into memory and then are unlikely to be used again. An example of this type of application[5] is presented in work by Manavski [2007], in which they implement the AES encryption algorithm using a stream language. This programming paradigm has been implemented in a series of *streaming languages* designed to easily exploit parallel processing. Streaming languages were originally (and concurrently) developed for specialized streaming processors, such as RAW [Taylor et al., 2004] and Imagine [Khailany et al., 2001]. Stream processing then found a home on the more commonly found graphics card (GPU) through CUDA [NVIDIA, 2007] and gaming processors, such as the Cell BE [Zhang, 2007], which led to the idea that streaming languages could provide the necessary structure to effectively map their code onto general-purpose processors. More recent streaming efforts like OpenCL [Munshi, 2009] have partnered several corporate entities to develop an open standard for specifying these languages.

Streaming languages are often defined very differently. They can be developed as simple libraries that target a specific processing unit. Rapidmind [Monteyne, 2008] (and later, Intel Array Building Blocks Intel Corp. [2010b]) do this for GPPs. CUDA specifically targets NVIDIA GPUs. OpenCL tries to optimize stream processing by targeting both GPPs and GPUs simultaneously. Some streaming languages, such as Brook [Buck et al., 2004], are defined by simply extending a more familiar language such as C/C++. Others, like StreamIt [Thies et al., 2002], were created with the belief that an entirely new language is essential to representing the stream model. Both platforms share common attributes such as facilities to read and write stream data, kernels (or filters), and control techniques such as reductions. *Kernels* are defined as the function that operates on all elements of a stream. Both platforms also have a compiler to generate code and a runtime (or back-end) to serve as the computational engine. Figure 2.6 shows the general process of compiling a stream program with Brook and Streamit.

The *computational intensity* property of streaming is represented by a kernel. Kernels operate on every element in the stream and are the only place individual stream

[5] And an application important to this dissertation

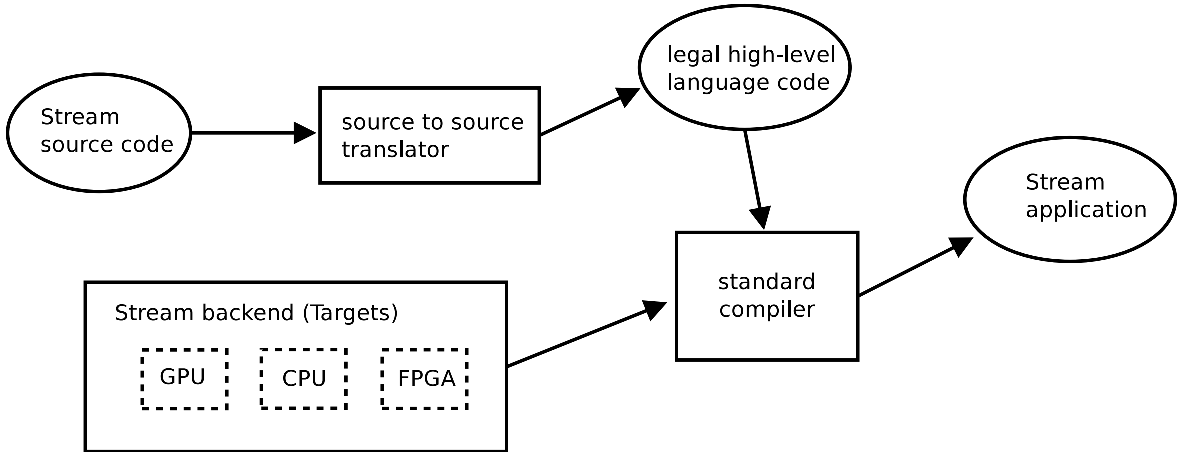


Figure 2.6: General Process of compiling a Streaming Program in Brook and Streamit.

elements can be accessed. It is usually required that streams be initialized and written back out to normal arrays to keep the streams separated between the main body of the program and kernels. In Brook, this is done with a set of runtime functions called `streamRead` and `streamWrite`. It ensures that the language implementation is in complete control of the data structures.

When compiling streaming code, a source-to-source translation is commonly used on the source with resulting output in another high-level language [Das et al., 2006; Amarasinghe et al., 2005]. This allows the C++ compiler to apply low-level optimizations. When translating the kernels, a loop is generated to iterate through the size of the input stream(s). The code contained within the kernel function is then evaluated within the loop for every single element of the stream. Further optimizations are applied, dependent on the target architecture.

The other part of compiling and executing a stream program is the runtime. The responsibilities of the runtime vary depending on the target architecture. The runtimes are stand-alone back-ends that allow a programmer to write stream programs independently of the architecture they run on. Specific optimizations are also built into runtimes to make certain features more efficient on a given platform [Advanced Micro Devices, Inc., 2007]. The runtimes are linked with the streaming application at compile time. Some of the responsibilities of a runtime include: checking the stream size during kernel invocation, bounds checks on array indexing, and managing threads. The runtime also manages the interaction between the kernels and the main body of the program.

A streaming application's *data parallelism* property, in its most basic sense, is a single instruction, multiple data construct. In stream processing, the single instruction

is the kernel which is operating on the multiple data represented by the stream itself. The term vectorization however, generally refers to the transformation from a scalar set of instructions to vector code that utilizes vector processors more effectively.

Chapter 3

Using Streaming Languages to Automatically Generate Vector Code for General Purpose Processors

Making good use of vector/SIMD hardware is a research topic that has been going for decades. Making the most of parallel execution is critical for significant speedups. The vector instructions on common x86 machines usually do 2 to 4 instructions in place of their scalar equivalents. Many vector instructions on modern architectures have similar throughput and latency times compared to their scalar equivalents as well. Often, disassembling code that has been compiled by gcc or icc will reveal that common multiply or add instructions simply use vector instructions and operate only on the lower bits, ignoring up to three possible operations that could be computed at the same cost. Much of the autovectorization research mentioned in Section 2.1.4 has many limitations and requirements that must be met for compilers to effectively vectorize scalar code. This chapter investigates using streaming languages to encourage better vector code generation for general purpose processors that have instruction-set extensions to accommodate SIMD processing.

Streaming languages were originally aimed at streaming architectures, but work has shown the stream programming model to be useful in exploiting parallelism on general purpose processors as described in Section 2.4. Research in mapping stream code onto GPPs deals with load balancing and generating threads based on hardware features. In this chapter, we look into improving problems associated with stream data locality and stream data parallelism on GPPs. We argue that automatically

generating vectorized code for these streaming operations is a potential solution. We use the streaming language Brook as our syntax base and augment it to generate vector intrinsics targeting the x86 architecture. This compiler uses both existing and novel strategies to transform high-level streaming kernel code into vector instructions without requiring additional annotations. We compare our system's results to existing mapping strategies aimed at using stream code on GPPs. When evaluating performance, we see a wide range of speedups from a few percent to over 2x and discuss the effectiveness of using vector code over scalar equivalents in specific application domains.

3.1 Introduction

As the demand for large scale multimedia processing increases, a high-level abstraction to represent the inherent parallelism found in certain applications has yet to be agreed upon. However, the stream programming model continues to be proposed as such a platform [Owens et al., 2002]. Faced with an ever increasing processor core count, streaming languages become an attractive solution as general-purpose processors (GPPs) start to mirror the architectural structure of the more traditional targets like GPUs. Traditional programmers require new tools and new representation models [Zhang et al., 2007] to make use of this advanced hardware. While some streaming languages have focused primarily on generating code for GPUs to perform general computation, others have put effort into adequately making use of the streaming model on GPPs [Buck et al., 2004; Amarasinghe, 2006] and it is not without challenges.

Previous research into mapping streaming languages on GPPs have led some [Talla et al., 2003; Gummaraju and Rosenblum, 2005; Gummaraju et al., 2007] to conclude that without the addition of extra hardware on the processor or chipset, it would be difficult for GPPs to match the performance of their GPU counterparts for programs that need high floating point operations per second (FLOPS) rates to process large chunks of memory—such as digital signal processing operations.

We believe there are software solutions to counteract the memory bandwidth bottlenecks and low peak FLOPS rates on GPPs. The stream programming model is most suitable for applications that include three main properties: computational intensity, data locality, and data parallelism. GPPs can handle heavy computational workloads. Data that is fetched from memory once or twice to feed kernel operations during runtime and never used again favours the stream processing model. Conversely, traditional GPP caching algorithms that emphasize temporal and spatial locality are not as beneficial in the stream model.

With work being done on load balancing and generating threads intelligently based on GPP hardware features—such as core count and cache sizes [Kudlur and Mahlke, 2008; Gummaraju et al., 2008; wei Liao et al., 2006]—we look at a technique to improve the problems with stream data locality and stream data parallelism on GPPs. We argue generating vectorized code for streaming operations is a potential solution. Due to the nature of reading, writing and executing streams, we believe that building a compiler that exploits the vector hardware on today's GPPs allows us to make a case for further research into mapping streams onto these architectures. In this chapter:

- We outline both existing and modified vectorization techniques that can be used within the stream framework.
- We show that vectorization is an effective strategy to map stream code to general purpose processors.
- We analyze application and algorithm properties to suggest reasons for different levels of vectorization effectiveness when generating stream code.

The chapter is organized as follows: Section 3.2 describes the Brook streaming language. In Section 3.3, we elaborate on the modifications required to the stream compiler and the stream runtime to integrate our vectorization techniques. The strategies and techniques required to generate vector code are outlined in Section 3.4. Section 3.5 presents and offers analysis on our experimental results including speedup and cache miss data over several backends. Work related to this chapter is mentioned in Section 3.6. Section 3.7 summarizes the conclusions presented in this chapter.

3.2 Brook

The work presented in this chapter extends a streaming language backend for general purpose processors. Specifically, it extends streaming backends that use a compilation process that operates similarly to Brook. Brook is a streaming language (see Section 2.4) designed to incorporate the streaming model of data parallel processing and computational intensity as an extension to ANSI C. This allows the programmer to have a comfortable and familiar environment to build streaming applications. Brook compiles a stream program using a process shown in Figure 3.1. For work described in this chapter, the stream runtime is the CPU backend, and the compiler targets this instead of other runtimes (such as the GPU). Any additions or extensions to Brook, often require modifying both the runtime and the stream code generator. We need to modify both to implement autovectorization techniques.

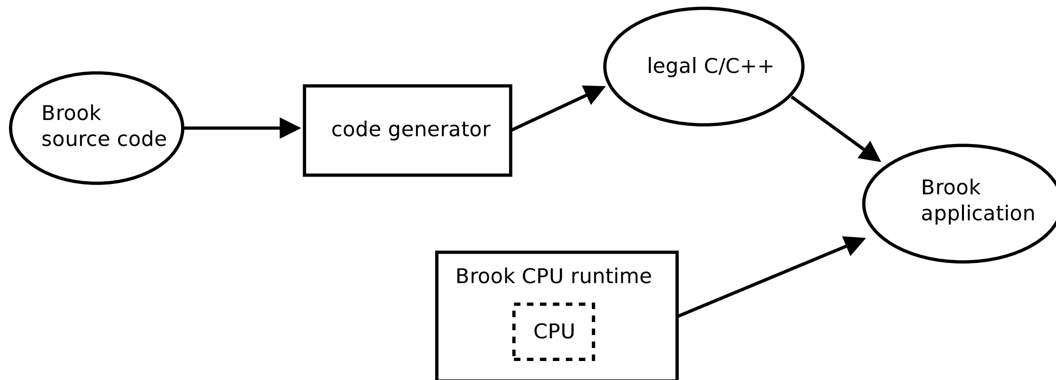


Figure 3.1: Compiling a Streaming Program in Brook using the CPU backend.

3.3 Modifications to Brook

Due to the cooperation of the generated code and the runtime when executing a stream program, changes need to be reflected in the stream compiler. In our case, we build upon and modify Brook to implement classical and our own vectorization techniques. Generating this vectorized code requires us to modify the runtime to support the difference in the way a stream was being represented in memory. Likewise, augmenting the runtime to support certain hardware features requires code to be generated differently. While we talk about the specific techniques of generating vector code later on, we modify both the code generator and the runtime. That work is discussed in Section 3.4. The implementation of our techniques uses Brook v0.5 Beta1 [Buck et al., 2004] as the base platform and our code targets the x86 architecture.

Autovectorization

Autovectorization is a technique that automatically converts scalar code into vector code, as described in Section 2.1.5. The autovectorizer we implement with Brook generates code that uses the most common mathematical operations. However, more advanced instructions become equally useful to us for specific problems. The majority of instructions we use are the ones that process single-precision floating-point values (SP FP). For example, with Intel's C/C++ intrinsics (detailed in Section 2.1.4), these values are stored in 128-bit `__m128` vector registers. The registers hold four 32-bit SP FP values. Data is loaded into a vector register by the `_mm_load_ps(float*)` intrinsic which takes a pointer to the start of a block of memory which contains four consecutive SP FP values. Generating vector intrinsics requires us to make modifications to a stream compiler.

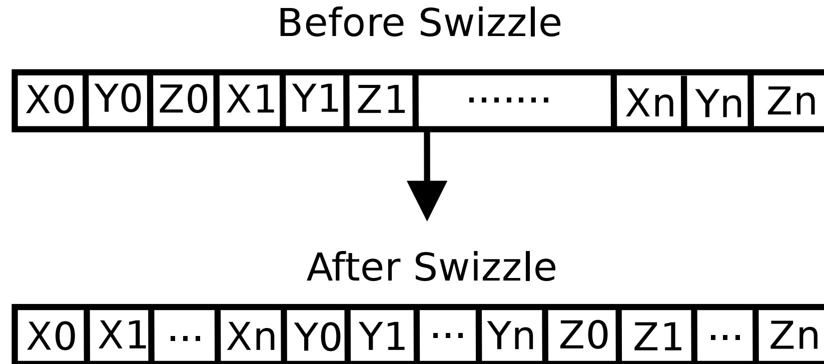


Figure 3.2: Swizzling data from an Array of Structures to a Structure of Arrays.

3.3.1 Modifying the Code Generator

The code generator creates the "for-loop function" that iterates through the size of the stream and parses the internal kernel code. The runtime requires a function to be created that calls the "for-loop function" and kernel code. We modify the stream loop and the data representation as both are necessary modifications.

We change the loop to index every n elements, where n is the SIMD size, or the number of values which are packed into the vector register. The value n changes depending on the data type used. As the intrinsics require a pointer to the start of the block of values, we are required to change the argument list since we now iterate over n elements. If the streaming language allows structures to be defined, like Brook does, we must modify the argument list further by expanding the arguments to as many separate streams as necessary representing each data field of the structure.

3.3.2 Modifying the Runtime

The runtime manages several things during execution that alleviates certain burdens from generating verbose code. However, changing the code generator requires us to make changes to the runtime and vice-versa due to their co-dependence. Multi-dimensional data representation needs to be implemented by both. For example, a stream of structures (containing three floats named x, y and z) will essentially be an array of structures (AoS). To make use of vectorization techniques, data needs to be *swizzled* so it can be represented as a structure of arrays (SoA). Swizzling is a technique that shuffles horizontal data in memory to be represented vertically. In other words, data is realigned so we can perform vector operations on `x[0-3]`, `y[0-3]`, and `z[0-3]` all at the same time, rather than its original representation of `x[0]`, `y[0]`, `z[0]`, `x[1]`, `y[1]`, `z[1]` ... and so on. This is visually represented in Figure 3.2.

Listing 3.1: Stream source code, general operations.

```
kernel void k(float inStream1<>, float inStream2<>,
                        out float outStream<>){
  outStream += inStream1 * inStream2;
}
```

Since stream languages require that all streams be initialized before being used by a kernel, we swizzle the data at this point. To swizzle the data, we also use vector intrinsics. After the kernel executes and we write the stream back to memory, the data is swizzled back to AoS format. This allows the behaviour of structures to never change for the programmer. Intel's optimizations manual [Intel Corp., 2011] details an algorithm for swizzling multi-dimensional data.

3.4 Generating Vector Code

When translating the Brook kernel, the block statement containing the original code is parsed. To generate our vectorized code, we use a C++ parser and code generator. It also has the ability to exit out of our vector mode and generate the standard Brook generated code if our compiler cannot produce equivalent vectorizable code. The remainder of this section deals with some of the general and more interesting techniques used to automatically generate vectorized code within a stream compiler.

3.4.1 General operations

Generating the intrinsics for general computation, assignments, and declarations are fairly straightforward. When the grammar finds an expression like `expr1 '+' expr2`, it generates the equivalent intrinsics instruction (`_mm_add_ps(expr1, expr2)`).

Assignment expressions such as `unary_expr = expr` use the previously defined `__m128` variables as much as possible. Since output streams are the only modifiable streams in a kernel, we only write variables back to memory at the very end of each kernel iteration. With results not needing to be used in subsequent iterations we store data using `_mm_stream_ps()` which does not pollute the cache. General operations are produced from the source seen in Listing 3.1 and the generated code is seen in Listing 3.2.

Listing 3.2: Generated vector code, general operations.

```
1  static void __k_cpu_inner(const float* inStream1,
2                            const float* inStream2,
3                            float* outStream){
4    __m128 __v0, __v1, __v2;
5    __v0 = _mm_load_ps(inStream1);
6    __v1 = _mm_load_ps(inStream2);
7    __m128 __t0 = _mm_mul_ps(__v0, __v1);
8    __m128 __t1 = _mm_add_ps(__v2, __t0);
9    __v2 = __t1;
10   _mm_stream_ps(outStream, __v2);
11 }
```

Listing 3.3: Stream source code, array lookups.

```
1  kernel void k(float streamIndex<>, float array[],
2                out float outStream<>){
3    outStream = array[streamIndex];
4  }
```

3.4.2 Arrays

Lookups

The challenge with vectorized array lookups occurs when indexing on a value that is not known during compilation time. If the index is known, then we use constant expansion to load the same value into all four slots of the vector register. However, real world problems need to index on more than constants and we see this in programs using a stream of values as the index for an array. Even knowing the addresses of the four elements to load into a vector register will not help as there is no instruction to load values from four different memory locations. The compiler generates scalar code to accomplish the task.

When generating code for an array lookup, the index vector currently holds the four index values. In Brook, these can be, and often are, floating point values. If there is some computation needed to calculate the index, this can all be done in vector mode before the four values are requested from memory. Iterating through with a for-loop, the four array values are stored into a second array. Upon loop completion, these values are then loaded into vector format so they can be used with other vectorized operations. Array lookup code is produced from the source seen in Listing 3.3, and the generated code is seen in Listing 3.4.

Listing 3.4: Generated vector code, array lookups.

```
1 static void  __k_cpu_inner(const float* streamIndex,
2                  float* array, float* outStream){
3   // ...
4   float TVF0[4];
5   for(i=0;i<4;i++)
6     TVF0[i] = array[(int)streamIndex[i]];
7   __v2 = __t0 = _mm_load_ps(TVF0);
8   // ...
9 }
```

Float2 Indexing

Augmenting floating point indexing support, Brook allows a float2 structure to act as an index for 2-dimensional arrays. Float2 structures contain float x and float y components. These act as the first and second index values when accessing an array. Where `i` is of type float2, `array[i]` is essentially equivalent to `array[i.x][i.y]`. Our compiler treats float2s as two separate float variables and does the appropriate pointer math required to access the data. We then use the same strategy as described earlier for loading array values.

3.4.3 Conditional Assignments

Vectorizing `if` and `if/else` statements is difficult. When a conditional statement is evaluated in vectorized code, the "true" elements need to perform different actions to those of the evaluated "false" elements. This technique is not suited for easy vectorization. Conditional assignments suffer from a similar problem. Conditional assignments reduce to a single true expression and a single false expression. We only need to assign those single values to the appropriate elements. Conversely, an entire block of code may follow when vectorizing `if` statements.

To enable conditional assignments in kernels, we augment a well known bit-masking technique as shown in Figure 3.3 for inclusion in a streaming compiler. Defined roughly as `conditional ? true_expr : false_expr`, the conditional evaluation will produce a vector filled with four values, each having a value of `0xFFFFFFFF` or `0x0`. We bitwise `and` this with `true_expr` and produce a temporary vector. The `false_expr` is then used with the `andnot` value of the `conditional`, producing a second vector. The two vectors are combined with the `or` operation giving us a result vector filled with the correct assignments for each element.

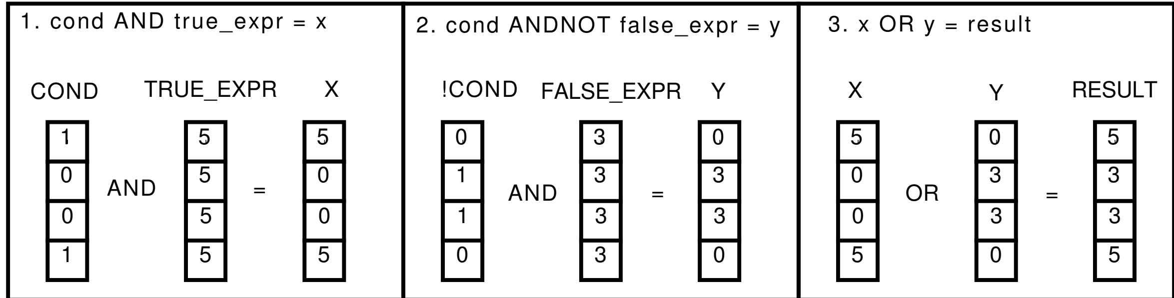


Figure 3.3: Conditional assignments in vector mode.

Listing 3.5: Stream source code, for-loops

```
kernel void k(float inStream<>, float outStream<>){
  // ...
  for(i = 0.0; i < 5.0; i += 1.0)
      outStream += inStream;
  // ...
}
```

3.4.4 For loops inside kernels

At first glance, vectorizing loops found inside kernels seemingly run into the same problems that face normal loop vectorization. However, since the loop inside a kernel is really focusing on at most one element of the stream, we can assume a few things. To clarify, we do not vectorize inner loops[1]. We generate scalar code to process them with the vectorization code that would likely surround the inner loop bodies to process the streams.

Our code generator will produce code for loops that are countable[2]. While it is conceivable that the loop's conditional statement could use a stream or some other dynamic element, we feel accounting for this would occur in very extreme cases and would generally break the essential concepts about streaming applying the same kernel on every element in a stream. Given this concession, we generate loop code using a set of `goto` statements as we can see in Listing 3.5 and Listing 3.6.

3.4.5 Reduce Kernels

A reduction kernel in the stream model is simply a function that performs a mathematical operation over every element in a stream and returns a result or a set of results. Due to this, reduction kernels have very restrictive rules. They only support operations

[1]Inner loops are loops that are inside kernels.

[2]Countable loops have lengths that are known at compilation time.

Listing 3.6: Generated vector code, for-loops

```
1  static void __k_cpu_inner(const float* inStream,
2                              const float* outStream){
3    // ...
4    void* __for_0_target;
5  __FOR_0_TOP:
6    __t0 = _mm_cmplt_ps(__v2, _mm_set1_ps(5.000000f));
7    __for_0_target = (_mm_cvttss_si32(__t0)) ?
8                       &&__FOR_0_BLOCK : &&__FOR_0_END;
9    goto *__for_0_target;
10 __FOR_0_POST:
11   __v2 = __t0 = _mm_add_ps(__v2, _mm_set1_ps(1.000000f));
12   goto __FOR_0_TOP;
13 __FOR_0_BLOCK:
14   __v1 = __t1 = _mm_add_ps(__v1, __v0);
15   goto __FOR_0_POST;
16 __FOR_0_END:
17   // ...
18 }
```

that are associative and commutative. In Brook, two types of reduction kernels are supported. The first reduces a stream of input to an output stream. The size of the output stream must divide evenly into the size of the input stream. Providing this is the case, the stream will reduce into its smaller counterpart by combining every n elements, where n is $inputsize/outputsize$. For example, an input stream of size 50 will reduce to a 5 element stream by combining every 10 elements into the 5 elements.

The second type of reduction kernel takes an entire input stream and reduces it to a single scalar value. Our compiler only attempts to vectorize the reduction of this type of reduction. A variable is used to keep a running total through each “4-element” iteration of the loop. Upon completing the loop, a simple summation of the four elements contained in the vector is returned from the “for-loop function” as a single value.

3.4.6 Selective Vectorization

Selective vectorization is considered when performance penalties may be incurred when vector code is used in place of scalar code. From examining the properties of streaming applications we notice that there will be times when vectorization is not the best option. We make a conscious decision to not always vectorize things like array accesses, reduction kernels and loops that occur inside kernels. Our compiler can be passed a flag to generate scalar code in place of vector code for these cases.

In programs where a value from an array is assigned to an output stream, the entire kernel could be left in scalar mode. This will depend on application properties. Certain reduction kernels can also be left in scalar mode, depending on their complexity. The size of the input and output streams of the reduction kernel will determine whether or not it is worthwhile to vectorize. We talk more about our reasoning for selective vectorization in detail from our results in Section 3.5.

3.5 Results

Using original Brook source code and running it through our vectorizing compiler, our experiments test the techniques we described in Section 3.4. The vectorized code generated is compiled with our modified runtime. A streamlined scalar and single-threaded CPU version is used as the baseline. The original Brook CPU backend is an unsuitable baseline as it contains large amounts of suboptimal code which heavily skews performance results. The original Brook CPU backend was included mainly for debugging purposes and is provided for interest only. Recognizing this, we felt it was important to provide a more realistic comparison. The baseline we use is essentially a scalar version of the code we generate for the vector version. Using this as the baseline highlights the differences between the scalar and vector implementations. A set of multi-core backend results is also provided. The OpenMP pragmas are generated at compile time. This backend multi-threads the scalar baseline on a quad core machine—again, not multi-threading the original Brook CPU code.

We take an in-depth look at benchmarks `bitonicsort`, a parallel sorting algorithm; `bsearch`, a parallel binary search; `conjgrad`, an iterative sparse matrix solver; `imgproc`, an application to apply a 3x3 mask across an image; `throughput`, a benchmark calculating memory bus bandwidth; `spMatrix`, a sparse matrix solver; `jpg`, for compressing images; and `dct`, a discrete cosine transform. The benchmark suite is a combination of applications included with Brook and others ported from StreamIt. The experiment was carried out on a Intel Xeon quad-core machine at 2.16GHz with 12 GB of RAM running Ubuntu. All backends were compiled by icc with `-O3`. Thirty iterations were executed for each benchmark and the median times were used for comparison. The variance of the results in both the optimized baseline and vectorized backend are quite small. With `bitonicsort` for example, the variance in running time is less than 1%. The largest variance occurs with `jpg`, and this is only 3% of total runtime.

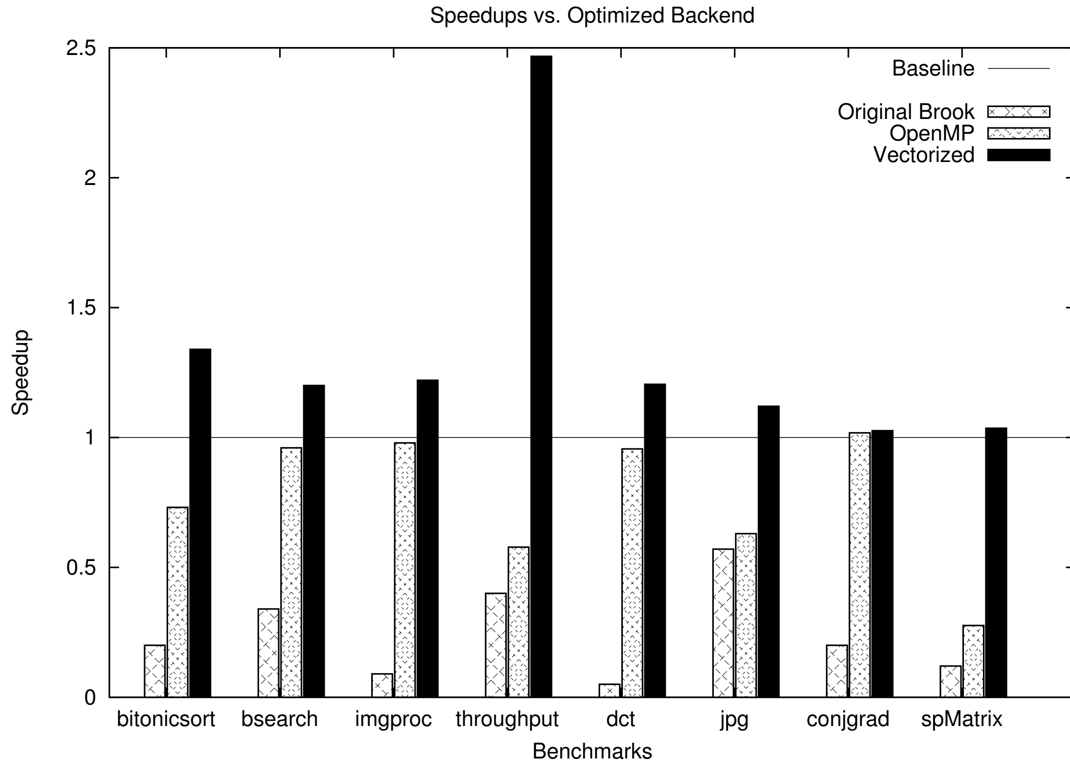


Figure 3.4: Performance of the original Brook implementation, OpenMP, and our vectorized backend against the scalar baseline.

3.5.1 Analyzing Performance

For our comparison, we generate streamlined scalar CPU code for our baseline. This is compared to a baseline version with OpenMP and our vectorized backend. We provide numbers for the original Brook backend, but do not discuss these results due to reasons mentioned earlier. The eight benchmarks we chose exhibit the vector code generating techniques discussed in Section 3.4. Results of these benchmarks are shown in Figure 3.4. The vectorized version of stream code which our system generates performs better than an optimized single element backend and the optimized OpenMP backend running on four CPU cores. The range of speedups is quite high. We see a minimal speedup in `conjgrad` of just a few percent and a maximum speedup of 2.46x with `throughput`. Analyzing the properties of the benchmarks allows us to suggest reasons for the varying levels of effectiveness vectorization has on stream code. We also provide Level 1 and Level 2 cache miss data on the median runs of the benchmarks, found in Table 3.1 and Table 3.2, respectively. These results are generated with PapiEx [Mucci, 2009].

Table 3.1: Table of Level 1 cache misses incurred during kernel execution.

L1 Misses			
Benchmark	**Baseline**	**w/ OpenMP**	**Vectorized**
bsort	8,819,090	8,840,260	6,000,600
bsearch	2,495,320	3,498,000	3,498,000
conjgrad	2,543,280	2,550,040	2,132,620
dct	49,981	62,700	93,354
jpg	100,057	290,654	180,665
imgproc	4,485	10,228	4,291
spMatrix	4,937	23,802	5,050
throughput	13,044,800	7,602,920	1,130,710

On Random Accesses

Data locality is an important property of the streaming model. As such, when random data is requested in stream kernels, performance takes a hit. This behaviour is highlighted in `conjgrad`, `bsearch`, and `jpg`. All these benchmarks feature a large dependency on non-sequential array accesses. With `bsearch`, performance loss is stymied because it is an algorithm suitable for parallelization. Since four values are running through the algorithm at the same time in vector mode, if an element is not found, it will still be able to use that information in the next iteration. We observe little performance lost to the overhead of the `and`/`andnot`/`or` algorithm displayed in Section 3.4.3. The number of cache misses in the vectorized version with randomized data tends to be quite high in comparison.

Benchmark `jpg` performs slower than its `dct` counterpart. This is interesting since the discrete cosine transform is used in the jpeg algorithm. The decrease in performance is a result of quantization and zig-zag bit encoding after the DCT is applied. Array accesses dominate these functions, however, our system also does selective vectorization. In this case, the quantization/zig-zag kernel executing only scalar code resulted in a 4% increase in performance over the version that was completely vectorized. The overall speedup using selective vectorization is 1.12x.

Similar use of arrays occur in `conjgrad` and `spMatrix` as several small kernels execute sequentially to iteratively solve sparse matrices. Some of the small kernels are automatically vectorized by icc when the `-O3` flag is set. This explains why our generated intrinsics version is only slightly faster than the scalar—turned vector by icc—version. In other benchmarks, kernels are too complex to be exploited by the automatic vectorization found in both icc and gcc. We still achieve a small speedup when we generate the vector code for these two benchmarks ourselves. The Intel C

Table 3.2: Table of Level 2 cache misses incurred during kernel execution.

L2 Misses			
Benchmark	**Baseline**	**w/ OpenMP**	**Vectorized**
bsort	709,605	5,166,490	3,101,300
bsearch	44	70	200,239
conjgrad	631	500	753,862
dct	22,460	25,778	63,193
jpg	56,960	345,790	138,062
imgproc	502	2,518	616
spMatrix	637	6,056	683
throughput	1,263,530	3,444,500	694,378

Compiler will do some autovectorization optimizations if possible in the benchmark code. The baseline numbers, then, provide a fair comparison between what a standard compiler can do with aggressive optimization using its built-in vectorization strategies and the code *our* autovectorizer can generate.

On Computational Intensity

Heavy workloads with large data sets test all three streaming properties of computational intensity, data locality, and data parallelism. The benchmarks that include kernels which depend on large amounts of computation show us better results. This occurs because as more cycles are used to complete the kernel, the overhead of swizzling data has less impact on overall runtime. Since we load 16 bytes per parameter in vector mode and kernels require at minimum two parameters (one input and one output stream), we have already (at best) requested a full cache-line of data. Subsequent iteration values are more likely to be in cache when the kernel completes. Our L1 and L2 cache figures suggest this to be the case.

Both `imgproc` and `dct` are similar in this way. They both contain a computationally expensive kernel with the former applying a mask across an image and later being a mathematically intensive function with internal summation loops. With `imgproc`, there are array accesses, but they are more predictable and also index with `float2` values. We use the technique in Section 3.4.2 and all of the pointer math is done in vector mode.

`Dct` also contains array accesses. These accesses reference a relatively small table of values necessary to perform the DCT algorithm. Floating point mod operations and other long latency instructions are used around these table lookups and the benefits are seen in the CPU pipeline. In `throughput`, the benchmark reads in 4 MB of 4-

dimensional data, performs 2 billion operations and writes the data back to memory. This gives us our best speedup at 2.46x. Even our best performing benchmark has a quite modest speedup considering that the SIMD size is 4—thus, expecting figures closer to a 4x speedup. With `throughput`, this is the only benchmark that is clearly dominated by data parallel processing and its performance is affected by swizzling. In this case, swizzling data both before and after kernel execution accounts for nearly 65% of `throughput`'s total execution time.

Some research has gone into vectorizing interleaved data [Nuzman et al., 2006] and we may be able to apply some of those techniques to our compiler. Streaming languages often have structures to represent multi-dimensional data and finding ways to diminish the penalty, or perhaps eliminating some of the need, of swizzling data. We do see in Figures 3.1 and 3.2 that re-swizzling the data ends up causing fewer cache misses in computationally intensive benchmarks like `throughput`. It is a matter of finding the right balance between kernel execution time and minimizing the amount of data cache misses.

On Other Mapping Strategies

While we have spent most of our discussion on comparing our vectorized version to the optimized scalar baseline, a few notes need to be said about the multi-core backend. This backend takes our baseline implementation and adds multi-threading support provided by OpenMP. When executing different parts of the stream kernel on different cores, it may be expected that multi-threading would achieve some speedup. We found this to be very unlikely in experiments. We do use the existing threading implementation that Brook uses, so we concede that a more efficient implementation is possible. Using the scalar CPU backend for Brook is very slow as it was originally designed for testing purposes. Applying multi-core execution to this original backend showed healthy speedups and argued that mapping Brook stream code using threads was an excellent solution. We found when multi-threading an optimized scalar baseline does not actually achieve comparable speedups.

Work with MCUDA [Stratton et al., 2008], which retargets GPU code by transforming kernel functions into CPU threaded loops, shows that using four threads fails to outperform serially executed code. When adding OpenMP directives to our scalar baseline, results show a significant overhead of using threads. However, it is important to reiterate that performance of the multi-threaded version also suffers from benchmarks that are not dominated by data parallel processing. Our data suggests that vectorization is a useful alternative to parallelize stream code (even when the pro-

grams are not easily parallelizable) and could be investigated further and possibly in conjunction with multi-core support.

One of the reasons of exploring multi-core and vectorization at the same time would be to improve code areas which are not easily vectorizable. Our compiler generates vector code for all the kernels present in the benchmarks provided. Although, constructs like irregular array accesses and if statements are difficult to vectorize effectively. Scalar code is generated for those lines, leaving the remaining code inside the kernel in vector mode. This is why we see speedups (although small) in benchmarks which are not completely parallelizable. We must generate vector code directly with intrinsics because autovectorizers in gcc and icc are unable to recognize that most stream kernels are even possible to vectorize.

3.6 Related Work

The last few years has seen research in streaming languages targeting general purpose processors. One streaming language that has been targeting GPP support is StreamIt [Thies et al., 2002]. StreamIt was designed to bridge the gap between performance and programmability. The semantics of the language are designed for hierarchical stream structures like pipelines, split-joins, and the feedback loop. Using these constructs paves the way for better analysis and optimization when StreamIt source is translated to a general language—especially when dividing workloads across several cores. Other stream implementations also have similar GPP mapping strategies. Streamware [Gummaraju et al., 2008] suggests splitting workloads across hardware resources while considering processor and cache configurations. The runtime takes cache hierarchy into consideration and ensures that stream kernels are executed on the core closest to where the data is currently stored. Our work differs to these implementations as we increase GPP performance of streaming languages using vector registers rather than exclusively utilizing multiple cores.

While little work has been published directly relating with the use of vectorization in streaming languages, examining the internals of Brook and presentation notes regarding commercial stream implementation RapidMind[3] [Monteyne, 2008] suggests that there is some generation of SIMD instructions. However, there is a significant body of literature that presents strategies, limitations, and problems when compiling for multimedia and vector processors from general high-level language code such as FORTRAN and C++. This work offers some direction despite focusing on classical

[3]RapidMind was merged with Intel Ct to form Intel Array Building Blocks.

vectorization problems such as: loop analysis, unaligned memory references, and data dependency. These problems are generally not present given the programming constraints of the stream model. As the stream model is essentially a for-loop, work on vectorizing loops has been extensively examined. In work presented by Krall and Lelait [2000], they generate vector code from a combination of their own methods and "classic" vectorization algorithms developed by Allen and Kennedy [1987]. In the system by Krall and Lelait, vector code is generated after unrolling the loop as it decreases the need for data dependency analysis. Loops are unrolled in their system dependent on the data type that is being used and what is supported by the UltraSPARC Visual Instruction Set.

Ren et al. [2003] provided insight on the widening gap between traditional vectorization and generating vector code for newer multimedia ISAs. By studying code in the Berkeley Multimedia Workload (BMW), the authors were able to identify limitations of high-level languages that inhibit the exploitation of classical vectorization techniques. However, they also concede that multimedia extension sets contain specific instruction sets that limit functionality for general programming use. Larsen et al. [2005] consider another approach and exploit parallelism by utilizing both vector and scalar constructs with selective vectorization. This reduces the semantic limitations of high-level languages for vector code generation and makes use of the vector instructions available. We expand on this idea with vectorizing on streaming languages by allowing certain parts of the kernel (and even entire kernels) to remain in scalar form. This approach could cause the portability of vectorization to decrease even further than current methods.

Multi-platform vectorization was investigated by Nuzman and Henderson [2006]. They present work on gcc's autovectorizer and the implications in developing it. While a decent amount of the work deals with memory alignment issues across different architectures, the discussion on using basic and general instructions was applicable to us due to our own experience in generating code with an up-to-date and even future ISAs (Intel AVX, SSE4) and lower generational ones (MMX, SSE). The paper also has some material on reduction idioms which are similar to the reduction kernels found in streaming languages.

3.7 Conclusion

In this chapter, we investigated the use of GPPs as a target architecture for compiling streaming languages through vectorization. We augment traditional techniques with

some of our own to generate vector intrinsics within the framework of the streaming model. We showed that due to some of the restrictions inherited by the stream programming model, our compiler produces code that takes advantage of often dormant SIMD/vector processing power found on today's general purpose processors.

Using the vector registers allows GPPs to better manage data locality and accounts for significant speedups in stream applications. Using vectorization to map stream code to GPPs under our framework runs faster compared to other mapping strategies. While multi-threaded implementations are useful and applicable in many stream programs, exploiting vectorization should not be as overlooked as a strategy for mapping stream code to GPPs. We also found that performance problems which plague vectorization, such as swizzling memory, can be minimized depending on the characteristics of the application and using a combination of both scalar and vector code can also be useful.

We see vectorization as a driving force behind increasing GPP performance of stream programs. With little research in streams and vectorization, we believe there is an even greater performance potential as additional optimization ideas are investigated. All of our vector code generation is done transparently to the programmer. In other autovectorizing implementations, loop analysis and/or hinting code is required to generate code effectively. Analysis often finds that loops are not primed for vectorization. The stream model is a platform ripe for vectorization as it alleviates these pressures. The model also outlines rules and restrictions needed to effectively vectorize scalar code.

The work presented in this chapter showed that implementing an autovectorizing mechanism resulted in good speedups when the algorithms used a stream of data that did not have to be rearranged in memory before processing them. It was important to know that making transformations from scalar to vector representations could give overall performance improvements. It encouraged further thought on the subject. But, could we be using the instruction-set extensions effectively? We already could generate the vector instructions, but we needed to improve *those* generated instructions. One further idea was to integrate improved instruction selection techniques. Another idea was to integrate loop fusion so fewer memory operations would be required. Ultimately, we thought that focusing on more specific algorithms would give greater insight on how to best optimize in a more general case.

An interesting problem arose when discussing our streaming research with Intel. A soon-to-be-released instruction-set extension included several long latency vector instructions to support an algorithm that operated much like a streaming program. The properties of the algorithm would lend itself well to being optimized as a stream pro-

gram, but could the work we did be applied to this problem? Ultimately, we felt there was a different and better approach to optimize this code. With the (then) upcoming release of a new microarchitecture from Intel specifically designed for encryption, we investigated how we could best generate algorithms that used this instruction set.

