Mississippi State University

Scholars Junction

5-3-2008

A framework for automatically generating optimized digital designs from C-language loops

Wesley James Holland

Follow this and additional works at: https://scholarsjunction.msstate.edu/td

Recommended Citation

Holland, Wesley James, "A framework for automatically generating optimized digital designs from C-language loops" (2008). *Theses and Dissertations*. 130.
https://scholarsjunction.msstate.edu/td/130

A FRAMEWORK FOR AUTOMATICALLY GENERATING OPTIMIZED

DIGITAL DESIGNS FROM C-LANGUAGE LOOPS

By

Wesley James Holland

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computer Engineering
in the Department of Electrical and Computer Engineering

Mississippi State, Mississippi

May 2008

A FRAMEWORK FOR AUTOMATICALLY GENERATING OPTIMIZED

DIGITAL DESIGNS FROM C-LANGUAGE LOOPS

By

Wesley James Holland

Approved:

<div style="display:flex;">

Yoginder Dandass
Assistant Professor of Computer Science
and Engineering
(Major Advisor and Director of Thesis)

J.W. Bruce
Associate Professor of Electrical and
Computer Engineering
(Committee Member)

Robert Reese
Associate Professor of Electrical and
Computer Engineering
(Committee Member)

Nicholas H. Younan
Professor of Electrical and Computer
Engineering
(Graduate Coordinator)

</div>

W. Glenn Steele
Dean of the Bagley College
of Engineering

Name: Wesley James Holland

Date of Degree: May 2, 2008

Institution: Mississippi State University

Major Field: Computer Engineering

Major Professor: Dr. Yoginder Dandass

Title of Study: A FRAMEWORK FOR AUTOMATICALLY GENERATING OPTI-
             MIZED DIGITAL DESIGNS FROM C-LANGUAGE LOOPS

Pages in Study: 56

Candidate for Degree of Master of Science

Reconfigurable computing has the potential for providing significant performance in-
creases to a number of computing applications. However, realizing these benefits requires
digital design experience and knowledge of hardware description languages (HDLs). While
a number of tools have focused on translation of high-level languages (HLLs) to HDLs, the
tools do not always create optimized digital designs that are competitive with hand-coded
solutions. This work describes an automatic optimization in the C-to-HDL transformation
that reorganizes operations between pipeline stages in order to reduce critical path lengths.
The effects of this optimization are examined on the MD5, SHA-1, and Smith-Waterman
algorithms. Results show this technique results in performance gains of 13%-37% and that
the automatically-generated solutions perform comparably to hand-coded solutions.

Key words: reconfigurable computing, electronic design automation, temporal relocation

DEDICATION

To my loving wife.

ACKNOWLEDGMENTS

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS, ABBREVIATIONS, AND NOMENCLATURE

core modular, function-specific, and easily distributable hardware designs typically expressed in an HDL

EDA Electronic design automation

FPGA Field-programmable gate array

FSM Finite-state machine

HDL Hardware-description language

HLL High-level language

LUT Lookup table

MD5 Message Digest 5

SHA-1 Secure Hash Algorithm 1

CHAPTER 1

INTRODUCTION

The current generation of multi-core processors is capable of increased performance for properly designed parallel applications. Nonetheless, even for naturally parallel problems, adding conventional processor cores typically provides a linear speedup at best. One approach for further increasing computational throughput is to utilize reconfigurable computing hardware elements [24].

Field-programmable gate array (FPGA) devices are the most prevalent reconfigurable computing platform. These devices can be configured to implement custom processing units, among other things, in hardware. These so-called *soft* computing elements present an alternative to equivalent functionality implemented in software.

The primary advantage of reconfigurable computing as compared to software is that such systems can be tailored to exploit problem-specific concurrency through hardware parallelism and pipelining. A secondary advantage of reconfigurable computing systems is the lack of overhead of a general-purpose computer (*e.g.*, operating system, memory management, etc.). Reconfigurable computing solutions have consistently been found to have performance advantages for inherently parallel problems [3]. The availability of FPGA devices allows affordable access to custom-designed and dedicated computing hardware.