Chapter 4

AES Encryption in Software and Hardware

Cryptography is a concept dating back to Ancient Egypt and was essentially synonymous with *encryption*. Encryption is the translation of data into a secret code. This is done in an attempt to keep information secure. To read an encrypted file, you must have access to a secret key or password that enables you to decrypt it. Third parties without access to the shared secret key are unable to easily access the data or information that has been encrypted. Unencrypted data is called plain text while encrypted data is referred to as cipher text. There are two main types of encryption: asymmetric encryption (also known as public key encryption) and symmetric encryption.

Asymmetric encryption is a form of encryption where keys come in pairs. What one key encrypts, only the other can decrypt. The public key is usually used as the encryption key and distributed freely, as only the holder of the private key is able to read the data that has been encrypted with the public key. Symmetric encryption is a form of encryption where the same key is used for both encryption and decryption. The key must be kept secret, and is shared by the message sender and recipient. This form of encryption can usually be performed much more quickly and efficiently than asymmetric encryption. In practice, asymmetric encryption is usually used to encrypt an insecure communication channel in order to allow for the exchange of the secret key for the symmetric encryption that will be used for the rest of the communications. This technique effectively combines the strengths of the two forms of encryption and is the basis of the SSL/TLS family of encryption protocols, including HTTPS, that are used everyday for online banking and shopping.

The material presented in this chapter serves as background on the AES encryption mode and includes a survey of research related to optimizing the AES algorithm.

Listing 4.1: Pseudo-AES code using ECB block cipher mode

```
def AES_Encrypt(plaintext, ciphertext, key_sched, key_size, blocks) :

    key = key_sched

    if(key_size == 128) :
        keys = 10
    elif(key_size == 192) :
        keys = 12
    elif(key_size == 256) :
        keys = 14

    for i in range (0, blocks) :      # for each block
        result = xor(plaintext[i], key[0])

        for j in range (1, keys-1): # key rounds
            result = SubBytes(result)
            result = ShiftRows(result)
            result = MixColumns(result)
            result = AddRoundKey(result, key[j])

        result = SubBytes(result)
        result = ShiftRows(result)
        result = AddRoundKey(result, key[keys])

        ciphertext[i] = result
```

Chapters 5 and 6 of this dissertation deal directly with optimizing the various AES modes. Understanding the algorithmic properties of these modes alone is important to understanding why the problem is important and how others have approached it.

An overview of the AES mode is featured in Section 4.1 and roughly explains the encryption process that actually converts `plaintext` into `ciphertext`. AES is used to encode every 16-byte block of data. In order to encode a stream of blocks, a number of block-cipher modes can be used and their properties are described in Section 4.2. Optimization approaches for AES in both software and hardware are featured in Sections 4.3 and 4.4, respectively. The Intel AES New Instructions are described in Sectiongen-aes-ni. A summary and conclusion is offered in Section 4.6.

4.1 AES

AES is one of the most popular algorithms used in symmetric encryption. Originally published as Rijndael [Daemen and Rijmen, 2000], AES was adopted as a standard by the U.S. government in November 2001 [The National Institute of Standards and

Technology (NIST), 2001]. The standard uses a fixed block size of 128-bits (16 bytes) and uses key sizes 128-, 192-, or 256-bits. Further mathematical properties of AES encryption are defined in the standard but are not provided here as this dissertation deals with optimizing AES implementations. The AES algorithm can be described with four major steps:

1. **Key Expansion:** Using the Rijndael key schedule, the k round keys are extracted from the 128-, 192-, or 256-bit key.

2. **Initial Round:** The initial round uses bitwise `xor` to combine the `plaintext` with the first round key; this is known as `AddRoundKey`.

3. **Rounds:** $k - 1$ rounds operate on the result state from the initial round. The state is defined as a 4×4 array of bytes and the following occurs to this 16 byte block of data, also shown in Figure 4.1:

    (a) `SubBytes`—each byte of the state is substituted with another byte, referred to as the Rijndael S-box, using a lookup table.

    (b) `ShiftRows`—each row of the state is cyclically rotated a constant number of steps.

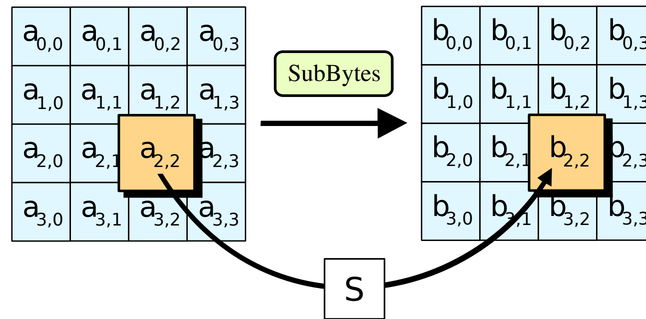
    (c) `MixColumns`—each column of the state is multiplied by a fixed polynomial.

    (d) `AddRoundKey`—Similar to initial round, add the round key to the current state with bitwise `xor`.

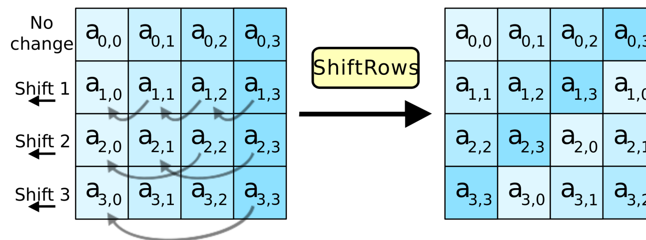
4. **Final round:** The final round is identical to previous rounds, without applying `MixColumns`.

The AES algorithm uses these steps to convert `plaintext` into `ciphertext` . Applying these rounds in reverse will decrypt the `ciphertext` back into plaintext. Practical implementations of AES often require encrypting and decrypting of input sizes greater than 128-bits. Encrypting `plaintext` data that is larger than the defined block size[1] requires the use of a block cipher mode of operation [Ehrsam et al., 1976]. It should be noted that in AES, the key expansion only has to be performed once for any given secret key. In these block ciphers, the round keys are computed during key expansion and are reused for each block.

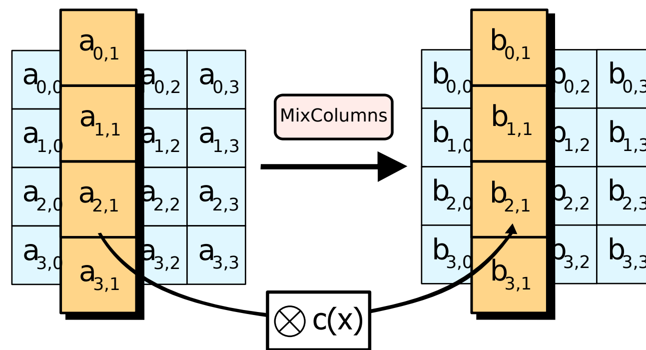
[1] In other words, encrypting a series of blocks—for example, AES encrypts 128-bit (16-byte) blocks. So, 64 bytes will represented as a series of four 16-byte blocks.



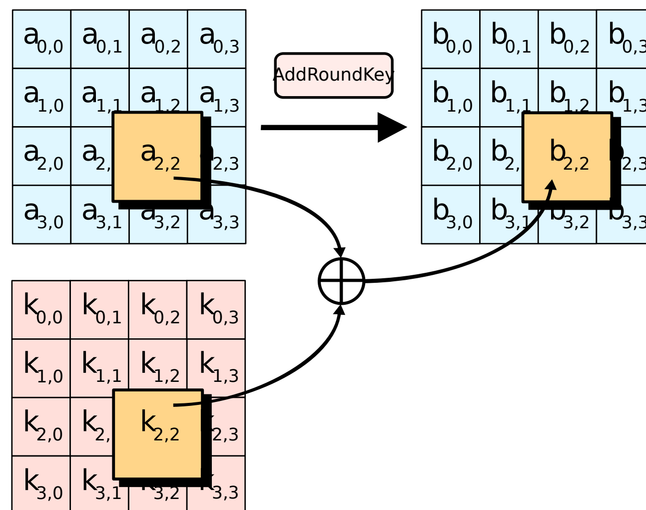
(a) SubBytes



(b) ShiftRows



(c) MixColumns



(d) AddRoundKey

Figure 4.1: AES Round encryption process. Images are public domain.

4.2 Block Cipher Modes

A block cipher is a cipher that operates on fixed-length groups of bits, termed blocks. Block cipher *modes of operation* [Flynn, 1972] are used to encrypt variable length messages with a single key without sacrificing security. A block cipher mode can implement other encryption methods other than AES, but for the needs of this dissertation, we refer to these modes as defined for use with AES. Block ciphers can also be defined for use with authentication. There are nine National Institute of Standards and Technology (NIST) approved block ciphers [Dworkin, 2001, 2005, 2007b,a, 2010]:

- Six **Confidentiality modes**—ECB, CBC, OFB, CFB, CTR, XTS-AES.
- One **Authentication mode**—CMAC.
- Two **Combined confidentiality and authentication modes**—CCM and GCM.

The following modes are important to provide background and related research as they are mentioned throughout this dissertation. The modes which are discussed in Chapters 5 and 6 are explained in the following sections. They are divided into three categories that have unique properties that affect how we generated code: Parallel Modes, Cyclic Modes, and Authentication Modes.

4.2.1 Parallel Modes

Block cipher modes that can encrypt multiple blocks simultaneously have the obvious benefit of being able to be computed in parallel. On out-of-order architectures, the modes featured in this section run faster than the cyclic dependent modes featured in the following section. These modes can benefit from instruction-level optimizations like software pipelining, which is an important part of this dissertation. The simplest parallel block cipher mode to implement is ECB.

Electronic Code Book (ECB)

Electronic code book (ECB) is a confidentiality mode described by Dworkin [2001] as a mode that operates in a similar fashion to using code words in a code book. Using a key, the ECB mode applies an encryption block cipher directly to the `plaintext` and outputs the `ciphertext` . There is no additional code in the block unrelated to the AES rounds and no dependency from one block to the next, making ECB perhaps the simplest and most straight-forward block cipher mode. This is shown in Figure 4.2. While this dissertation does not deal primarily with the security benefits of these

block cipher modes, ECB has one undesirable trait that should be mentioned. Each block is encrypted entirely independently, so a given block of `plaintext` will always result in the same `ciphertext`. This means that repeating patterns in the `plaintext` may result in repeating patterns in the `ciphertext`. Such an outcome makes ECB susceptible to certain types of attack. This makes ECB undesirable for use in many applications that require encryption. However, the security problem with ECB can be overcome by adding an pseudo-random element.

Counter (CTR)

As introduced by Diffie and Hellman [1979], Counter (CTR) is a confidentiality mode that takes the block cipher concept and converts it in a *stream cipher*. A stream cipher combines `plaintext` data with pseudo-random data. This is usually done with a bitwise `xor` operation, as can be seen in the CTR diagram found in Figure 4.3. While the counter can actually be any sequence of bits that will not repeat often, a simple increment is adequate and is generally used. This counter value is often combined with the *initialization vector* (IV) or nonce. The IV/nonce are values used once to pseudo-randomly seed the encryption process. Figure 4.3 details the other parts of the Counter algorithm. The AES rounds are applied to the combined nonce and incremented value. After the last round of encryption, the result is combined with the `plaintext` using a bitwise `xor` and stored to `ciphertext`. Note that blocks in CTR mode can be encrypted in parallel as there is no cyclic dependency from one iteration to the next. [Lipmaa et al., 2001] presented a cryptanalysis on the security and performance of CTR mode. In the cryptanalysis, the authors argue that CTR should be standardized by the NIST because it is a mode that can encrypt blocks in parallel without sacrificing security. They go on to claim that Counter mode's perceived disadvantages are often based on misinformation.

4.2.2 Cyclic Modes

We classify Block cipher modes that use a "feedback" value in the loop as cyclic modes. These modes have a long chain of dependent instructions when using AES, which prevents block parallelization. In our work with cyclic modes, we focus on reducing this dependency chain as much as possible. The most widely used cyclic mode is CBC.

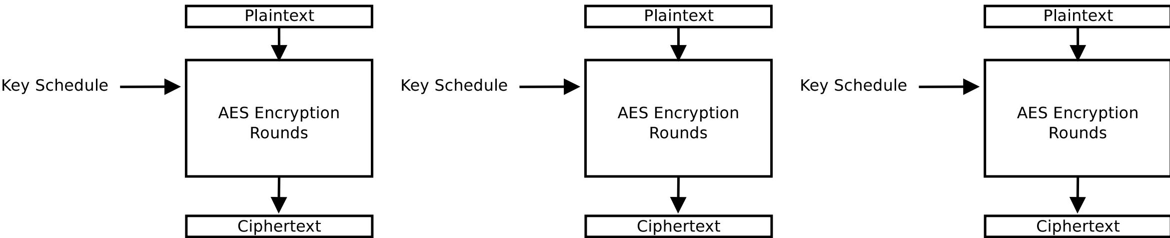


Figure 4.2: Electronic Code Book (ECB) mode encryption.

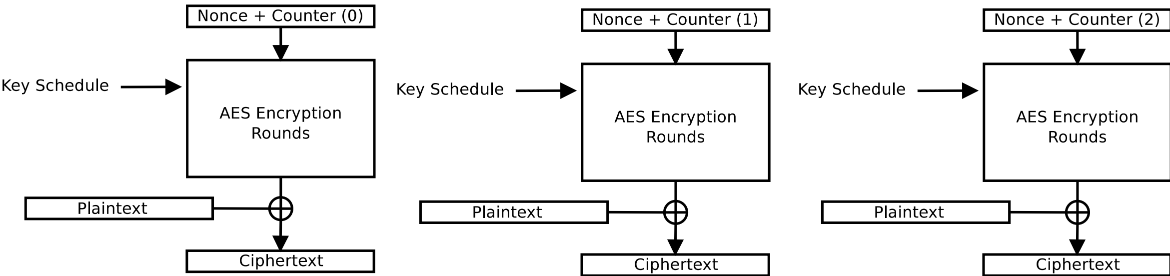


Figure 4.3: Counter (CTR) mode encryption.

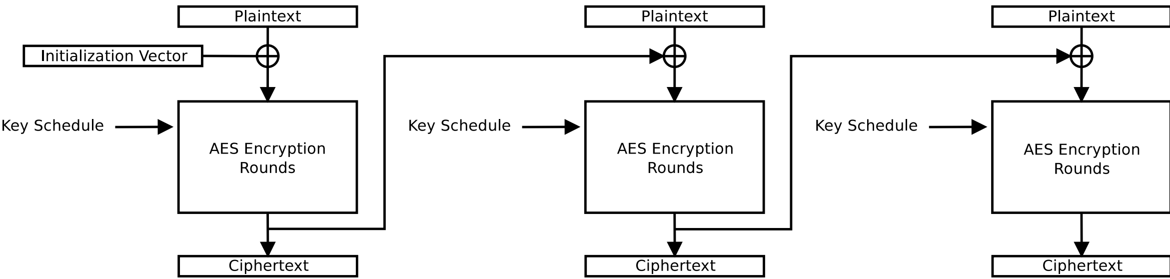


Figure 4.4: Cipher-block Chaining (CBC) mode encryption.

Cipher-Block Chaining (CBC)

Cipher-block chaining (CBC) mode was developed by [Ehrsam et al., 1978] and is a cyclic dependent mode of operation that requires that each block of `plaintext` is combined with the previous block's `ciphertext` using an `xor` operation. The visual representation of CBC encryption is shown in Figure 4.4. The first block uses an initialization vector to make identical messages unique when encrypting a message. All other blocks will be dependent on all preceding blocks that have been encrypted so far. In serial implementations, CBC would not be much slower than CTR code. On processors (like certain embedded systems) that do not have superscalar or out-of-order architectures, the cyclic dependency does not limit performance. However, on architectures that do support those features, other optimizations need to be applied to those modes to improve performance. CBC does have increased security benefits as even small changes in the `plaintext` will cause dramatic differences in the `ciphertext` of all successive blocks. However, it is possible to decrypt `ciphertext` using only two adjacent blocks and this allows CBC decryption to be executed in parallel.

PCBC, CFB, and OFB

Propagating Cipher-Block chaining (PCBC) mode is not included in the list of standardized block ciphers published by the NIST mentioned in the beginning of this section. The origins and exact definition of PCBC are not clearly known [Mitchell, 2005]. A generally accepted definition of the algorithm, such as the one used by [Meyer and Matyas, 1982], calls for PCBC to prevent small errors isolated to adjacent blocks to not affect the all remaining blocks. As shown in Figure 4.5, this is done with a second `xor` of the current block of `plaintext` *after* it has been encrypted using AES. It is not a widely used mode.

Cipher Feedback (CFB) mode is another confidentiality mode that is also cyclic dependent. It is very similar to CBC mode, as can be seen in Figure 4.6. The main algorithmic difference is the `plaintext` is `xor`'d just prior to storing the result to `ciphertext`. The `ciphertext`, like the other cyclic modes is used as feedback to encrypt the next block. Like CBC, it can also decrypt in parallel.

Output Feedback (OFB) mode is designed to be a synchronous stream cipher by only generating key stream blocks. The result of the key stream blocks is combined with the `plaintext` using an `xor` just prior to storing to `ciphertext`. However, the

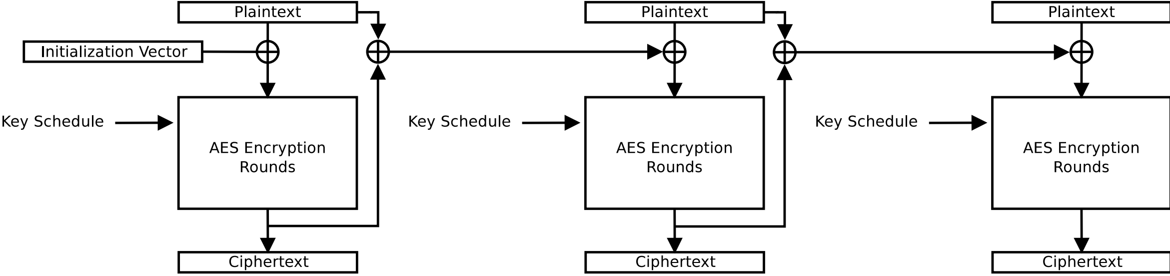


Figure 4.5: Propagating Cipher-block Chaining (PCBC) mode encryption.

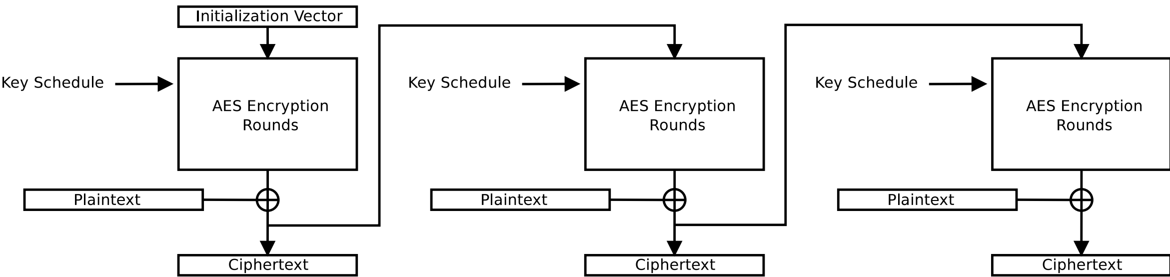


Figure 4.6: Cipher Feedback (CFB) mode encryption.

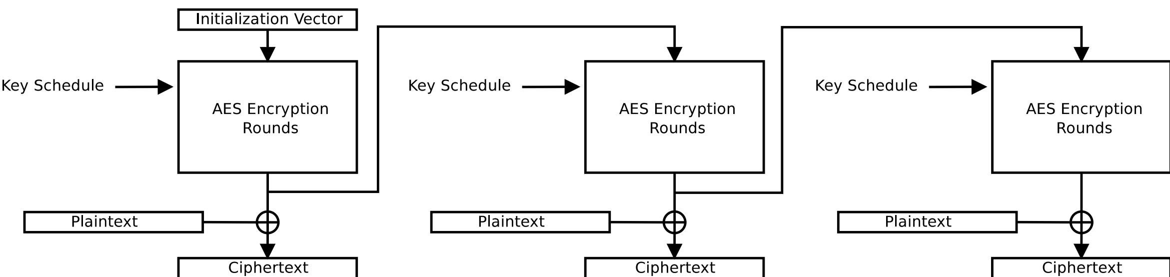


Figure 4.7: Output Feedback (OFB) mode encryption.

`plaintext` never directly influences the encryption stream. The IV is encrypted, and the result from the block cipher is re-encrypted for each block. This can be seen clearly in Figure 4.7. This mode does allow for some error correction as changing a single bit in the `plaintext` will flip the bit in the corresponding location of the `ciphertext`.

4.2.3 Combined Encryption and Authentication

Most of the confidentiality modes have been listed in the sections that deal with parallel and cyclic modes. The NIST, as mentioned at the start of this section, defines block cipher standards for authentication and combined encryption/authentication. Authentication, in terms of computer based cryptology, is verifying that data has been transmitted and received correctly. CMAC (or CBC-MAC) is a NIST approved authentication mode [Dworkin, 2005]. From the name, it could be deduced that CMAC works essentially like CBC mode. Instead of converting blocks into `ciphertext`, CMAC updates an authentication tag after encrypting every block that computes a value that can be used to error check an entire encryption stream. Further modes are defined that combine the encryption and authentication process, allowing one to securely send text and ensure it was received correctly.

CCM (Counter with CBC-MAC) is a NIST defined [Dworkin, 2007b] combined encryption and authentication block cipher mode. As the name suggests, the mode combines Counter (for encryption) and CBC-MAC (for authentication). This mixes a parallel mode with a cyclic mode and suffers from the same problems inherited from CBC mode. However, as AES implementations targeted more parallel architectures, a mode that could encrypt and authenticate in parallel was developed.

Galois/Counter Mode (GCM)

Galois/Counter Mode (GCM) is a widely used combined encryption/authentication mode defined by Dworkin [2007a]. GCM combines CTR encryption mode with the Galois authentication mode. GCM uses CTR to encrypt the actual data, while Galois field multiplication used for authenticating the message can be easily computed in parallel, allowing better performance than when authentication is done using chaining modes (such as CBC). From Figure 4.8, we can see that CTR encryption is performed and the `ciphertext` is fed into an authentication function. This function, known as the GHASH, is defined by McGrew and Viega [2004]. The GHASH function creates a tag at the start of encryption based on the initial counter value. Going into the loop, a new tag is updated after encrypting each block. Upon completion of the loop, GHASH

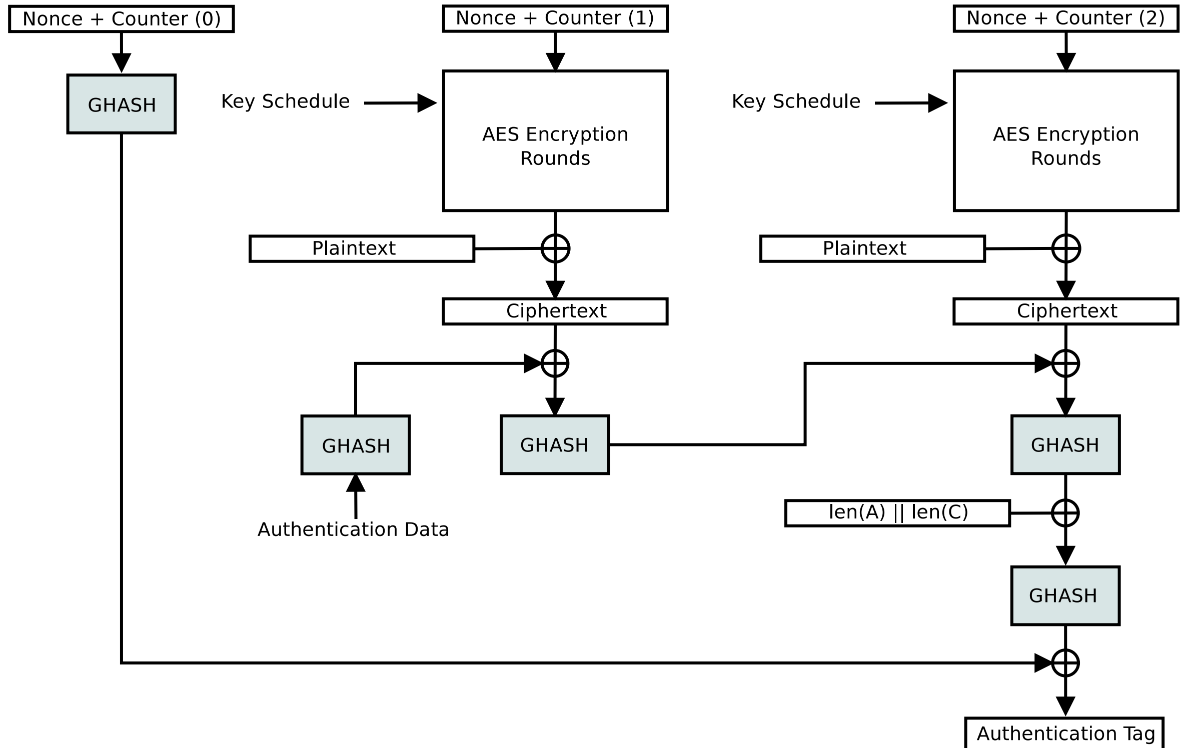


Figure 4.8: Galois/Counter Mode (GCM) encryption and authentication mode.

is applied once more and combined with the tag generated at the start of the loop. However, this shows that there is a dependency between tags within the encryption loop.

4.3 AES Software Acceleration

The crossover of cryptography into computer science and engineering started with the advent of computers and open (or unclassified) research is relatively recent. A good body of text could be devoted to a history lesson on cryptography on modern architectures, dating back to IBM's development of the formerly standardized Data Encryption Standard [(DES), 1977] in the mid-1970s. However, this dissertation deals specifically with using code generation to exploit instruction-level parallelism to optimize the AES algorithm, so this section presents a literary overview on research that deals with improving the performance of encryption code. We start our survey of literature of software optimizations in this section and follow it with hardware optimizations. One of the most common software optimizations for AES is the use of one or more lookup tables [Bartolini et al., 2009].

Lookup Tables

In general, the `SubBytes`, `ShiftRows`, and `MixColumns` round operations can be combined in software to increase the performance of AES encryption. The instructions can be combined and transformed into a series of table lookups. These tables contain the pre-computed values of the round operations that trade computational instructions for memory operations. This trade-off often results in good code on platforms that can handle the extra space required for the lookup tables. In many studies in both this section and in Section 4.4, speedups listed are compared against non-optimized AES[2] and against a well known software-optimized version of AES, published by Gladman [2003, 2009]. Gladman's AES implementations are written in both C and x86 assembly and use table lookups to reduce execution time spent during AES encryption rounds. His implementations are usually used as they are considered to be the fastest AES implementations written in C, which can be compiled for multiple architectures.

Instruction Selection

Work by Bernstein and Schwabe [2008] presents new AES software speed records using several documented techniques. They provide a breakdown of the execution of an AES round as such:

- 16 shift instructions
- 16 mask instructions
- 16 load instructions (for table lookups)
- 4 load instructions (for round keys)
- 16 `xor` instructions

This is a total of 68 instructions with 20 loads and 28 integer operations. In AES-128 (10 rounds), they listed that each block of `plaintext` in the loop body will contain 680 AES round related instructions and a total of 720 instructions (additional `xor`, loads and stores for block cipher modes), which acts as their baseline figure. Their work aims to reduce the total number of cycles at the assembly level. One way they achieve this is through the use of more complicated instructions that could perform several standard instructions in place of one (similar to vectorization). For example, the PowerPC architecture has a "combined shift-and-mask instruction" that does both

[2]Generally using scalar instructions, or others if currently available on the target architecture.

tasks in a single instruction which will reduce the combined 320 shifts and masks to 160 instructions per block. They also look at swapping long latency instructions with cheaper equivalents. Round key re-computation is an example of this, as loading each round key can be more costly than simply expanding the key on-the-fly. Keeping keys in registers[3] is another technique which we implement and discuss in more detail later on in Section 5.2.2.

Hamburg [2009] presents work that replaces parts of the AES algorithm with vector permute instructions found on Intel's SSSE3 and PowerPC Altivec. He claims that using this method makes AES immune to known timing attacks because the techniques avoid data-dependent and key-dependent branches as well as memory references (like those used in lookup tables). On Intel SSSE3, he uses the `pshufb` instruction which implements a 16-way shuffle. On the Altivec, this same method is computed via the `vperm` instruction. This vector permutation is used to implement several types of inversion and effectively replace the `MixColumns` procedure. The resulting matrix calculations using vector permute is faster via interleaving the scalar and vector code required to perform inversion. Hamburg also considered using scalar operations to perform some of the `xors`, but found that passing data between vector and scalar units reduced any improvement. His work finds a 1.5x speedup over implementations on the PowerPC G4e, but slower on x64 by 41% compared to bitsliced implementations.

Bitslicing

Bitslicing, as used here, is also referred to as SWAR (or SIMD within a register). In this context, bitslicing essentially means dividing a single register into bitwise chunks and processing the parts in parallel. For example, a register would hold a value that uses all 128-bits, but is divided into four 32-bit slices so the value can be processed on a 32-bit processor in parallel. Work by both Rebeiro et al. [2006] and Matsui and Nakajima [2007] showed how using bitslices could dramatically improve the AES algorithm through existing architectures.

Rebeiro et al. [2006] present a bitslice implementation of AES and benchmark it on several common superscalar architectures and their SIMD instruction sets–Intel Pentium 4, AMD Athlon 64, and the Intel Core 2. With the Pentium 4 and its 31 stage pipeline, they interleave 128-bit bundles and 64-bit bundles to allow the SIMD and logical (scalar) operations to effectively compute in parallel. Because of the deep pipeline, this strategy results in good encryption per second rates. The same inter-

[3] Keeping keys in registers is also referred to as "key caching", and we use the term "localkeys" in later chapters.

leaving strategy does not work as well on the Athlon 64 and Core 2 processors. The Athlon 64 splits 128-bit instructions into independent 64-bit instructions which saves little clock time. The authors note that the presence of 3 address generation units save the implementation due to an increase of efficiency with memory operations. The Core 2 has three SSE execution units that will process 128-bit data natively. This causes their bitslicing implementation to be less efficient than just processing everything using 128-bit chunks.

Work by Matsui and Nakajima [2007] shows that the Core 2 *can* achieve significant speedups through bitslicing due to its improved SIMD architecture. The authors cite that the problems with bitslicing in prior research that targeted 64-bit processors (like the Core 2) are based on implementations that used 64-bit general registers. They rewrote the implementation for full 128-bit register support and targeted a 64-bit processor. This opens 16 total 128-bit (SIMD) registers for use. Additional registers reduce register pressure by storing keys in registers and using fewer table lookups which results in a 2% smaller code size. The work by Matsui and Nakajima [2007] interleaves this improved SIMD and scalar code and they achieve 9.2 cycles/block (or 0.96 cycles/byte). However, there are two problems with this strategy. First, because the `plaintext` and `ciphertext` must be converted to bitsliced format, the cost of this transformation is not included. Secondly, the `plaintext` can only be 2048-byte chunks due to the transformation.

Memory Re-organization

Bertoni et al. [2003] approach the AES software implementation problem from a different direction (while comparing their work with Gladman's). Recall the 4x4 state matrix in Figure 4.1. The focus in the work by Bertoni et al. [2003] is to transpose the data representation of the state matrix in software to exploit data locality and use fewer instructions. Fewer instructions are used for the computationally expensive `ShiftRows` and `MixColumns` steps of the AES algorithm. They allocate smaller lookup tables for S-box and inverse S-box transformations while depending on the processor to compute all other operations. As `SubBytes` operates on single bytes, this operation changes little. They change `ShiftRows` to now operate on columns and they introduce a completely revised `MixColumns` operation that reduces the number of Galois Field calculations. They also improve `AddRoundKey` step when AES implementations use an on-the-fly key expansion (as opposed to pre-calculating the key schedule). The speedups they achieve are mostly accomplished through the improvement of `MixColumns` operations. These operations consist of bitwise `xor`, masking, shift, conditional `xor` instructions,

but their transposition of the state matrix allows some of these operations from a single column to compute in parallel. Gladman's implementation applies these transformations sequentially.

4.4 AES Hardware Acceleration

The previous section outlined many pieces of work that have proposed various software optimizations to increase AES performance. From this work, it is clear how difficult it is to improve the AES algorithm through software. AES is a computationally expensive algorithm and has multiple steps that need to be performed repeatedly on large blocks of data. None of that description suggests that native AES implementations will run well on general purpose processors. One belief is that new hardware should be designed specifically for encryption processes. Crucial to achieving this, Paar [2002] argues that with the complexity of interdisciplinary cryptography research, implementing future encryption modes should use hardware-software co-design. In fact, as Paar declares, "The efficiency of an implementation algorithm often depends heavily on the details of the target platform, e.g., on the instruction set or the pipeline structure of a processor." This is very true of the AES algorithm, as we see in Chapters 5 and 6 of this dissertation, as we target specific platforms that use cryptographic instructions with defined latencies and throughput values.

As mentioned earlier in this dissertation in Section 2.1.4, multimedia instruction-set extensions are added to existing general purpose processors to compute vector data. Adding hardware support for encryption through an instruction-set extension has been suggested by many as a possible solution. [Bartolini et al., 2009] give an overview of the implications of designing and implementing an ISE that is tailored for cryptographic applications. They discuss that despite support for basic rotate operations required by most encryption processes, Intel, Alpha, and SPARC processors have significant limitations without instructions to handle permutations, substitutions, and extraction.

Implementing S-box and MixColumns on GPPs

According to Bartolini et al. [2009], 32-bit microprocessors and 8-bit micro-controllers were thought to handle AES well, but the steps implemented for each round key (as mentioned in Section 4.1) require several memory re-orderings and are sufficiently complex enough to limit encryption bandwidth. Optimized software implementations use lookup tables, but can be a problem for devices that are memory constrained. Even on general systems that do not have memory constraints, software implementations can be

quite slow. For applications that require maximum speed, special "crypto"-processors have been used but these often can only support specific algorithm parameters—such as key size. With these problems in mind, new special instruction-set extensions that dedicate instructions for the `S-box` and `MixColumns` operations of the AES algorithm are offered as potential solutions.

Bertoni et al. [2006] proposes a generalized instruction-set extension for 32-bit processors, using an ARM processor simulator for proof of concept. The instructions they propose to add come from analyzing the clocktime used by different parts of the AES algorithm during its encryption phase (and not key generation). The AES implementation they analyze uses three lookup tables. They find that most time is spent doing the individual steps of the AES rounds we detailed in Section 4.1 and decide to dedicate some instructions to hardware. They propose instructions `SBox` and `SMix` to perform their respective transformations in two different ways: a set to apply the transformations one byte at a time, and a separate set of instructions for processing 32-bit words. The authors achieve a speedup of 3.45x on one-byte processing and 2.56x on word size processing. They achieve this by adding four additional registers to a CPU that allow access to single bytes from each register in parallel.

Work by Tillich and Groschdl [2006] proposes similar byte and word size processing extension instructions for 32-bit processors (simulated with SPARC). Their equivalent `SBox` and `SMix` instructions differ by calculating one byte of the result while being able to choose the source and destination locations. In Bertoni et al. [2006], the source and destination operands are not needed because the changes are handled by the register file. Their word size processing, noted as `sbox4` and `smix4`, are used to parallelize four-byte operations. The `sbox4` instruction can also perform a byte-wise rotation that will make the particular transformations needed to support key expansion during encryption or decryption (`S-box` or inverse `S-box`). The `smix4` instruction performs the necessary `MixColumns` (or its inverse for decryption) on four-bytes at a time as well. The authors find a speedup of 8.35x using these new instructions with a pre-computed key schedule and unrolling the loop an unspecified number of times.

Implementing S-box and MixColumns on Other Architectures

Implementing `S-box` and `MixColumns` has also been proposed for architectures other than commonly found processors. Tillich and Herbst [2008] expand on their previous work and suggest instructions for `S-box` and `MixColumns` for micro-controllers, such as those used used in wireless sensor networks. These "tiny processor cores" usually need to use less power, process smaller blocks of code, use less RAM, while instruction

latencies are kept low. In order to meet these demands, they propose three instructions to speed up AES encryption on a customized pipelined functional unit. AESENC(1,2) and AESDEC(1,2) are designed to handle the encryption and decryption, respectively, of a single AES round processes with two invocations. A third helper instruction is included for `S-box` and its inverse.

Kosaraju et al. [2006] suggests acceleration techniques for AES on VLSI implementations. They propose two major architecture pieces. The first piece of the architecture is for the data unit. This looks and feels very similar to the work mentioned in the last few paragraphs in relation to the `S-box` and `MixColumns` instructions. The second piece of architecture is used for key scheduling. With `S-box` instructions already implemented in the other piece of hardware, additional hardware is included to perform shifts on the least significant 32 bits of the key, a `ByteSub`, and `xor` with the "round constant". Tillich and Groschdl [2007] have further work on accelerating AES on VLSI, with specific instructions and hardware tailored for computing elliptic curve cryptography (ECC), which is a critical mathematical component when using symmetric ciphers like AES. In this work, the focus is on optimizing `MixColumns` and its inverse function.

Full Hardware Implementations

Other special hardware has been designed to fully support AES encryption. These are platform dependent and are known as *cryptographic accelerators*. The UltraSPARC T1 in 2005 included a cryptographic accelerator [Sun and Lin, 2007]. On the UltraSPARC T1, the accelerator targeted modular arithmetic operations and accelerated public-key cryptography. The UltraSPARC T2 added additional features to accelerate bulk encryption (used by AES), secure hashing, and additional public key algorithms that used ECC. The use of an accelerator on-chip[4] reduces CPU utilization, I/O bandwidth requirements, and limit additional latencies. Another on-chip cryptographic accelerator has been developed by Intel. This instruction-set extension is called AES-NI and, as we focus a lot of work on optimizing AES implementations using this instruction set, it is explained in much greater detail in Section 4.5. Due to these features, on-chip accelerators are particularly effective for encrypting small-data inputs—such as packets. However, off-chip implementations also exist.

Algredo-Badillo et al. [2006] present work on implementing CBC on an FPGA. They specifically design the processor with the cyclic dependency in mind and keeping hardware utilization low. They retain basic algorithm functionality, so basic modules like registers or multiplexers need not be removed. To gain higher throughput, registers

[4]As opposed to off-chip accelerators—such as an FPGA.

were actually added for data multiplexing and storing keys. In combination with that and replacing distributed memory with dual-port memory this strategy reduced the critical paths, resulting in less hardware and faster code.

As mentioned, a software optimization for AES is using lookup tables. An idea presented by McLoone and McCanny [2003] extends this idea to the FPGA. They designed a fully pipelined AES implementation on an FPGA that replaces slow and complex instructions with lookup table values. The values in the lookup table are precomputed and stored on chip and this strategy nets a 1.2x speedup with other lookup table based implementations and 6x overall speedup against other FPGA implementations.

Multimedia extensions such as SSE and MMX on general purpose processors have been covered in Section 2.1.4. A relatively recent instruction set extension called AES-NI debuted in January 2010 on certain Core i5 and Core i7 processors that included the "Westmere" microarchitecture[5].

4.5 Intel AES-NI

The Intel Advanced Encryption Standard New Instructions (AES-NI) includes six instructions [Gueron, 2010] to support AES encryption, decryption and key expansion. These instructions are listed in Table 4.1. AES-NI also includes an instruction performing carry-less multiplication, which would aid in performing finite field arithmetic—used in advanced block cipher encryption (and, as seen later, GCM authentication). These hardware-based primitives provide a security benefit apart from their speed advantage by avoiding table-lookups and hence protecting against software side channel attacks (attempts to discover the secret key by observing and analyzing the flow of information in the computer during the encryption process). The `aesenc` instruction performs a full "round" of encryption in a single instruction. The AES algorithm consists of 10, 12 or 14 "rounds of encryption", which corresponds to using key sizes of 128-, 192-, or 256-bits, respectively. With these new instructions, the code for encrypting a single block consists of a chain of 10, 12, or 14 `aesenc` instructions. The output of each round becomes the input to the next round, so each round must complete before the subsequent one can start. This dependency can be seen in the loop of Listing 4.2, which implements AES Counter mode (see Section 4.2.1).

In the current hardware implementation of AES-NI, the `aesenc` instruction has a latency of six cycles [Akdemir et al., 2010]. Encrypting a 16-byte block using a 128-bit key requires ten rounds or at least 60 cycles to complete. However, a new

[5]And later on "Sandy Bridge" microarchitecture.

Table 4.1: AES-NI Instruction Set as described by [Gueron, 2010].

Instruction	Description
AESENC	Perform one round of AES encryption
AESENCLAST	Perform final round of AES encryption
AESDEC	Perform one round of AES decryption
AESDECLAST	Perform final last round of AES decryption
AESKEYGENASSIST	Assist in generating the AES key schedule for encryption
AESIMC	Assist in converting key schedule for decryption using inverse mix columns
PCLMULQDQ	Carry-less multiply

Listing 4.2: Using AES-NI in AES-128 CTR mode.

```
void AES_CTR_Encrypt(__m128i* plaintext, __m128i* ciphertext,
              __m128i* key, long long ivec, long nonce,  int blocks){
  int i = 0;
  __m128i one = _mm_set_epi32(0,1,0,0);
  __m128i BSWAP = _mm_setr_epi8(7,6,5,4,3,2,1,0,15,14,13,12,11,10,9,8);
  __m128i result;
  __m128i plain;

  __m128i counter_block = _mm_setzero_si128();

  counter_block = _mm_insert_epi64(counter_block, ivec, 1);
  counter_block = _mm_insert_epi32(counter_block, nonce, 1);
  counter_block = _mm_srli_si128(counter_block, 4);
  counter_block = _mm_shuffle_epi8(counter_block, BSWAP);

  for(i = 0; i < blocks; i++){
    counter_block = _mm_add_epi64(counter_block,one);
    result = _mm_shuffle_epi8(counter_block, BSWAP);
    result = _mm_xor_si128(result, key[0]);
    result = _mm_aesenc_si128(result, key[1]);
    result = _mm_aesenc_si128(result, key[2]);
    result = _mm_aesenc_si128(result, key[3]);
    result = _mm_aesenc_si128(result, key[4]);
    result = _mm_aesenc_si128(result, key[5]);
    result = _mm_aesenc_si128(result, key[6]);
    result = _mm_aesenc_si128(result, key[7]);
    result = _mm_aesenc_si128(result, key[8]);
    result = _mm_aesenc_si128(result, key[9]);
    result = _mm_aesenclast_si128(result, key[10]);
    result = _mm_xor_si128(result, plaintext[i]);
    ciphertext[i] = result;
  }
}
```

`aesenc` instruction can be started every two cycles. Therefore, with parallel modes of operation, such as AES CTR, the encryption of multiple blocks can be overlapped.

Intel provides a library of hand-tuned assembly routines which overlap the execution of AES instructions [Gueron, 2010]. The library uses standard techniques for exploiting instruction-level parallelism (ILP), such as manually unrolling loops and manually interleaving the instructions from the unrolled iterations. For example, the AES CTR version unrolls the loop four times and interleaves the resulting instructions. The assembly code is carefully written to make good use of processor resources and achieves excellent performance.

4.6 Conclusion

Optimizing AES has been a well covered and researched topic since its standardization in 2001. It is clear that optimizations in both hardware and software look to improving the logical operations of the rounds which are executed repeatedly. These instructions are relatively simple, but are often costly. We have discussed work that sought to improve this through the use of new instruction-set architectures—by augmenting existing ones or using VLSI/FPGAs. Implementing `S-box` and `MixColumns` as hardware instructions would be incredibly beneficial as AES spends the majority of clock time performing these tasks. Other research looked at more practical solutions through the use of software optimizations. Substituting computations for table lookups and substituting two small instructions for a longer one were commonly explored techniques. Most of the the instruction-set extensions proposed in Section 4.4 were theoretical. Instructions were assigned various latency and throughput values for virtual pipelined functional units for general purpose processors, coprocessors, micro-controllers, VLSI and FPGA architectures. Related to the hardware research, data transformations and bitslicing were explored as ways to increase parallel computation of AES.

Until the release of cryptographic accelerators, speeding up the AES algorithm for practical implementations needed to be done via software. The advent of Intel's AES-NI reduced the need for faster software-optimized AES. However, despite the instructions being fast and easy to implement, little work in general has been published on scheduling these instructions that exploit instruction-level parallelism. In this dissertation, and specifically in Chapters 5 and 6, we explore the usefulness of using software pipelining and interleaving as possible techniques to speed up block cipher modes in AES.

Chapter 5

A Program Generator for Hardware Accelerated AES CTR and CBC Modes

Implementations of the AES algorithm often contain several code sections that can be fine tuned for optimal performance. However, these optimizations are usually done by hand, which can be a lengthy, labour intensive process. We present a system that can generate billions of variants of the AES encryption code to find the best solution for a particular microarchitecture. We apply both common loop optimizations and ones specific to AES. We evaluate the generated code on hardware with built-in AES support using both selective-brute force and guided searches. Our generator achieves significant speedups over straightforward implementations of the CTR and CBC block cipher modes.

5.1 Introduction

Data must be encrypted if it is to remain confidential when sent over computer networks. Encryption solves many problems involving invasion of privacy, identity theft, fraud, and data theft. However for encryption to be widely used, it must be fast. The problem is so important that recent Intel processors provide hardware support for encryption. These instructions implement key stages of the Advanced Encryption Standard (AES), allowing encryption to be completed more quickly and using less power. The AES algorithm consists of several 'rounds' of encryption, each of which involves a relatively complicated computation. This new hardware support allows an entire round to be implemented with just a single instruction. This chapter presents a

code generator that creates CTR and CBC block-cipher mode implementations using AES encryption.

The use of code generators to find which combination of optimization techniques yields an optimized result is an established technique for solving problems in optimizing for modern architectures. It is ideal for optimizing a small piece of code that uses a vast amount of processing time and to which the best optimizations are not obvious. Code generators have been very successfully applied to several domains such as ATLAS, Spiral, and FFTW (see Section 2.3). The motivation for a code generator is to avoid the problems with code maintenance and code readability that almost inevitably result from hand-tuned assembly specific to the architecture it is written for. A code generator can tune itself to the architecture it is running on to find the best combination of optimizations for that architecture, while remaining readable and maintainable as it can be written in a high-level language, such as C++.

AES encryption costs are greatly reduced with Intel's Westmere microarchitecture [Benadjila et al., 2009] and its instruction set extension [Gueron, 2010] (AES-NI) implements key stages of the AES algorithm. Due to this, our code generator focuses on optimizing the AES algorithm itself, rather than the mathematics required to transform `plaintext` into encrypted data—which is handled by these new instructions. Our generator can create billions of different AES implementations, regardless of architecture, using vector intrinsics based on several optimizations to the AES algorithm, while maintaining correctness. We believe our system is a valuable resource to find an optimized AES implementation on a given target architecture.

In this chapter:

- We show our generator finds optimized variants with an average speedup of 1.29x with 128-bit key sizes and 1.38x with 256-bit key sizes over all standard compiler baselines.
- We show that a simple generator can find a good variation of the code without any specific knowledge of the target microarchitecture.
- We offer a viable alternative to maintaining multiple versions of hand-optimized code.
- We show simulated annealing is an effective and quick method to find a solution in a wide search space.

The remainder of this chapter is organized as follows: The optimization techniques we implement in the code generator to create CTR and CBC implementations are

Listing 5.1: Using AES-NI in AES-128 CBC mode.

```
1  void AES_CBC_Encrypt(__m128i *plaintext, __m128i *ciphertext,
2                       __m128i *key, __m128i feedback, int blocks){
3
4    int i = 0;
5    __m128i result = feedback;
6
7    for(i = 0; i < blocks; i++){
8      result = _mm_xor_si128(plaintext[i], result);
9      result = _mm_xor_si128(result, key[0]);
10     result = _mm_aesenc_si128(result, key[1]);
11     result = _mm_aesenc_si128(result, key[2]);
12     result = _mm_aesenc_si128(result, key[3]);
13     result = _mm_aesenc_si128(result, key[4]);
14     result = _mm_aesenc_si128(result, key[5]);
15     result = _mm_aesenc_si128(result, key[6]);
16     result = _mm_aesenc_si128(result, key[7]);
17     result = _mm_aesenc_si128(result, key[8]);
18     result = _mm_aesenc_si128(result, key[9]);
19     result = _mm_aesenclast_si128(result, key[10]);
20     ciphertext[i] = result;
21   }
22 }
```

detailed in Section 5.2. Discussion of our experimental results on multiple platforms can be found Section 5.3. We summarize the contributions of this chapter and offer our conclusions in Section 5.4.

5.2 CTR and CBC Code Generation

Our generator (GEN1) creates C source code variations of the AES encryption loop in both CTR and CBC mode which have been described in Section 4.2. Standard implementations of CTR and CBC, using AES-NI instructions can be seen in Listings 4.2 and 5.1, respectively[1]. This generator is more "static" in comparison to our generator mentioned in Chapter 6. It generates CTR and CBC code from scratch without any input source code as a guide. Any algorithm changes to CTR and CBC modes would require adding additional functionality to GEN1. GEN1 generates C code and uses compiler intrinsics to specify the Intel AES instructions. Optimizations are turned on and off by a set of flags that traverse a wide search space. These variants are further optimized at low-level by gcc or icc.

GEN1 can create implementations of CTR and CBC with all three key sizes. CTR

[1]Code for both modes are shown with inner loop unrolled.

Listing 5.2: Interleaving three iterations in CTR mode.

```
for(i = 0; i < blocks - 3; i += 3){
  it0_result = nonce = _mm_add_epi64(nonce,one);
  it1_result = nonce = _mm_add_epi64(nonce,one);
  it2_result = nonce = _mm_add_epi64(nonce,one);
  it0_result = _mm_xor_si128(it0_result, key[0]);
  it1_result = _mm_xor_si128(it1_result, key[0]);
  it2_result = _mm_xor_si128(it2_result, key[0]);
  it0_result = _mm_aesenc_si128(it0_result, key[1]);
  it1_result = _mm_aesenc_si128(it1_result, key[1]);
  it2_result = _mm_aesenc_si128(it2_result, key[1]);
  it0_result = _mm_aesenc_si128(it0_result, key[2]);
  it1_result = _mm_aesenc_si128(it1_result, key[2]);
  it2_result = _mm_aesenc_si128(it2_result, key[2]);
  // remaining rounds
}
```

mode is a stream cipher which encrypts a counter value and is `xor`'d with `plaintext`, making it ripe for parallelizable optimizations. CBC mode has a cyclic dependency that occurs because the result of each encrypted block is then used when encoding the next block. This dependency requires different optimization strategies than CTR.

While some optimizations are specific to each mode, there are some that are common to both, such as: completely unrolling the inner loop, software prefetching, and holding a varying number of key values in registers. Also, interleaving can be applied to both CTR and CBC modes—though they do slightly different things. The generator has several optimizations that make both major and minor differences in performance.

The search space for optimizations is very large. Using only interleaving options yields 221,184 variants. With a 256-bit key, and considering holding individual keys in registers in conjunction with interleaving, this number is multiplied by 2^{15} yielding a search space of over seven billion possible combinations in CTR mode alone. There are additional variants with software pipelining. In CBC mode, there are over 2.6 million variants with a search space of using up to five different encryption streams. Again, this number is multiplied by 2^{15} yielding a search space of over 85 billion possible CBC implementations that GEN1 can produce.

5.2.1 CTR Optimizations

A full set of possible optimizations to CTR mode are as follows:

- **localkeys** (0-14) [2^{15}] — Value encourages keys to be kept in registers.
- **interleave factor** (0-15) — How many iterations to unroll and interleave.

- **interleave distance** (0-17) — Affects the how and where the code is interleaved after unrolling.
- **software pipelining initiation interval** (0-17) — Varies the *ii* value.
- **software prefetching to cache OR register** — Adds software prefetching instructions.
- **prefetch source** (1-[interleave factor]) — Determines which `plaintext` to fetch.
- **prefetch distance** (0-[interleave factor-1]) — Determines where fetch instruction is placed.
- **restrict pointers** (on/off) — Adds restrict pointers.
- **streaming store** (on/off) — Adds streaming stores in place of normal stores.

These optimizations are explained in greater detail in the sections immediately following. The order in which they are presented is roughly based on the effectiveness of optimizations when applied to CTR code. Further discussion of the effectiveness of applying these optimizations is provided in Section 5.3.

Interleaving Iterations in CTR

In CTR mode, **interleaving** means unrolling the encryption loop by a factor of x and instructions for these unrolled iterations are arranged in a non-contiguous manner. CTR mode has no dependency from encrypting one block to the next, so this is easily accomplished. This strategy allows key values to be used more frequently in succession and to encourage the compiler to keep them in a register. An example of interleaving in CTR can be seen in Listing 5.2. In this example, the loop has been unrolled three times, and round 1 from different iterations of the loop are interleaved, with key1 being used in rapid succession.

Interleave distance plays an important part in adjusting the "cushion" between interleaved iterations in the loop body. Setting this variable distance will push the next iteration down x lines. The higher the value, the more of the previous iteration will have completed before the current iteration starts. When the interleave distance is maximized, this would generate code that looks exactly the same as simply unrolling the loop.

Listing 5.3: Software pipelining with an initiation interval of 2

```
/* pre software pipelining code (prologue) */

//software pipelining loop (kernel)
for(i = 0; i < (blocks-8); i += 1){
  sp0_result = _mm_xor_si128(sp0_result, plaintext[i]);
  sp1_result = _mm_aesenc_si128(sp1_result, key[13]);
  sp2_result = _mm_aesenc_si128(sp2_result, key[11]);
  sp3_result = _mm_aesenc_si128(sp3_result, key[9]);
  sp4_result = _mm_aesenc_si128(sp4_result, key[7]);
  sp5_result = _mm_aesenc_si128(sp5_result, key[5]);
  sp6_result = _mm_aesenc_si128(sp6_result, key[3]);
  sp7_result = _mm_aesenc_si128(sp7_result, key[1]);
  sp8_result = _mm_xor_si128(sp8_result, key[0]);
  ciphertext[i] = sp0_result;
  sp1_result = _mm_aesenclast_si128(sp1_result, key[14]);
  sp2_result = _mm_aesenc_si128(sp2_result, key[12]);
  sp3_result = _mm_aesenc_si128(sp3_result, key[10]);
  sp4_result = _mm_aesenc_si128(sp4_result, key[8]);
  sp5_result = _mm_aesenc_si128(sp5_result, key[6]);
  sp6_result = _mm_aesenc_si128(sp6_result, key[4]);
  sp7_result = _mm_aesenc_si128(sp7_result, key[2]);
  sp0_result = sp1_result;
  sp1_result = sp2_result;
  sp2_result = sp3_result;
  sp3_result = sp4_result;    // clean up copy variables
  sp4_result = sp5_result;
  sp5_result = sp6_result;
  sp6_result = sp7_result;
  sp7_result = sp8_result;
}
/* post software pipelining code (epilogue) */
```

Software Pipelining

Software pipelining divides several iterations into distinct parts. Software pipelining is described in detail in Section 2.2.3. The size of the parts depend on the initiation interval. The *ii* value can be a maximum of 17 (in 256-bit mode) as the intervals are based on the number of lines of code in the loop body. In Listing 5.3, which shows Counter in 256-bit mode, the *ii* is set to 2. This creates nine pipeline intervals. However, inside the loop, only one full set of instructions used in the AES algorithm exist while they operate on several different blocks. This limits code growth within the loop body and can be seen in Listing 5.3.

Both software pipelining and interleaving Counter code exploit ILP. Generating interleaved code creates good ILP code in the middle of the loop but messy schedules exist in the beginning and end of the loop block due to the `xor` instructions and the

load and store instructions. Software pipelining will instead look for a cleaner solution that has only one set of instructions, operating on parallel iterations. Since the same keys are used to encode every block and there is no cyclic dependency with successive iterations, software pipelining can be used as a way to decrease the dependency between encryption rounds within each encryption block. However, this optimization strategy can only be used with generating CTR code.

5.2.2 Optimizations for Both Modes

The remaining optimizations generally work for both modes. The streaming store option is an intrinsic function that writes vector data to memory without polluting the cache. Using C *restrict pointers* for the `plaintext`, `ciphertext`, and key value arrays can also be turned on and off to provide the compiler with pointer aliasing information. We also make use of other software optimizations by specifically using prefetching and preloading of `plaintext` data. Prefetching uses an x86 instruction to specifically move data (in our case, `plaintext`) into a specified level of cache. Preloading code generates code that specifically assigns `plaintext` to a variable to encourage it to stay in cache and reduce cache misses. The location of prefetch and preload instruction in code is varied by the source and distance parameters. The source declares which iteration it will preload/prefetch and the distance is how far away it will place this directive.

Localkeys

The number of keys that are assigned as variables can also be varied. As mentioned, 128, 192, and 256-bit key sizes use 10, 12, and 14 keys respectively. We encourage the compiler to store these keys in registers by assigning them to local variables. This is useful because depending on the number of free vector registers, n keys can be loaded into registers and rarely (if ever) evicted during the duration of the encryption process. The number of registers available varies depending on the operating system or hardware. So where all keys could be stored in registers on one machine may not work as well on another machine. Another variation is not only the number of keys, but which specific keys. We utilize a bit-vector to decide which individual keys will be assigned to local variables. For example, a bit-vector of `110101` will assign keys 0, 2, 4, and 5 to local variables while the rest remain in memory.

5.2.3 CBC Optimizations

A full set of possible optimizations to CBC mode are as follows:

- **streams** (1-5) — Generate n streams.
- **localkeys** (0-14)[2^{15}] — same as CTR mode.
- **unroll** (0-15) — Unrolls n iterations for all streams.
- **interleave** (on/off) — Interleaves the streams.
- **interleave distance** (0-17) — Affects placement of interleaved code.
- **software prefetching to cache OR register** — same as CTR mode.
- **prefetch source** (2-[interleave factor]) — same as CTR mode.
- **prefetch distance** (0-[interleave factor-1]) — same as CTR mode.
- **restrict pointers** (on/off) — same as CTR mode.
- **streaming store** (on/off) — same as CTR mode.
- **xor** (on/off) — change first two `xor` operations.

The optimizations specific to CBC are explained in greater detail in the sections immediately following. As before, the order in which they are presented is roughly based on the effectiveness of optimizations when applied to CTR code. Further discussion of the effectiveness of applying these optimizations is provided in Section 5.3.

Interleaving Streams in CBC

Due to the cyclic dependency of CBC, any improvements must be made to the implementation itself. To get better overall performance with CBC code, multiple encryption streams must be used. These are independent streams with separate key schedules and `plaintext` inputs. As CTR operated on multiple iterations to hammer the AES unit, interleaving streams in CBC is a similar strategy. Instead of multiple iterations, a single iteration from each encryption stream is interleaved in the loop body and minimizes the cyclic dependency on each stream from one iteration to the next. An example of this method of interleaving streams is shown in Listing 5.4.

Interleave distance plays an important part as GEN1 can also modify the distance between the start of each interleaved stream. Due to the cyclic dependence, this tends to have a greater impact on CBC performance than increasing the interleave distance with CTR mode. Increasing the distance in CBC mode spreads out the long-latency `plaintext` loads generated at the top of the loop and improves performance.

Listing 5.4: Interleaving three encryption streams in CBC mode.

```
__m128i st0_result = st0_feedback;
__m128i st1_result = st1_feedback;
__m128i st2_result = st2_feedback;

for(i = 0; i < blocks; i += 1){
  st0_result = _mm_xor_si128(st0_plaintext[i], st0_result);
  st1_result = _mm_xor_si128(st1_plaintext[i], st1_result);
  st2_result = _mm_xor_si128(st2_plaintext[i], st2_result);
  st0_result = _mm_xor_si128(st0_result, st0_key[0]);
  st1_result = _mm_xor_si128(st1_result, st1_key[0]);
  st2_result = _mm_xor_si128(st2_result, st2_key[0]);
  st0_result = _mm_aesenc_si128(st0_result, st0_key[1]);
  st1_result = _mm_aesenc_si128(st1_result, st1_key[1]);
  st2_result = _mm_aesenc_si128(st2_result, st2_key[1]);
  // remaining rounds
}
```

XOR Modification

As mentioned above, the cyclic dependency of CBC prevents certain optimization opportunities. However, anything that we can do to reduce the length of the dependency chain for each block would be of great use to us. We reduce the dependency chain of the `xor` tree of the original implementation in Figure 5.1a by using `xor` on the `plaintext` and `key0` and placing it in a temporary variable. The feedback variable `result` causes the cyclic dependency, so the `xor` and load of `plaintext` can use cycles before `result` is needed. This is seen in Figure 5.1b.

5.2.4 Simulated Annealing

To manage all the different optimizations when generating CTR and CBC code, we use simulated annealing. As mentioned in Section 2.3.3, simulated annealing is a heuristic search algorithm that employs probabilistic reasoning to increase the search space [Skiena, 1998]. The goal of using simulated annealing in conjunction with our generated code is to use a guided search to reduce the time it takes to find a solution. Exhaustively trying all possible flags is not computationally feasible. In Listing 5.5, we outline the pseudo code we use to implement simulated annealing. The listing includes one of our slight modifications to the classic algorithm to keep track of a global best solution.

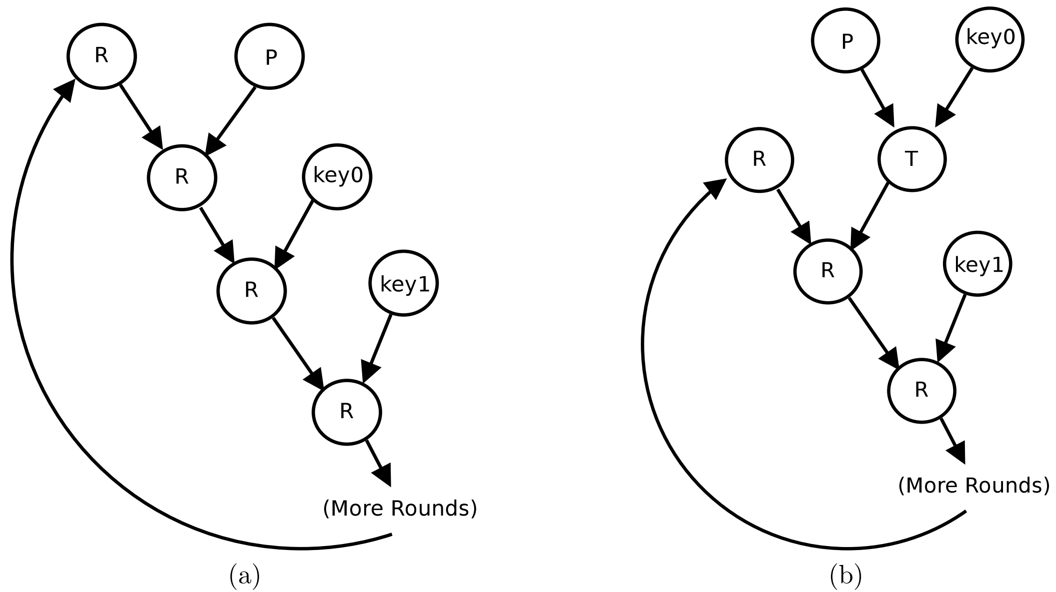


Figure 5.1: Reducing the dependency chain in CBC Mode by modifying the XOR tree—5.1a without xor modification, 5.1b with xor modification.

5.3 Results

Our experimental results with GEN1 show how AES, in both CTR and CBC modes, perform on various platforms. On non-AES enabled microarchitectures, GEN1 is able to simulate the times using various replacement instructions that have documented latencies. While these experiments cannot *actually* perform encryption, they give us two insights. First, from a practical point of view, using non-AES instructions allowed us to develop and test our generator before the AES hardware became publicly available in early January 2010. Second, the visual representation of the results in this section are shown using the same parameters, but with three different latencies. These graphs show the difficultly in predicting performance in advance when trying to find a solution and also shows how slight changes in the latency value can dramatically alter the potential solutions.

This allows the generator to be run with a number of different latency values for each round of encrypting a block. The idea behind simulating encryption with various latency figures is to determine how the AES algorithm would perform, given instructions take x cycles, on a particular microarchitecture. As part of this research pre-dated the commercial release of the Westmere microarchitecture, this allowed us to examine potential results of our work and theorize how these implementations work if latency decreases for the hardware instructions. We later gained access to Westmere hardware

Listing 5.5: Psuedocode of our implementation of simulated annealing.

```
1  anneal()
2       c0 = cost(default_arguments)
3       t = start_temperature
4       g = c0 #global best set to intial arguments
5       while i < iterations :
6            j = 0
7            while j < cooling_steps :
8                     new_args = random_arg * weight
9                 c1 = cost(new_args)
10                delta = (c1 - c0) / c0
11                if c1 < g :  #keep global best
12                     g = c1
13                if delta < 0 :
14                     c0 = c1
15                else if (e^(-delta/(k*t) >= random[0,1)) :
16                     c0 = c1
17                j++
18           t = t * reduce
19           i++
```

and tested GEN1 with actual AES instructions and these experimental results are also found in this section.

Timing using RDTSC and Median Values

The values included in the results for GEN1 are found by selecting the median time of the encryption loop over 150,000 times. The performance values are captured by surrounding the encryption loop with reads of the time stamp counter (RDTSC). We run the experiment so many times to eliminate outliers, because our variance can be quite high. The individual encryption runs are so fast (in terms of clock time), the smallest influence can affect the data. We are still very confident in our results for the following reason: When timing AES implementations at over 150,000 runs per experiment, over 99.9% of the results ran within 1% of the reported values[2] shown in this chapter and in Chapter 6. A very small 2/100*ths* of a percent (or less than 50 runs) of the experiments ran greater than 1% of our reported value. These outliers are 10 to 20 times slower, which cause a very high variance and this performance decrease is caused by other tasks on the machine, such as context switches.

The Core 2 data was gathered on a quad-core 2.4GHz machine running 64-bit Ubuntu emulating the AES instructions with other long-latency instructions. The Core

[2]In other words, using an example, CTR-128's median result is a runtime of about 1310 cycles. Over 99.9% of our tests ran within 13 cycles of this value. Similar behaviour occurred with CBC.

i5 is a dual core machine running at 3.2GHz with 64-bit Ubuntu and implementing the actual AES instructions in hardware. Generated code is compiled by icc with `-O3` enabled.

5.3.1 Cycles per Byte

When benchmarking and discussing the performance of encryption implementations on a system, one of the most common units to use is *cycles per byte*. Simply, cycles per byte (c/b) is the number of clock cycles it requires to encrypt (or decrypt) 1 byte of data. As AES is a block-cipher, and these blocks are 16-bytes each, we compute this value with:

$$\frac{\text{total \# of cycles}}{\text{blocks} \times 16}$$

The lower this value is, the less time an encryption algorithm takes to do its job. One of the benefits of using this unit is that it allows one to weigh the performance/security trade off. For example, AES 128-bit implementations will likely be faster than AES-256 as there are 4 additional rounds to compute in the algorithm. However, using a 256-bit key may be more secure.

Megabits per Second

Megabits per Second is a unit that is also commonly used when discussing encryption performance on GPUs, such as in work by Harrison and Waldron [2008]. GPUs often transfer large amounts of data back and forth from the chip and have the ability to have many pipelines to process the data in parallel, so they require a larger scale to compare their effectiveness. To convert cycle per byte figures to megabits per second requires the following equation:

$$\left(\frac{\text{CPU Clock Speed (Ghz)}}{\text{Cycles per Byte}}\right) \div 10^9 \times 8 \times 1024 = \text{Megabits per Second}$$

5.3.2 Selective-Exhaustive Searches

As mentioned earlier, GEN1 can create literally billions of implementation variations on both the CTR and CBC modes. After trial and error, and observation, we did a number of "selective-exhaustive" searches. What this means is that we saw patterns in the data that showed us which optimization parameters had more influence on performance than others. Before we implemented simulated annealing, we did selective-exhaustive searches and from the results, narrowed the scope of *current* optimization values while

we introduced other optimizations that had smaller effects on performance.

In the experimental phase, this data pointed us in directions that further could improve performance. The data compiled from this strategy also enabled us to generate graphs to show an important contribution of this thesis—that generating code for for AES-NI instructions is non-obvious and non-trivial. As we chased 10ths and 100ths of a cycle per byte improvements in running time, it became even more obvious that we cannot project exactly how smaller optimizations affected the performance of GEN1 scheduled code for Counter and CBC.

Platform: Intel Core 2

The microarchitecture on Intel's Core 2 does not include support for AES. When running the generator on the Core 2 platform, we present two sets of results that substitute the AES instructions with other instructions. We present results with the `PMADDWD`, documented with a 2 cycle latency and with the `MULSD` instruction, documented with a 5-cycle latency [Intel Corp., 2011].

In Counter mode, there are billions of possible variations. We ran experiments on small cross sections of these possibilities that gave us results ranging from 0.98 to 5.0 cycles per byte. The general observations about the data suggest that both software pipelining and interleaving are very useful in Counter mode. Only 30 variants run within 0.10 c/b of the fastest version. Of those, ten are software pipelined versions with initiation intervals of 2 or 3 with the remainder being various interleaved factors of 5 and 8 through 16. The keys held in registers are generally quite high to the max of 14. Both 13 and 14 keys are held in registers in over half of the top performing runs.

Simulated AES on the Core 2 seems to run best with variants that exploit higher levels of instruction-level parallelism (ILP). Tight software pipelining and modest values of interleaving run best. We also find when using software pipelining or interleaving, streaming store becomes a noticeable and positive attribute in performance. In both scenarios, the newly created `ciphertext` must be stored back to memory and streaming store works best when there are several stores at the same time, which there are with software pipelining and interleaving with small interleave distances.

In Figure 5.2, the graph shows a subset of implementations simulating AES implemented with an instruction that has a latency and throughput of 2 cycles. The effect of the number of keys held in a registers is not represented well graphically in the figure. The number of keys does have some effect, but the initiation interval value has a much greater influence on performance. The best ii values are in the 2 to 5 range. This value essentially sets the granularity of using ILP. Figure 5.3 has a similar graphical pattern.

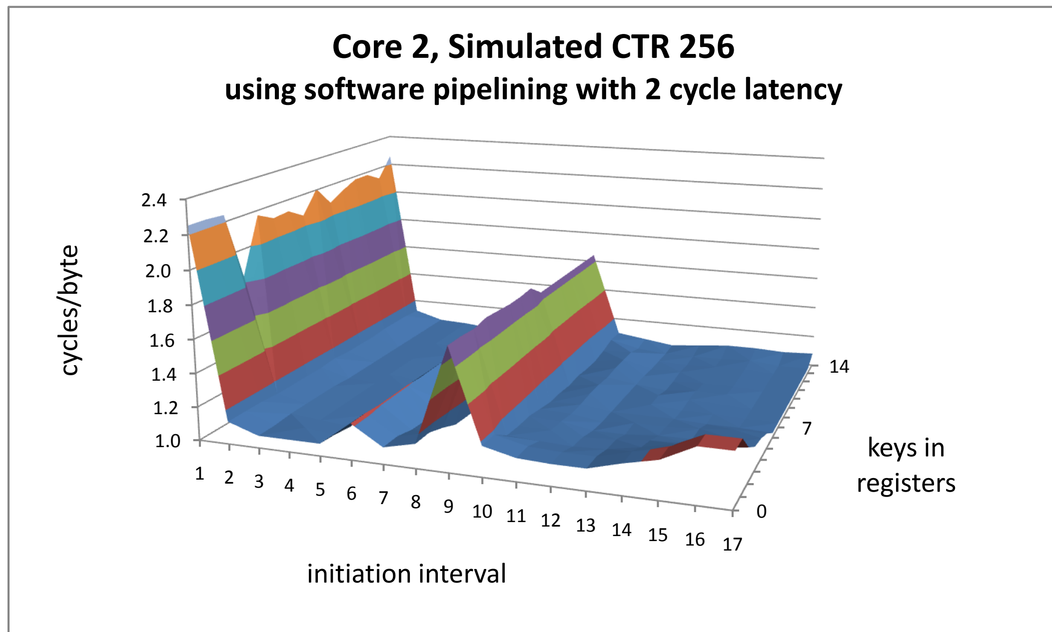


Figure 5.2: Simulated CTR in 256-bit mode results on a Core 2 using a 2 cycle latency and a 1K input buffer. Shown using different software pipelining intervals and various number of keys (inclusive) in registers.

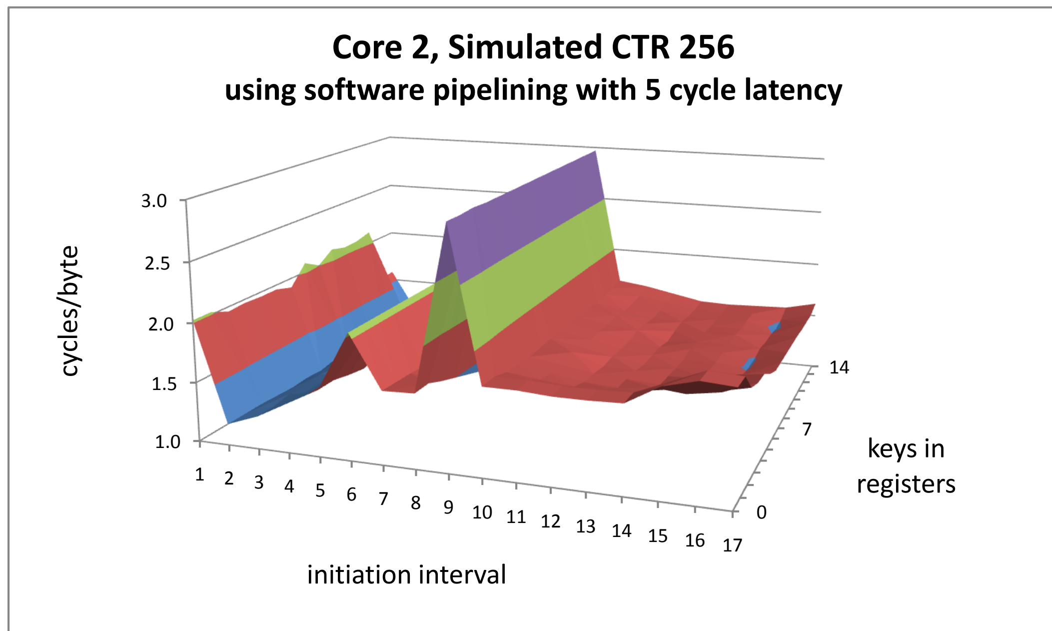


Figure 5.3: Simulated CTR in 256-bit mode results on a Core 2 using a 5 cycle latency and a 1K input buffer. Shown using different software pipelining intervals and various number of keys (inclusive) in registers.

Keys in registers make only small differences while *ii* values from 2-5 are clearly the better solutions.

In both figures, generating code with lower *ii* values created schedules that can pipeline instructions well. These *ii* values create groups of 2 to 4 instructions with 3 to 5 intervals or so. Notice the peaks with *ii* values of 1 and 9. With an *ii* value of 1, each instruction in the loop has its own interval and requires 17 copies to send results down the pipeline. This also causes an incredible amount of code growth that spends very little time inside the steady state with the small input value of 1024 bytes. The peak at the *ii* value of 9 generates code that strictly interleaves two halves of the AES algorithm. This strict interleaving does not help much and keeps the load and store (despite being in different intervals) as side by side instructions, resulting in very poor performance.

When simulating CBC mode on the Core 2, the performance is even less predictable than the Counter results. However, from Figures 5.4 and 5.5 which display results from varying interleaving degrees and localkeys for four streams, there are still some noticeable patterns. With 1, 2, 3 and 4 streams, we see cycles/byte ranges of 5.13-5.49, 2.57-2.93, 1.74-3.0, and 1.71-3.03 cycles/byte respectively. Looking at the data gathered with $streams > 2$ data, there is a clear trend that the best performing implementations use far fewer keys in registers. Considering that every stream will try to assign k number of keys to registers, as the value of k gets larger, register pressure increases. While CBC has a large chain of dependent instructions, as with the *ii* values from Counter code that will optimally use 2 to 4 pipeline intervals, we see that with 2 to 5 streams (2 to 5 pipelines) of parallel encryption also works well.

Figures 5.4 and 5.5 show the results of simulating CBC in 256-bit mode using four streams of data with 2 and 5 cycle latency instructions, respectively. Keys in registers is shown to have a much greater effect in these figures as we have mentioned. What contorts the graph more, however, is the interleave distance. This value pushes the start of the next stream down a certain number of instructions. This encourages the compiler to schedule loads farther apart and not in succession before streams get strictly interleaved.

In both figures, the lower and higher end interleave distance values perform badly. In the first case, low distance values force the loads and stores to execute in succession. With four streams, this is a really bad solution. With the other case, high distance values will essentially create the same affect as unrolling the loop. CBC4 has several long chains of dependent instructions that will execute in succession. Despite this, we still get good performance in comparison to single- or even two-stream numbers.

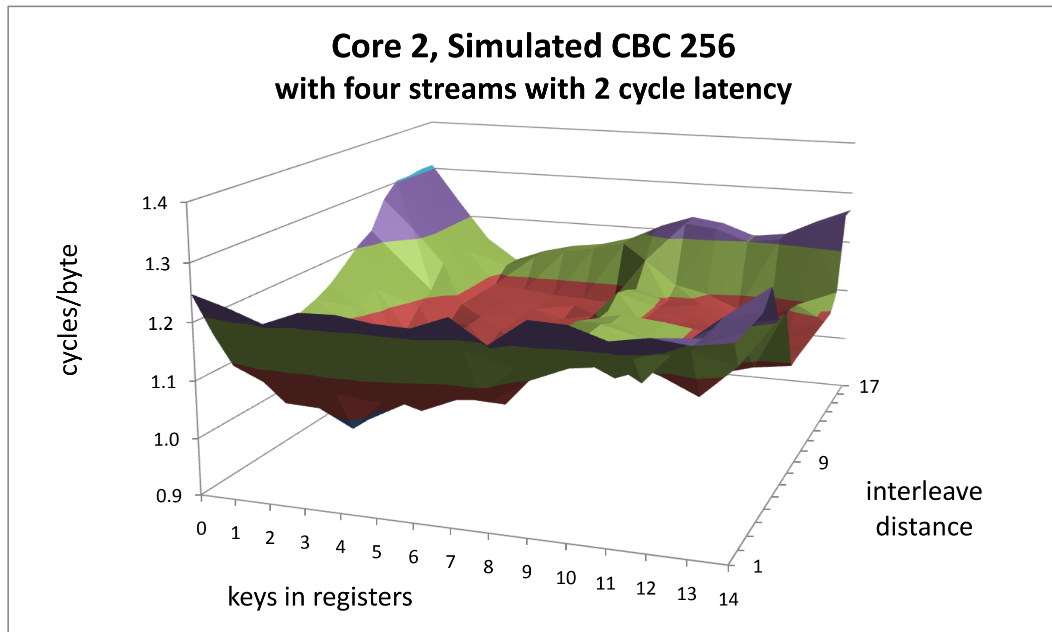


Figure 5.4: Simulated CBC with four streams in 256-bit mode results on Core 2 using a 2 cycle latency and a 1K input buffer. Shown using different interleave distances and various number of keys (inclusive) in registers.

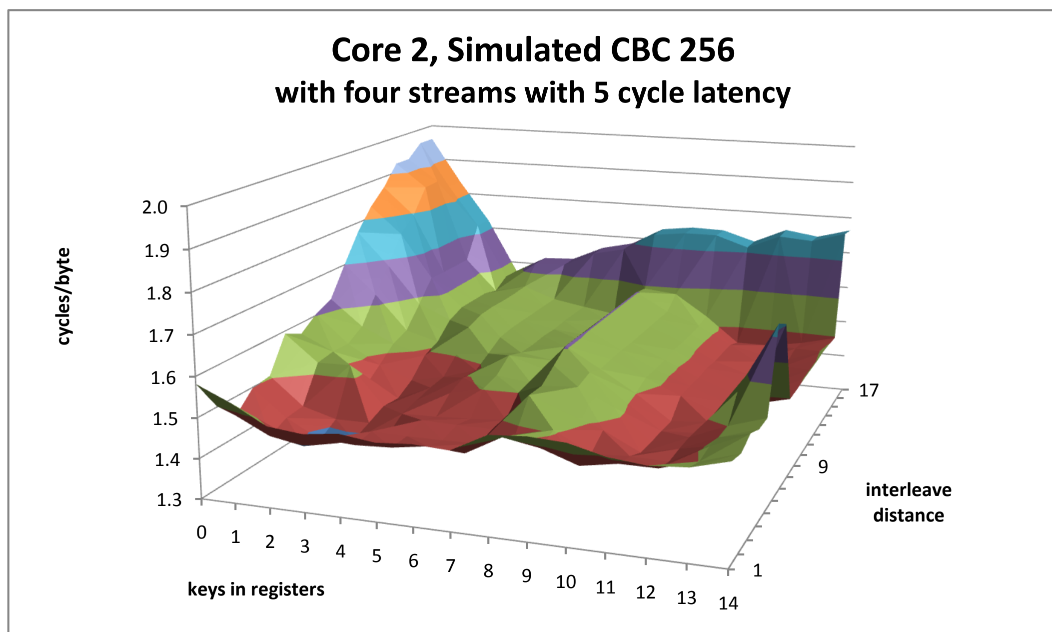


Figure 5.5: Simulated CBC with four streams in 256-bit mode results on Core 2 using a 5 cycle latency and a 1K input buffer. Shown using different interleave distances and various number of keys (inclusive) in registers.

The compiler can still generate code which allows for out-of-order architectures like the Core 2 to parallelize the encryption of the four streams even if the high-level code is not interleaved. There are distance values that work well, however. These tend to be in the 2 to 6 range. This will allow the initial `xor` with `plaintext` from each stream to be separated in the loop, waiting to load while the other AES instructions complete and limiting the number of stalls.

The instructions we used to simulate the (then forthcoming) AES instructions, provided key results on how to improve this and our future generator. But, after running these experiments on AES-enabled hardware, we quickly found that these simulations were just that—simulations. The behaviour and results on the Core i5 were much different and much more volatile.

Platform: Intel Core i5 (Westmere)

Initially, our generator was built for hardware that did not include special hardware instructions to complete AES encryption. We were fortunate to be given access to an internal Intel Westmere machine early on to get preliminary numbers. With the public release of the Core i5 "Clarkdale" processors, we were able to test our numbers more thoroughly on the Westmere microarchitecture. We found interesting results and some surprising behaviours when replacing simulated instructions with the actual `aesenc` calls, due to the difference of defined throughput and latency values for each set of instructions. We simulated results on Core 2 with instructions with small latency and small throughput values. The `aesenc` instruction has a 2 cycle throughput, but a 6 cycle latency [Akdemir et al., 2010].

The Counter results on Westmere reflected the latency and throughput of the instruction set. Using a selective-exhaustive search on a subset of possible variants, we find a range of running times from 1.73 c/b to over 6.0 c/b. Only 25 variants run within 0.1 c/b of each other, so Westmere numbers are more greatly affected by the optimizations used by our generator. Similarly to the Core 2 numbers however, we find that mid-range (6-12) interleaving values tend to run best. Software pipelining initiation intervals of 2 and 3 are also good. Small *ii* values increase ILP (as does interleaving with minimal distances of 1-3), but it also increases register pressure. Software pipelining also requires a number of copies at the end of the loop. These were not optimized out at low level compilation by icc.