Consequently, FPGA devices stand to provide significant benefits to a wide variety of research and commercial applications.

1.1 Problem Statement and Motivation

Unfortunately, designing efficient processing elements using hardware description languages (HDLs) requires considerable specialized digital system design knowledge. Researchers without this expertise are left to implement algorithms, even those with inherent parallelism, on a multiprocessor system of limited parallelism at best or on an unnecessarily sequential software system on a general-purpose computer at worst. Examples of some applications which could benefit from hardware parallelism include:

- hash functions for encryption used in communication
- proteomic mapping for homology modeling in bioinformatics
- pattern matching in digital forensics
- atomic bond modeling in chemistry

Efforts at reducing the complexity of digital system design for reconfigurable platforms have been made. Many such efforts have focused on high-level language (HLL) compilation. These solutions allow system developers to leverage existing programming knowledge and programming tools (*e.g.*, compilers and debuggers) to define an algorithm in an HLL (typically ANSI C), which is then translated to an HDL representation for implementation on an FPGA platform. While solutions based on HLL-to-HDL translation show promise, the current generation of HLL- and C-based electronic design automation

(EDA) tools is not always effective at fully exploiting the concurrency inherent in the application sources, resulting in suboptimal performance of the generated design.

Additionally, these tools currently focus on optimizing performance of the resulting hardware by generating *pipelined* designs in which combinational logic is broken into registered stages which can operate concurrently to increase throughput. Pipelining does not work for the class of problems exhibiting *chained* variable access, in which early operations of a stage require the output of later operations of the same stage. Consequently current tools often generate a suboptimal design when faced with problems of this type.

Tools that reduce the complexity of FPGA-based design to the level of a high-level programming language while continuing to generate optimized hardware solutions would allow researchers in many fields to quickly design hardware to meet specialized computing needs. However, the current generation of C-to-HDL tools is too immature to provide the ease-of-use necessary to meet the needs of non-expert digital designers. By developing techniques for automatic optimization in C-based EDA tools, the need for understanding of digital design principles can be reduced. Furthermore, such techniques would also benefit experienced designers by allowing fast design-space exploration for complex designs.

This work describes a technique, *temporal relocation*, for reordering and reorganizing computations performed in successive pipeline stages such that path delays are balanced between the stages. The primary goal of this optimization is to decrease the *critical path* lengths in the resulting design. The critial path in any design is the longest combinational logic path; this path determines the maximum clock frequency at which a design can

3

operate. Consequently, decreasing the critial path length increases the clock frequency and overall throughput. While this optimization is applicable to any design, it is of particular importance in applications containing loops with chained variables accesses. For such applications, the optimization techniques attempt to construct pipeline stages from the remaining non-chained computations while moving the chained accesses to the final stage in the pipeline.

1.2 Hypothesis

The hypothesis of this work is that HLL-based EDA tools can, through the use of automatic temporal relocation optimizations, generate hardware solutions which perform competitively with with hand-generated designs.

1.3 Summary of Main Contributions

The main contributions of this work are as follows:

1. Identification of a temporal relocation optimization that will increase the performance and flexibility of designs generated by C-based EDA tools for problems exhibiting chained variable access

2. Development of methods for generalizing and automating this optimization

3. Testing of this automated optimization both alone and in conjunction with automatic loop unrolling

4. Investigation of the performance advantages of this optimization and comparison of automatically-generated implementations to manually-designed solutions with emphasis on the following algorithms:

   - the MD5 hash function
   - the SHA-1 hash function

4

- the Smith-Waterman algorithm

5. Results that show comparable performance for automatically-generated designs as compared to manual designs

1.4 Organization of Thesis

The organization of this thesis is as follows. Chapter 2 presents background information and a survey of related literature. First, the current state of C-based EDA tools is explored. Next, the algorithms of interest and respective hand-tailored hardware solutions are examined. Chapter 3 presents a novel hardware optimization, as well as an explanation of the loop-unrolling optimization. Also examined is the prototype EDA tool and its operation. After an overview of the tool, the generalization and automation of the novel optimization and loop-unrolling is explored. Chapter 4 presents analysis of the automatically generated solutions and comparisons to hand-tailored solutions. Chapter 5 presents conclusions and questions for further study.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 C-Based EDA Tools