Nearly all of the fastest 200 variants use 10 or more keys. Among this selection, prefetching upcoming `plaintext` seems to also be beneficial. If the loop is unrolled 10 times, then GEN1 can insert instructions to prefetch the a single piece of `plaintext`

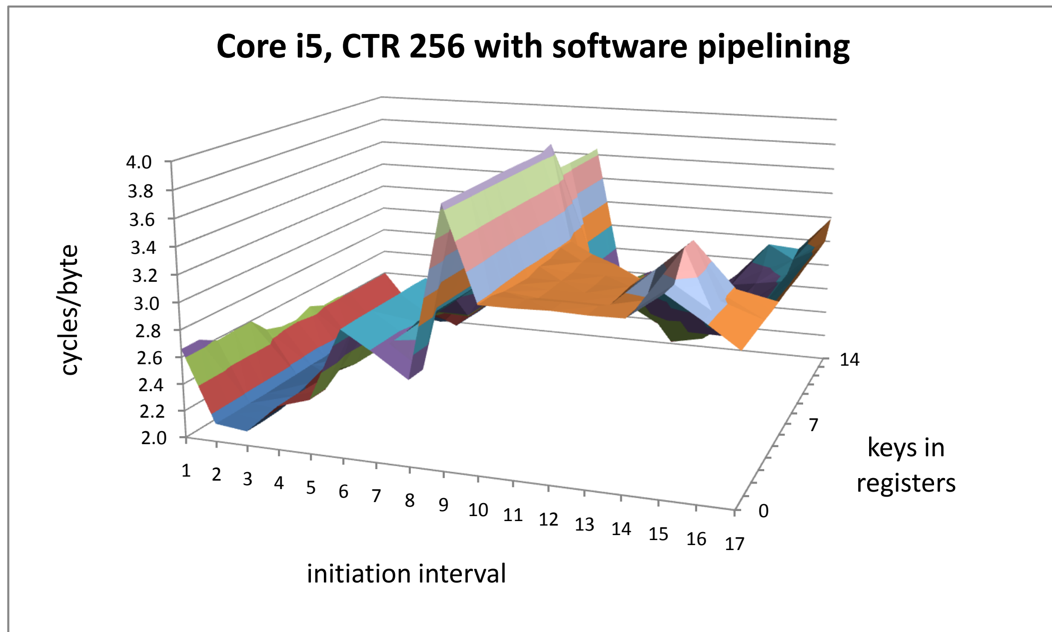


Figure 5.6: CTR in 256-bit mode results on a Core i5 using a 1K input buffer. Shown with different software pipelining intervals and various number of keys (inclusive) in registers.

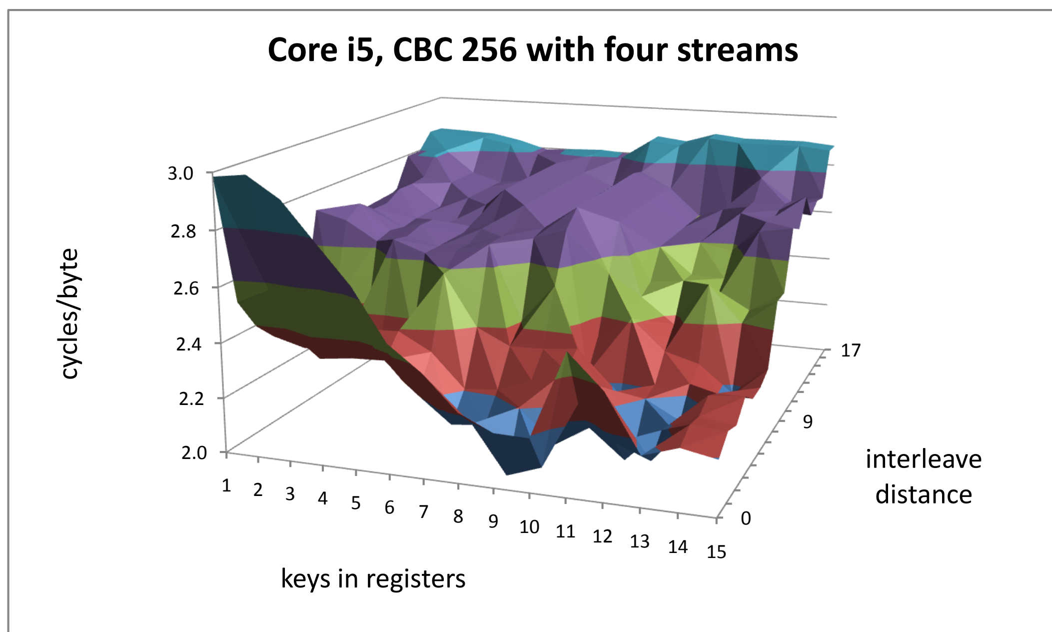


Figure 5.7: CBC with four streams in 256-bit mode results on Core i5 using a 1K input buffer. Shown with different interleave distances and various number of keys (inclusive) in registers.

for `[i+1]` through `[i+10]`. Which iteration of `plaintext` that is prefetched seems to be irrelevant, but the distance between where the `plaintext` is loaded into a register and where it is actually used tends to perform best at a distance of 3 iterations. This means that if `plaintext[i+6]` is going to be loaded into a register, the instruction to do so is placed during the execution of the third iteration. Prefetching tends to reduce c/b by small margins, but can squeeze out a few percent to get that extra bit of performance across the board.

In Figure 5.6, the results are shown for software pipelining on Westmere. There is a clear trend that smaller *ii* values are better with valleys in the graph showing local minimums at *ii* values of 2, 3 and 8. The *ii* value of 9 shows sub-par performance and adjusting the number of registers makes little difference. As with the Core 2 data, the *ii* value of 9 essentially divides the AES loop in half, thus offering little optimization. Lower *ii* values mean a higher number of iterations that have little to no dependency inside the encryption loop and maximize the ability of exploiting ILP. Figure 5.6 shows that *ii* values less than 9 work best for pipelining the new instructions. The performance tends to level out with *ii* values above 9, as the code generated does not change much with higher values. The number of keys held in registers has a greater affect on the Core i5 with *ii* values >= 9 than the affect they have on the Core 2 results for software pipelining with much rougher edges on the graph.

In CBC mode, we see Westmere numbers stabilize and even run slightly faster than CTR numbers (when working with 4 stream buffers). From 1 to 4 streams, we see a range of 1.71 c/b to 5.23 c/b. Figure 5.7 showcases data for 4 stream buffers in CBC mode with varying interleave distances and key values held in registers. There is a clear trend towards small interleave distances. Smaller interleave distances in CBC increases ILP by reducing the dependency from one instruction to the next—both the key value and the result value used are independent for each stream buffer. Figure 5.8 reiterates this value by keeping optimization values constant and focusing on the difference of performance between the number of streams and interleave distance. CBC with 3 or 4 streams rapidly lose performance when distance is increased over 7 which would generate code with large amounts of data dependency.

Optimizations like streaming store, restrict and prefetching/preloading have not been mentioned much in detail as they generally improve already improved optimizations. They can improve a variant by a few percent. This shows us that a generator is useful for finding which set of these minor optimizations can improve a variant—even if minimal. The combination of these smaller optimizations with the larger impact ones like interleaving, software pipelining, and keys in registers give us a significant im-

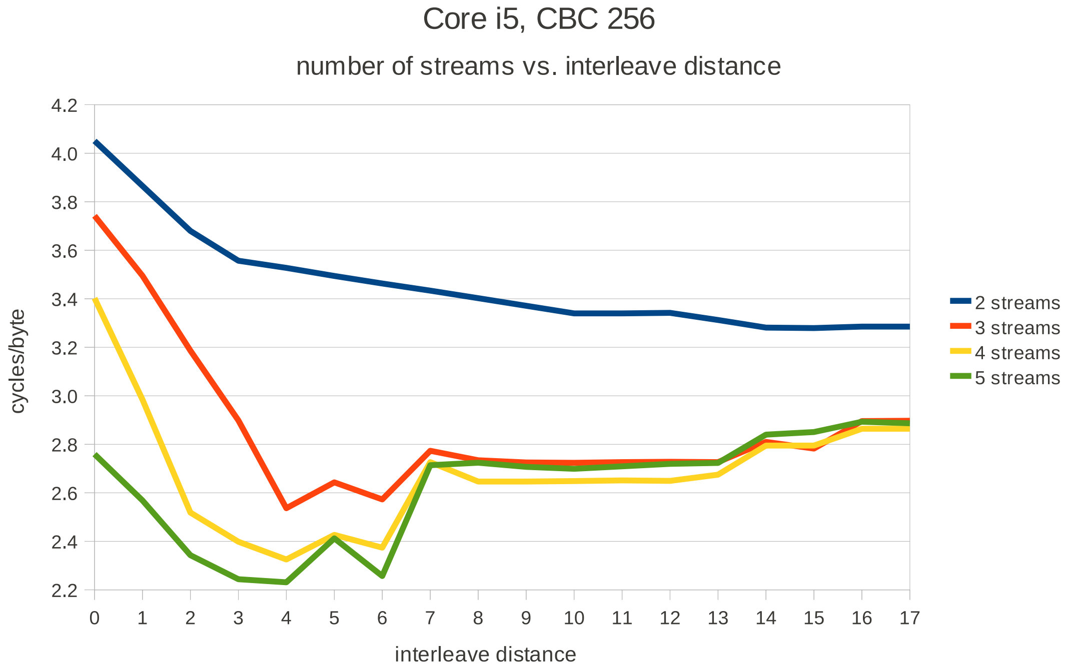


Figure 5.8: Performance graphs streams vs. interleaving distance in CBC.

provement over the standard compiler baselines. This is discussed in the next section with more detail.

5.3.3 Guided Search using Simulated Annealing

We experimented with simulated annealing on Westmere. The selective-exhaustive experiments we conducted ran in batches of often more than 40,000 variants. This is a **very small** subset of possible combinations and even one of these small subsets takes several hours to complete (including compilation time). Our simulated annealing implementation tries 1500 solutions and completes in under an hour. Its initial solution applies no optimizations (flags to zero or off). When flags are changed, greater weights are given to the number of localkeys and the interleaving factor. Lower weights are given to on/off options. In both CTR and CBC modes, annealing was able to traverse much more of the search space to find variants that ran faster than the selective-exhaustive searches found. Running exhaustive searches on even small subsets took hours to complete, whereas simulated annealing took under an hour to complete and came up with a better result.

Applying simulated annealing to variants with CTR code with a 1K input buffer yielded a top performance of 1.434 c/b with software pipelining and 1.398 c/b using

Table 5.1: Best found results generating AES-128 and AES-256 code using Simulated Annealing with GEN1 using 1K and 32K input buffers. Values shown in cycles/byte.

	AES-128		**AES-256**	
Encryption Mode	1K	32K	1K	32K
CTR – general version	1.398	1.264	1.867	1.771
CTR – software pipelined	1.434	1.318	1.887	1.756
CBC – 1 stream of encryption	4.156	4.066	5.656	5.566
CBC – 2 parallel streams of encryption	2.083	2.033	2.834	2.785
CBC – 3 parallel streams	1.394	1.357	1.910	1.876
CBC – 4 parallel streams	1.305	1.279	1.831	1.787
CBC – 5 parallel streams	1.305	1.266	1.794	1.766

Table 5.2: Results in comparison with documented Intel HPL results reported by [Gueron, 2010] using 1K input buffers in AES-128 and AES-256 modes. Values shown in cycles/byte.

	AES-128		**AES-256**	
Encryption Mode	Intel	GEN1	Intel	GEN1
CTR	1.38	1.398	1.88	1.867
CBC – 1 stream	4.15	4.156	5.65	5.656
CBC – 4 streams	1.33	1.305		

Table 5.3: Comparing GEN1 to icc baseline results in AES-128 and AES-256 modes using 32K input buffers. Results shown in cycles/byte.

	AES 128 (32K buffer)			**AES 256 (32K buffer)**		
Mode	**Baseline**	**Best**	**Speedup**	**Baseline**	**Best**	**Speedup**
CTR	1.85	1.264	1.46	2.79	1.771	1.58
CTR-SP	1.85	1.318	1.40	2.79	1.756	1.59
CBC-1	4.67	4.066	1.15	6.06	5.666	1.07
CBC-2	2.02	2.033	0.99	2.72	2.785	0.98
CBC-3	1.71	1.357	1.26	2.67	1.876	1.42
CBC-4	1.76	1.279	1.38	2.69	1.787	1.51
CBC-5	1.77	1.266	1.40	2.69	1.766	1.52

all other possible optimizations as seen in Table 5.1. As CTR is parallelizable, its inherently performs quite well when we apply software pipelining to the code, the number of keys held in registers is often a factor that is far more "searchable" when using annealing. As possible key configurations can be up to 2^{15} in 256-bit mode, this is a massive number of possible solutions if searched exhaustively.

The selective-exhaustive experiments we ran with CBC was again only a subset of billions of possible combinations and annealing is used to reduce search time. We modify the argument weights to account for the different properties of CBC mode. The `xor` optimization is shown to be helpful in most situations so we give it a smaller weight than even turning on and off streaming store and restrict. Interleaving is either on or off, but the algorithm's trace shows that it rarely steps out of interleaved mode. We tune the argument weights for CBC-specific optimizations. Table 5.1 shows GEN1 CBC results encrypting 1 to 5 streams at a time. GEN1 finds that encrypting two streams of CBC instead of one produces nearly a 2x speedup when using a 1K input buffer. Encrypting 3 to 5 streams in parallel does so at a rate around 1.3 c/b.

When AES-NI was introduced by Intel [Gueron, 2010], the work included hand-tuned assembly listings of CTR and CBC and related performance figures. In Table 5.2, we provide several comparison points with this work and the best running variants found by GEN1. The figures reported by Gueron [2010] use 1K input buffers. Table 5.2 shows that GEN1 CTR code performs slightly slower in 128-bit mode and slightly faster in 256-bit mode. With the cyclic dependency of CBC, we are glad to essentially match the reported Intel numbers with a single stream and perform slightly faster with four streams.

An argument for the usefulness of the generator is shown in the results found in Table 5.3. Here, the baselines are the CTR and CBC implementations compiled by icc. Using a 128-bit key size, there is average speedup of 1.292x over the baselines. In 256-bit mode, the average speedup is higher at 1.38x. This table shows clearly that a standard compiler has difficulty optimizing the encryption loop. Our largest speedups are for CTR and CBC implementations that use four or five streams. GEN1 creates code that will exploit ILP better than icc can—whether it is with interleaving or software pipelining. With CBC4 and CBC5, the code becomes very complicated and icc is unable to schedule the streams optimally, causing the loads and stores to be grouped together. Interleave distance is an important optimization with GEN1. The CBC2 results show some interesting behaviour. GEN1 only finds code that is slightly slower. Encrypting two streams, even in parallel, will not maximize AES instruction throughput. A standard compiler like icc can easily schedule the basic block for CBC2,

because like CBC1, there is little we can do to improve the performance of those modes.

5.4 Conclusion

In this chapter, we presented a code generator that creates optimized AES implementations for CTR and CBC modes. To find an optimized implementation, GEN1 generates billions of versions the AES encryption loop. Searching even small subsets of the possible combinations yields speedups of 1.42x. GEN1 uses common optimizations, such as loop unrolling and software prefetching to generate these implementations. In addition, it also makes optimizations specific to implementing the AES algorithm, such as round interleaving and adjusting the number of keys held in registers. These optimization strategies are featured in detail in Section 5.2 of this chapter.

To evaluate this system, we evaluated the generated code on architectures that both did and did not include hardware supported AES instructions. On architectures that did not include AES-NI, performance was simulated using other documented latency instructions. This was done to show how architectural properties can influence the search for the best optimizations. The generator can search this space without any insight into hardware specifics, such as cache sizes, number of registers, or even the instruction set available to implement AES. GEN1 can find CTR and CBC variants by simply re-running the tool on any architecture.

In order to encrypt data using AES with a block cipher mode efficiently, implementations are often optimized by hand. The generator is a viable alternative to maintaining hand-optimized code when new microarchitectures are introduced—saving both time and money. We saw in our results that GEN1 was able to produce CTR and CBC implementations that performed comparably to hand-tuned versions provided by Intel.

In an effort to save time to find an optimized AES loop, we implemented simulated annealing as a guided search heuristic. We found and showed that it performed very well on both platforms and in both CTR and CBC modes. In addition to running a very small fraction of the search space (only 1500 variants), the algorithm found variants that performed better than ones found in selective-exhaustive searches. Results found with annealing showed that standard compilers cannot optimize block-ciphers effectively in most cases.

Using a code generator to find optimized implementations is a good system to find which version runs fast on a target platform in order to perform an already very common everyday task—AES encryption.

Chapter 6

A Generalized AES Program Generator for Instruction-Level Parallelism and Algorithmic Choice

Recent Intel processors include hardware instructions that implement a full AES round in a single instruction. Existing libraries use hand-tuned assembly language to overlap the execution of multiple AES instructions and extract maximum performance. We present a program generator that creates optimized AES code automatically from a simple, annotated C version of the code. We show how this generator can be used to rapidly create highly optimized versions of AES in several modes including CTR, ECB, CBC, PCBC, CFB, OFB, GCM and CCM. The resulting code generated has performance that is equal to, or up to 8% faster than the hand-tuned assembly libraries from Intel.

6.1 Introduction

As described in Chapter 5, Intel has recently extended its popular x86 architecture to support the Advanced Encryption Standard (AES) [The National Institute of Standards and Technology (NIST), 2001; Daemen and Rijmen, 2002]. The new instructions perform a full AES round in a single instruction. Making good use of these instructions is not simple because they have a latency of several cycles. Good performance depends on the execution of multiple AES instructions being overlapped in the processor's pipeline, while also carefully managing other resources such as registers. Existing compilers do not do this well, so Intel provides a library of highly-optimized assembly routines implementing various AES modes [Gueron, 2009, 2010].

Assembly code is both expensive and difficult to understand, maintain or modify. The need for assembly language programming is a major barrier to experimenting with new variations of the code. New versions of architectures often require changes in the assembly. Maintaining multiple versions of the same basic piece of assembly code is a costly software engineering problem. As newer versions of architectures appear, sub-optimal code may be used simply to avoid creating yet another version.

Assembly programming also makes it difficult to combine multiple algorithms into a single piece of code. For example, it is often faster to do encryption and authentication together rather than separately, by interleaving the code from each algorithm [Gopal et al., 2010a]. However, if the code for each algorithm consists of hand-tuned assembly library routines, manually creating a combined version is a cumbersome engineering task. In contrast, our generator can automatically intermingle two pieces of code with little programmer effort.

This chapter describes a program generator that takes an annotated C version of AES code, generates many different variants and automatically finds a variant of the code that runs well on the target processor. The system is flexible enough that the cryptographer can specify different ways of varying the code using several strategies. These strategies are also described. The generator uses an iterative, feedback-directed approach to finding efficient code for the particular architecture.

The main contributions of this chapter are:

- We automate the application of low-level optimizations used in the Intel library.
- We present an automated system to search for the best variation of optimizations, and show that the resulting code can be equal to or faster than hand-tuned assembly.
- We show that using mixed-mode CTR can run "faster than optimal" by replacing some AES-NI instructions with "traditional" table lookups.
- We show that exploiting the properties of `xor` can speed up cyclic modes like CBC, PCBC, CFB and OFB.
- We show the ease of optimizing combined encryption/authentication algorithms like GCM and CCM using function stitching.
- We show that this system can be re-targeted and adapted for hardware optimizations like simultaneous multithreading.

The remainder of this chapter is organized as follows: An overview comparison of the generators in this chapter (AES-GEN) and the previous chapter (GEN1) is provided in Section 6.2. Our implementation of the program generator is described in Section 6.3. A breakdown of optimizations and features are in Sections 6.4—6.6. The flexibility of the generator with parallel algorithms, such as CTR and ECB modes, is discussed in Section 6.4. Making algorithmic variations with cyclic dependent modes is discussed in Section 6.5. Combining algorithms using function stitching is discussed in Section 6.6. An analysis of our experimental results appears in Section 6.7 with related work mentioned in Section 6.8, respectively. Our conclusions are offered in Section 6.9.

6.2 GEN1 vs. AES-GEN

Building the previous generator detailed in Chapter 5, our results were good because we implemented code generation ideas for the high latency AES-NI instructions that were first emulated on hardware that did not support them, and later on actual hardware. Under time pressures of publication, these results were extremely good compared to the baseline, and decent when comparing them against the hand-tuned assembly version. Closer inspection of the disassembled *generated* code revealed shortcomings. GEN1 also lacked the ability to directly influence the way the code was scheduled with the exception of using interleave distance, but the distance would only move the first instruction of an iteration down n lines and the remaining iteration instructions would be generated in lockstep with parallel iterations from that starting point. When we wanted to apply software pipelining techniques to the loop, we could only apply it to the number of lines and could not allow instructions to span several intervals.

Any modifications and further optimizations would need to be hard coded into the generator and specifically tailored for the way they would work in CTR and CBC modes. Directly related to those shortcomings, GEN1 could only support two encryption modes. In addition, new and faster results with other modes were being published by Intel and we were curious if we could match and/or beat these values. Our solution was to create a completely different tool from scratch that would provide a framework that could:

1. Allow us to generate code for other encryption modes, and
2. Greater flexibility over how the code is generated.

From the experience of implementing our ideas with GEN1 and the supporting experimental results, we found optimizations could be classified as either major or mi-

nor. Major optimizations like interleaving, unrolling, and software pipelining yielded the highest influence on encryption performance. These instruction-level parallelism optimizations were implemented in a separate stage of the new system. The minor optimizations like key values and `xor` modifications were implemented in a different stage of the system that could modify the algorithm easily. This separation of optimization allowed us to experiment with different implementations. These two key tools allow us to generate code with incredible flexibility—including the ability to easily scheduled multiple loops.

6.2.1 Function Stitching

As mentioned in Section 4.2.3, authentication is often used in conjunction with encryption. This ensures that the message is both secure and correct when it is transmitted from one source to another. The authentication process can be executed separately from encryption in different loops. Combined authentication and encryption modes, like the ones we discuss in this chapter will incorporate both processes in the same basic block.

To improve execution time of combined modes, Gopal et al. [2010a] from Intel suggest a process called *function stitching*. This process literally interleaves sections of both processes with each other. We emulate this process and are able to adjust the level of "stitching" between processes in the generator by applying instruction latencies in the generator to a data dependency graph.

6.2.2 Cycles per Round

In the discussion of results for GEN1, most of the figures are stated in cycles/byte. While this is the common measurement for dealing with cryptographic performance, we present an additional figure—cycles/round. This is the number of cycles it takes per round of AES encryption. Recall that it takes 10 rounds of encryption with a 128-bit key, 12 rounds with a 192-bit key, and 14 with a 256-bit key. To determine cycles/round, we use:

$$\frac{\text{total \# of cycles}}{\text{rounds(key size)} \times \text{blocks}}$$

.

We provide this unit of measurement because it helps us to determine how close to optimal the AES-GEN generated code is. As we know that an `aesenc` instruction can be issued every two cycles, and an `aesenc` instruction is used for every round, our

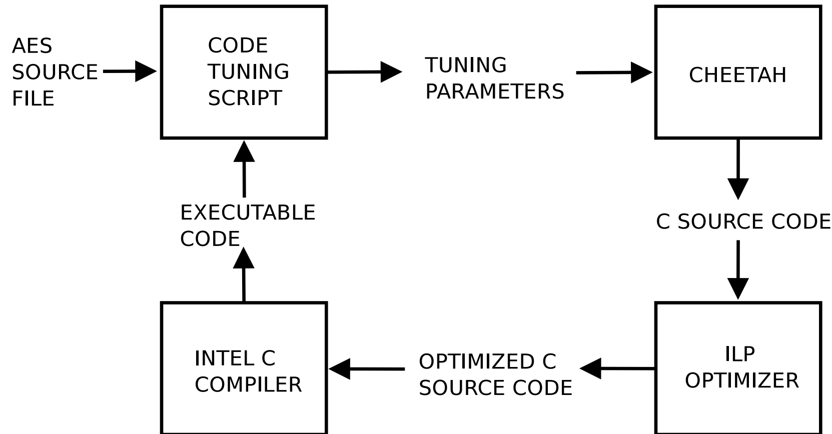


Figure 6.1: Structure of the AES-GEN System.

performance goal is 2 cycles per round regardless of the mode, the input size, and the key schedule.

6.3 The AES-GEN Program Generator

Our generator, which we refer to as AES-GEN, automates many of the optimization techniques used in GEN1 and adds some additional ones. The main benefit of this system of creating variants is that it is more generalized. AES-GEN can be used to generate code for any of the AES modes and can find efficient code for any processor with the AES-NI instruction set[1].

AES-GEN uses a multi-stage system to find a potential solution to implement a particular AES mode. The structure of the system is shown in Figure 6.1. The input to the system is annotated C code which implements a certain AES mode, which can be seen in Listing 6.1. These annotations can be used to express algorithmic variants and other information. The general process is as follows:

1. The algorithmic variations are handled by a tool called Cheetah. Cheetah outputs a legal C/C++ file which is sent to the ILP optimizer.

2. The ILP optimizer schedules the code after converting to a simple version of static single assignment (SSA) form [Cytron et al., 1991] and building a data dependency graph (DDG).

[1] Of course, as described in Section 5.3.2, encryption round instructions could still be replaced with other instructions for simulation purposes.

3. The optimized scheduled code is then compiled by a standard compiler like gcc or icc.

4. The unoptimized native solution seeds initial solution required by the code tuning script, which is powered by an augmented simulated annealing algorithm. Arguments are modified by the code tuning script and a new version of the code is created.

This iterative process is repeated thousands of times, until an efficient variant of the code is found. A more detailed description of the AES-GEN system is described in the sections immediately following.

6.3.1 Algorithmic Choices with Cheetah

Our generator builds code using two stages. The first stage relates to choices in the basic source code that is generated. For example, setting the number of key values to be stored in registers by assigning them to variables, using streaming stores, and making various `xor` modifications. We call these *algorithmic variants*. Cheetah [Rudd, 2007] is a Python-powered template engine, which we use essentially as a macro-expansion tool. We use Cheetah to make high-level source code changes to the input which is used by the ILP optimizer. The primary use of Cheetah is to make these optimizations to the various AES implementations:

- **Manage local key values:** A keymask is used to determine which key values will be assigned to variables.

- **Manage constants:** Constants within the loop can also be assigned as a constant to a register, rather than kept in memory. This can trade off lookup time with register pressure, just like key values.

- **Manage the number of streams used in cyclic modes:** Create function arguments, separate names for each stream's `plaintext`, `ciphertext` and key schedule.

- **Manage the `xor` optimization:** More on this is found later in 6.5.1, with varying degrees of manipulating `xor` within the AES encryption loop.

Cheetah manages and generates these various small, but large number of, optimizations that can affect the performance of AES code. This allows the generator to handle all the remaining optimizations through scheduling and other changes

Listing 6.1: Cheetah templated AES-128 Counter code.

```
1  void AES_CTR_Encrypt(__m128i* plaintext, __m128i* ciphertext,
2               __m128i* key, long long ivec, long nonce,  int blocks){
3    int i = 0;
4    __m128i one = _mm_set_epi32(0,1,0,0);
5    __m128i BSWAP = _mm_setr_epi8(7,6,5,4,3,2,1,0,15,14,13,12,11,10,9,8);
6    __m128i result;
7    __m128i plain;

9    $keys_to_be_declared // Cheetah variable for key declarations

11   __m128i counter_block = _mm_setzero_si128();

13   counter_block = _mm_insert_epi64(counter_block, ivec, 1);
14   counter_block = _mm_insert_epi32(counter_block, nonce, 1);
15   counter_block = _mm_srli_si128(counter_block, 4);
16   counter_block = _mm_shuffle_epi8(counter_block, BSWAP);

18   for(i = 0; i < blocks; i++){
19     counter_block = _mm_add_epi64(counter_block,one);
20     result = _mm_shuffle_epi8(counter_block, BSWAP);
21     result = _mm_xor_si128(result, $key0);
22     result = _mm_aesenc_si128(result, $key1);
23     result = _mm_aesenc_si128(result, $key2);
24     result = _mm_aesenc_si128(result, $key3);
25     result = _mm_aesenc_si128(result, $key4);  // All keys are also
26     result = _mm_aesenc_si128(result, $key5);  // Cheetah variables
27     result = _mm_aesenc_si128(result, $key6);
28     result = _mm_aesenc_si128(result, $key7);
29     result = _mm_aesenc_si128(result, $key8);
30     result = _mm_aesenc_si128(result, $key9);
31     result = _mm_aesenclast_si128(result, $key10);
32     result = _mm_xor_si128(result, plaintext[i]);
33     ciphertext[i] = result;
34   }
35 }
```

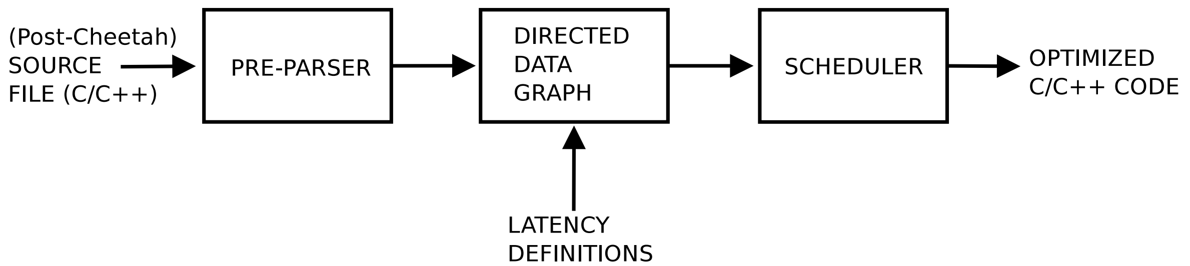


Figure 6.2: Structure of the ILP Optimizer.

that affect the encryption loop itself. A sample template is shown in Listing 6.1. Cheetah variables are preceded by a `$` sign. In the listing, `$key0` will either be replaced with `key[0]` if the key will be kept in memory or `key0` with an accompanying `const __m128i key0 = key[0];` instruction included where `$keys_to_be_declared` is shown.

6.3.2 The ILP Optimizer

After all the algorithmic variations have been applied to the input source code, further optimizations that are aimed at exploiting instruction-level parallelism are applied in the second stage. These optimizations include loop unrolling, interleaving, and software pipelining. These optimizations are applied and then scheduled by the ILP optimizer. The process of applying ILP optimizations is shown in Figure 6.2. The ILP optimizer parses the input using lexer and grammar tools. This parsing stage converts the code to SSA form and builds the DDG.

After the DDG has been created, the optimizer uses two scheduling phases. The basic block scheduler applies a standard list scheduling algorithm [Skiena, 1998] to the data dependence graph to generate code that can be executed in parallel. For example, if the loop is unrolled with an unrolling factor of two, and there are no data dependencies between iterations, the basic block scheduler builds a data dependence graph that includes statements from both iterations. The instructions from both iterations are interleaved in the generated code, which usually allows them to execute in parallel. The generator uses variable renaming to eliminate false data dependencies. Figure 6.2 shows how the ILP optimizer works.

The ILP optimizations applied in this stage are based on standard compiler optimizations for VLIW architectures [Fisher, 1983]. Like a standard VLIW compiler, we model data dependencies and build schedules of statements based on these data depen-

Listing 6.2: Modulo scheduling example, standard CTR.

```
1  /* prologue */
2    for(i = 0; i < blocks - 4; i++ ){
3      sp0_result11 = _mm_aesenclast_si128(sp0_result10, key[10]);
4      sp1_result8 = _mm_aesenc_si128(sp1_result7, key[7]);
5      sp2_result5 = _mm_aesenc_si128(sp2_result4, key[4]);
6      sp3_result2 = _mm_aesenc_si128(sp3_result1, key[1]);
7      sp4_counter_block = counter_block = _mm_add_epi64(counter_block, one);
8      sp0_result12 = _mm_xor_si128(sp0_result11, plaintext[i]);
9      sp1_result9 = _mm_aesenc_si128(sp1_result8, key[8]);
10     sp2_result6 = _mm_aesenc_si128(sp2_result5, key[5]);
11     sp3_result3 = _mm_aesenc_si128(sp3_result2, key[2]);
12     sp4_result = _mm_shuffle_epi8(sp4_counter_block, BSWAP);
13     ciphertext[i] = sp0_result12;
14     sp1_result10 = _mm_aesenc_si128(sp1_result9, key[9]);
15     sp2_result7 = _mm_aesenc_si128(sp2_result6, key[6]);
16     sp3_result4 = _mm_aesenc_si128(sp3_result3, key[3]);
17     sp4_result1 = _mm_xor_si128(sp4_result, key[0]);
18     // variable copy cleanup
19   }
20 /* epilogue */
```

dencies. However, unlike a standard VLIW compiler, we do not attempt to accurately model processor resources. We assume that we do not have a clear idea of the available resources and generate a large number of different variants of the code instead. We use iterative feedback and machine learning to find a variant of the code that fits the actual machine resources. The result is that our generator does a reasonable job of finding good code for any current or future processors supporting the Intel AES instructions.

Nonetheless, we need some way to affect the amount of ILP that is exposed by the generated code. Exposing more ILP usually results in more code growth, more variable renaming and greater register pressure. We can adjust the ILP in several ways, while limiting code growth. We often use modulo scheduling [Rau, 1994] instead of loop unrolling. Modulo scheduling is a form of software pipelining where the length of the loop kernel is fixed in advance and instructions are scheduled based on a measure of their dependency modulo value.

Each statement in the source code has a latency value, which is a notional number of cycles that it will take to theoretically execute. AES-GEN uses these notional latencies when building the schedule. By increasing or decreasing the latencies of statements, the ordering of the statements in the schedule can be changed, which in turn affects the amount of ILP in the schedule. For basic block scheduling, the ordering of statements is determined entirely by the latencies.

Our modulo scheduler can also control the amount of exposed ILP by varying

the initiation interval. Note, we redefine the use of ii for this chapter[2]. When used with AES-GEN, the ii is the number of notional cycles that elapse between starting successive iterations of the loop. Lower ii values result in higher amounts of instruction-level parallelism that exist in the generated code. By varying both the ii and the latency of the instructions in the loop, a very large space of possible modulo schedules for the loop can be explored. Figure 6.2 shows a possible transformation of the standard CTR loop (from Listing 4.2) when latencies are set to 2 cycles per instruction and an ii of 6. This new system also allows us to span instructions over several intervals. For example, we can set the latency of an instruction to 12 cycles, but set the ii to 4, causing the instruction to run through three intervals before it is used again.

AES-GEN uses annotated C code as input, as was shown in Figure 6.1. Therefore, prototype implementations of AES code can be fed into the generator immediately to give us quick feedback on its performance on the target architecture. This allows an exploratory approach when thinking about new ways to write the AES code. With an assembly language implementation, the cost of experimenting with new ways of writing the code is very high. Every change made may require a completely different schedule and register allocation. AES-GEN allows us to write a straightforward C implementation of the code and then it handles all low-level details associated with finding a good schedule.

Using different ii values in conjunction with different latency sets yields very different results. A subset of instructions will be pushed into different intervals of the schedule and can make different changes. Of course, the interval in which these instructions are placed will be dependent on the latency set values that correspond to each instruction. This effect is illustrated in the 3-D surface graph seen in Figure 6.3. This graph shows the turbulent search space with adjusting only two (albeit, important) parameters and does not consider the influence of other optimizations, such as key values in registers.

The resulting scheduled code can look very intimidating and the two examples found in Appendix C show two "solutions" to CTR code, which are discussed later. It is clear from a even a quick glance that this code would be very difficult to schedule by hand. The ILP optimizer and AES-GEN does this seamlessly with ease. AES-GEN finds a solution after traversing a very large search space, and this requires the final stage of the system which ties everything together—the code tuning stage.

[2]The definition of ii in Chapter 5 assumes that the latency of all instructions is one cycle

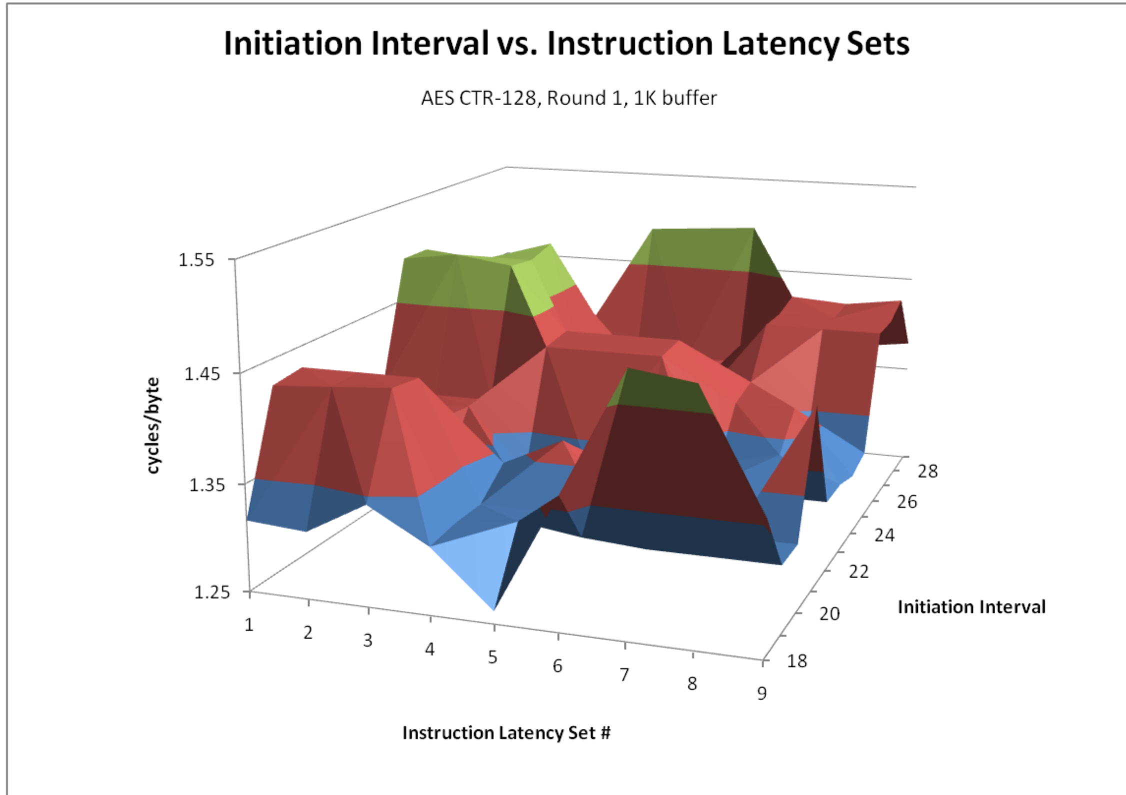


Figure 6.3: CTR R1 128, Initiation Interval vs. Latency Set.

6.3.3 Code Tuning

After the optimizations are applied in the first two stages, AES-GEN then compiles and benchmarks the newly generated code. The performance of this variation is used by the code tuning script to modify the optimization parameters to generate the next potential variation. In Section 5.2, we discussed the broad scope of billions of possible variations for generating CTR and CBC implementations with GEN1. With AES-GEN, many of the optimizations like interleaving are done with the data dependency graph and interleave distance is accomplished through setting latencies. Because of these options and the generalizations with both the algorithmic and ILP optimization options, the scope of variations that AES-GEN can generate is effectively unlimited.

This code tuning strategy is part of the larger AES-GEN system, as shown in Figure 6.1. The code tuning process starts with a pseudo-C source file template with Cheetah annotations. Cheetah handles algorithmic variation parameters to set the number of constants used in the encryption loop to be assigned to registers. This creates a legal C source file that is used as input into AES-GEN. AES-GEN, as detailed earlier in Chapter 6.3, applies the latency values to the data dependency graph and uses the

initiation interval to schedule the code. AES-GEN output is compiled by icc, executed, and using the simulated annealing algorithm mentioned in Section 2.3.3, optimization parameters are adjusted in an effort to improve the code. We go through 1500 variations using annealing, before doing a second 1500 iteration pass with a refined set of possible arguments—based on the best optimization values from the first pass.

For the implementations generated by AES-GEN, we redesigned the simulated annealing tuning script that we used with GEN1. As AES-GEN is a more general-case program generator, the code tuning script also needs to suit general-case algorithms as well. This script now uses an argument file that will anneal with two sets of arguments: those to Cheetah and those to the ILP optimizer. Cheetah template variables will be assigned separate weights and modified independently of the parameters to AES-GEN. This allows AES-GEN to remain isolated to scheduling the code based on latencies and the number of pipeline intervals for any piece of code, while custom arguments to Cheetah which are algorithm dependent can still be adjusted by the annealing script with ease.

6.4 Generator Flexibility with Parallel Algorithms

The results with GEN1 showed us that Counter, which is a parallelizable algorithm, performed extremely well when the generator created code that used the interleaving strategy, and good when using software pipelining. This section focuses on how AES-GEN can easily generate very good (and even better than optimal) code for different implementations of parallelizable algorithms such as CTR and ECB.

6.4.1 Counter (CTR)

A key limitation in GEN1 was the ability to easily and quickly create and test different AES implementations. However, with GEN1, we were only able to reasonably test a plain, straight-forward version of Counter. Our results in Section 5.3 showed us that we had a good concept and we could come close, if not match the performance of hand-tuned assembly code.

Mixed-mode operation

As mentioned in Section 4.5, the `_mm_aesenc_si128` and its related instructions have a 6 cycle latency and can be issued every 2 cycles. GEN1 created Counter implementations with performance that approached the theoretical maximum performance of 2

Table 6.1: Performance of AES-GEN CTR code in cycles per AES round. Results for CTR (round 1) and CTR (round 2) reference 4080B, while CTR (standard) references 32K.

Encryption Mode	**AES-128**		**AES-256**	
	1K buffer	4080B buffer*	1K buffer	4080B buffer*
CTR (standard)	2.194	2.013	2.143	2.007
CTR (round 1)	2.043	1.864	2.013	1.905
CTR (round 2)	2.237	2.077	1.982	1.854

cycles/round. We needed to modify the Counter algorithm in addition to adding more optimization strategies to the generator to further improve the results. Our idea for improving Counter performance was to combine both AES-NI and scalar instructions for round encryption. The scalar instructions are based on a more traditional AES implementation that would have been required to perform an AES block cipher prior to the release of AES-NI using table lookups.

Our implementations used counter-mode caching [Bernstein and Schwabe, 2008] to reduce the number of table lookups in combination with AES-NI. We created two versions using this *mixed-mode* strategy: CTR Round 1 (shown in Listing 6.3) which uses a scalar counter and table lookups to perform the first round of encryption. CTR Round 2 (seen in Listing 6.4) uses the same strategy, but for the first two rounds.

These versions also convert the counter increment to scalar operations. Those operations can now be combined with the scalar code for encrypting the first round. This reduces the use of the AES unit and frees up one to three vector registers as the counter and round key(s) can now be stored in scalar registers.

These versions of CTR work only for inputs of up 4080 bytes (255 blocks). This limitation can be overcome, but at the cost of a more complicated implementation. However, despite the size limitation, the mixed-mode AES CTR code is still valuable. AES is commonly used to encrypt data packets in wireless networks, which are typically less than 2K in length. Tables 6.1 and 6.2 show results of these implementations with input sizes of 1024 bytes and 4080 bytes[3].

Faster than "Optimal" CTR

We used this flexibility to create implementations of AES CTR mode that are faster than the "optimal" code. The Intel AES instructions implement a full round of AES in a single instruction. On the Intel Westmere microarchitecture, the AES instructions

[3]CTR (standard) is reported with 1K and 32K input buffers

Listing 6.3: AES CTR (round 1) encryption in C using AES-NI instructions.

```
1  void AES_CTR_Encrypt(__m128i* plaintext, __m128i* ciphertext,
2                  __m128i* key, long long ivec, long nonce,
3                                  int blocks){

5    __m128i result, result0, result1;
6    __m128i fake_key, saved_r1, table_mask;
7    __m128i counter_block = _mm_setzero_si128();
8    unsigned scalar_key0, scalar_result0, scalar_result1;
9    unsigned my_counter, scalar_counter = 0;
10   int i = 0;

12   counter_block = _mm_insert_epi64(counter_block, ivec, 1);
13   counter_block = _mm_insert_epi32(counter_block, nonce, 1);
14   counter_block = _mm_srli_si128(counter_block, 4);

16   result0 = _mm_xor_si128(counter_block, key[0]);
17   scalar_key0 = _mm_extract_epi32(key[0], 3);
18   result1 = _mm_insert_epi32(result0, scalar_key0 & 0xFFFFFF, 3);
19   saved_r1 = _mm_aesenc_si128(result1, key[1]);
20   table_mask = _mm_cvtsi32_si128(table3[0]);
21   saved_r1 = _mm_xor_si128(saved_r1, table_mask);

23   for(i = 0; i < blocks; i++){
24     my_counter = scalar_counter++;
25     scalar_result0 = ((_bswap(scalar_key0)) & 0xFF) ^ my_counter;
26     table_mask =  _mm_cvtsi32_si128(table3[scalar_result0]);
27     result1 = _mm_xor_si128(saved_r1, table_mask);

29     result = _mm_aesenc_si128(result1, key[2]);
30     result = _mm_aesenc_si128(result, key[3]);
31     result = _mm_aesenc_si128(result, key[4]);
32     result = _mm_aesenc_si128(result, key[5]);
33     result = _mm_aesenc_si128(result, key[6]);
34     result = _mm_aesenc_si128(result, key[7]);
35     result = _mm_aesenc_si128(result, key[8]);
36     result = _mm_aesenc_si128(result, key[9]);

38     fake_key = _mm_xor_si128(key[10], plaintext[i]);
39     result = _mm_aesenclast_si128(result, fake_key);
40     ciphertext[i] = result;
41   }
42 }
```

Listing 6.4: AES CTR (round 2) encryption in C using AES-NI instructions.

```c
void AES_CTR_Encrypt(__m128i* plaintext, __m128i* ciphertext,
    __m128i* key, long long ivec, long nonce, int blocks){

  int i = 0;
  __m128i plain, result, counter_block = _mm_setzero_si128();
  unsigned scalar_counter = 0, scalar_key0, scalar_r0, scalar_r1;
  unsigned idx0, idx1, idx2, idx3, v0, v1, v2, v3, my_counter;
  __m128i r0, fake_key, t0, t1, t3, t4, t5, t6;
  __m128i r1, saved_r1, result2, saved_result2;
  __m128i table_entries, second_round_output;
  unsigned first_round_output_x0;

  counter_block = _mm_insert_epi64(counter_block, ivec, 1);
  counter_block = _mm_insert_epi32(counter_block, nonce, 1);
  counter_block = _mm_srli_si128(counter_block, 4);
  r0 = _mm_xor_si128(counter_block, key[0]);
  scalar_key0 = _mm_extract_epi32(key[0], 3);
  r1 = _mm_insert_epi32(r0, scalar_key0 & 0xFFFFFF, 3);
  saved_r1 = _mm_aesenc_si128(r1, key[1]);
  first_round_output_x0 = _mm_extract_epi32(saved_r1, 0) ^ table3[0];
  result2 = _mm_insert_epi32(saved_r1, 0, 0);
  saved_result2 = _mm_aesenc_si128(result2, key[2]);
  table_entries = _mm_set_epi32(table1[0],table2[0],table3[0],table0[0]);
  second_round_output = _mm_xor_si128(saved_result2, table_entries);

  for(i = 0; i < blocks; i += 1){
    my_counter = scalar_counter++;
    scalar_r0 = (_bswap(scalar_key0) & 0xFF) ^ my_counter;
    scalar_r1 = table3[scalar_r0] ^ first_round_output_x0;
    idx0 = scalar_r1 & 0xFF;
    idx1 = (scalar_r1 >> 8) & 0xFF;
    idx2 = (scalar_r1 >> 16) & 0xFF;
    idx3 = (scalar_r1 >> 24);
    t0 = _mm_cvtsi32_si128(table0[idx0]);
    t1 = _mm_cvtsi32_si128(table3[idx3]);
    t3 = _mm_unpacklo_epi32(t0, t1);
    t4 = _mm_cvtsi32_si128(table2[idx2]);
    t5 = _mm_cvtsi32_si128(table1[idx1]);
    t6 = _mm_unpacklo_epi32(t4, t5);
    table_entries = _mm_unpacklo_epi64(t3, t6);
    result2 = _mm_xor_si128(second_round_output, table_entries);
    result = _mm_aesenc_si128(result2, key[3]);
    result = _mm_aesenc_si128(result, key[4]);
    result = _mm_aesenc_si128(result, key[5]);
    result = _mm_aesenc_si128(result, key[6]);
    result = _mm_aesenc_si128(result, key[7]);
    result = _mm_aesenc_si128(result, key[8]);
    result = _mm_aesenc_si128(result, key[9]);
    fake_key = _mm_xor_si128(key[10], plaintext[i]);
    result = _mm_aesenclast_si128(result, fake_key);
    ciphertext[i] = result;
  }
}
```

Table 6.2: Performance of AES-GEN CTR code measured in cycles per byte. Results for CTR (round 1) and CTR (round 2) reference 4080B, while CTR standard references 32K.

Encryption Mode	AES-128		AES-256	
	1K buffer	4080B buffer*	1K buffer	4080B* buffer
CTR (standard)	1.371	1.258	1.875	1.756
CTR (round 1)	1.277	1.165	1.761	1.667
CTR (round 2)	1.398	1.298	1.734	1.622
CTR Intel (HPL)	1.38		1.88	

have a throughput of one instruction every two cycles. Therefore, if an implementation of AES CTR completes each round in an average of a little over 2 cycles per round, we consider it close to optimal. Table 6.1 shows the cycles/round for our generated versions of AES-CTR. These implementations are almost optimal, especially for large inputs.

Our implementations of AES achieve good performance by keeping the hardware AES unit running at maximum capacity. However, while the AES unit is saturated, the rest of the processor core is largely idle. If we could use these other idle resources to implement some of the AES CTR rounds, we might be able to exceed the speed of the optimal code.

The results in Table 6.2 show that replacing round 1 with scalar instructions and table lookups is faster than using the AES-NI equivalent. With a 1K buffer, the 128- and 256-bit implementations of AES using this method achieves a 7.3% and a 6.5% reduction in runtime, respectively. Eliminating a single AES instruction could theoretically reduce runtime in 128-bit mode by 10%, and by 7.1% in 256-bit mode; we consider this to be an excellent result. Replacing 2 rounds with table lookups performs worse when using 128-bit keys—running slower than both Intel HPL and CTR (standard), but achieves a 1.08x speedup using a 256-bit key with both 1K and 4080Byte buffers. We believe that the results in Tables 6.1 and 6.2 demonstrate that it is possible for AES CTR to run faster than the optimal possible runtime when using just Intel AES instructions. These are also the first implementations—to our knowledge—that complete an AES round in under two cycles. Listings C.1 and C.2 in Appendix C show these "faster-than-optimal" solutions in full[4]. They are incredibly complex and would be a significant software engineering task to attempt to build something similar.

[4]These listings were generated from the initial source files shown in Listings 6.3 and 6.4, respectively.

6.4.2 Electronic Codebook (ECB)

Electronic Codebook (ECB) is a simple block-cipher mode with parallelizable iterations like Counter (see Section 4.2.1). However, as Listing 6.5 shows, ECB does not have a counter variable and inserts `plaintext` data at the beginning of the loop, rather than at the end. Due to the simplicity of ECB's algorithm, there are not as many useful latency configurations to experiment with. The load and store in ECB are at either end of the loop, however, the modulo scheduler should place the load in a good location several intervals ahead.

Listing 6.5: AES-128 ECB Encryption loop

```
1  for(i = 0; i < blocks; i++){
2    result = _mm_xor_si128 (plaintext[i], key[0]);
3    result = _mm_aesenc_si128 (result, key[1]);
4    /* more rounds */
5    result = _mm_aesenclast_si128 (result, key[10]);
6    ciphertext[i] = result;
7  }
```

Despite the simple algorithm, AES-GEN is able to maximize the performance of ECB. At this point, all we can do is adjust the latencies up and down so that the scheduler will make slightly different generation combinations. The modulo scheduler as expected, does a good job on spreading out the load and store over several intervals. We test with the load and store taking anywhere from 6 to 10 cycles each while all the `aesenc` rounds are uniformly set at 10, 12 or 14 cycles each.

The results yielded by AES-GEN did not come very close to the reported results from Intel's hand-tuned assembly presented by Gueron [2010]. We denote these values as Intel HPL. The fastest result AES-GEN found with ECB 128 with was 1.355 c/b with a 1K input buffer. We more or less expect this result because the normal CTR implementation comes in at 1.371 c/b. As ECB has one less `xor` instruction and spreads out the load and store, this seems reasonable. We became curious as to why our results were so different to the published results as we have previously had great success with generating similar numbers in other modes. We decided to test the assembly code listed by Gueron on our own platform (this is denoted as Intel ASM).

We were unable to reproduce the numbers documented in the Intel HPL [Gueron, 2010] despite using the exact same code as listed in that work. The reported result from Intel HPL cites 1.26 c/b for ECB 128 with a 1K buffer, but running their assembly code on our testing platform (Intel ASM) ran at 1.359 c/b. Similarly, this behaviour repeated with AES-192 and AES-256 modes, as illustrated in Figure 6.4 with exact

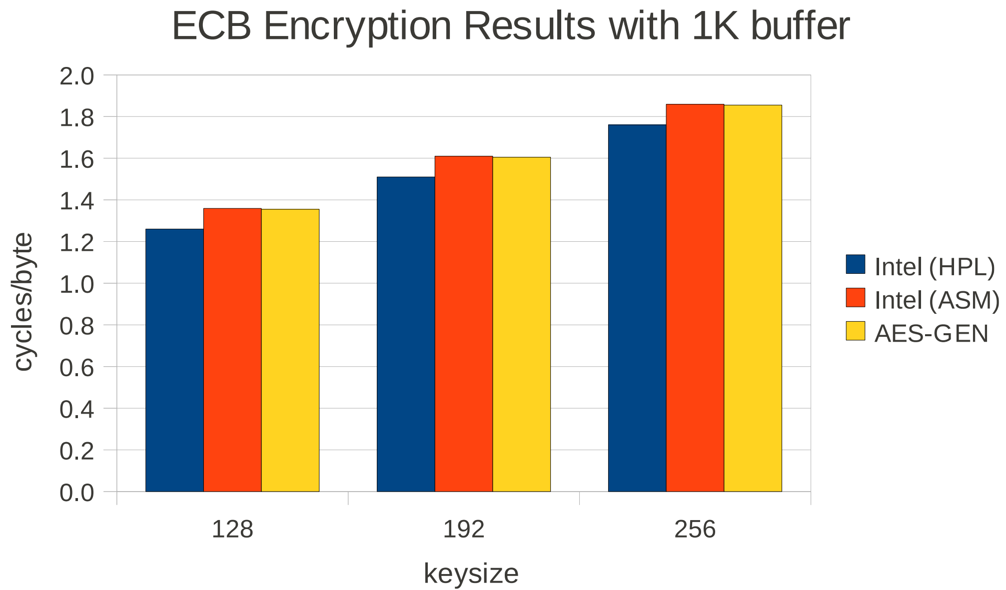


Figure 6.4: ECB Results with 1K buffer as reported by Intel [Gueron, 2010] (Intel HPL), Intel assembly code running on our testing platform (Intel ASM), and AES-GEN.

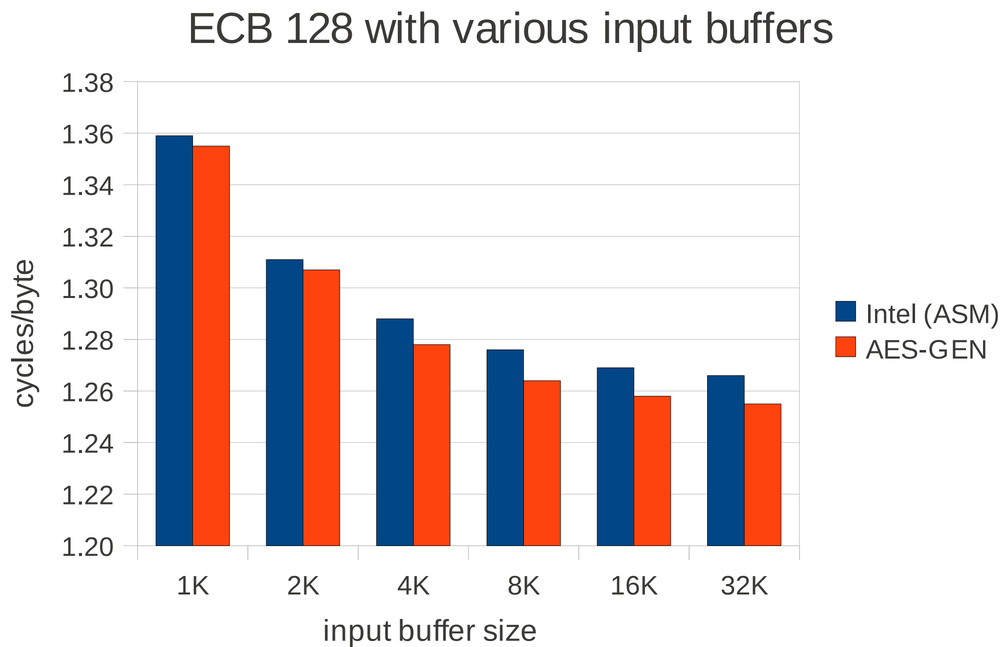


Figure 6.5: ECB 128 results with various input buffer sizes. Shown with Intel assembly code running on our testing platform (Intel ASM) and AES-GEN.

values in Table B.1. The results from using Intel ASM on our machine are in lockstep with the AES-GEN results, as expected. We believe this shows that AES-GEN is a far more portable system than hand-tuned assembly. Further proof of this is benchmarking against the assembly for different sizes. The results in Figure 6.5 show that with lower input we generate implementations with slightly better performance in comparison to how the Intel assembly runs on our machine, but our speedups increase as the input size increases as AES-GEN searches for different suitable implementations.

6.4.3 Performance Observations for CTR and ECB Modes

The CTR and ECB code AES-GEN builds shows that the same modulo scheduled code works well for both small and large input buffer sizes. With GEN1, we often saw that tests with small input sizes had smaller improvements over the results reported by Intel than experiments with large inputs. In both GEN1 and assembly implementations, the encryption loop is often unrolled. This requires a second loop to encrypt remaining blocks when the number of blocks does not divide evenly into the unrolling degree. This second loop is slow, which is a problem for very small inputs which spend a large proportion of time in the second loop. With modulo scheduling there is a single loop, which flows smoothly from one iteration to another.

With the ease of generating code variants from different source files using AES-GEN, we are able to explore optimizations very specific to each AES mode without significant rewrites as would be necessary with hand-tuned assembly. Specifically to CTR, the 16-byte counter value must be incremented for each block. If the loop is unrolled, counter increments are scheduled together as they are the first statements of each iteration. However, this creates a data dependency from one increment to the next, which slightly limits ILP. Splitting the counter into multiple variables is a possible solution [Gueron, 2010], but this increases register pressure. With modulo scheduling, the statements from multiple iterations are overlapped seamlessly and there is only a single copy of each statement within the generated loop body. For a loop such as the one for CTR with a 128-bit key schedule, where an iteration of the loop takes around 20 cycles, the cyclic data dependence from the counter increment is too small to affect the schedule.

6.5 Algorithmic Variations with Cyclic Algorithms

Unlike CTR and ECB, cyclic modes do not pipeline well for obvious reasons. The four cyclic block-ciphers are CBC, PCBC, CFB, and OFB. They are all very similar to each other. These cyclic block-ciphers encrypt the input `plaintext` directly and in order to eliminate patterns in the `ciphertext`, the output of encrypting one 16-byte block is used as input into the next block. This results in a large cyclic data dependency in the code, which prevents pipelining of the encryption. This dependence makes these modes slow and optimization is difficult. On the other hand, the difficulty of optimizing these modes make any speedup very welcome. With GEN1, we optimized by encrypting multiple streams concurrently. With AES-GEN, we take the same approach but consider algorithmic variations of the cyclic code to test new ideas for faster cyclic block-cipher performance.

6.5.1 Cipher-Block Chaining (CBC)

We are already quite familiar with CBC from the previous chapter. It first performs an `xor` on the current block's `plaintext` with the previous block's `ciphertext`. It then does a second `xor` on the result and `key0`. We implemented this in GEN1 as an optimization for CBC as mentioned in Section 5.2.3. Further investigation yielded an idea to decrease the length of the dependency chain by additionally exploiting the `xor` operation.

Figure 6.6a shows a C translation of the assembly code in the Intel hand-optimized high-performance AES encryption library [Gueron, 2010]. The code is a direct implementation of standard descriptions of CBC mode [Ehrsam et al., 1976]. It combines the result (variable `r` in the code) from the previous block of encryption with `plaintext` of the current block, using an `xor` operation. It then applies the standard AES encryption algorithm to the result. The cyclic dependency in the loop includes all statements except two: the first statement in the loop, which loads the `plaintext` and the last statement, which stores the encrypted block to memory. This long cyclic dependency prevents any significant exploitation of instruction-level parallelism in the CBC loop.

It is possible to slightly shorten the length of the cyclic dependence in the CBC loop. The first two statements in the cyclic data dependency are `xor` operations. As the `xor` operator is both commutative and associative, these expressions can be rewritten to increase the amount of exploitable ILP.

Neither the value in `plain` nor the value in `key0` depends on any value computed within the cyclic data dependence. Figure 6.6b shows a version of the code with the

load of the plaintext and the first xor operation scheduled to execute in parallel with the encryption code for the previous block.

```
while(i < size){
 p = plain[i];
 r = xor(r, p);
 r = xor(r, key0);
 r = aesenc(r, key1);
 r = aesenc(r, key2);
 /* more rounds */
 r = aesenc(r, key8);
 r = aesenc(r, key9);
 r = enclast(r, key10);
 cipher[i] = r;
 i++;
}
```

(a)

```
while(i < size){
 r = xor(r, tmp);
 r = aesenc(r, key1);
 r = aesenc(r, key2);
 /* more rounds */
 r = aesenc(r, key8);
 p = plain[i+1];
 tmp = xor(p, key0);
 r = aesenc(r, key9);
 r = enclast(r, key10);
 cipher[i] = r;
 i++;
}
```

(b)

```
while(i < size){
 r = aesenc(r, key1);
 r = aesenc(r, key2);
 /* more rounds */
 r = aesenc(r, key9);
 p = plain[i+1];
 tmp = xor(p, key0);
 fake = xor(tmp, key10);
 r = enclast(r, fake);
 correct = xor(r, tmp);
 cipher[i] = correct;
 i++;
}
```

(c)

Figure 6.6: AES CBC algorithm implementations with (a) no xor modifications, (b) partial xor optimizations, and (c) full xor modifications

With very careful programming, it is possible to further reduce the length of the cyclic dependency. Recall from Section 4.1 that the final round of AES encryption involves three steps:

1. substitute bytes,
2. shift rows,
3. add round key.

The add round key step is an xor operation, where the result of the previous steps are combined with the key using xor. All three steps are implemented with a single x86 aesenclast instruction. The result of the aesenclast is used to store the ciphertext to memory and is also fed into the next round using an xor operation. Given that the last stage of aesenclast is an xor computation, we can combine the aesenclast and xor statements.

Performing this transformation removes the second xor operation from the cyclic data dependence. However, a problem occurs because the result of the aesenclast operation is not just used in the encryption of the next block. The result is also stored to memory as the ciphertext. It is possible to repair the result coming from the aesenclast operation, at the cost of adding another xor operation to the code.

Table 6.3: Performance of AES CBC versions found in Figure 6.6 in cycles per byte.

Encryption Mode	**AES-128**		**AES-256**	
	1K buffer	32K buffer	1K buffer	32K buffer
CBC version (a)	4.539	5.315	6.042	6.815
CBC version (b)	4.156	4.067	5.656	5.567
CBC version (c)	3.851	3.759	5.351	5.259
CBC Intel (HPL)	4.15		5.65	

Figure 6.6c shows code implementing this strategy. This version of the code contains an additional `xor` operation in comparison with all the other versions, so more work must be done on each iteration of the loop. However, both original `xor` operations have been removed from the cyclic data dependence. To our knowledge, we are the first to propose this way of writing AES CBC code.

CBC XOR Performance

We compared the three different versions of the AES CBC code, each with their own strategy for dealing with the `xor` operations. Table 6.3 shows the performance of each of the three versions, alongside performance numbers from Intel's high-performance, hand-tuned assembly version [Gueron, 2010].

As these results show, removing the `xor` operations from the cyclic dependence chain makes a significant difference to performance. The version that removes both instructions from the cyclic dependence gives the best performance, even though it increases code size. Moving the load of the `plaintext` and the first `xor` operation forward affects the performance of our CBC implementations.

The influence of moving the load operation forward is particularly visible for the 32K inputs as version (a) of Figure 6.6 does not move the load and cache misses cause poor performance. Although Intel's hand-tuned assembly version of the code uses the same approach as in Figure 6.6a, it performs faster. Our version in Figure 6.6c is nonetheless faster than the assembly, because it uses a fundamentally more efficient approach.

A final question is whether we could further improve the performance of our code in Figure 6.6c by more careful programming or writing in assembly. To address this question we compute the number of cycles per round of AES encryption. Given that the code is dominated by a large cyclic dependency of AES instructions, the best we can hope to do is perform the encryption in 6 cycles per round. Additional CBC results in Table 6.6 show the performance of each version of our CBC code in cycles per round

```
while(i < size){
 p = plain[i];
 r = xor(r, p);
 r = xor(r, key0);
 r = aesenc(r, key1);
 r = aesenc(r, key2);
 /* more rounds */
 r = aesenc(r, key8);
 r = aesenc(r, key9);
 r = enclast(r, key10);
 cipher[i] = r;
 r = xor(r, p);
 i++;
}
```

(a)

```
while(i < size){
 r = aesenc(r, key1);
 r = aesenc(r, key2);
 /* more rounds */
 r = aesenc(r, key8);
 r = aesenc(r, key9);
 tmp = xor(plain[i], key0);
 tmp = xor(plain[i+1], tmp);
 fake = xor(tmp, key10);
 r = enclast(r, fake);
 correct = xor(r, tmp);
 cipher[i] = correct;
 i++;
}
```

(b)

Figure 6.7: AES PCBC algorithm implementations—(a) without `xor` modifications, (b) with `xor` modifications.

completed and suggest that our CBC version (c) is already quite close to optimal and that any possible remaining speedups from careful coding are likely to be small.

6.5.2 PCBC, CFB, and OFB

PCBC, CFB, and OFB are all additional cyclic-modes with various security benefits as was covered in Section 4.2.2. Like CBC, we also present two different versions of code for these cyclic modes. We have seen with CBC results that reducing the dependency chain as much as possible is the best idea so we present figures on both the non-`xor` version and the fully exploited `xor` versions. As the cyclic modes share many algorithm properties with each other, the `xor` optimization strategy is very similar for all the modes. Again, due to the flexibility of the generator, we can test both versions for each of these implementations simply by using another high-level source input file.

Propagating Cipher-Block Chaining (PCBC)

The Propagating Cipher-Block Chaining mode of operation is identical to CBC but has an additional `xor` of the result with the same `plaintext` *after* it has been stored as `ciphertext`. This is done before it feeds back into the start of the encryption process for the next block, as seen in Figure 6.7a. Instead of simply adding an additional `xor` to the optimized version of CBC code in Figure 6.6c, we can combine the second `xor` with `plaintext` in PCBC with the `tmp` value. When we need to correct the output for

```
while(i < size){
 p = plain[i];
 r = xor(r, p);
 r = xor(r, key0);
 r = aesenc(r, key1);
 r = aesenc(r, key2);
 /* more rounds */
 r = aesenc(r, key9);
 r = enclast(r, key10);
 cipher[i] = r;
 r = xor(r, p);
 i++;
}
```

(a)

```
while(i < size){
 r = aesenc(r, key1);
 r = aesenc(r, key2);
 /* more rounds */
 r = aesenc(r, key9);
 p = plain[i];
 tmp = xor(p, key0);
 fake = xor(tmp, key10);
 r = enclast(r, fake);
 correct = xor(r, key0);
 cipher[i] = correct;
 i++;
}
```

(b)

Figure 6.8: AES CFB algorithm implementations—(a) without `xor` modifications, (b) with `xor` modifications.

the `ciphertext`, we use `tmp` to `xor` the bits of `plaintext[i+1]` out of the result. This shortens the dependency chain and now allows the load instructions for `plaintext[i]` and `plaintext[i+1]` to be scheduled while the encryption rounds complete. The optimized PCBC algorithm can be seen in Figure 6.7b.

Cipher Feedback (CFB)

Cipher Feedback mode is similar to CBC but its implementation moves the `xor` of the result and the `plaintext` *after* the `aesenc` rounds. This can be seen in Figure 6.8a. Optimizing `xor` with CFB requires slight changes to the to the fully `xor`'d CBC code in Figure 6.6c. The main difference between the optimized CBC and CFB code is the result must be corrected, prior to storing to `ciphertext`. With the `xor` of the `plaintext` already at the end of the loop, we correct the `ciphertext` with an `xor` with `key0`, as seen in Figure 6.8b.

Output Feedback (OFB)

Output Feedback is similar to Cipher Feedback but the `xor` of the result and the `plaintext` at the end of the encryption rounds is only stored to the `ciphertext` and is not used as feedback into the next block of encryption. This can be seen in Figure 6.9a. Optimizing `xor` with OFB is different to the other cyclic modes as the dependency chain is isolated to the encryption rounds. We are still able to make optimizations by applying `xor` to both `key0` and `key10`. Since keys 0 and 10 will never change for each

```
while(i < size){
 p = plain[i];
 r = xor(r, p);
 r = xor(r, key0);
 r = aesenc(r, key1);
 r = aesenc(r, key2);
 /* more rounds */
 r = aesenc(r, key8);
 r = aesenc(r, key9);
 r = enclast(r, key10);
 cipher[i] = xor(p, r);
 i++;
}
```

(a)

```
fake = xor(key0, key10);
while(i < size){
 r = aesenc(r, key1);
 r = aesenc(r, key2);
 /* more rounds */
 r = aesenc(r, key8);
 r = aesenc(r, key9);
 p = plain[i];
 r = enclast(r, fake);
 correct = xor(r, key0);
 cipher[i] = xor(p, correct);
 i++;
}
```

(b)

Figure 6.9: AES OFB algorithm implementations—(a) without `xor` modifications, (b) with `xor` modifications.

stream of encryption, we can move this instruction outside the loop completely and treat it as a constant. Again, we need to correct the result before storing the result. We do this by performing an `xor` on `key0`. Prior to storing the data, an `xor` is down with the corrected data and the `plaintext` without affecting the result to be fed back as input to encrypt the next block. These changes can be seen in Figure 6.9b.

6.5.3 Applicability of XOR Optimizations to Other AES Modes

Note that in AES CTR mode the result of the last AES round is combined with the `plaintext` using an xor operation. The technique we use in Figure 6.6c can also be used to reverse the order of these two operations in AES CTR. This slightly reduces the length of the long chain of data dependencies in AES CTR. The code in Listings 6.3 and 6.4 show Counter in mixed mode operation which include this optimization. Given that AES CTR is fully parallelizable, there is no large-scale execution speed benefit. However, shortening the long dependence chain in CTR mode has three benefits.

First, it reduces the execution time of very short inputs. With short inputs the execution time is dominated by latency of one execution rather than throughput of many blocks. Secondly, where loop unrolling is used to implement AES-CTR, there needs to be a final sequential loop which deals with any iterations that are not an even multiple of the unrolling degree (this is one of the major reasons we do not advocate loop unrolling). Shortening the long dependence chain speeds up this loop. Finally, when creating modulo schedules, the number of iterations overlapped in the schedule

depends on the length of the dependence chain that is being pipelined. A slightly shorter dependence chain means slightly less overlapping of iterations, which means slightly less code growth and register pressure.

6.6 Combining Algorithms via Function Stitching

Many applications require both encryption and authentication of the same data. In AES, there are several modes of operation that perform this task. Galois/Counter Mode (GCM) [McGrew and Viega, 2004] combines both the aforementioned Counter mode (CTR) of encryption with Galois authentication. Counter with CBC-MAC (CCM) uses CTR for encrypting data and CBC for the authentication. Both these modes are described in more detail in Section 4.2.3.

The GCM algorithm operates by applying a GHASH function after a block of data has been encrypted in CTR mode. Due to this behaviour, there are opportunities to get speedups by overlapping the execution of both algorithms using a process called function stitching [Gopal et al., 2010a]. Function stitching is similar to loop fusion. However, where loop fusion generally takes entire selections of multiple loops and places them within a single loop body, function stitching attempts to interleave instructions from their original loop bodies to encourage ILP.

Writing code that schedules statements from two algorithms together in the same loop is tedious and time-consuming. Often the two algorithms contain different mixes of instructions, which create good opportunities for exploiting ILP between the algorithms. However, manual scheduling is always difficult. AES-GEN simplifies this problem greatly. We simply provide sequential code for the two algorithms in the same loop. The generator builds the data dependency graph and attempts to find a good software pipeline that overlaps the execution of the two algorithms.

6.6.1 Galois/Counter Mode (GCM)

Intel provides C code listings of GCM code [Gueron and Kounavis, 2010] and we have adapted both the one- (1x) and four-at-a-time (4x) implementations for use in our generator. GCM encrypts data using Counter mode and authenticates it by applying the GHASH to the encrypted blocks. The main loop body of our GCM 1x and 4x implementation both contain CTR encryption and fully inlines the GHASH function. The modulo scheduler mixes statements from both “functions” in the schedule. With larger input sizes, we are able to achieve speedups by applying AES-GEN to function

Table 6.4: Performance of AES GCM 128 with 1K, 4K, and 16K input buffers in cycles/byte.

Encryption Mode	**AES-128**		
	1K buffer	4K buffer	16K buffer
GCM 1x AES-GEN	5.175	4.892	4.883
GCM 1x Intel C	5.49	5.36	5.33
GCM 4x AES-GEN	3.964	3.597	3.505
GCM 4x Intel C	4.16	3.88	3.70
GCM 4x Intel HPL	3.85	3.60	3.54

stitched code. This can be seen in Table 6.4.

The results in Table 6.4 show that our generator is able to overlap the execution of the AES encryption code and the Galois field multiplication using modulo scheduling. As GCM is essentially CTR encryption followed by authentication, we can safely say that the GHASH function takes the majority of run time. The GHASH function has a large dependency chain but despite this, speedups are achieved in both 1x and 4x versions by pulling out several instructions from GHASH and replacing them with constants.

We improved the C input code for GCM by using associativity rules to replace sequential chains of `xor` operations with balanced trees of `xor` operations. There are a number of constants that can be computed outside the loop. Like assigning key values to variables, we introduce a mechanism to control which of these constants will be stored in memory (or in registers). Even with these improvements, it is difficult to fully match the GCM 4x performance reported by Intel [Gueron and Kounavis, 2010]. Each loop iteration contains 40 to 56 long-latency AES-NI instructions. We find variants that have slight speedups with 4K and 16K buffers, but are unable to quite match the performance of the 1K buffer. We suspect that the biggest reason for not being able to match the hand-tuned assembly version is due to the compiler. We suspect that low-level optimizations are not being exploited in full. This is a basis for one of several ideas to improve our GCM results as we mention in Section 7.2.

Further work from Gopal et al. [2010b] provided results of function stitched GCM code running two separate threads of encryption using Intel's hyper-threading (or simultaneous multithreading). As this paper documented the fastest GCM implementations known to us, we implemented a test platform to search for the GCM implementation variant that worked best with two threads running simultaneously. With very small input sizes, we were unable to come very close to their performance.

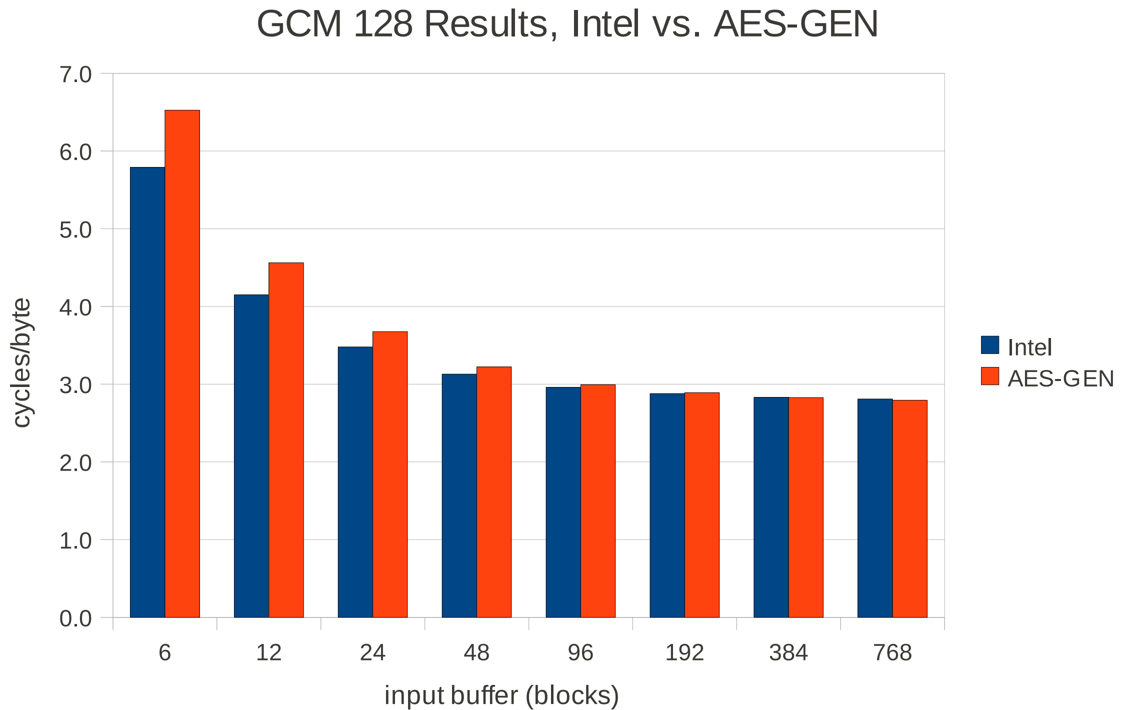


Figure 6.10: GCM 128 Results with various input sizes using simultaneous multithreading, Intel function stitching vs. AES-GEN.

We have seen that performance gains when using these AES hardware instructions are due to exploiting instruction-level parallelism. The modulo scheduler does an excellent job at generating code for this purpose. However, the good code that is often generated uses many latency intervals. If the number of intervals is 6, this requires at least six blocks of data to execute properly and would need several times the number of blocks to see any sort of speedup. This behaviour is visible in Figure 6.10. At six blocks (or 96 Bytes), AES-GEN is slower by 0.735 c/b. This difference is cut nearly in half when the input goes to 12 blocks, then over half again with 24 blocks. At 48 blocks, AES-GEN comes within 0.1 cycles a byte and eventually performs slightly better after 192 blocks. The discrete results of Figure 6.10 are listed in Table B.3 with additional SMT data found in Section 6.7.1.

6.6.2 Counter with CBC-MAC (CCM)

CCM is another authentication that combines two algorithms. With CTR being used for encryption, the authentication is done with CBC-MAC. CBC-MAC is identical to CBC but does not store `ciphertext`, as it is just trying to compute an authentication tag. In this particular mode, the amount of `aesenc` rounds necessary to encrypt and

Table 6.5: Performance of AES CCM, AES-GEN CCM vs. CBC-1 (theoretical best) in cycles/byte.

Encryption Mode	**AES-128**		**AES-256**	
	1K buffer	32K buffer	1K buffer	32K buffer
CCM	3.867	3.754	5.378	5.272
CBC-1	3.851	3.759	5.351	5.259

authenticate each block is double. For instance, with a 128 bit key size, there are 20 rounds required for each block in CCM mode. Though, as we have seen with multiple streams, the AES unit can be flooded with instructions while maintaining good performance. Since AES instructions can be issued every two cycles, we actually find that CCM will effectively be performance bound by the cyclic dependence of the CBC-MAC authentication and is unable to be pipelined well. In this case, our baseline for these instructions, would be our best results for CBC-1. The closer we come to CBC-1 results, the better our CCM performance is. These values can be compared in Table 6.5. Also in that table are cycle/round figures for CCM which are half of the CBC-1 figures. This is due to twice as many AES rounds in the encryption loop.

6.7 Experimental Results

All of the implementations generated by AES-GEN were tested on an Intel 3.2GHz Core i5 650 machine (cpu family 6, model 37), with all power management disabled, lightly loaded, and running 64-bit Linux. When comparing to Intel's performance, we denote Intel HPL as documented numbers as found in [Gueron, 2010; Gueron and Kounavis, 2010; Gopal et al., 2010b] and we denote Intel assembly code compiled and executed on our test platform (as mentioned) as Intel ASM.

All code is compiled with the Intel C Compiler (icc) with `-O2`. We found that `-O3` often slightly decreased performance—particularly on the cyclic and authentication modes. Our thoughts on why this happens and related material are found in our concluding chapter. We call our generated code from a simple timing harness that provides a random input of `plaintext`, a random key schedule, and a random initialization vector. We use the processor's time stamp counter to measure timings and report the median time of over 150,000 runs[5]. Our timings include only the encryption itself, not the key expansion. A slightly different timing mechanism is used for results

[5] See Section 5.3 for a detailed explanation and validation of using the median value. The additional experiments for other block-cipher modes conducted in this chapter also exhibit similar variance behaviour.

that use simultaneous multithreading, which is discussed later.

6.7.1 Generated Code Performance

Benchmarking AES-GEN's performance requires some thought. Abstractly, a good metric to show the usefulness of the AES-GEN system is the fact that it can generate implementations with optimizations that would normally be done by hand. However, more concretely, the results in Table 6.6 show AES-GEN to be a far better system for making optimizations automatically that would normally be done manually. We consider three main comparisons:

(a) Comparing against calculated (or theoretical) optimal results.

(b) Comparing against the fastest documented numbers of these modes.

(c) Comparing against the implementation baselines which are optimized and compiled by standard compilers.

We believe that these comparisons show multiple supporting arguments for the use of a code generator to solve the AES algorithm that use the AES-NI instructions. This is because (a) coming close to theoretical best results means good performance overall without the need for known results to compare against, (b) comparing with hand-tuned assembly shows we have some of the fastest implementations known at this time *and* that we can automate manual optimizations, and (c) shows how current tools cannot optimize as easily or as well as some might think, thus showing the need for using a code generator to solve this problem.

General Results

The results in Table 6.6 show the execution time of generated code for all implementations in cycles per byte and cycles per completed AES round—using both 128 and 256 key sizes. As mentioned, as an AES instruction can be issued every two cycles, the theoretical optimal figure for any AES implementation should be about 2 cycles/round. Of course, there are additional instructions in these algorithms, but as the `aesenc` instructions have a long-latency, we want AES-GEN to schedule these effectively. Consulting Table 6.6, we see that we come very close to 2 cycles/round for most of our implementations. Using the cycle/round figure is also helpful as the baseline number holds no matter which key size we use.

Table 6.6: Performance of AES-GEN generated code in cycles/byte and cycles/round. (*) Results for CTR (round 1) and CTR (round 2) reference 4080B, all others reference 32K.

Encryption Mode	AES-128				AES-256			
	cycles/byte		cycles/round		cycles/byte		cycles/round	
	1K	32K*	1K	32K*	1K	32K*	1K	32K*
Parallel Modes								
CTR	1.371	1.258	2.194	2.013	1.875	1.756	2.143	2.007
CTR-R1*	1.277	1.165	2.043	1.864	1.761	1.667	2.013	1.905
CTR-R2*	1.398	1.298	2.237	2.077	1.734	1.622	1.982	1.854
ECB	1.363	1.258	2.181	2.013	1.867	1.759	2.134	2.010
Cyclic Modes								
CBC1	3.851	3.753	6.162	6.005	5.351	5.253	6.115	6.003
CBC2	1.933	1.877	3.093	3.003	2.683	2.627	3.066	3.002
CBC3	1.282	1.262	2.051	2.019	1.791	1.762	2.047	2.014
CBC4	1.288	1.256	2.061	2.010	1.783	1.752	2.038	2.002
CBC5	1.288	1.270	2.061	2.032	1.784	1.757	2.039	2.008
PCBC1	3.851	3.760	6.162	6.016	5.351	5.264	6.115	6.016
PCBC2	1.933	1.877	3.093	3.003	2.684	2.627	3.067	3.002
PCBC3	1.293	1.261	2.069	2.018	1.796	1.764	2.053	2.016
PCBC4	1.286	1.256	2.058	2.010	1.822	1.787	2.082	2.042
PCBC5	1.300	1.297	2.080	2.075	1.846	1.826	2.110	2.087
CFB1	3.851	3.753	6.162	6.005	5.351	5.264	6.115	6.016
CFB2	1.928	1.877	3.085	3.003	2.680	2.627	3.002	3.002
CFB3	1.293	1.260	2.069	2.016	1.796	1.762	2.053	2.014
CFB4	1.287	1.254	2.059	2.006	1.808	1.772	2.066	2.025
CFB5	1.284	1.260	2.054	2.016	1.784	1.762	2.039	2.014
OFB1	3.851	3.760	6.162	6.016	5.531	5.265	6.321	6.017
OFB2	1.930	1.877	3.088	3.003	2.680	2.627	3.063	3.002
OFB3	1.294	1.258	2.070	2.013	1.800	1.763	2.057	2.015
OFB4	1.293	1.261	2.069	2.018	1.824	1.792	2.085	2.048
OFB5	1.312	1.299	2.099	2.078	1.784	1.787	2.039	2.042
Authentication Modes								
GCM 1x	5.175	4.825	8.280	7.720	5.574	5.262	6.370	6.014
GCM 4x	3.964	3.627	6.342	5.803	4.543	4.037	5.192	4.614
CCM	3.867	3.754	3.094	3.003	5.378	5.272	3.073	3.013

There are a few exceptions to this 2 cycle/round number. First, there are the numbers that come in less than 2 cycles/round. This was discussed in depth in Section 6.4.1, as our CTR variants were actually able to break this barrier by using a mixed-mode implementation. Secondly, there are cyclic modes that encrypt a single input buffer and run at about 6 cycles/round. This is also a calculated best. Since the rounds in cyclic mode cannot be executed in parallel, the best possible outcome with a 6 cycle latency instruction is 6 cycles per instruction—and that is what we achieve. Cyclic modes encrypting two streams halves this value, which is again expected. Thirdly, authentication modes have different theoretical bests depending on the implementation. With GCM, it is difficult to calculate a best possible result as we have noted in previous discussion that the bulk of the execution is done in the GHASH function, which is dependent on the encryption being finished. However, with CCM, we compared its performance with that of CBC1 performance. As CCM uses CBC for authentication, AES-GEN cannot generate code that performs faster than CBC1. However, it should be noted that the cycle/round figures are half what we would expect from CBC1. This is because CTR is performing the encryption and thus is using the AES unit as well.