C-based EDA tools are not new, and have been pursued since the 90's with varying

degrees of success. In 1999, [22] outlined some of the difficulties of hardware synthesis

from C/C++. These difficulties, which are still relevant today, include:

- C/C++ are designed to express sequential algorithms, while the advantage of hardware is the ability to exploit a large degree of concurrency;

- hardware circuits require some structural specification that cannot be expressed within C/C++, one example of which is bit-vectors of non-byte-multiple lengths;

- timing constraints, while important in hardware design, cannot be easily expressed within C/C++.

Nonetheless, many continue to see C-based tools as a viable and convenient hardware

design technique. In the following section, an overview of the most popular past and

present C-based tools is presented. However, many C-based EDA tools not discussed here

exist, some with novel aspects. An insightful enumeration of most such tools can be found

in a [9]. This paper, like [22], explores the practicality and current-state of C-like HDLs.

6

2.1.1 Existing Tools

In the following sections, the existing C-based EDA tools are introduced. The operation of these tools and the types of optimizations they make are explored.

2.1.1.1 HardwareC

One of the earliest C-like hardware description languages was HardwareC, described in [23]. Olympus was a synthesis system and one of the earlier tools in field of electronic design automation (EDA). HardwareC served as the HDL for the Olympus synthesis system. Synthesis in the Olympus system was in three stages: the first stage translated HardwareC into an implementation-independent graph-based representation. In this graph, vertices represented operations to be performed and edges represented dependencies. The second stage performed scheduling and resource binding and mapped the graph representation into a logic-level representation. The final stage translated this logic-representation into a technology-dependent netlist. This stage was accomplished as two tasks: resource binding and scheduling. After each stage, the Olympus system provided simulation tools for verification purposes. The Olympus system also performed some automatic transformations/optimization, including:

- fixed-iteration loops were unrolled to provide optimization opportunities;
- variables correctly referenced last assigned values;
- multiple and conditional assignments used multiplexers in hardware;
- redundant operations were removed;
- dead code was eliminated;

- data and control dependencies were identified.

While HardwareC was more C-like than previous HDLs, it fell short of being a true C language or even a subset of the C language. One language-related failing was the inclusion of both "declarative semantics" and "procedural semantics", equivalent to the notion of concurrent statements and sequential statements. This placed the burden of exploiting parallelism on the programmer, rather than the synthesis system.

2.1.1.2 PRISM-II

After HardwareC, there were a number of attempts at a C-subset HDL. This included the PRISM-II Configuration compiler [30], which automatically partitioned a C program into software modules designed to run on a microprocessor and hardware modules implemented in FPGAs. This compiler used the GCC frontend in order to take advantage of the standard optimizations it provided. While innovative, this was not a general-purpose algorithm-to-hardware solution as a microprocessor was required and only the simplest and most easily parallelizable operations were translated to hardware.

2.1.1.3 Transmogrifier C

Another such C-subset HDL was the Transmogrifier C HDL, detailed in [12]. This paper presents as motivation the idea that hardware-based techniques would see more widespread use if they could be made available to the many programmers proficient in C. Transmogrifier C was an improvement over previous C-like HDLs in that it was a

strict subset of C. Additionally, it could implement sequential circuits, a feature not always present in C-like HDLs up to that point. Transmogrifier C used pragma statements and special functions to specify synthesis parameters for which the C language was not equipped. The compiler did not provide many timing configuration options. The timing scheme was essentially as follows:

1. *if* statements and assignment statements corresponded to combinational logic;
2. new FSM stages were started only at the tops of *while* loops and at function calls.

Transmogrifier C also makes no effort to share low-level primitives like comparators and adders, although such structures can be forced to be shared by putting them inside a function. The language also does not allow the use of pointers or arrays.