Given these exceptions, we see that in nearly all cases, AES-GEN can generate implementations that achieve effectively-optimal performance. Regardless of input buffer size or the number of keys, AES-GEN finds near optimal solutions for nearly every mode.

Simultaneous Multithreading

It was mentioned in Section 6.6.1, that work from Intel documented very fast results in GCM mode when used with hyper-threading technology. We discussed specific result comparisons with Intel in that section, but the experiment left us curious about other SMT performance numbers for the other AES implementations. We compiled these results by modifying our current testing platform to run two instances of an implementation concurrently on separate threads that are pinned to a single core using processor affinities. Each of the two threads would execute over 150,000 times, with separate key schedules and input values randomly generated at runtime. To keep in line with the experimental setup used by Gopal et al. [2010b], we use:

$$\frac{\text{total \# of cycles}}{2 \times \text{input buffer size}}$$

to calculate cycles/byte for the SMT results. SMT results are shown in Table 6.7.

With parallel modes, we see relatively small improvements compared to scalar ex-

Table 6.7: Performance of AES-GEN generated code in cycles/byte and cycles/round in SMT mode, pinned to a single core. (*) Results for CTR (round 1) and CTR (round 2) reference 4080B, all others reference 32K.

Encryption Mode	AES-128				AES-256			
	cycles/byte		cycles/round		cycles/byte		cycles/round	
	1K	32K*	1K	32K*	1K	32K*	1K	32K*
Parallel Modes								
CTR	1.264	1.250	2.022	2.000	1.754	1.749	2.005	1.999
CTR-R1*	1.150	1.132	1.840	1.811	1.644	1.630	1.879	1.863
CTR-R2*	1.270	1.214	2.032	1.942	1.724	1.635	1.970	1.869
ECB	1.234	1.250	1.974	2.000	1.729	1.746	1.976	1.995
Cyclic Modes								
CBC1	1.941	1.878	3.106	3.005	2.691	2.627	3.075	3.002
CBC2	1.275	1.259	2.040	2.014	1.772	1.755	2.025	2.006
CBC3	1.257	1.252	2.011	2.003	1.749	1.755	1.999	2.006
CBC4	1.257	1.267	2.011	2.027	1.760	1.763	2.011	2.015
CBC5	1.261	1.285	2.018	2.056	1.758	1.766	2.009	2.018
PCBC1	1.941	1.888	3.106	3.021	2.689	2.634	3.073	3.010
PCBC2	1.278	1.259	2.045	2.014	1.751	1.763	2.001	2.015
PCBC3	1.260	1.254	2.016	2.006	1.770	1.767	2.023	2.019
PCBC4	1.263	1.263	2.021	2.021	1.769	1.774	2.022	2.027
PCBC5	1.271	1.286	2.034	2.058	1.757	1.777	2.008	2.031
CFB1	1.940	1.877	3.104	3.003	2.689	2.636	3.073	3.013
CFB2	1.273	1.259	2.037	2.014	1.772	1.757	2.025	2.008
CFB3	1.253	1.252	2.005	2.003	1.758	1.757	2.009	2.008
CFB4	1.255	1.256	2.008	2.010	1.762	1.760	2.014	2.011
CFB5	1.256	1.272	2.010	2.035	1.755	1.759	2.006	2.010
OFB1	1.939	1.885	3.102	3.016	2.701	2.635	3.087	3.011
OFB2	1.273	1.257	2.037	2.011	1.787	1.771	2.042	2.024
OFB3	1.256	1.254	2.010	2.006	1.757	1.756	2.008	2.007
OFB4	1.254	1.254	2.006	2.006	1.756	1.761	2.007	2.013
OFB5	1.257	1.271	2.011	2.034	1.751	1.764	2.001	2.016
Authentication Modes								
GCM 1x	4.111	3.979	6.578	6.366	4.586	4.381	5.241	5.007
GCM 4x	3.107	2.775	4.971	4.440	3.515	3.161	4.017	3.613
CCM	2.637	2.603	2.110	2.082	3.793	3.757	2.167	2.147

ecution. While nearly all the CTR and ECB versions run extremely close to or under 2 c/b, many patterns seen in scalar execution do not appear here. Perhaps the most curious behaviour is ECB with 32K input buffers. With SMT, AES code in both 128- and 256-bit mode showed a general pattern of larger input sizes actually taking longer to complete. With CTR-R1, our fastest implementation of Counter, we see *very* small improvements when going from 1K to 32K inputs of less than 0.02 c/b in both 128- and 256-bit mode. This starts to suggest that we have basically hit maximum possible performance and there is simply no room to improve on an already parallel, already optimized algorithm. Nonetheless, these versions still run faster than their scalar counterparts.

Cyclic modes also show some curious behaviour. In all of the cyclic modes that encrypt a single stream, we immediately see 50% reduction in runtime. Encrypting two streams with SMT also reduces runtime by a healthy yet also expected margin. What is curious is encrypting with 3–5 streams in SMT mode. These numbers only improve a mere 0.05 c/b, and in some cases, perform worse than their scalar counterparts. Consider encrypting 5 streams in CBC; running two threads of this implementation means one core must process 10 streams of encryption, each with their own set of keys, and their own input buffer. At this point, we are flooding the AES unit with work, and it is not surprising that we are unable to get any better than 2 cycles/round without making algorithmic changes, as we do with Counter.

Simultaneous multithreading is potentially interesting because it is likely it would be easier to develop a real world system that processed many different scenarios. With multiple stream cyclic modes, input sizes must be of identical size across the streams. SMT could potentially be used to encrypt different types of data in different modes. Under our system, we can easily check and optimize for that scenario. By simply re-running the code tuning mechanism and we could see how to best optimize with different modes with different input sizes or key sizes that need to be run simultaneously. We also believe these results show the flexibility of AES-GEN to generate good variants. Like the scalar results, most of these variants clock at close to 2 cycle/byte and shows the code generator's ability to generate good results in (effectively) an entire different set of implementations.

6.7.2 AES-GEN vs Hand-tuned Assembly

One of the aims of AES-GEN is to automatically generate code that makes good use of the AES-NI instructions, rather than having to write the assembly code by hand. If the generated code is as efficient as the assembly language, then we get the benefits

of maintainability, retargetability and flexibility without sacrificing performance. The results in Table 6.6 show that we have largely achieved that goal. In all cases but one, our generated code achieves performance that is at least as good as the documented assembly results. The exception is GCM, where our code is slightly slower for 1K inputs, but faster for larger input sizes (see Table 6.4).

Results in previous sections show our generator produces variants that are as fast or faster than the documented Intel hand-tuned results [Gueron, 2010; Gueron and Kounavis, 2010; Gopal et al., 2010b]. To significantly improve upon the assembly numbers in both CTR and CBC, we made algorithmic variations to the C code before we let the ILP optimizer try to find an optimal schedule. This shows the usefulness and versatility of AES-GEN in general. With CTR, we re-write the counter from a vector to a scalar value. With CBC, optimizing with additional encryption streams requires minimal effort. Retuning the code with `xor` optimizations is equally easy with our system. For GCM, we collapse the `xor` chains into trees and let the modulo scheduler interleave the encryption and GHASH function as much as possible. These high-level changes, written in C, are easy to make and Cheetah can generate these algorithmic variants—as described in Section 6.3.1—automatically.

During our discussion of the ECB assembly results in Section 6.4.2, we mentioned that we found some peculiar data when we tried to reproduce the documented Intel numbers for ECB encryption using Intel code on our machine. Running their assembly code on our machine yielded nearly the exact same cycle/byte figures for 1K input buffers as we were able to get using AES-GEN. We thought it would be interesting to see how the assembly listings in their white papers would perform on our test machine and stack them up against what AES-GEN can produce. The CTR and CBC results found in Figures 6.11 and 6.12 come from using code found in the Intel AES-NI sample library [Intel Corp., 2010a] which includes high-performance assembly listings.

Figure 6.11 displays the results of running Intel's assembly and our generated Counter code in variety of key sizes and input buffers. We see that in all key size modes, AES-GEN is able to generate faster running code compared to running the assembly on our machine. Exact numbers can be found in Table B.4. In 128-bit mode, we find an average speed up with 1K to 32K buffers to be nearly 6%. This is our highest average speedup, but also the smallest cycle/byte decrease in performance from 1K to 32K, reducing execution time by less than 0.3%. Both 192- and 256-bit modes have about a 2% reduction in execution time with larger input buffers. Our average speedup decreases slightly when using more keys. With a 192-bit key size, it drops to 5.2% and 4.6% with 256-bits. Despite this, we see speedups of over 6% with 1K input buffers for

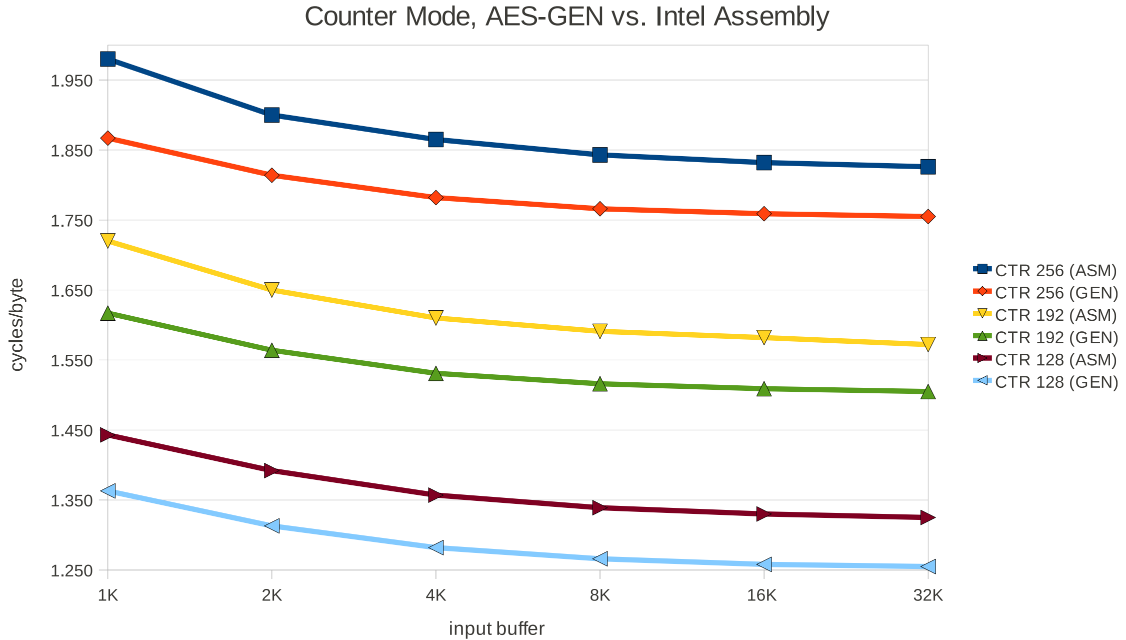


Figure 6.11: Results for Counter using normal mode operation with various key sizes and input buffers. Shown comparing AES-GEN vs. Intel Assembly running on our machine.

these two modes.

Figure 6.12 displays a similar comparison of Intel assembly and AES-GEN. In this figure, we compare the performance of single stream CBC. Again, in all modes, AES-GEN generates code that is much faster than the assembly. Further numbers are shown in Table B.5. We see very small changes in CBC performance with different input sizes in all modes while having overall speedups against the assembly version. This is once again due to the cyclic dependency. There is not much opportunity to optimize the CBC with a single stream. We see almost identical speedups comparing different input sizes for each key size mode: using 128-bits, there is an 11.8% average speedup; in 192-bit mode, there is a 10.1% average speedup; in 256-bit mode, there is an 8.7% average speedup.

With the Intel assembly listings that we could get to compile and run properly, the published numbers from Intel are much better than running the implementations they used to get those numbers on our own machine. We think this shows that a single assembly listing for a particular problem is not necessarily a good idea. Engineers spend large amounts of time crafting these implementations to run as fast as possible on a particular machine and it is clear that those numbers are not replicable on our testing platform. Using a code generator with tuning can tailor an implementation

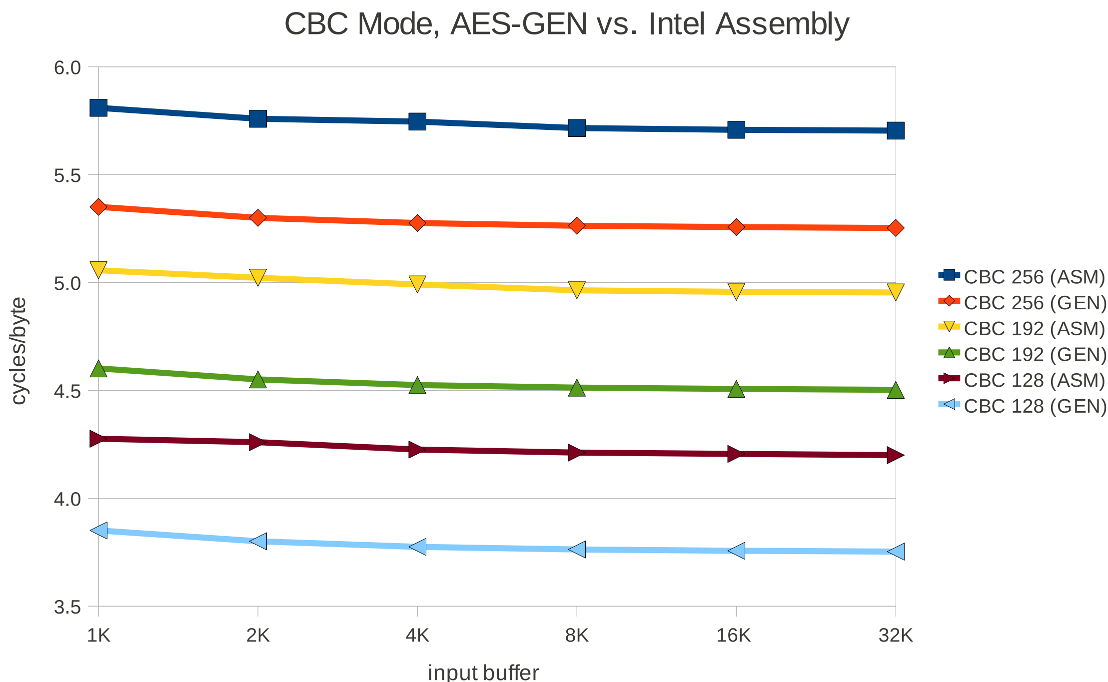


Figure 6.12: Results for single-stream CBC with no xor optimizations with various key sizes and input buffers. Shown comparing AES-GEN vs. Intel Assembly running on our machine.

without knowing a systems available, exact, real-time resources and configuration.

In many cases we are still able to find faster running implementations than the *documented* numbers as well. A pattern that tends to emerge for these assembly implementations is that they seem to be optimized for small input buffers. With GCM, we were unable to find a matching or faster 1K variant compared to Intel's, but as seen in Table 6.4 we are faster on both the 4K and 16K values documented by Intel. The code inside the encryption and authentication loop is fairly complicated and it would not be easy to manually write different versions for optimal performance. We were not able to get the GCM assembly code from Intel working on our machine.

With ECB, our results are essentially identical for 1K values across the board. However, with higher input sizes, we see once again that our implementations are faster. ECB is a very straightforward algorithm with essentially a load, a store and 10 `aesenc` instructions. While we are satisfied with matching the performance of the Intel assembly, we believe there are possible optimizations to ECB[6] that could be tested easily with AES-GEN and perhaps perform better than even documented numbers.

[6]Such as using mixed-mode or `xor` optimizations.

6.7.3 Why Not Optimize with a Standard Compiler?

Many of the techniques we use to optimize the AES implementations are standard compiler optimizations to exploit instruction-level parallelism. Most of these techniques have been implemented in compilers, particularly compilers for VLIW architectures [Fisher, 1983]. In principle, a standard compiler could do most of these optimizations. However, exploiting instruction-level parallelism is not a priority for compilers that target out-of-order architectures. We include Table 6.8 to show that a standard compiler (icc) cannot compete with our code generator. The results from both testing platforms in Table 6.8 use 1K input buffers. We find an overall average speedup of 1.37x in 128-bit mode and a 1.45x speedup in 256-bit mode.

Even a quick glance at Table 6.8 shows the large performance gap between even aggressive optimization with a standard compiler compared to AES-GEN. In the previous chapter, it was mentioned that doing selective-exhaustive searches would help us determine where to optimize the code further. Based on the final results from AES-GEN, it would be hard to predict performance based on the results obtained from using a standard compiler. We see this behaviour with several data points in Table 6.8.

If we only looked at the icc results, we would never consider using CTR (round 2) over other Counter modes. It runs very slow in comparison to the CTR-R1 code and is even slightly slower than the straight forward CTR code. However, from Table 6.6, we see that CTR (round 2) with AES-GEN gives us the fastest Counter results with a 256-bit key size with various input sizes. In addition, CTR-R2 gives us our highest speedup against a standard compiler of nearly 1.9x. There is a similar finding with cyclic modes.

For *streams* > 2 with all cyclic modes, we find that a standard compiler follows a similar trend to how AES-GEN generated code performs. AES-GEN clearly generates faster code, but it is not really obvious how what number of streams is optimal to use with either platform. This pattern has been seen in both the overall results in Table 6.6 and the SMT results in Table 6.7. This is not necessarily surprising given the cyclic dependency, but allows us to once again mention that generating good code for cyclic mode is tricky, any speedups are welcome, and issuing as many AES instructions in parallel as possible is generally a good idea.

The cyclic results also show another interesting pattern. While we could simply interleave n streams and compile them with icc, this is not likely to produce good code. Multiple stream implementations issue many AES instructions, however, if the basic block of all streams are strictly interleaved (as shown in Listing 6.6), the loads and stores of all streams will be executed in succession. Waiting for four or five loads

Table 6.8: Performance of AES-GEN generated code compared `icc -O3` compiled code in cycles/byte using 1K input buffers for all key sizes.

Encryption Mode	AES-128 1K input buffer			AES-256 1K input buffer		
	icc	gen	speedup	icc	gen	speedup
Parallel Modes						
CTR	2.000	1.371	1.459	3.273	1.875	1.746
CTR-R1*	1.828	1.277	1.431	2.902	1.761	1.648
CTR-R2*	2.160	1.398	1.545	3.285	1.734	1.894
ECB	2.023	1.363	1.484	2.695	1.867	1.443
Cyclic Modes						
CBC1	4.281	3.851	1.112	5.781	5.351	1.080
CBC2	2.258	1.933	1.168	3.029	2.683	1.129
CBC3	1.892	1.282	1.476	2.935	1.791	1.639
CBC4	1.920	1.288	1.491	2.916	1.783	1.635
CBC5	1.916	1.288	1.488	2.926	1.784	1.640
PCBC1	4.402	3.851	1.143	5.902	5.351	1.103
PCBC2	2.285	1.933	1.182	3.098	2.684	1.154
PCBC3	2.091	1.293	1.617	3.103	1.796	1.728
PCBC4	2.067	1.286	1.607	3.018	1.822	1.656
PCBC5	2.023	1.300	1.556	3.035	1.846	1.644
CFB1	4.281	3.851	1.112	5.785	5.351	1.081
CFB2	2.256	1.928	1.170	3.035	2.680	1.132
CFB3	1.953	1.293	1.510	2.978	1.796	1.658
CFB4	1.904	1.287	1.479	2.938	1.808	1.625
CFB5	1.930	1.284	1.503	2.977	1.784	1.669
OFB1	4.222	3.851	1.096	5.723	5.531	1.035
OFB2	2.152	1.930	1.115	3.033	2.680	1.132
OFB3	1.970	1.294	1.522	2.943	1.800	1.635
OFB4	1.900	1.293	1.469	2.938	1.824	1.611
OFB5	1.927	1.312	1.469	2.916	1.784	1.635
Authentication Modes						
GCM 1x	7.820	5.175	1.511	9.340	5.574	1.676
GCM 4x	4.300	3.964	1.085	4.777	4.543	1.052
CCM	4.254	3.867	1.100	5.793	5.378	1.077

Average speedup	1.37x	1.45x

Listing 6.6: Strict interleaving four streams of CFB mode

```
for(i = 0; i < blocks; i++){
  /* xor with results */
  /* encryption rounds 1 to 9 */
  st0_result = _mm_aesenclast_si128(st0_result, st0_key[10]);
  st1_result = _mm_aesenclast_si128(st1_result, st1_key[10]);
  st2_result = _mm_aesenclast_si128(st2_result, st2_key[10]);
  st3_result = _mm_aesenclast_si128(st3_result, st3_key[10]);
  st0_result = _mm_xor_si128(st0_plaintext[i], st0_result);
  st1_result = _mm_xor_si128(st1_plaintext[i], st1_result);
  st2_result = _mm_xor_si128(st2_plaintext[i], st2_result);
  st3_result = _mm_xor_si128(st3_plaintext[i], st3_result);
  st0_ciphertext[i] = st0_result;
  st1_ciphertext[i] = st1_result;
  st2_ciphertext[i] = st2_result;
  st3_ciphertext[i] = st3_result;
}
```

will significantly stall the pipeline. AES-GEN will create a data dependency graph that looks like this but, to improve the code, we define thousands of possible latency sets and let code tuning find where to best place these loads and stores.

The relatively small speedups for GCM 4x and CCM can be explained. GCM has a massive bottleneck in the GHASH function. To counteract this bottleneck, Gueron and Kounavis unrolls the loop four times to encrypt four blocks in parallel and uses a modified GHASH that computes authentication tag for the `ciphertext` of all four blocks. The GHASH function must complete before the *next* four blocks finish encrypting. AES-GEN will create an additional pipeline interval to encrypt the next four blocks as the previous four blocks' tag is being computed, but as most of the time is spent in the GHASH function, AES-GEN will not create a schedule as good as CTR-R1 or even the normal CTR code. CCM also has a cyclic dependency in computing the tag, and thus there is the expected, about the same, 10% speedup as we see for CBC1. With GCM 1x, we see a speedup of over 1.5x as we can have more pipeline intervals as the GHASH function on a single block is faster to compute than the GCM 4x version.

The standard compilation results show that AES-GEN an important tool to fully experiment with different AES implementations. Implementing efficient AES code would be much more straightforward if standard compilers incorporated good modulo schedulers. However, adding such a scheduler and other sophisticated optimizations needed for a compiler to reorder memory operations would be a significant engineering task. Out-of-order pipelines are sufficiently complex that a compiler can rarely predict,

in advance, exactly which code ordering is likely to be best, so an approach that uses experimentation is valuable. AES-GEN was built to be flexible enough to explore specific optimizations for general problems.

6.8 Related Work

The most closely related to work to ours has been from various outlets at Intel, including its hand-tuned assembly language library [Gueron, 2010; Gueron and Kounavis, 2010] for the AES-NI instructions. Both this library and our generator use standard techniques for optimizing assembly code in the presence of long-latency instructions. The main difference is that our generator automates the process of applying these and other optimizations. Although our performance is similar, the generated code is actually very different. We use modulo scheduling to execute multiple iterations together, whereas the Intel code uses loop unrolling to achieve the same goal. The Intel library code also achieves significant code size savings by combining the AES code for different key sizes.

Akdemir et al. [2010] present performance results for CBC using AES-NI on multicore processors. To achieve speedups with CBC, they utilize multiple cores and multiple threads. They also present a method for overlapping the execution of the key expansion with the encryption. Our implementations do not consider this technique, but would be worth looking into as our attempts with function stitching with GCM provided excellent results.

Gopal et al. [2010a] proposed using function stitching as cryptographic applications often process pairs of independent algorithms. In AES GCM, for example, both encryption and authentication algorithms are serially executed. In their work, they present several different stitching techniques, among them stitching CBC with SHA-1 [Eastlake and Jones, 2001] which are executed in parallel within a single composite function to achieve better speeds—often at the speed of only the longer executing portion. Gopal et al. [2010b] optimized GCM further in a white paper released while our paper was under review by improving the GHASH function. They treat constants differently while encrypting four blocks at a time. We have implemented a similar technique as an algorithmic variant to assign any permutation of constants used in the GHASH function to registers.

6.9 Conclusion

AES-GEN is used to generate code that uses the Intel AES instructions. The generator reads in annotated source code and uses iterative methods to try to find a variant of the source code which executes fast on the target hardware. The generator is also a useful experimental tool for programmers. The ability to make small, exploratory changes to an algorithm that can be easily scheduled differently and quickly evaluated is extremely valuable.

Our results show that AES-GEN creates faster code without a massive increase in code size, due to its good modulo scheduler. We have implemented several algorithmic variants of AES CTR, CBC and GCM modes. Our standard implementations of AES counter perform almost exactly the same as the hand-tuned assembly code, which is a good result. However, we also discuss additional implementations that are faster than anything published by Intel; to our knowledge, we have the fastest cycle per round implementations of AES CTR and AES CBC on the Westmere architecture.

With AES-GEN, it is easy for us to generate CBC code that operates on any number of input streams, as we simply make those changes in high-level C code and re-run the generator. We also presented dependency reduction strategies for the cyclic-dependent algorithm by exploiting the properties of the `xor` operation. These strategies yielded good results and testing these changes would have been a time-consuming task if one were to make the changes in assembly.

Similarly with GCM, we were able to try a number of techniques in an attempt to improve GCM code generation with minimal effort. Trying to optimize multiple algorithms running in the same loop body as GCM does would be tedious in assembly. AES-GEN's modulo scheduler is very effective at building good function stitched code.

Our goal was to maximize the performance of the Intel AES-NI instructions. We believe that we have done that. AES-GEN generates many variations of the AES algorithm and a good solution tuned to any particular architecture that supports AES-NI is found quickly.

Chapter 7

Final Thoughts

In the previous chapter, we presented a system that generated many variations of AES code with excellent results. However, are our results worth anything if the implementations are secure to use? We dedicate some discussion to the security issues in Section 7.1. As we found at the conclusion of our first generator in Chapter 5, AES-GEN also has limitations when generating good code. In Section 7.2 of this chapter, we present some ideas to increase the functionality to AES-GEN that could compete with hand-tuned assembly more closely. We also discuss the potential of using our system in a more generalized context in Section 7.3. Section 7.4 contains an assessment of our contributions that we believe support our thesis. We conclude the chapter, and this dissertation, in Section 7.5.

7.1 Security

While this dissertation focuses on the using AES-New Instructions effectively, this particular instruction-set extension is used for cryptography, and this means there are potential security issues. As mentioned in Section 5.3.1, the cycles/byte unit can be used to see which modes run the fastest overall. This can possibly come at the cost of security—using 128-bit key is much faster than using a 256-bit key, but a 256-bit key takes longer to break. While this dissertation makes no claim of expertise in security benefits or weaknesses of AES, a few key points should be discussed to alleviate security concerns. Most of the security issues surrounding our work is inherited by the Intel AES instructions.

In Gueron [2010], the security issues of Intel AES-NI are addressed. The use of on-chip, dedicated AES instructions actually mitigate several types of attacks. Software side channel attacks are common when targeting a multi-tasking platform (such

as an x86 processor). Because cryptography processing would be shared with multiple processes on common processors, information can unintentionally "leak" and allow unauthorized processes to gain insight on memory access patterns or execution flow. The long execution and processing times of scalar operations can encourage cache timing attacks. Fast software implementations also use lookup tables which span multiple cache lines. The cache timing attacks continually perform data reads to fill the cache with its own data. It then measures the latency of these read operations and can identify which cache lines which have been evicted by the encryption process running in parallel. Analyzing this data can be used to determine which parts of the lookup tables are being used. This information would then be used to determine the secret key.

AES-NI is designed to prevent these types of attacks. This is because the latency of the encode and decode instructions are data independent[1]. No lookup tables are required as all computations used in the AES are done on-chip. Our work uses these instructions "properly" and generating them in different combinations do not affect the security strengths as claimed by Intel. However, we present one caveat. For our CTR (round 1) and (round 2) modes, we use a mixed-mode strategy to implement "faster than optimal" CTR solutions. This was done by instructing 1 or 2 AES rounds to be done using conventional lookup tables. There is a potential danger in using lookup tables. These implementations *can* leak data. We have no proof exactly how much of a risk there would be using these implementations, as only 1 or 2 rounds would be using memory operations. However, if these modes were found to be unsafe, they can still be useful on non-multi-tasking platforms. These modes could be used for isolated platforms, such as embedded systems.

7.2 Future Work

In Section 6.3, we described AES-GEN as a multi-stage system that generates AES implementations. It uses a combination of algorithmic choices, ILP optimizations, and an adapted simulated annealing algorithm to find near optimal solutions. As such, there are multiple points for potential improvement to this automated program generator.

[1] This is shown by cycle/round results in Chapter 6

7.2.1 High-level Choices

When GEN1 and AES-GEN returned AES implementations that achieved similar performance to Intel's documented numbers, we were satisfied with the results. This showed that we could automate the techniques used by assembly programmers to achieve comparable performance. However, as we mention in Section 6.4.1, generating even faster code required high-level algorithm modifications. Similarly, achieving noticeably better results for the cyclic dependent block ciphers required a high-level modification as mentioned in Section 6.5.

GCM is a "function stitched" algorithm. We would have liked to have taken more time to explore possible algorithmic choices to GCM to increase performance under our system. While we did make some changes, as described in Section 6.6.1, our results show that AES-GEN produces GCM code that runs slightly slower than those published by Gueron and Kounavis [2010] and Gopal et al. [2010b] with using some small input sizes.

The authentication tag used in GCM has a cyclic dependency from one block to the next[2] due to the GHASH function. AES-GEN cannot schedule the GHASH code effectively. When AES-GEN attempts to look for a solution, it includes most of this code as a giant block without scheduling it over several pipeline intervals. However, it does interleave GHASH instructions with Counter instructions. We believe that a closer look at using AES-NI with the GHASH function will yield some opportunities for faster code. Luckily, AES-GEN is incredibly flexible in taking exploratory approaches with high-level algorithmic change.

Petabricks [Ansel et al., 2009] is a language for describing algorithmic choices, but it works by allowing multiple definitions of functions. Our work has shown that for generating AES code, the algorithmic choices need to be described at a much finer grain. If we were to simply define multiple versions of the AES functions, we would need to provide many thousands (or more) versions. An interesting problem is how fine grained algorithmic choice might be described in a language similar to Petabricks.

7.2.2 ILP Optimizations

Our program generator process takes input source code, makes algorithmic changes, applies ILP optimizations and schedules code, which is then compiled by a standard compiler. We found the quality of assembly code generated with both gcc and icc was somewhat poor. While we still achieve good results in Section 6.7, they show that

[2] In GCM 4x, the dependency is every four blocks

there is very little performance opportunity to be gained. However, when compiling code that used software pipelining, we noticed assembly code included extra copies that we would have thought to be removed by the compiler. We would like greater control of the assembly code. A tool like MAO: An Extensible Micro-Architectural Optimizer [Thuresson, 2010] could be useful in helping to gain better control of the assembly. MAO takes basic blocks in assembly, converts them to an IR, applies low-level optimizations and returns transformed assembly code.

"Sandy Bridge"

It would be interesting to see how AES-GEN produces code when targeting a different microarchitecture. Attached in Appendix D are two tables of results that show the performance of AES-GEN on the "Sandy Bridge" microarchitecture[3]. Sandy Bridge also includes AES-NI, but documents the `aesenc` instruction at a very different 8 cycle latency and 1 cycle throughput [Intel Corp., 2011]. This hardware change produces quite different results compared to the Westmere architecture (results which are well documented throughout this dissertation). The overall running times (cycles/byte) are much better on Sandy Bridge, but as they are preliminary results, based on little changes to the AES-GEN system, built on experience with Westmere, the results are preliminary. We do, however, believe these to be very good results, but without direct comparisons from Intel, we cannot say with any certainty how good they are. We can say, that according to Intel's optimization manual [Intel Corp., 2011], maximizing CTR performance requires a new assembly implementation which now unrolls the loop 8 times. To find our solution, we simply ran the generator on the new system to find an optimal variant.

It would also be interesting to see how AES-GEN produces code when targeting a very different microarchitecture. We have shown AES-GEN to work well on the **out-of-order** Westmere Core i5 and probably well on Sandy Bridge. However, AES-NI could possibly be included on new generations of Intel Atom processors. The Atom also implements simultaneous multithreading, but uses a 16 stage **in-order** pipeline. Targeting an in-order execution processor would show that our contributions are important, as the system should be able to adapt to the new architecture easily.

[3]These results were compiled after the initial submission of this dissertation, due to hardware not being released until January 2011. Table D.1 shows normal execution results, while Table D.2 shows code running in SMT mode. The test setup is the very similar to the setup mentioned in Section 6.7, but the clock speed is slower at 2.1ghz.

7.2.3 Traversing the Search Space

Benchmarking every possible solution that AES-GEN can generate is computationally infeasible. In order to reduce the time, AES-GEN uses an adapted simulated annealing algorithm to find a solution. Again, we achieve good results so it is hard to criticize a system that works well. The "trace" logs show what parameter is changed during each cooling step, the current set of optimizations and how fast they run, and whether or not it has been accepted or rejected by the algorithm as a good solution. Simulated annealing, as described in Section 2.3.3, involves a random element that affects the decision to accept or reject. This characteristic means that starting with the same source file and the same set of tuning arguments is likely to return a very different trace—but ultimately a near optimal result as we have shown.

We modify the traditional simulated annealing algorithm to keep track of a global best. This was necessary as the runtime differences between solutions can be 10ths or 100ths of a cycle per byte. This ends up causing many of the 1500 variations created each time AES-GEN is run to be rejected; this causes the search to become stuck in local minima. We introduced a second change to the algorithm to counteract this behaviour which we did by forcing the system to "jump" back to one of the global solutions if nothing had improved after a certain number of steps.

Small additions to the simulated annealing algorithm could increase the number of good solutions. This could be accomplished by the following:

- **Establishing Ranges** — The trace of annealing Counter code shows that good initiation interval value ranges are established quickly. When the *ii* value is outside the optimal range, other minor, but important optimizations could be turned on to generate good code. The annealing process often stays in these local minima when it would preferable for it to explore applying these minor optimizations inside the good *ii* range. Detecting patterns and establishing these good ranges in an effort to further optimize code would be a key improvement. This could possibly be implemented by assigning argument weights dynamically.

- **Smarter jumps** — As the search progresses, the annealing algorithm changes the argument set to a neighbouring solution. What we found while investigating the traces was that despite the functionality to revert back to a good solution if progress has not been made, it often just retraces the bad steps. When these jumps happen, it would be beneficial for a different step size or a different weight to ensure that this path is not retraced.

- **Additional tuning sets** — In some cases, we generated code that gcc and icc could not further improve. As we noted in Section 6.7, compiling with `-O2` often yielded better performance than compiling with `-O3`. Because of this observation, including an additional step may further improve compiled code. It could be included as a separate argument set used by the code tuning mechanism. These could be done similarly to the aforementioned Acovea [Ladd, 2009], which tries different combinations of compiler flags to find an optimal set.

These techniques would make simulated annealing a "smarter" algorithm. It would prevent a large number of solutions from being rejected while exploring greater portions of the search space. The combination of these simulated annealing improvements and adding additional argument sets for both compiler flags and/or post-compiled optimizations would create a much stronger system. This stronger system could also be slightly adapted to generate optimal solutions for general applications. As mentioned in Section 2.3.2, there is a large body of research that explores using learning algorithms to generate code. It would also be interesting to try different learning algorithms with our system to see if we could find solutions in shorter time-frames. Currently, our generator explores 1500 variants per pass. However, using simulated annealing works well. It returns near optimal solutions in a relatively short time frame.

7.3 Applicability to Other Applications

Fast implementation of problems other than AES often rely on high-performance libraries that are tuned to specific architectures. These are often highly-optimized, hand-tuned assembly routines which cause significant software engineering issues. Maintaining assembly language routines is difficult, because even small changes in assembly routines may require a substantial rewrite, even if only to redo the allocation of registers. Also, when new versions of the microarchitecture appear, existing routines may be sub-optimal, and new versions may need to be built. In addition, when multiple versions of assembly routines exist the maintenance problem becomes much larger. One approach to this problem that has been successful is to build program generators that automatically tune the generated code to the microarchitecture.

7.3.1 Generality

In this dissertation, we found that using a program generator to find good AES implementations was a successful approach. It seamlessly merges three techniques to

generate good code. From the experience gained compiling this work, we believe that this strategy can be extended to a wider range of applications. What we propose is a generalized system that could do iterative instruction scheduling, while considering algorithmic choices.

Algorithmic choices are problem-specific choices that the programmer makes. It is often possible to make small changes in the algorithm, or even use a completely different algorithm to solve a problem. For example, there are many sorting algorithms and it is often faster to switch between algorithms depending on the size of data and the machine architecture. Sorting program generators exist which select good algorithms and switch-over points for a particular machine [Li et al., 2004; Bida and Toledo, 2007]. An important choice appears in CTR code, where the programmer must chose between several different strategies for incrementing the counter. A complication arises because the counter is a big-endian value, but Intel architectures are little-endian, and one must choose how to address this problem. Exploring this choice was crucial in finding the best CTR code as found in Section 6.4.1. Generalizing the choices that could be made by the generator is a small engineering task that could prove very interesting.

One of the major results of this dissertation is a "proof of concept". The concept is a program generator can be used to effectively optimize an algorithm (AES) that uses instruction-set extensions (AES-NI) to maximize performance. There is another dissertation worth of research that could be done applying our concept to more general algorithms. Specifically, programs that contain loops that contain long-latency and varying throughput times would be an ideal candidate to run through the generator. The ultimate goal of AES-GEN is to exploit as much ILP as possible by scheduling independent groups of instructions close together. There is potential in the ILP optimizer alone for easy rescheduling of multiple algorithms in a single loop.

Even more specifically, digital Signal Processing (DSP) problems and floating-point intensive algorithms can usually make use of advanced multimedia functions (which are instruction-set extensions) found on common x86 processors. Many of the instructions, found in SSE4.1 and SSE4.2 for example, have long latency, but fast throughput times [Intel Corp., 2011]. These streaming extensions are included to improve DSP and floating-point operations with dedicated hardware in a fraction of the time it would take using just scalar operations (just like AES). The behaviour of these algorithms (and the instructions they use) could be similar enough to AES. One would have to extend our program generation techniques to support these problems effectively. However, this dissertation focuses on improving AES implementations and we have no results to unequivocally say that this system can be applied to any problem that

uses instruction-set extensions. Ultimately, our system is very generalized, and an interesting followup to this research will be to find suitable algorithms to test it with. The high-level choices enable exploratory approaches to be take, but the choices cannot be used effectively without a good schedule.

7.3.2 Instruction Scheduling for Out-of-Order Architectures

Instruction scheduling is a well-studied problem, and a wealth of very sophisticated compiler techniques has been developed for simple basic blocks, for loops, and for more general code. These techniques have been developed primarily for VLIW architectures, where resource constraints are very well defined. Instruction scheduling is also used by compilers for out-of-order superscalar architectures, to help the processor exploit instruction-level parallelism. Our generator builds quite complex schedules, using techniques such as modulo scheduling, reordering memory operations and scheduling operations speculatively. These sorts of schedules are much more complex than the basic block scheduling found in most compilers. We found that this iterative approach to instruction scheduling has some significant advantages for the Intel processor we used in our experiments.

The first major advantage of iterative instruction scheduling is that a good model of the execution resources of the machine is not needed. The detailed operation of modern out-of-order processors is often not well documented. The broad scheduling rules can be found in Intel manuals. However, there may be special cases and exceptions that are simply undocumented. More importantly, the program generator can adapt itself to future, unknown architectures without modification.

A second advantage of iterative instruction scheduling arises with out-of-order processors. Out-of-order processors dynamically reorder and schedule instructions and rename registers within the reorder buffer. In other words, out-of-order processors do a lot of the same things that our program generator is trying to do. This is important because instruction schedulers always try to reorder instructions so that fully independent instructions are scheduled together. However, re-ordering instructions in software is not free. It usually increases register pressure and techniques such as modulo scheduling and global instruction scheduling cause code growth. On an out-of-order processor, it is not always necessary to schedule independent instructions together. We just need to schedule them close enough to each other so the out-of-order logic will execute them together. We know of no other compiler approach to instruction scheduling that is able to figure out how close together instructions need to be for them to be executed in parallel because out-of-order processors are too complex to easily analyze

such things. But an iterative scheduling approach can, by trial and error, find schedules that bring instructions close enough for parallel execution, while minimizing the costs of reordering instructions in software.

A third advantage of iterative instruction scheduling is that it can adapt to different execution environments, such as processors with simultaneous multi-threading. When tuning an iterative code generator, we run many variations of the code on the processor and measure the performance. If we plan to run the code in parallel with other known code on a SMT processor, we could run that other code during training. This will allow the generated code to adapt itself to the behaviour and resource usages of the other code[4]. The result may not be perfect, because the behaviour of either or both codes may depend on when they start and on user input. But it is possible that generated code can adapt itself to running efficiently alongside the other code. Furthermore, we can also take into account the running time of the other code, and try to find a variation of the generated code that has minimum impact on the other code. We know of no other instruction scheduling strategy that can adapt itself to SMT execution.