The Transmogrifier C compiler was constructed using lex and yacc. It scanned over the input C looking for combinational expressions that represented the values of each variable in each state. The combinational expression for each variable was saved as a tree of 4-input lookup tables (LUT)s. The LUTs in each tree were then minimized, and each state's netlist could be outputted as the combinational logic present in each variable's tree. Some simple optimizations performed by this compiler were:

- the boolean expressions for the 4-LUTs were minimized using K-maps (up to 4 variables);
- duplicate 4-LUTs were removed;
- dead code was eliminated.

Overall, the Transmogrifier C compiler was functional, but simplistic in its view of timing configurability and lack of high-level optimizations.

9

2.1.1.4 C2Verilog

In 1998, a new commercial compiler C2Verilog was introduced [33]. C2Verilog translates ANSI C to RTL Verilog. It allows structures, loops, and function calls. With an external memory module, addressable variables such as pointers and arrays are supported. Illustrated in [33] is an example in which the LZW compression and decompression algorithm is implemented in reconfigurable hardware. This algorithm includes arrays, indirections, pointers, loops, and functions. The conclusion presented in [33] was that compiled hardware from C2Verilog was generally slower than the equivalent sequential program running on a Pentium II. Nonetheless, the C2Verilog compiler consistently produced functional hardware implementations with better performance than other automatically-generated solutions. While C2Verilog is one of the most complete solution to date, it has a number of flaws. One criticism of C2Verilog is that it relies completely on the compiler to recognize and exploit parallelism and provides no natural mechanisms to the programmer for expressing parallelism [9]. While C2Verilog provides a mechanism by which timing constraints can be specified, this is an unnatural and unwieldy method of affecting parallelism. Consequently, C2Verilog can, at most, exploit instruction-level parallelism. There are fundamental limits on the amount of concurrency that can be achieved in this manner [35].

2.1.1.5   SystemC

Introduced in 1999 was SystemC [34]. SystemC is an object-oriented language based on C++ that allows description of modules and their interaction. Although a small subset of the language can be synthesized, the purpose of SystemC is not to provide compilation to hardware, but to model and simulate hardware systems. Since 1999, SystemC has been established as an industry standard for system specification and many IP vendors, in addition to providing hardware implementations, provide SystemC specifications for fast simulation of designs.

2.1.1.6   Streams-C

In 2000, the Streams-C programming model was developed [13]. This model is best applied to stream-oriented applications. Such applications are characterized by high data-rate flow through compute-intensive operations. Streams-C works by dividing an application into processes, streams, and signals. A process is "an independently executing object with a process body that is given by a C subroutine." It may run on either a host processor or in synthesized hardware, though processes mapped to synthesized hardware must use only a subset of ANSI C. Streams and signals are used to connect processes. [13] concludes with an example application in which the automatically generated design is found to be three times the area of and half the clock frequency of an equivalent handcrafted design. A 2001 paper [11] further evaluated the Streams-C compiler through four applications and found results consistent with [13], namely that compiler-generated designs

were 1.37-4 times the area and 0.5-1 times the clock frequency of handcrafted designs. Both [13] and [11] found that handcrafted designs took approximately ten times longer to develop than compiler-generated designs. Criticisms of the Stream-C compiler include its deviation from the structural programming model, the limited subset of C allowed for hardware processes, and its focus on stream-oriented applications.

2.1.1.7 Bach EDA

In 2001, a paper was published detailing the Bach EDA tool [10]. This compiler provides mechanisms for explicit parallelism and bit-width specification of data types and arithmetic. Additionally, Bach included the ability to easily handle multiple communicating processes. One criticism of the Bach compiler is that it is not a strict subset of ANSI C; additions were made to the language to support explicit parallelism and non-byte-multiple bit widths. These additions could instead have been implemented as pragma statements. While the difference may seem superficial, the consequence of making additions to ANSI C is that a specialized simulator is required for testing operation of the C code. However, this fault is eased somewhat in that a Bach to ANSI C translator is provided which removes these non-standard structures. An additional criticism of Bach is that it does not support pointer types. One final criticism is that the Bach system performs minimal automatic parallelism and mostly relies on the programmer for explicit parallelism.