7.4 Assessment of Contributions

In Chapter 3, we explored the idea of automatic vector code generation for general purpose processors. This dissertation deals with using code generation systems to optimize instruction-set extensions. We found using streaming languages as a framework to generate code that used SIMD instructions was an effective solution. But, what we thought was a robust environment to work with, we found that better program generation techniques could be found by focusing on a particular algorithm—AES. We built a "static" generator to implement two modes, but eventually developed AES-GEN which is flexible and robust to high-level source code changes.

Being able re-generate a solution from high-level changes to the source code has some crucial advantages. Exploring new AES implementations is easy. In Chapter 5, we first introduced the "local keys" concept. The generators can assign very specific subsets of the round keys to registers. We found that the effect of these configurations can make small, but important reductions in runtime. There are 2^{15} possible local key configurations and this explodes the search space. Also in Chapter 5, we showed how easily we could generate good code with a set number of streams for CBC mode.

[4]We when benchmarked encryption using SMT, we used two copies of the same code; it could easily be adapted to test different AES implementations (or different input sizes, key sizes, etc.) at once.

Adapting assembly code to use multiple streams is very difficult. Intel's assembly library includes code listings for CBC that use 1 and 4 streams. Intel's CBC-4 numbers run close to 2 cycles per round. However, we later showed in Section 6.7 that, with our optimizations, using 3 streams is actually enough to achieve the "optimal" 2 cycles/round figure. These types of algorithmic choices are just not easy to explore with assembly implementations.

Another very important algorithmic choice the system considers is exploiting the properties of the `xor` operator. This is an important contribution as cyclic modes have a long dependency chain of operations from one block to the next and any speedup is desirable. While multiple streams can be used to reduce cycle per byte times, single stream implementations are more practical. Obtaining any speedup in these modes is desirable. We improve single stream[5] execution of the cyclic modes by combining `xor` operations that load each block of `plaintext` with the final round of the AES algorithm. To our knowledge, we are the first to propose this strategy, as detailed in Section 6.5. Results in Table 6.3 showed that fully exploiting this property with CBC achieved a 1.17x speedup in 128-bit mode, and a 1.13x speedup in 256-bit mode over not applying `xor` optimizations. In correspondence with Vinodh Gopal[6] at Intel, he related to us that he believes we have the fastest known CBC implementations using AES-NI. However, just like all the algorithmic changes mentioned in this dissertation, encoding these choices at low-level has unknown effects for the rest of the assembly. Making these choices will affect the way the code is scheduled.

Another important technique critical to generating code AES implementations is generating a good schedule for the code. In Chapter 5, we exploited ILP by interleaving blocks and software pipelining in CTR mode and interleaving streams in CBC mode. Scheduling loads and stores apart from each other in both modes was important to achieve good performance. In Chapter 6, we improved this technique with an ILP optimizer, as detailed in Section 6.3.2. This ILP optimizer used modulo scheduling to generate code within the loop and allowed much greater flexibility on the placement of instructions. With this technique we were able to quickly generate schedules for our mixed-mode CTR code. The Counter implementations, shown in Listings 6.3 and 6.4, mixed scalar table lookup-based instructions with the AES-NI instructions. The ILP optimizer was able to schedule the memory operations required by table lookups effectively with the remaining rounds that used AES-NI instructions. These CTR implementations yielded our fastest CTR numbers[7].

[5] We also improve multiple streams through the usage of `xor`, but the impact is less noticeable.
[6] His work has been cited throughout this dissertation: [Gopal et al., 2010a,b; Akdemir et al., 2010].
[7] To our knowledge, they are also the fastest published CTR implementations.

Scheduling code for function stitched loops has also been shown to be a viable strategy with the ILP optimizer. With combined authentication/encryption modes, such as GCM and CCM, two functions that would normally be placed in separate loops are fused together and must be scheduled together accordingly. Fusing these functions together in assembly is difficult, but has been done for GCM in work by Gueron and Kounavis [2010]; Gopal et al. [2010a]. The GCM code from Gueron and Kounavis [2010] unrolls the loop four times and places the GHASH function after the encryption is done. Our ILP optimizer pipelines some of the Counter encryption code while interleaving it with the GHASH code. The performance we achieve from using the ILP optimizer *and* the algorithmic choices could not be achieved without the use of a code tuning algorithm.

The generators in Chapter 5 and 6 used an adapted simulated annealing algorithm to traverse the search space. We found that using selective-exhaustive searches could be used to quickly test whether optimizations were worth investigating. However, even selective-exhaustive searches test 40,000+ implementations at time. Benchmarking that many variants takes hours to complete and generally does not return a near optimal result. Using simulated annealing on very small subsets of only 1500 variants returned solutions in a fraction of the time of selective-exhaustive searches. By tuning both major and minor optimizations to each AES implementation, we showed that seamlessly integrating these three techniques into a program generator is a suitable alternative to generating assembly code.

7.5 Conclusion

Using AES-NI instructions effectively is not a trivial process. Obtaining good performance from the instructions requires the use of hand-tuned high-performance assembly libraries. The libraries, provided by Gueron [2010]; Gueron and Kounavis [2010]; Intel Corp. [2010a], use a simple schedule to achieve good performance from the AES-NI instructions. It appears that the code runs well on out-of-order architecture, but it remains to be seen if performance would hold on different architectures. Assembly code can require significant modifications as new hardware emerges, which is an expensive and time-consuming task. This dissertation argues that a program generator can be a suitable alternative to generating assembly code for AES implementations. In Chapter 5, we were able to produce similar performing code to hand-tuned assembly for both CTR and CBC code. In Chapter 6, we expanded the generator to implement additional block cipher modes to, in some cases, build superior code to assembly.

Chapter 7: Final Thoughts

This dissertation showed that a program generator is an effective solution for building near optimal AES implementations that use AES-NI instructions. A program generator accomplishes this by using three important techniques: algorithmic choices, easily exploiting instruction-level parallelism, and code tuning. The flexibility of the generator allowed high-level changes to the code to be explored as possible optimizations, while instruction-level parallelism was exploited through the use of a good scheduler that targeted a number of execution environments. Work on optimal scheduling is usually reserved for in-order architectures. To our knowledge, there is no literature that deals with finding optimal schedules on out-of-order architectures as they are too complex for analysis since they dynamically reorder instructions. Independent instructions only need to be grouped "close together" to exploit ILP. This sums up why our program generator is so important. Our program generator applies many different optimizations to a single piece of code without the need to profile the target system. As machines with special purpose instructions such as AES-NI become more widespread, our techniques are likely to become increasingly important for generating high-performance code. A program generator produces fast code in a short time.

Bibliography

Aarts, E. and J. Korst (1988). Simulated Annealing and Boltzmann machines.

Advanced Micro Devices, Inc. (2007, November). *AMD Brook+*. Advanced Micro Devices, Inc. `http://ati.amd.com/technology/streamcomputing/AMD-Brookplus.pdf`.

Akdemir, K., M. Dixon, W. Feghali, P. Fay, V. Gopal, J. Guilford, E. Ozturk, G. Wolrich, and R. Zohar (2010). *Breakthrough AES Performance with Intel AES New Instructions (White Paper)*. Intel Corp. `http://software.intel.com/file/27067`.

Algredo-Badillo, I., C. Feregrino-Uribe, and R. Cumplido (2006). Design and Implementation of an FPGA-Based 1.452-Gbps Non-pipelined AES Architecture. In M. Gavrilova, O. Gervasi, V. Kumar, C. Tan, D. Taniar, A. Lagan, Y. Mun, and H. Choo (Eds.), *Computational Science and Its Applications - ICCSA 2006*, Volume 3982 of *Lecture Notes in Computer Science*, pp. 456–465. Springer Berlin / Heidelberg.

Allen, R. and K. Kennedy (1987). Automatic Translation of FORTRAN Programs to Vector Form. *ACM Transactions on Programming Languages and Systems 9*, 491–542.

Amarasinghe, S. (2006, November). *StreamIt A Programming Language for the Era of Multicores.*

Amarasinghe, S., M. I. Gordon, M. Karczmarek, J. Lin, D. Maze, R. M. Rabbah, and W. Thies (2005). Language and Compiler Design for Streaming Applications. *Int. J. Parallel Program. 33*(2), 261–278.

Ansel, J., C. Chan, Y. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe (2009). PetaBricks: a language and compiler for algorithmic choice. *ACM SIGPLAN Notices 44*(6), 38–49.

Bartolini, S., R. Giorgi, and E. Martinelli (2009). Instruction Set Extensions for Cryptographic Applications. In . K. Ko (Ed.), *Cryptographic Engineering*, pp. 191–233. Springer US.

Benadjila, R., O. Billet, S. Gueron, and M. J. B. Robshaw (2009). The Intel AES Instructions Set and the SHA-3 Candidates. In *Advances in Cryptology - ASIACRYPT 2009*, Lecture Notes in Computer Science 5912, pp. 162–178. Springer Verlag.

Bernstein, D. J. and P. Schwabe (2008). New AES Software Speed Records. In *INDOCRYPT '08: Proceedings of the 9th International Conference on Cryptology in India*, Berlin, Heidelberg, pp. 322–336. Springer-Verlag.

Bertoni, G., L. Breveglieri, P. Fragneto, M. Macchetti, and S. Marchesin (2003). Efficient Software Implementation of AES on 32-Bit Platforms. In *CHES '02: Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, London, UK, pp. 159–171. Springer-Verlag.

Bertoni, G. M., L. Breveglieri, F. Roberto, and F. Regazzoni (2006). Speeding Up AES By Extending a 32 bit Processor Instruction Set. *Application-Specific Systems, Architectures and Processors, IEEE International Conference on 0*, 275–282.

Bida, E. and S. Toledo (2007). An automatically-tuned sorting library. *Software: Practice and Experience 37*(11), 1161–1192.

Buck, I., T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan (2004). Brook for GPUs: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, New York, NY, USA, pp. 777–786. ACM.

Cytron, R., J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst. 13*(4), 451–490.

Daemen, J. and V. Rijmen (2000). The Block Cipher Rijndael. In *CARDIS '98: Proceedings of the The International Conference on Smart Card Research and Applications*, London, UK, pp. 277–284. Springer-Verlag.

Daemen, J. and V. Rijmen (2002). *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag.

Das, A., W. J. Dally, and P. Mattson (2006). Compiling for stream processing. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, New York, NY, USA, pp. 33–42. ACM.

Davidson, J. W. and C. W. Fraser (1984). Automatic generation of peephole optimizations. In *Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, SIGPLAN '84, New York, NY, USA, pp. 111–116. ACM.

(DES), D. E. S. (1977). FIPS PUB 46-3. *US NBS*.

Diffie, W. and M. Hellman (1979, March). Privacy and authentication: An introduction to cryptography. *Proceedings of the IEEE 67*(3), 397 – 427.

Dworkin, M. (2001). Recommendation for Block Cipher Modes of Operation. NIST Special Publication 800-38A. `http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf`.

Dworkin, M. (2005). Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication . NIST Special Publication 800-38B. `http://csrc.nist.gov/publications/nistpubs/800-38B/SP_800-38B.pdf`.

Dworkin, M. (2007a). Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. NIST Special Publication 800-38D. `http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf`.

Dworkin, M. (2007b). Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality . NIST Special Publication 800-38C. `http://csrc.nist.gov/publications/nistpubs/800-38C/SP800-38C_updated-July20_2007.pdf`.

Dworkin, M. (2010). Recommendation for Block Cipher Modes of Operation: The XTS-AES Mode for Confidentiality on Storage Devices. NIST Special Publication 800-38E. `http://csrc.nist.gov/publications/nistpubs/800-38E/nist-sp-800-38E.pdf`.

Eastlake, D. E. and P. E. Jones (2001). US Secure Hash Algorithm 1 (SHA1). `http://www.ietf.org/rfc/rfc3174.txt?number=3174`.

Eggers, S., J. Emer, H. Leby, J. Lo, R. Stamm, and D. Tullsen (1997). Simultaneous multithreading: a platform for next-generation processors. *Micro, IEEE 17*(5), 12 –19.

Ehrsam, W., C. Meyer, J. Smith, and W. Tuchman (1978, February 14). Message verification and transmission error detection by block chaining. US Patent 4,074,066.

Ehrsam, W. F., C. H. W. Meyer, R. L. Powers, J. L. Smith, and W. L. Tuchman (1976, 06). Product block cipher system for data security. Patent. US 3962539.

Fisher, J. A. (1983). Very Long Instruction Word architectures and the ELI-512. In *ISCA '83: Proceedings of the 10th annual international symposium on Computer architecture*, New York, NY, USA, pp. 140–150. ACM.

Flynn, M. (1972). Some Computer Organizations and Their Effectiveness. *IEEE Trans. Comput. C-21*, 948+.

Franchetti, F. and M. Püschel (2003). Short Vector Code Generation for the Discrete Fourier Transform. In *International Parallel and Distributed Processing Symposium (IPDPS)*.

Franchetti, F. and M. Püschel (2007). SIMD Vectorization of Non-Two-Power Sized FFTs. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Volume 2, pp. II–17.

Fraser, C. W. and A. L. Wendt (1988). Automatic generation of fast optimizing code generators. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, New York, NY, USA, pp. 79–84. ACM.

Freescale Semiconductor (1999). *AltiVec Technology Programming Interface Manual*. Freescale Semiconductor. `http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf`.

Frigo, M. and S. G. Johnson (1998). FFTW: An adaptive software architecture for the FFT. In *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, Volume 3, pp. 1381–1384. IEEE.

Frigo, M., Steven, and G. Johnson (2005). The design and implementation of FFTW3. In *Proceedings of the IEEE*, pp. 216–231.

Fursin, G., C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, B. Mendelson, P. Barnard, E. Ashton, E. Courtois, F. Bodin, E. Bonilla, J. Thomson, H. Leather, C. Williams, and M. O'Boyle (2008, June). MILEPOST GCC: machine learning based research compiler. In *Proceedings of the GCC Developers' Summit*.

Gladman, B. (2003). Implementations of AES (Rijndael) in C/C++ and assembler. `http://gladman.plushost.co.uk/oldsite/cryptography_technology/rijndael/`.

Gladman, B. (2009). AES and Combined Encryption/Authentication Modes. `http://gladman.plushost.co.uk/oldsite/AES`.

Gopal, V., W. Feghali, J. Guilford, E. Ozturk, G. Wolrich, M. Dixon, M. Locktyukhin, and M. Perminov (2010a, April). Fast Cryptographic Computation on Intel Architecture Via Function Stitching (White Paper). `http://download.intel.com/design/intarch/PAPERS/323686.pdf`.

Gopal, V., E. Ozturk, W. Feghali, J. Guilford, G. Wolrich, and M. Dixon (2010b, August). Optimized Galois-Counter-Mode Implementation on Intel Architecture Processors. `http://download.intel.com/design/intarch/PAPERS/324194.pdf`.

Gregg, D. (2001, June). *Compilation Techniques for Instruction Level Parallelism in the Presence of Loops and Branches*. Ph. D. thesis, Technische Universitaet Wien.

Gueron, S. (2009). Intel's New AES Instructions for Enhanced Performance and Security. In *Fast Software Encryption - FSE 2009*, Lecture Notes in Computer Science 5665, pp. 51–66. Springer Verlag.

Gueron, S. (2010). *Intel Advanced Encryption Standard (AES) Instructions Set (White Paper)*. Intel Corp. `http://software.intel.com/file/24917`.

Gueron, S. and M. E. Kounavis (2010). *Intel Carry-Less Multiplication Instruction and its Usage for Computing the GCM Mode (White Paper)*. Intel Corp. `http://software.intel.com/file/24918`.

Gummaraju, J., J. Coburn, Y. Turner, and M. Rosenblum (2008). Streamware: programming general-purpose multicore processors using streams. *SIGOPS Oper. Syst. Rev. 42*(2), 297–307.

Gummaraju, J., M. Erez, J. Coburn, M. Rosenblum, and W. J. Dally (2007). Architectural Support for the Stream Execution Model on General-Purpose Processors. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, Washington, DC, USA, pp. 3–12. IEEE Computer Society.

Gummaraju, J. and M. Rosenblum (2005). Stream Programming on General-Purpose Processors. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, Washington, DC, USA, pp. 343–354. IEEE Computer Society.

Hamburg, M. (2009). Accelerating AES with Vector Permute Instructions. In *Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '09, Berlin, Heidelberg, pp. 18–32. Springer-Verlag.

Harrison, O. and J. Waldron (2008). Practical symmetric key cryptography on modern graphics hardware. In *Proceedings of the 17th conference on Security symposium*, Berkeley, CA, USA, pp. 195–209. USENIX Association.

Hennessy, J. L. and D. A. Patterson (1992). *Computer Architecture; A Quantitative Approach* (1st ed.). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Hinton, G., D. Sager, M. Upton, D. Boggs, et al. (2001). The Microarchitecture of the Pentium® 4 Processor. In *Intel Technology Journal*. Citeseer.

Hoste, K. and L. Eeckhout (2008). COLE: Compiler Optimization Level Exploration. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, New York, NY, USA, pp. 165–174. ACM.

Intel Corp. (2010a). *Download the Intel AESNI Sample Library*. Intel Corp. `http://software.intel.com/en-us/articles/download-the-intel-aesni-sample-library/`.

Intel Corp. (2010b). *Intel Array Building Blocks for Linux* OS Users Guide*. Intel Corp. `http://software.intel.com/sites/products/documentation/arbb/arbb_userguide_linux.pdf`.

Intel Corp. (2011). *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corp. `http://www.intel.com/Assets/PDF/manual/248966.pdf`.

Khailany, B., W. J. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang, and S. Rixner (2001). Imagine: Media Processing with Streams. *IEEE Micro 21*(2), 35–46.

Kirkpatrick, S. (1984). Optimization by simulated annealing: Quantitative studies. *Journal of Statistical Physics 34*(5), 975–986.

Kohn, L., G. Maturana, M. Tremblay, A. Prabhu, and G. Zyner (1995). The visual instruction set (VIS) in UltraSPARC. In *Proceedings of the 40th IEEE Computer Society International Conference*, COMPCON '95, Washington, DC, USA, pp. 462–. IEEE Computer Society.

Kosaraju, N. M., M. Varanasi, and S. P. Mohanty (2006). A High-Performance VLSI Architecture for Advanced Encryption Standard (AES) Algorithm. *VLSI Design, International Conference on*, 481–484.

Krall, A. and S. Lelait (2000). Compilation Techniques for Multimedia Processors. *International Journal of Parallel Programming 28*, 347–361.

Kudlur, M. and S. Mahlke (2008). Orchestrating the execution of stream programs on multicore platforms. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, New York, NY, USA, pp. 114–124. ACM.

Ladd, S. R. (2009). Acovea: Using Natural Selection to Investigate Software Complexities. `http://www.coyotegulch.com/products/acovea/`.

Lam, M. (1988). Software pipelining: an effective scheduling technique for VLIW machines. *SIGPLAN Not. 23*(7), 318–328.

Larsen, S., R. Rabbah, and S. Amarasinghe (2005). Exploiting Vector Parallelism in Software Pipelined Loops. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, Washington, DC, USA, pp. 119–129. IEEE Computer Society.

Lawler, E., J. Lenstra, C. Martel, B. Simons, and L. Stockmeyer (1987). Pipeline Scheduling: A Survey. *Research Report RJ-5738, IBM*.

Leather, H., E. Bonilla, and M. O'Boyle (2009). Automatic Feature Generation for Machine Learning Based Optimizing Compilation. In *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, Washington, DC, USA, pp. 81–91. IEEE Computer Society.

Li, X., M. Garzarán, and D. Padua (2004). A dynamically tuned sorting library. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pp. 111–122. IEEE.

Li, X., M. J. Garzaran, and D. Padua (2005). Optimizing Sorting with Genetic Algorithms. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, Washington, DC, USA, pp. 99–110. IEEE Computer Society.

Lipmaa, H., P. Rogaway, and D. Wagner (2001). CTR-mode encryption.

Lowney, P., S. Freudenberger, T. Karzes, W. Lichtenstein, R. Nix, J. O'donnell, and J. Ruttenberg (1993). The multiflow trace scheduling compiler. *The journal of Supercomputing 7*(1), 51–142.

Manavski, S. (2007, November). CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography. In *Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on*, pp. 65 –68.

Massalin, H. (1987). Superoptimizer: A Look at the Smallest Program. *SIGPLAN Not. 22*(10), 122–126.

Matsui, M. and J. Nakajima (2007). On the Power of Bitslice Implementation on Intel Core2 Processor. In P. Paillier and I. Verbauwhede (Eds.), *Cryptographic Hardware and Embedded Systems - CHES 2007*, Volume 4727 of *Lecture Notes in Computer Science*, pp. 121–134. Springer Berlin / Heidelberg.

McGrew, D. A. and J. Viega (2004). The Galois/Counter Mode of Operation (GCM). `http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/gcm/gcm-spec.pdf`.

McLoone, M. and J. V. McCanny (2003). Rijndael FPGA Implementations Utilising Look-Up Tables. *The Journal of VLSI Signal Processing 34*, 261–275.

Meyer, C. H. and S. M. Matyas (1982). *Cryptography: A New Dimension in Computer Data Security*. Wiley, New York :.

Mitchell, C. J. (2005). Cryptanalysis of Two Variants of PCBC Mode When Used for Message Integrity. In C. Boyd and J. M. Gonzlez Nieto (Eds.), *Information Security and Privacy*, Volume 3574 of *Lecture Notes in Computer Science*, pp. 560–571. Springer Berlin / Heidelberg.

Monteyne, M. (2008). RapidMind Multi-Core Development Platform. *RapidMind Inc., Waterloo, Canada, February*.

Mucci, P. J. (2009). *PapiEx - Execute arbitrary application and measure hardware performance counters with PAPI*. `http://icl.cs.utk.edu/~mucci/papiex/`.

Munshi, A. (2009). The OpenCL Specification. *Khronos OpenCL Working Group*, l1–15.

Naishlos, D. (2004, June). Autovectorization in GCC. In *GCC Summit*.

Nuzman, D. and R. Henderson (2006). Multi-platform Auto-vectorization. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, Washington, DC, USA, pp. 281–294. IEEE Computer Society.

Nuzman, D., I. Rosen, and A. Zaks (2006). Auto-vectorization of interleaved data for SIMD. *SIGPLAN Not. 41*(6), 132–143.

Nuzman, D. and A. Zaks (2006, June). Autovectorization in GCC - two years later. In *GCC Summit*.

NVIDIA (2007). Compute Unified Device Architecture Programming Guide. *NVIDIA: Santa Clara, CA*.

Owens, J. D., S. Rixner, U. J. Kapasi, P. Mattson, B. Towles, B. Serebrin, and W. J. Dally (2002). Media Processing Applications on the Imagine Stream Processor. *Computer Design, International Conference on 0*, 295.

Paar, C. (2002). The future of the art of cryptographic implementations. In *Position Statement for the STORK Workshop, Brussels*.

Pan, Z. and R. Eigenmann (2006a). Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, Washington, DC, USA, pp. 319–332. IEEE Computer Society.

Pan, Z. and R. Eigenmann (2006b). Fast, automatic, procedure-level performance tuning. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, PACT '06, New York, NY, USA, pp. 173–181. ACM.

Püschel, M., J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo (2005). SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation" 93*(2), 232–275.

Rau, B. and C. Glaeser (1981). Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the 14th annual workshop on Microprogramming*, pp. 183–198. IEEE Press.

Rau, B. R. (1994). Iterative modulo scheduling: an algorithm for software pipelining loops. In *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture*, New York, NY, USA, pp. 63–74. ACM.

Rebeiro, C., D. Selvakumar, and A. Devi (2006). Bitslice Implementation of AES. In D. Pointcheval, Y. Mu, and K. Chen (Eds.), *Cryptology and Network Security*, Volume 4301 of *Lecture Notes in Computer Science*, pp. 203–212. Springer Berlin / Heidelberg.

Ren, G., P. Wu, and D. Padua (2003). A Preliminary Study on the Vectorization of Multimedia Applications for Multimedia Extensions. In *In 16th International Workshop of Languages and Compilers for Parallel Computing*, pp. 420–435.

Rudd, T. (2007). Cheetah - The Python-Powered Template Engine. `http://www.cheetahtemplate.org/`.

Skiena, S. S. (1998). *The Algorithm Design Manual.* New York, NY, USA: Springer-Verlag New York, Inc.

Smid, M. and D. Branstad (1988, May). Data Encryption Standard: past and future. *Proceedings of the IEEE 76*(5), 550 –559.

Stratton, J., S. Stone, and W. mei Hwu (2008, July). MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs. In *21st Annual Workshop on Languages and Compilers for Parallel Computing (LCPC'2008).*

Sun, N. and C.-C. Lin (2007, November). Using the Cryptographic Accelerators in the UltraSPARC® T1 and T2 Processors. `http://www.sun.com/blueprints/0306/819-5782.pdf`.

Talla, D., L. K. John, and D. Burger (2003). Bottlenecks in Multimedia Processing with SIMD Style Extensions and Architectural Enhancements. *IEEE Trans. Comput. 52*(8), 1015–1031.

Taylor, M. B., W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal (2004). Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, Washington, DC, USA, pp. 2. IEEE Computer Society.

The National Institute of Standards and Technology (NIST) (2001). *Specification for the Advanced Encryption Standard (AES).* The National Institute of Standards and Technology (NIST).

Thies, W., M. Karczmarek, and S. P. Amarasinghe (2002). StreamIt: A Language for Streaming Applications. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, London, UK, pp. 179–196. Springer-Verlag.

Thuresson, M. (2010). MAO - An Extensible Micro-Architectural Optimizer. `http://code.google.com/p/mao/`.

Tillich, S. and J. Groschdl (2006). Instruction Set Extensions for Efficient AES Implementation on 32-bit Processors. In L. Goubin and M. Matsui (Eds.), *Cryptographic Hardware and Embedded Systems - CHES 2006*, Volume 4249 of *Lecture Notes in Computer Science*, pp. 270–284. Springer Berlin / Heidelberg.

Tillich, S. and J. Groschdl (2007). VLSI Implementation of a Functional Unit to Accelerate ECCandAES on 32-Bit Processors. In C. Carlet and B. Sunar (Eds.), *Arithmetic of Finite Fields*, Volume 4547 of *Lecture Notes in Computer Science*, pp. 40–54. Springer Berlin / Heidelberg.

Tillich, S. and C. Herbst (2008). Boosting AES Performance on a Tiny Processor Core. In T. Malkin (Ed.), *Topics in Cryptology CT-RSA 2008*, Volume 4964 of *Lecture Notes in Computer Science*, pp. 170–186. Springer Berlin / Heidelberg.

Tournavitis, G., Z. Wang, B. Franke, and M. F. O'Boyle (2009). Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, New York, NY, USA, pp. 177–187. ACM.

Tullsen, D. M., S. J. Eggers, and H. M. Levy (1995). Simultaneous multithreading: maximizing on-chip parallelism. In *Proceedings of the 22nd annual international symposium on Computer architecture*, ISCA '95, New York, NY, USA, pp. 392–403. ACM.

wei Liao, S., Z. Du, G. Wu, and G.-Y. Lueh (2006). Data and Computation Transformations for Brook Streaming Applications on Multiprocessors. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, Washington, DC, USA, pp. 196–207. IEEE Computer Society.

Whaley, C., A. Petitet, and J. J. Dongarra (2000). Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing 27*, 2001.

Wolfe, M. J. (1990). *Optimizing Supercompilers for Supercomputers.* Cambridge, MA, USA: MIT Press.

Xiong, J., J. Johnson, R. W. Johnson, and D. Padua (2001). SPL: A Language and Compiler for DSP Algorithms. In *Programming Languages Design and Implementation (PLDI)*, pp. 298–308.

Zhang, X. D. (2007, August). A Streaming Computation Framework for the Cell Processor. M.eng. thesis, Massachusetts Institute of Technology, Cambridge, MA.

Zhang, X. D., Q. J. Li, R. Rabbah, and S. Amarasinghe (2007, December). A Lightweight Streaming Layer for Multicore Execution. In *Workshop on Design, Architecture and Simulation of Chip Multi-Processors*, Chicago, IL.

Appendix A

Commonly Used Acronyms

AES Advanced Encryption Standard

AES-NI Advanced Encryption Standard New Instructions

AoS Array of Structures

ASM Assembly (Intel)

CBC Cipher-Block Chaining

CBC-MAC Cipher-Block Chaining Message Authentication Code

CCM Counter with CBC-MAC

CFB Cipher Feedback

CPU Central Processing Unit

CTR Counter

DCT Discrete Cosine Transform

DFT Discrete Fourier Transform

DDG Data Dependency Graph

DSP Digital Signal Processing

ECB Electronic Code Book

FFTW Fastest Fourier Transform in the West

GCC GNU C Compiler

Appendix A: Commonly Used Acronyms

GCM Galois/Counter Mode

Ghz Gigahertz

GPP General Purpose Processor

GPU Graphics Processing Unit

HPL High Performance Library (Intel)

ICC Intel C Compiler

ILP Instruction Level Parallelism

ISA Instruction-set Architecture

ISE Instruction-set Extension

OFB Output Feedback

OOO Out-of-Order (architecture)

PCBC Propagating Cipher-Block Chaining

SSA Single Static Assignment

SSE Streaming SIMD Extensions

SIMD Single Instruction, Multiple Data

SMT Simultaneous Multi-Threading

SoA Structure of Arrays

VLIW Very Long Instruction Word

XOR Exclusive Or

Appendix B

Additional Tables

Table B.1: ECB Results, as shown in Figure 6.4. Results are shown in cycles/byte.

	Keysize		
	128	**192**	**256**
Intel (HPL)	1.26	1.51	1.76
Intel (ASM)	1.359	1.61	1.859
AES-GEN	1.355	1.605	1.855

Table B.2: ECB Results, as shown in Figure 6.5. Results are shown in cycles/byte.

	Input Buffer Size					
	1K	**2K**	**4K**	**8K**	**16K**	**32K**
Intel (ASM)	1.359	1.311	1.288	1.276	1.269	1.266
AES-GEN	1.355	1.307	1.278	1.264	1.258	1.255

Table B.3: AES-GEN vs. Intel GCM 128 SMT Results, as shown in Figure 6.10. Results are shown in cycles/byte.

GCM 128 (Cycles/Byte)			
Buffer Size (B)	**Intel**	**AES-GEN**	**Difference**
96	5.79	6.525	0.735
192	4.15	4.562	0.412
384	3.48	3.677	0.197
768	3.13	3.223	0.093
1536	2.96	2.994	0.034
3072	2.88	2.891	0.011
6144	2.83	2.828	-0.002
12288	2.81	2.794	-0.016

Table B.4: CTR Results as shown in Figure 6.11. Results are shown in cycles/byte.

	Input Buffer					
Encryption Mode	**1K**	**2K**	**4K**	**8K**	**16K**	**32K**
CTR 128 (GEN)	1.363	1.313	1.282	1.266	1.258	1.255
CTR 128 (ASM)	1.443	1.392	1.357	1.339	1.330	1.325
CTR 192 (GEN)	1.617	1.564	1.531	1.516	1.509	1.505
CTR 192 (ASM)	1.720	1.650	1.610	1.591	1.582	1.572
CTR 256 (GEN)	1.867	1.814	1.782	1.766	1.759	1.755
CTR 256 (ASM)	1.980	1.900	1.865	1.843	1.832	1.826

Table B.5: CBC Results as shown in Figure 6.12. Results are shown in cycles/byte.

	Input Buffer					
Encryption Mode	**1K**	**2K**	**4K**	**8K**	**16K**	**32K**
CBC 128 (GEN)	3.851	3.801	3.775	3.763	3.757	3.753
CBC 128 (ASM)	4.276	4.260	4.226	4.212	4.206	4.200
CBC 192 (GEN)	4.602	4.551	4.525	4.513	4.507	4.503
CBC 192 (ASM)	5.057	5.022	4.991	4.964	4.957	4.954
CBC 256 (GEN)	5.351	5.300	5.276	5.263	5.257	5.253
CBC 256 (ASM)	5.810	5.759	5.746	5.716	5.708	5.704

Appendix C

Additional Source Code Listings

Listing C.1: Fastest scheduled CTR (round 1) code found by AES-GEN.

```
1  void AES_CTR_Encrypt(__m128i *plaintext,__m128i *ciphertext,
2            __m128i *key,long long ivec,long nonce,int blocks){
3  int i = 0;
4  __m128i resultx;
5  __m128i plain;
6  const __m128i key0 = key[0 ];
7  const __m128i key1 = key[1 ];
8  const __m128i key3 = key[3 ];
9  const __m128i key4 = key[4 ];
10 const __m128i key5 = key[5 ];
11 const __m128i key6 = key[6 ];
12 const __m128i key7 = key[7 ];
13 const __m128i key8 = key[8 ];
14 const __m128i key9 = key[9 ];
15 __m128i counter_block = _mm_setzero_si128();
16 unsigned scalar_counter = 0, scalar_key0, scalar_result0, scalar_result1;
17 unsigned my_counter;
18 __m128i result0;
19 __m128i fake_key;
20 __m128i result1, saved_r1, table_mask;
21 counter_block = _mm_insert_epi64(counter_block , ivec , 1 );
22 counter_block = _mm_insert_epi32(counter_block , nonce , 1 );
23 counter_block = _mm_srli_si128(counter_block , 4 );
24 result0 = _mm_xor_si128(counter_block , key0 );
25 scalar_key0 = _mm_extract_epi32(key0 , 3 );
26 result1 = _mm_insert_epi32(result0 , scalar_key0 & 0xFFFFFF , 3 );
27 saved_r1 = encrypt_round(result1 , key1 );
28 table_mask = _mm_cvtsi32_si128(table3[0 ] );
29 saved_r1 = _mm_xor_si128(saved_r1 , table_mask );
30  __m128i sp0_fake_key;
31  unsigned sp0_my_counter;
32  __m128i sp0_result1;
33  __m128i sp0_resultx;
34  __m128i sp0_resultx1;
35  __m128i sp0_resultx2;
36  __m128i sp0_resultx3;
```

```
37  __m128i sp0_resultx4;
38  __m128i sp0_resultx5;
39  __m128i sp0_resultx6;
40  __m128i sp0_resultx7;
41  __m128i sp0_resultx8;
42  unsigned sp0_scalar_counter;
43  unsigned sp0_scalar_result0;
44  __m128i sp0_table_mask;
45  __m128i sp1_fake_key;
46  unsigned sp1_my_counter;
47  __m128i sp1_result1;
48  __m128i sp1_resultx;
49  __m128i sp1_resultx1;
50  __m128i sp1_resultx2;
51  __m128i sp1_resultx3;
52  __m128i sp1_resultx4;
53  __m128i sp1_resultx5;
54  __m128i sp1_resultx6;
55  __m128i sp1_resultx7;
56  __m128i sp1_resultx8;
57  unsigned sp1_scalar_counter;
58  unsigned sp1_scalar_result0;
59  __m128i sp1_table_mask;
60  __m128i sp2_fake_key;
61  unsigned sp2_my_counter;
62  __m128i sp2_result1;
63  __m128i sp2_resultx;
64  __m128i sp2_resultx1;
65  __m128i sp2_resultx2;
66  __m128i sp2_resultx3;
67  __m128i sp2_resultx4;
68  __m128i sp2_resultx5;
69  __m128i sp2_resultx6;
70  __m128i sp2_resultx7;
71  __m128i sp2_resultx8;
72  unsigned sp2_scalar_counter;
73  unsigned sp2_scalar_result0;
74  __m128i sp2_table_mask;
75  __m128i sp3_fake_key;
76  unsigned sp3_my_counter;
77  __m128i sp3_result1;
78  __m128i sp3_resultx;
79  __m128i sp3_resultx1;
80  __m128i sp3_resultx2;
81  __m128i sp3_resultx3;
82  __m128i sp3_resultx4;
83  __m128i sp3_resultx5;
84  __m128i sp3_resultx6;
85  __m128i sp3_resultx7;
86  __m128i sp3_resultx8;
87  unsigned sp3_scalar_counter;
88  unsigned sp3_scalar_result0;
89  __m128i sp3_table_mask;
90  __m128i sp4_fake_key;
91  unsigned sp4_my_counter;
```

```
92  __m128i sp4_result1;
93  __m128i sp4_resultx;
94  __m128i sp4_resultx1;
95  __m128i sp4_resultx2;
96  __m128i sp4_resultx3;
97  __m128i sp4_resultx4;
98  __m128i sp4_resultx5;
99  __m128i sp4_resultx6;
100 __m128i sp4_resultx7;
101 __m128i sp4_resultx8;
102 unsigned sp4_scalar_counter;
103 unsigned sp4_scalar_result0;
104 __m128i sp4_table_mask;
105 __m128i sp5_fake_key;
106 unsigned sp5_my_counter;
107 __m128i sp5_result1;
108 __m128i sp5_resultx;
109 __m128i sp5_resultx1;
110 __m128i sp5_resultx2;
111 __m128i sp5_resultx3;
112 __m128i sp5_resultx4;
113 __m128i sp5_resultx5;
114 __m128i sp5_resultx6;
115 __m128i sp5_resultx7;
116 __m128i sp5_resultx8;
117 unsigned sp5_scalar_counter;
118 unsigned sp5_scalar_result0;
119 __m128i sp5_table_mask;
120 __m128i sp6_fake_key;
121 unsigned sp6_my_counter;
122 __m128i sp6_result1;
123 __m128i sp6_resultx;
124 __m128i sp6_resultx1;
125 __m128i sp6_resultx2;
126 __m128i sp6_resultx3;
127 __m128i sp6_resultx4;
128 __m128i sp6_resultx5;
129 __m128i sp6_resultx6;
130 __m128i sp6_resultx7;
131 __m128i sp6_resultx8;
132 unsigned sp6_scalar_counter;
133 unsigned sp6_scalar_result0;
134 __m128i sp6_table_mask;
135  sp0_fake_key= _mm_xor_si128(key[10 ] , plaintext[i ] )  ;
136 sp0_scalar_counter  =  scalar_counter = scalar_counter + 1 ;
137  sp0_my_counter= sp0_scalar_counter ;
138  sp0_scalar_result0= ((_bswap(scalar_key0 )) & 0xFF) ^ sp0_my_counter ;
139  sp0_table_mask= _mm_cvtsi32_si128(table3[sp0_scalar_result0] )  ;
140  sp0_result1= _mm_xor_si128(saved_r1 , sp0_table_mask)  ;
141  sp0_resultx= encrypt_round(sp0_result1, key[2 ] )  ;
142  sp1_fake_key= _mm_xor_si128(key[10 ] , plaintext[1 + i ] )  ;
143 sp1_scalar_counter  =  scalar_counter = scalar_counter + 1 ;
144  sp1_my_counter= sp1_scalar_counter ;
145  sp1_scalar_result0= ((_bswap(scalar_key0 )) & 0xFF) ^ sp1_my_counter ;
146  sp1_table_mask= _mm_cvtsi32_si128(table3[sp1_scalar_result0] )  ;
```

```
147  sp0_resultx1= encrypt_round(sp0_resultx , key3 )  ;
148  sp1_result1= _mm_xor_si128(saved_r1  , sp1_table_mask)  ;
149  sp1_resultx= encrypt_round(sp1_result1 , key[2 ] )  ;
150  sp0_resultx2= encrypt_round(sp0_resultx1 , key4 )  ;
151  sp2_fake_key= _mm_xor_si128(key[10 ] , plaintext[2 + i ] )  ;
152 sp2_scalar_counter  =  scalar_counter = scalar_counter + 1 ;
153  sp2_my_counter= sp2_scalar_counter ;
154  sp2_scalar_result0= ((_bswap(scalar_key0 )) & 0xFF) ^ sp2_my_counter ;
155  sp2_table_mask= _mm_cvtsi32_si128(table3[sp2_scalar_result0] )  ;
156  sp1_resultx1= encrypt_round(sp1_resultx , key3 )  ;
157  sp0_resultx3= encrypt_round(sp0_resultx2 , key5 )  ;
158  sp2_result1= _mm_xor_si128(saved_r1  , sp2_table_mask)  ;
159  sp2_resultx= encrypt_round(sp2_result1 , key[2 ] )  ;
160  sp1_resultx2= encrypt_round(sp1_resultx1 , key4 )  ;
161  sp3_fake_key= _mm_xor_si128(key[10 ] , plaintext[3 + i ] )  ;
162 sp3_scalar_counter  =  scalar_counter = scalar_counter + 1 ;
163  sp0_resultx4= encrypt_round(sp0_resultx3 , key6 )  ;
164  sp3_my_counter= sp3_scalar_counter ;
165  sp3_scalar_result0= ((_bswap(scalar_key0 )) & 0xFF) ^ sp3_my_counter ;
166  sp3_table_mask= _mm_cvtsi32_si128(table3[sp3_scalar_result0] )  ;
167  sp2_resultx1= encrypt_round(sp2_resultx , key3 )  ;
168  sp1_resultx3= encrypt_round(sp1_resultx2 , key5 )  ;
169  sp0_resultx5= encrypt_round(sp0_resultx4 , key7 )  ;
170  sp3_result1= _mm_xor_si128(saved_r1  , sp3_table_mask)  ;
171  sp3_resultx= encrypt_round(sp3_result1 , key[2 ] )  ;
172  sp2_resultx2= encrypt_round(sp2_resultx1 , key4 )  ;
173  sp4_fake_key= _mm_xor_si128(key[10 ] , plaintext[4 + i ] )  ;
174 sp4_scalar_counter  =  scalar_counter = scalar_counter + 1 ;
175  sp1_resultx4= encrypt_round(sp1_resultx3 , key6 )  ;
176  sp4_my_counter= sp4_scalar_counter ;
177  sp4_scalar_result0= ((_bswap(scalar_key0 )) & 0xFF) ^ sp4_my_counter ;
178  sp0_resultx6= encrypt_round(sp0_resultx5 , key8 )  ;
179  sp4_table_mask= _mm_cvtsi32_si128(table3[sp4_scalar_result0] )  ;
180  sp3_resultx1= encrypt_round(sp3_resultx , key3 )  ;
181  sp2_resultx3= encrypt_round(sp2_resultx2 , key5 )  ;
182  sp1_resultx5= encrypt_round(sp1_resultx4 , key7 )  ;
183  sp4_result1= _mm_xor_si128(saved_r1  , sp4_table_mask)  ;
184  sp4_resultx= encrypt_round(sp4_result1 , key[2 ] )  ;
185  sp0_resultx7= encrypt_round(sp0_resultx6 , key9 )  ;
186  sp3_resultx2= encrypt_round(sp3_resultx1 , key4 )  ;
187  sp5_fake_key= _mm_xor_si128(key[10 ] , plaintext[5 + i ] )  ;
188 sp5_scalar_counter  =  scalar_counter = scalar_counter + 1 ;
189 for(i = 0; i < blocks -6; i += 1 ){
190  sp5_my_counter= sp5_scalar_counter ;
191  sp5_scalar_result0= ((_bswap(scalar_key0 )) & 0xFF) ^ sp5_my_counter ;
192  sp5_table_mask= _mm_cvtsi32_si128(table3[sp5_scalar_result0] )  ;
193  sp0_resultx8= encrypt_final(sp0_resultx7 , sp0_fake_key)  ;
194  sp1_resultx6= encrypt_round(sp1_resultx5 , key8 )  ;
195  sp2_resultx4= encrypt_round(sp2_resultx3 , key6 )  ;
196  sp4_resultx1= encrypt_round(sp4_resultx , key3 )  ;
197  sp5_result1= _mm_xor_si128(saved_r1  , sp5_table_mask)  ;
198  ciphertext[i ] = sp0_resultx8 ;
199  sp1_resultx7= encrypt_round(sp1_resultx6 , key9 )  ;
200  sp2_resultx5= encrypt_round(sp2_resultx4 , key7 )  ;
201  sp3_resultx3= encrypt_round(sp3_resultx2 , key5 )  ;
```

```
202   sp4_resultx2= encrypt_round(sp4_resultx1 , key4 )  ;
203   sp5_resultx= encrypt_round(sp5_result1 , key[2 ] )  ;
204  sp6_scalar_counter  =  scalar_counter = scalar_counter + 1 ;
205   sp6_fake_key= _mm_xor_si128(key[10 ] , plaintext[6 + i ] )  ;
206   sp0_fake_key = sp1_fake_key;
207   sp0_scalar_counter = sp1_scalar_counter;
208   sp0_my_counter = sp1_my_counter;
209   sp0_scalar_result0 = sp1_scalar_result0;
210   sp0_table_mask = sp1_table_mask;
211   sp0_result1 = sp1_result1;
212   sp0_resultx = sp1_resultx;
213   sp0_resultx1 = sp1_resultx1;
214   sp0_resultx2 = sp1_resultx2;
215   sp0_resultx3 = sp1_resultx3;
216   sp0_resultx4 = sp1_resultx4;
217   sp0_resultx5 = sp1_resultx5;
218   sp0_resultx6 = sp1_resultx6;
219   sp0_resultx7 = sp1_resultx7;
220   sp1_fake_key = sp2_fake_key;
221   sp1_scalar_counter = sp2_scalar_counter;
222   sp1_my_counter = sp2_my_counter;
223   sp1_scalar_result0 = sp2_scalar_result0;
224   sp1_table_mask = sp2_table_mask;
225   sp1_result1 = sp2_result1;
226   sp1_resultx = sp2_resultx;
227   sp1_resultx1 = sp2_resultx1;
228   sp1_resultx2 = sp2_resultx2;
229   sp1_resultx3 = sp2_resultx3;
230   sp1_resultx4 = sp2_resultx4;
231   sp1_resultx5 = sp2_resultx5;
232   sp2_fake_key = sp3_fake_key;
233   sp2_scalar_counter = sp3_scalar_counter;
234   sp2_my_counter = sp3_my_counter;
235   sp2_scalar_result0 = sp3_scalar_result0;
236   sp2_table_mask = sp3_table_mask;
237   sp2_result1 = sp3_result1;
238   sp2_resultx = sp3_resultx;
239   sp2_resultx1 = sp3_resultx1;
240   sp2_resultx2 = sp3_resultx2;
241   sp2_resultx3 = sp3_resultx3;
242   sp3_fake_key = sp4_fake_key;
243   sp3_scalar_counter = sp4_scalar_counter;
244   sp3_my_counter = sp4_my_counter;
245   sp3_scalar_result0 = sp4_scalar_result0;
246   sp3_table_mask = sp4_table_mask;
247   sp3_result1 = sp4_result1;
248   sp3_resultx = sp4_resultx;
249   sp3_resultx1 = sp4_resultx1;
250   sp3_resultx2 = sp4_resultx2;
251   sp4_fake_key = sp5_fake_key;
252   sp4_scalar_counter = sp5_scalar_counter;
253   sp4_my_counter = sp5_my_counter;
254   sp4_scalar_result0 = sp5_scalar_result0;
255   sp4_table_mask = sp5_table_mask;
256   sp4_result1 = sp5_result1;
```

```
257     sp4_resultx = sp5_resultx;
258     sp5_fake_key = sp6_fake_key;
259     sp5_scalar_counter = sp6_scalar_counter;
260   }
261     sp2_resultx4= encrypt_round(sp2_resultx3 , key6 )  ;
262     sp5_my_counter= sp5_scalar_counter ;
263     sp5_scalar_result0= ((_bswap(scalar_key0 )) & 0xFF) ^ sp5_my_counter ;
264     sp1_resultx6= encrypt_round(sp1_resultx5 , key8 )  ;
265     sp5_table_mask= _mm_cvtsi32_si128(table3[sp5_scalar_result0] )  ;
266     sp4_resultx1= encrypt_round(sp4_resultx , key3 )  ;
267     sp0_resultx8= encrypt_final(sp0_resultx7 , sp0_fake_key)  ;
268     sp3_resultx3= encrypt_round(sp3_resultx2 , key5 )  ;
269     sp2_resultx5= encrypt_round(sp2_resultx4 , key7 )  ;
270     sp5_result1= _mm_xor_si128(saved_r1 , sp5_table_mask)  ;
271     sp5_resultx= encrypt_round(sp5_result1 , key[2 ] )  ;
272     sp1_resultx7= encrypt_round(sp1_resultx6 , key9 )  ;
273     sp4_resultx2= encrypt_round(sp4_resultx1 , key4 )  ;
274     ciphertext[i ] = sp0_resultx8 ;
275     sp3_resultx4= encrypt_round(sp3_resultx3 , key6 )  ;
276     sp2_resultx6= encrypt_round(sp2_resultx5 , key8 )  ;
277     sp5_resultx1= encrypt_round(sp5_resultx , key3 )  ;
278     sp1_resultx8= encrypt_final(sp1_resultx7 , sp1_fake_key)  ;
279     sp4_resultx3= encrypt_round(sp4_resultx2 , key5 )  ;
280     sp3_resultx5= encrypt_round(sp3_resultx4 , key7 )  ;
281     sp2_resultx7= encrypt_round(sp2_resultx6 , key9 )  ;
282     sp5_resultx2= encrypt_round(sp5_resultx1 , key4 )  ;
283     ciphertext[1 + i ] = sp1_resultx8 ;
284     sp4_resultx4= encrypt_round(sp4_resultx3 , key6 )  ;
285     sp3_resultx6= encrypt_round(sp3_resultx5 , key8 )  ;
286     sp2_resultx8= encrypt_final(sp2_resultx7 , sp2_fake_key)  ;
287     sp5_resultx3= encrypt_round(sp5_resultx2 , key5 )  ;
288     sp4_resultx5= encrypt_round(sp4_resultx4 , key7 )  ;
289     sp3_resultx7= encrypt_round(sp3_resultx6 , key9 )  ;
290     ciphertext[2 + i ] = sp2_resultx8 ;
291     sp5_resultx4= encrypt_round(sp5_resultx3 , key6 )  ;
292     sp4_resultx6= encrypt_round(sp4_resultx5 , key8 )  ;
293     sp3_resultx8= encrypt_final(sp3_resultx7 , sp3_fake_key)  ;
294     sp5_resultx5= encrypt_round(sp5_resultx4 , key7 )  ;
295     sp4_resultx7= encrypt_round(sp4_resultx6 , key9 )  ;
296     ciphertext[3 + i ] = sp3_resultx8 ;
297     sp5_resultx6= encrypt_round(sp5_resultx5 , key8 )  ;
298     sp4_resultx8= encrypt_final(sp4_resultx7 , sp4_fake_key)  ;
299     sp5_resultx7= encrypt_round(sp5_resultx6 , key9 )  ;
300     ciphertext[4 + i ] = sp4_resultx8 ;
301     sp5_resultx8= encrypt_final(sp5_resultx7 , sp5_fake_key)  ;
302     ciphertext[5 + i ] = sp5_resultx8 ;
303 }
```

Listing C.2: Fastest scheduled CTR (round 2) code found by AES-GEN.

```
1  void AES_CTR_Encrypt(__m128i *plaintext,__m128i *ciphertext,
2                  __m128i *key,long long ivec,long nonce,int blocks){
3  int i = 0;
4  __m128i plain;
5  const __m128i key0 = key[0 ];
6  const __m128i key3 = key[3 ];
7  const __m128i key6 = key[6 ];
8  const __m128i key9 = key[9 ];
9  const __m128i key11 = key[11 ];
10 const __m128i key12 = key[12 ];
11 const __m128i key13 = key[13 ];
12 __m128i counter_block = _mm_setzero_si128();
13 unsigned scalar_counter = 0, scalar_key0, scalar_r0, scalar_r1;
14 unsigned idx0, idx1, idx2, idx3, v0, v1, v2, v3, my_counter;
15 __m128i r0, fake_key;
16 __m128i t0, t1, t3, t4, t5, t6;
17 __m128i r1, saved_r1, result2, saved_result2,
18                 table_entries, second_round_output;
19 unsigned first_round_output_x0;
20 __m128i resultx;
21 counter_block = _mm_insert_epi64(counter_block , ivec , 1 );
22 counter_block = _mm_insert_epi32(counter_block , nonce , 1 );
23 counter_block = _mm_srli_si128(counter_block , 4 );
24 r0 = _mm_xor_si128(counter_block , key0 );
25 scalar_key0 = _mm_extract_epi32(key0 , 3 );
26 r1 = _mm_insert_epi32(r0 , scalar_key0 & 0xFFFFFF , 3 );
27 saved_r1 = encrypt_round(r1 , key[1 ] );
28 first_round_output_x0 = _mm_extract_epi32(saved_r1 , 0 ) ^ table3[0 ];
29 result2 = _mm_insert_epi32(saved_r1 , 0 , 0 );
30 saved_result2 = encrypt_round(result2 , key[2 ] );
31 table_entries = _mm_set_epi32(table1[0], table2[0], table3[0], table0[0]);
32 second_round_output = _mm_xor_si128(saved_result2 , table_entries );
33  __m128i sp0_fake_key;
34  unsigned sp0_idx0;
35  unsigned sp0_idx1;
36  unsigned sp0_idx2;
37  unsigned sp0_idx3;
38  unsigned sp0_my_counter;
39  __m128i sp0_result2;
40  __m128i sp0_resultx;
41  __m128i sp0_resultx1;
42  __m128i sp0_resultx10;
43  __m128i sp0_resultx11;
44  __m128i sp0_resultx2;
45  __m128i sp0_resultx3;
46  __m128i sp0_resultx4;
47  __m128i sp0_resultx5;
48  __m128i sp0_resultx6;
49  __m128i sp0_resultx7;
50  __m128i sp0_resultx8;
51  __m128i sp0_resultx9;
52  unsigned sp0_scalar_counter;
53  unsigned sp0_scalar_r0;
```

```
54  unsigned sp0_scalar_r1;
55  __m128i sp0_t0;
56  __m128i sp0_t1;
57  __m128i sp0_t3;
58  __m128i sp0_t4;
59  __m128i sp0_t5;
60  __m128i sp0_t6;
61  __m128i sp0_table_entries;
62  __m128i sp1_fake_key;
63  unsigned sp1_idx0;
64  unsigned sp1_idx1;
65  unsigned sp1_idx2;
66  unsigned sp1_idx3;
67  unsigned sp1_my_counter;
68  __m128i sp1_result2;
69  __m128i sp1_resultx;
70  __m128i sp1_resultx1;
71  __m128i sp1_resultx10;
72  __m128i sp1_resultx11;
73  __m128i sp1_resultx2;
74  __m128i sp1_resultx3;
75  __m128i sp1_resultx4;
76  __m128i sp1_resultx5;
77  __m128i sp1_resultx6;
78  __m128i sp1_resultx7;
79  __m128i sp1_resultx8;
80  __m128i sp1_resultx9;
81  unsigned sp1_scalar_counter;
82  unsigned sp1_scalar_r0;
83  unsigned sp1_scalar_r1;
84  __m128i sp1_t0;
85  __m128i sp1_t1;
86  __m128i sp1_t3;
87  __m128i sp1_t4;
88  __m128i sp1_t5;
89  __m128i sp1_t6;
90  __m128i sp1_table_entries;
91  __m128i sp2_fake_key;
92  unsigned sp2_idx0;
93  unsigned sp2_idx1;
94  unsigned sp2_idx2;
95  unsigned sp2_idx3;
96  unsigned sp2_my_counter;
97  __m128i sp2_result2;
98  __m128i sp2_resultx;
99  __m128i sp2_resultx1;
100 __m128i sp2_resultx10;
101 __m128i sp2_resultx11;
102 __m128i sp2_resultx2;
103 __m128i sp2_resultx3;
104 __m128i sp2_resultx4;
105 __m128i sp2_resultx5;
106 __m128i sp2_resultx6;
107 __m128i sp2_resultx7;
108 __m128i sp2_resultx8;
```

```
109  __m128i sp2_resultx9;
110  unsigned sp2_scalar_counter;
111  unsigned sp2_scalar_r0;
112  unsigned sp2_scalar_r1;
113  __m128i sp2_t0;
114  __m128i sp2_t1;
115  __m128i sp2_t3;
116  __m128i sp2_t4;
117  __m128i sp2_t5;
118  __m128i sp2_t6;
119  __m128i sp2_table_entries;
120  __m128i sp3_fake_key;
121  unsigned sp3_idx0;
122  unsigned sp3_idx1;
123  unsigned sp3_idx2;
124  unsigned sp3_idx3;
125  unsigned sp3_my_counter;
126  __m128i sp3_result2;
127  __m128i sp3_resultx;
128  __m128i sp3_resultx1;
129  __m128i sp3_resultx10;
130  __m128i sp3_resultx11;
131  __m128i sp3_resultx2;
132  __m128i sp3_resultx3;
133  __m128i sp3_resultx4;
134  __m128i sp3_resultx5;
135  __m128i sp3_resultx6;
136  __m128i sp3_resultx7;
137  __m128i sp3_resultx8;
138  __m128i sp3_resultx9;
139  unsigned sp3_scalar_counter;
140  unsigned sp3_scalar_r0;
141  unsigned sp3_scalar_r1;
142  __m128i sp3_t0;
143  __m128i sp3_t1;
144  __m128i sp3_t3;
145  __m128i sp3_t4;
146  __m128i sp3_t5;
147  __m128i sp3_t6;
148  __m128i sp3_table_entries;
149  __m128i sp4_fake_key;
150  unsigned sp4_idx0;
151  unsigned sp4_idx1;
152  unsigned sp4_idx2;
153  unsigned sp4_idx3;
154  unsigned sp4_my_counter;
155  __m128i sp4_result2;
156  __m128i sp4_resultx;
157  __m128i sp4_resultx1;
158  __m128i sp4_resultx10;
159  __m128i sp4_resultx11;
160  __m128i sp4_resultx2;
161  __m128i sp4_resultx3;
162  __m128i sp4_resultx4;
163  __m128i sp4_resultx5;
```

```
164  __m128i sp4_resultx6;
165  __m128i sp4_resultx7;
166  __m128i sp4_resultx8;
167  __m128i sp4_resultx9;
168  unsigned sp4_scalar_counter;
169  unsigned sp4_scalar_r0;
170  unsigned sp4_scalar_r1;
171  __m128i sp4_t0;
172  __m128i sp4_t1;
173  __m128i sp4_t3;
174  __m128i sp4_t4;
175  __m128i sp4_t5;
176  __m128i sp4_t6;
177  __m128i sp4_table_entries;
178  __m128i sp5_fake_key;
179  unsigned sp5_idx0;
180  unsigned sp5_idx1;
181  unsigned sp5_idx2;
182  unsigned sp5_idx3;
183  unsigned sp5_my_counter;
184  __m128i sp5_result2;
185  __m128i sp5_resultx;
186  __m128i sp5_resultx1;
187  __m128i sp5_resultx10;
188  __m128i sp5_resultx11;
189  __m128i sp5_resultx2;
190  __m128i sp5_resultx3;
191  __m128i sp5_resultx4;
192  __m128i sp5_resultx5;
193  __m128i sp5_resultx6;
194  __m128i sp5_resultx7;
195  __m128i sp5_resultx8;
196  __m128i sp5_resultx9;
197  unsigned sp5_scalar_counter;
198  unsigned sp5_scalar_r0;
199  unsigned sp5_scalar_r1;
200  __m128i sp5_t0;
201  __m128i sp5_t1;
202  __m128i sp5_t3;
203  __m128i sp5_t4;
204  __m128i sp5_t5;
205  __m128i sp5_t6;
206  __m128i sp5_table_entries;
207   sp0_fake_key= _mm_xor_si128(key[14 ] , plaintext[i ] )  ;
208  sp0_scalar_counter  =  scalar_counter = scalar_counter + 1 ;
209   sp0_my_counter= sp0_scalar_counter ;
210   sp0_scalar_r0= (_bswap(scalar_key0 ) & 0xFF) ^ sp0_my_counter ;
211   sp0_scalar_r1= table3[sp0_scalar_r0] ^ first_round_output_x0  ;
212   sp0_idx3= (sp0_scalar_r1 >> 24)  ;
213   sp0_idx2= (sp0_scalar_r1 >> 16) & 0xFF  ;
214   sp0_idx1= (sp0_scalar_r1 >> 8) & 0xFF  ;
215   sp0_idx0= sp0_scalar_r1& 0xFF  ;
216   sp0_t1= _mm_cvtsi32_si128(table3[sp0_idx3] )  ;
217   sp0_t4= _mm_cvtsi32_si128(table2[sp0_idx2] )  ;
218   sp0_t5= _mm_cvtsi32_si128(table1[sp0_idx1] )  ;
```

```
219   sp0_t0= _mm_cvtsi32_si128(table0[sp0_idx0] )   ;
220   sp1_fake_key= _mm_xor_si128(key[14 ] , plaintext[1 + i ] )   ;
221  sp1_scalar_counter  =  scalar_counter = scalar_counter + 1 ;
222   sp1_my_counter= sp1_scalar_counter ;
223   sp1_scalar_r0= (_bswap(scalar_key0 ) & 0xFF) ^ sp1_my_counter ;
224   sp0_t6= _mm_unpacklo_epi32(sp0_t4 , sp0_t5)   ;
225   sp0_t3= _mm_unpacklo_epi32(sp0_t0 , sp0_t1)   ;
226   sp1_scalar_r1= table3[sp1_scalar_r0] ^ first_round_output_x0   ;
227   sp0_table_entries= _mm_unpacklo_epi64(sp0_t3 , sp0_t6)   ;
228   sp0_result2= _mm_xor_si128(second_round_output , sp0_table_entries)   ;
229   sp0_resultx= encrypt_round(sp0_result2 , key3 )   ;
230   sp1_idx3= (sp1_scalar_r1 >> 24)   ;
231   sp1_idx2= (sp1_scalar_r1 >> 16) & 0xFF   ;
232   sp1_idx1= (sp1_scalar_r1 >> 8) & 0xFF   ;
233   sp1_idx0= sp1_scalar_r1& 0xFF   ;
234   sp0_resultx1= encrypt_round(sp0_resultx , key[4 ] )   ;
235   sp1_t1= _mm_cvtsi32_si128(table3[sp1_idx3] )   ;
236   sp1_t4= _mm_cvtsi32_si128(table2[sp1_idx2] )   ;
237   sp1_t5= _mm_cvtsi32_si128(table1[sp1_idx1] )   ;
238   sp1_t0= _mm_cvtsi32_si128(table0[sp1_idx0] )   ;
239   sp0_resultx2= encrypt_round(sp0_resultx1 , key[5 ] )   ;
240   sp2_fake_key= _mm_xor_si128(key[14 ] , plaintext[2 + i ] )   ;
241  sp2_scalar_counter  =  scalar_counter = scalar_counter + 1 ;
242   sp2_my_counter= sp2_scalar_counter ;
243   sp2_scalar_r0= (_bswap(scalar_key0 ) & 0xFF) ^ sp2_my_counter ;
244   sp1_t6= _mm_unpacklo_epi32(sp1_t4 , sp1_t5)   ;
245   sp1_t3= _mm_unpacklo_epi32(sp1_t0 , sp1_t1)   ;
246   sp2_scalar_r1= table3[sp2_scalar_r0] ^ first_round_output_x0   ;
247   sp1_table_entries= _mm_unpacklo_epi64(sp1_t3 , sp1_t6)   ;
248   sp1_result2= _mm_xor_si128(second_round_output , sp1_table_entries)   ;
249   sp1_resultx= encrypt_round(sp1_result2 , key3 )   ;
250   sp0_resultx3= encrypt_round(sp0_resultx2 , key6 )   ;
251   sp2_idx3= (sp2_scalar_r1 >> 24)   ;
252   sp2_idx2= (sp2_scalar_r1 >> 16) & 0xFF   ;
253   sp2_idx1= (sp2_scalar_r1 >> 8) & 0xFF   ;
254   sp2_idx0= sp2_scalar_r1& 0xFF   ;
255   sp1_resultx1= encrypt_round(sp1_resultx , key[4 ] )   ;
256   sp0_resultx4= encrypt_round(sp0_resultx3 , key[7 ] )   ;
257   sp2_t1= _mm_cvtsi32_si128(table3[sp2_idx3] )   ;
258   sp2_t4= _mm_cvtsi32_si128(table2[sp2_idx2] )   ;
259   sp2_t5= _mm_cvtsi32_si128(table1[sp2_idx1] )   ;
260   sp2_t0= _mm_cvtsi32_si128(table0[sp2_idx0] )   ;
261   sp1_resultx2= encrypt_round(sp1_resultx1 , key[5 ] )   ;
262   sp3_fake_key= _mm_xor_si128(key[14 ] , plaintext[3 + i ] )   ;
263  sp3_scalar_counter  =  scalar_counter = scalar_counter + 1 ;
264   sp0_resultx5= encrypt_round(sp0_resultx4 , key[8 ] )   ;
265   sp3_my_counter= sp3_scalar_counter ;
266   sp3_scalar_r0= (_bswap(scalar_key0 ) & 0xFF) ^ sp3_my_counter ;
267   sp2_t6= _mm_unpacklo_epi32(sp2_t4 , sp2_t5)   ;
268   sp2_t3= _mm_unpacklo_epi32(sp2_t0 , sp2_t1)   ;
269   sp3_scalar_r1= table3[sp3_scalar_r0] ^ first_round_output_x0   ;
270   sp2_table_entries= _mm_unpacklo_epi64(sp2_t3 , sp2_t6)   ;
271   sp2_result2= _mm_xor_si128(second_round_output , sp2_table_entries)   ;
272   sp2_resultx= encrypt_round(sp2_result2 , key3 )   ;
273   sp1_resultx3= encrypt_round(sp1_resultx2 , key6 )   ;
```

```
274  sp0_resultx6= encrypt_round(sp0_resultx5 , key9 )  ;
275  sp3_idx3= (sp3_scalar_r1>> 24)  ;
276  sp3_idx2= (sp3_scalar_r1>> 16) & 0xFF  ;
277  sp3_idx1= (sp3_scalar_r1>> 8) & 0xFF  ;
278  sp3_idx0= sp3_scalar_r1& 0xFF  ;
279  sp2_resultx1= encrypt_round(sp2_resultx , key[4 ] )  ;
280  sp1_resultx4= encrypt_round(sp1_resultx3 , key[7 ] )  ;
281  sp3_t1= _mm_cvtsi32_si128(table3[sp3_idx3] )  ;
282  sp3_t4= _mm_cvtsi32_si128(table2[sp3_idx2] )  ;
283  sp3_t5= _mm_cvtsi32_si128(table1[sp3_idx1] )  ;
284  sp3_t0= _mm_cvtsi32_si128(table0[sp3_idx0] )  ;
285  sp0_resultx7= encrypt_round(sp0_resultx6 , key[10 ] )  ;
286  sp2_resultx2= encrypt_round(sp2_resultx1 , key[5 ] )  ;
287  sp4_fake_key= _mm_xor_si128(key[14 ] , plaintext[4 + i ] )  ;
288 sp4_scalar_counter  =  scalar_counter = scalar_counter + 1 ;
289  sp1_resultx5= encrypt_round(sp1_resultx4 , key[8 ] )  ;
290  sp0_resultx8= encrypt_round(sp0_resultx7 , key11 )  ;
291  sp4_my_counter= sp4_scalar_counter ;
292  sp4_scalar_r0= (_bswap(scalar_key0 ) & 0xFF) ^ sp4_my_counter ;
293  sp3_t6= _mm_unpacklo_epi32(sp3_t4 , sp3_t5)  ;
294  sp3_t3= _mm_unpacklo_epi32(sp3_t0 , sp3_t1)  ;
295  sp4_scalar_r1= table3[sp4_scalar_r0] ^ first_round_output_x0  ;
296  sp3_table_entries= _mm_unpacklo_epi64(sp3_t3 , sp3_t6)  ;
297  sp3_result2= _mm_xor_si128(second_round_output , sp3_table_entries)  ;
298  sp3_resultx= encrypt_round(sp3_result2 , key3 )  ;
299  sp2_resultx3= encrypt_round(sp2_resultx2 , key6 )  ;
300  sp1_resultx6= encrypt_round(sp1_resultx5 , key9 )  ;
301  sp0_resultx9= encrypt_round(sp0_resultx8 , key12 )  ;
302 for(i = 0; i < blocks -5; i += 1 ){
303  sp4_idx0= sp4_scalar_r1& 0xFF  ;
304  sp4_idx1= (sp4_scalar_r1>> 8) & 0xFF  ;
305  sp4_idx2= (sp4_scalar_r1>> 16) & 0xFF  ;
306  sp4_idx3= (sp4_scalar_r1>> 24)  ;
307  sp4_t0= _mm_cvtsi32_si128(table0[sp4_idx0] )  ;
308  sp4_t5= _mm_cvtsi32_si128(table1[sp4_idx1] )  ;
309  sp4_t4= _mm_cvtsi32_si128(table2[sp4_idx2] )  ;
310  sp4_t1= _mm_cvtsi32_si128(table3[sp4_idx3] )  ;
311  sp4_t3= _mm_unpacklo_epi32(sp4_t0 , sp4_t1)  ;
312  sp4_t6= _mm_unpacklo_epi32(sp4_t4 , sp4_t5)  ;
313 sp5_scalar_counter  =  scalar_counter = scalar_counter + 1 ;
314  sp5_fake_key= _mm_xor_si128(key[14 ] , plaintext[5 + i ] )  ;
315  sp0_resultx10= encrypt_round(sp0_resultx9 , key13 )  ;
316  sp1_resultx7= encrypt_round(sp1_resultx6 , key[10 ] )  ;
317  sp2_resultx4= encrypt_round(sp2_resultx3 , key[7 ] )  ;
318  sp3_resultx1= encrypt_round(sp3_resultx , key[4 ] )  ;
319  sp4_table_entries= _mm_unpacklo_epi64(sp4_t3 , sp4_t6)  ;
320  sp5_my_counter= sp5_scalar_counter ;
321  sp0_resultx11= encrypt_final(sp0_resultx10 , sp0_fake_key)  ;
322  sp1_resultx8= encrypt_round(sp1_resultx7 , key11 )  ;
323  sp2_resultx5= encrypt_round(sp2_resultx4 , key[8 ] )  ;
324  sp3_resultx2= encrypt_round(sp3_resultx1 , key[5 ] )  ;
325  sp4_result2= _mm_xor_si128(second_round_output , sp4_table_entries)  ;
326  sp5_scalar_r0= (_bswap(scalar_key0 ) & 0xFF) ^ sp5_my_counter ;
327  ciphertext[i ] = sp0_resultx11 ;
328  sp1_resultx9= encrypt_round(sp1_resultx8 , key12 )  ;
```

```
329    sp2_resultx6= encrypt_round(sp2_resultx5 , key9 )  ;
330    sp3_resultx3= encrypt_round(sp3_resultx2 , key6 )  ;
331    sp4_resultx= encrypt_round(sp4_result2 , key3 )  ;
332    sp5_scalar_r1= table3[sp5_scalar_r0] ^ first_round_output_x0  ;
333    sp0_fake_key = sp1_fake_key;
334    sp0_scalar_counter = sp1_scalar_counter;
335    sp0_my_counter = sp1_my_counter;
336    sp0_scalar_r0 = sp1_scalar_r0;
337    sp0_scalar_r1 = sp1_scalar_r1;
338    sp0_idx3 = sp1_idx3;
339    sp0_idx2 = sp1_idx2;
340    sp0_idx1 = sp1_idx1;
341    sp0_idx0 = sp1_idx0;
342    sp0_t1 = sp1_t1;
343    sp0_t4 = sp1_t4;
344    sp0_t5 = sp1_t5;
345    sp0_t0 = sp1_t0;
346    sp0_t6 = sp1_t6;
347    sp0_t3 = sp1_t3;
348    sp0_table_entries = sp1_table_entries;
349    sp0_result2 = sp1_result2;
350    sp0_resultx = sp1_resultx;
351    sp0_resultx1 = sp1_resultx1;
352    sp0_resultx2 = sp1_resultx2;
353    sp0_resultx3 = sp1_resultx3;
354    sp0_resultx4 = sp1_resultx4;
355    sp0_resultx5 = sp1_resultx5;
356    sp0_resultx6 = sp1_resultx6;
357    sp0_resultx7 = sp1_resultx7;
358    sp0_resultx8 = sp1_resultx8;
359    sp0_resultx9 = sp1_resultx9;
360    sp1_fake_key = sp2_fake_key;
361    sp1_scalar_counter = sp2_scalar_counter;
362    sp1_my_counter = sp2_my_counter;
363    sp1_scalar_r0 = sp2_scalar_r0;
364    sp1_scalar_r1 = sp2_scalar_r1;
365    sp1_idx3 = sp2_idx3;
366    sp1_idx2 = sp2_idx2;
367    sp1_idx1 = sp2_idx1;
368    sp1_idx0 = sp2_idx0;
369    sp1_t1 = sp2_t1;
370    sp1_t4 = sp2_t4;
371    sp1_t5 = sp2_t5;
372    sp1_t0 = sp2_t0;
373    sp1_t6 = sp2_t6;
374    sp1_t3 = sp2_t3;
375    sp1_table_entries = sp2_table_entries;
376    sp1_result2 = sp2_result2;
377    sp1_resultx = sp2_resultx;
378    sp1_resultx1 = sp2_resultx1;
379    sp1_resultx2 = sp2_resultx2;
380    sp1_resultx3 = sp2_resultx3;
381    sp1_resultx4 = sp2_resultx4;
382    sp1_resultx5 = sp2_resultx5;
383    sp1_resultx6 = sp2_resultx6;
```

```
384    sp2_fake_key = sp3_fake_key;
385    sp2_scalar_counter = sp3_scalar_counter;
386    sp2_my_counter = sp3_my_counter;
387    sp2_scalar_r0 = sp3_scalar_r0;
388    sp2_scalar_r1 = sp3_scalar_r1;
389    sp2_idx3 = sp3_idx3;
390    sp2_idx2 = sp3_idx2;
391    sp2_idx1 = sp3_idx1;
392    sp2_idx0 = sp3_idx0;
393    sp2_t1 = sp3_t1;
394    sp2_t4 = sp3_t4;
395    sp2_t5 = sp3_t5;
396    sp2_t0 = sp3_t0;
397    sp2_t6 = sp3_t6;
398    sp2_t3 = sp3_t3;
399    sp2_table_entries = sp3_table_entries;
400    sp2_result2 = sp3_result2;
401    sp2_resultx = sp3_resultx;
402    sp2_resultx1 = sp3_resultx1;
403    sp2_resultx2 = sp3_resultx2;
404    sp2_resultx3 = sp3_resultx3;
405    sp3_fake_key = sp4_fake_key;
406    sp3_scalar_counter = sp4_scalar_counter;
407    sp3_my_counter = sp4_my_counter;
408    sp3_scalar_r0 = sp4_scalar_r0;
409    sp3_scalar_r1 = sp4_scalar_r1;
410    sp3_idx3 = sp4_idx3;
411    sp3_idx2 = sp4_idx2;
412    sp3_idx1 = sp4_idx1;
413    sp3_idx0 = sp4_idx0;
414    sp3_t1 = sp4_t1;
415    sp3_t4 = sp4_t4;
416    sp3_t5 = sp4_t5;
417    sp3_t0 = sp4_t0;
418    sp3_t6 = sp4_t6;
419    sp3_t3 = sp4_t3;
420    sp3_table_entries = sp4_table_entries;
421    sp3_result2 = sp4_result2;
422    sp3_resultx = sp4_resultx;
423    sp4_fake_key = sp5_fake_key;
424    sp4_scalar_counter = sp5_scalar_counter;
425    sp4_my_counter = sp5_my_counter;
426    sp4_scalar_r0 = sp5_scalar_r0;
427    sp4_scalar_r1 = sp5_scalar_r1;
428  }
429    sp4_idx0= sp4_scalar_r1& 0xFF   ;
430    sp4_idx1= (sp4_scalar_r1>> 8) & 0xFF   ;
431    sp4_idx2= (sp4_scalar_r1>> 16) & 0xFF   ;
432    sp4_idx3= (sp4_scalar_r1>> 24)   ;
433    sp3_resultx1= encrypt_round(sp3_resultx , key[4  ] )   ;
434    sp2_resultx4= encrypt_round(sp2_resultx3 , key[7  ] )   ;
435    sp4_t0= _mm_cvtsi32_si128(table0[sp4_idx0] )   ;
436    sp4_t5= _mm_cvtsi32_si128(table1[sp4_idx1] )   ;
437    sp4_t4= _mm_cvtsi32_si128(table2[sp4_idx2] )   ;
438    sp4_t1= _mm_cvtsi32_si128(table3[sp4_idx3] )   ;
```

```
439    sp1_resultx7= encrypt_round(sp1_resultx6 , key[10 ] )  ;
440    sp0_resultx10= encrypt_round(sp0_resultx9 , key13 )  ;
441    sp3_resultx2= encrypt_round(sp3_resultx1 , key[5 ] )  ;
442    sp2_resultx5= encrypt_round(sp2_resultx4 , key[8 ] )  ;
443    sp1_resultx8= encrypt_round(sp1_resultx7 , key11 )  ;
444    sp0_resultx11= encrypt_final(sp0_resultx10 , sp0_fake_key)  ;
445    sp4_t3= _mm_unpacklo_epi32(sp4_t0 , sp4_t1)  ;
446    sp4_t6= _mm_unpacklo_epi32(sp4_t4 , sp4_t5)  ;
447    sp4_table_entries= _mm_unpacklo_epi64(sp4_t3 , sp4_t6)  ;
448    sp4_result2= _mm_xor_si128(second_round_output , sp4_table_entries)  ;
449    sp4_resultx= encrypt_round(sp4_result2 , key3 )  ;
450    sp3_resultx3= encrypt_round(sp3_resultx2 , key6 )  ;
451    sp2_resultx6= encrypt_round(sp2_resultx5 , key9 )  ;
452    sp1_resultx9= encrypt_round(sp1_resultx8 , key12 )  ;
453    ciphertext[i ] = sp0_resultx11 ;
454    sp4_resultx1= encrypt_round(sp4_resultx , key[4 ] )  ;
455    sp3_resultx4= encrypt_round(sp3_resultx3 , key[7 ] )  ;
456    sp2_resultx7= encrypt_round(sp2_resultx6 , key[10 ] )  ;
457    sp1_resultx10= encrypt_round(sp1_resultx9 , key13 )  ;
458    sp4_resultx2= encrypt_round(sp4_resultx1 , key[5 ] )  ;
459    sp3_resultx5= encrypt_round(sp3_resultx4 , key[8 ] )  ;
460    sp2_resultx8= encrypt_round(sp2_resultx7 , key11 )  ;
461    sp1_resultx11= encrypt_final(sp1_resultx10 , sp1_fake_key)  ;
462    sp4_resultx3= encrypt_round(sp4_resultx2 , key6 )  ;
463    sp3_resultx6= encrypt_round(sp3_resultx5 , key9 )  ;
464    sp2_resultx9= encrypt_round(sp2_resultx8 , key12 )  ;
465    ciphertext[1 + i ] = sp1_resultx11 ;
466    sp4_resultx4= encrypt_round(sp4_resultx3 , key[7 ] )  ;
467    sp3_resultx7= encrypt_round(sp3_resultx6 , key[10 ] )  ;
468    sp2_resultx10= encrypt_round(sp2_resultx9 , key13 )  ;
469    sp4_resultx5= encrypt_round(sp4_resultx4 , key[8 ] )  ;
470    sp3_resultx8= encrypt_round(sp3_resultx7 , key11 )  ;
471    sp2_resultx11= encrypt_final(sp2_resultx10 , sp2_fake_key)  ;
472    sp4_resultx6= encrypt_round(sp4_resultx5 , key9 )  ;
473    sp3_resultx9= encrypt_round(sp3_resultx8 , key12 )  ;
474    ciphertext[2 + i ] = sp2_resultx11 ;
475    sp4_resultx7= encrypt_round(sp4_resultx6 , key[10 ] )  ;
476    sp3_resultx10= encrypt_round(sp3_resultx9 , key13 )  ;
477    sp4_resultx8= encrypt_round(sp4_resultx7 , key11 )  ;
478    sp3_resultx11= encrypt_final(sp3_resultx10 , sp3_fake_key)  ;
479    sp4_resultx9= encrypt_round(sp4_resultx8 , key12 )  ;
480    ciphertext[3 + i ] = sp3_resultx11 ;
481    sp4_resultx10= encrypt_round(sp4_resultx9 , key13 )  ;
482    sp4_resultx11= encrypt_final(sp4_resultx10 , sp4_fake_key)  ;
483    ciphertext[4 + i ] = sp4_resultx11 ;
484  }
```

Appendix D

"Sandy Bridge" Results

Table D.1: Performance of AES-GEN generated code on Sandy Bridge. (*) Results for CTR (round 1) and CTR (round 2) reference 4080B, all others reference 32K.

Encryption	AES-128				AES-256			
	cycles/byte		cycles/round		cycles/byte		cycles/round	
Mode	1K	32K*	1K	32K*	1K	32K*	1K	32K*
Parallel Modes								
CTR	0.945	0.791	1.512	1.266	1.313	1.102	1.501	1.259
CTR-SX	1.020	0.845	1.632	1.352	1.344	1.151	1.536	1.315
CTR-R1*	0.994	0.890	1.590	1.424	1.309	1.182	1.496	1.351
CTR-R2*	1.238	1.119	1.981	1.790	1.523	1.353	1.741	1.546
ECB	0.910	0.754	1.456	1.206	1.238	1.056	1.415	1.207
Cyclic Modes								
CBC1	5.117	5.004	8.187	8.006	7.117	7.004	8.134	8.005
CBC2	2.561	2.502	4.098	4.003	3.561	3.502	4.070	4.002
CBC3	1.710	1.670	2.736	2.672	2.379	2.335	2.719	2.669
CBC4	1.285	1.255	2.056	2.008	1.787	1.755	2.042	2.006
CBC5	1.035	1.022	1.656	1.635	1.436	1.653	1.641	1.889
CBC6	0.911	0.907	1.458	1.451	1.225	1.217	1.400	1.391
CBC7	0.859	0.889	1.374	1.422	1.162	1.219	1.328	1.393
CBC8	0.898	0.937	1.437	1.499	1.205	1.191	1.377	1.361
Authentication Modes								
GCM 1x	3.930	3.669	6.288	5.870	4.242	3.894	4.848	4.450
GCM 4x	3.250	2.815	5.200	4.504	3.613	3.126	4.129	3.573
CCM	5.121	5.005	4.097	4.004	7.137	7.010	4.078	4.006

Table D.2: Performance of AES-GEN generated code on Sandy Bridge in SMT mode. (*) Results for CTR (round 1) and CTR (round 2) reference 4080B, all others reference 32K.

Encryption Mode	AES-128				AES-256			
	cycles/byte		cycles/round		cycles/byte		cycles/round	
	1K	32K*	1K	32K*	1K	32K*	1K	32K*
Parallel Modes								
CTR	0.865	0.809	1.384	1.294	1.189	1.129	1.359	1.290
CTR-SX	0.873	0.831	1.397	1.330	1.203	1.127	1.375	1.288
CTR-R1*	0.867	0.823	1.387	1.317	1.207	1.162	1.379	1.328
CTR-R2*	1.051	0.979	1.682	1.566	1.365	1.303	1.560	1.489
ECB	0.885	0.881	1.416	1.410	1.254	1.108	1.433	1.266
Cyclic Modes								
CBC1	2.568	2.504	4.109	4.006	3.568	3.505	4.078	4.006
CBC2	1.290	1.252	2.064	2.003	1.790	1.754	2.046	2.005
CBC3	0.910	0.884	1.456	1.414	1.250	1.228	1.429	1.403
CBC4	0.851	0.842	1.362	1.347	1.166	1.156	1.333	1.321
CBC5	0.846	0.868	1.354	1.389	1.164	1.189	1.330	1.359
CBC6	0.876	0.915	1.402	1.464	1.193	1.220	1.363	1.394
CBC7	0.893	0.968	1.429	1.549	1.222	1.278	1.397	1.461
CBC8	0.916	0.986	1.466	1.578	1.232	1.283	1.408	1.466
Authentication Modes								
GCM 1x	3.414	3.168	5.462	5.069	3.773	3.525	4.312	4.029
GCM 4x	2.701	2.367	4.322	3.787	3.059	2.766	3.496	3.161
CCM	2.604	2.523	2.083	2.018	4.477	4.426	2.558	2.529