2.1.1.8 CASH EDA

In 2002, the CASH EDA compiler was released [4]. CASH can generate standalone hardware or hardware to be used with a hard or soft general-purpose processor. CASH operates by dividing a program into hyperblocks. These structures can be used to uncover instruction-level parallelism as detailed in [21]. This approach allows CASH to uncover more parallelism than comparable compilers. However, CASH suffers from the same deficiency as C2Verilog, in that there is no natural method for specifying parallelism; identifying and exploiting parallelism is strictly the job of the compiler.

2.1.1.9 Handel-C

In 2003, a paper was published describing the use of Handel-C [2]. This paper steps through the methodology of implementing an application in Handel-C. Handel-C allows a small subset of ANSI C, excluding pointers and floating-point arithmetic; it also allows non-byte-multiple length data types. The main criticism of Handel-C is that, as many other EDA tools, it forces programmers to specify parallelism and offers few automatic optimizations.

2.1.1.10 ImpulseC

In 2003, a new commercial compiler was introduced called ImpulseC which translates ANSI C to Verilog [25]. Much like C2Verilog, this product is full-featured, with automatic pipelining and many built-in optimizations. Unlike C2Verilog, ImpulseC provides both

automatic optimization and a natural interface for the specification of manual optimization parameters. ImpulseC is arguably the most complete C-based EDA tool to date.

2.1.1.11   SUIF and DEFACTO

In 1996 the Stanford SUIF Compiler Group published a paper detailing their multiprocessor compiler SUIF which automatically identified and exploited parallelism on multiprocessor systems [15]. More recently, the Information Sciences Institute at the University of Southern California has published work on utilizing SUIF to identify parallelism for FPGA-based designs. Their system, called DEFACTO, serves as a test platform for the many optimizations they have explored, namely:

- automatic selection of loop unrolling factor [37]
- automatic replacement of common array accesses with scalar variables from registers [7]
- automatic division of arrays into separate block memories to maximize memory parallelism [32]
- automatic optimization of communication between sequential pipeline stages [37]

2.1.2   Pipelining and Chaining

In many applications the majority of the processing is performed in loop constructs. C loop semantics generally imply that an iterative process is required for performing the operations specified in the loop body. These operations can be implemented as a single repeated stage in a finite state machine. However, *pipelining* these operations has the potential for significantly speeding up the performance of the loop.

In a pipeline representing a C loop, the loop's body is split into several stages. Each stage is defined by a set of related computations (and the associated FPGA resources) that are performed in a single clock cycle. Stages can operate concurrently with each other such that each stage is operating on different loop iterations. For example, in a two-stage pipeline, when the second stage is operating on iteration i (*i.e.*, the second stage is finishing the processing for iteration i), the first stage is operating on the $(i + 1)^{th}$ iteration (*i.e.*, the first stage is initiating the processing for iteration $i + 1$).

In order to optimize performance of synchronous (*i.e.*, clocked) designs, the delay introduced by logic and routing must be minimized, resulting in high clock frequencies. Therefore, it is important for C-based EDA tools to generate pipelined stages with balanced (*i.e.*, as equal as possible) delay characteristics.

Most C-based EDA tools are capable of generating pipelined designs from C loops because improving throughput via pipelining is an important aspect of high-performance digital circuit design. However, existing tools typically perform a naive analysis of the C codes, often resulting in a non-optimal pipeline implementation. One commonly used scheme is to begin new pipeline stages at the beginning of nested controls structures such as **if** and **while** and at the end of the scopes of such statements. Furthermore, the tools also cannot effectively pipeline loops with chained variable accesses.

A chained variable is one which is modified at the end of a loop iteration and is read at the beginning of the next iteration. Clearly, generating pipelined design is difficult in this

situation because a stage cannot read the correct value of the chained variable while it is still being computed by a subsequent stage.

2.2 Example Applications

Presented below are the example applications chosen for evaluation of the presented temporal relocation optimization. These applications were chosen primarily due to their chaining nature and the difficulties they present to pipelining. Also relevant is their widespread use in real-world systems.

2.2.1 MD5

A hash algorithm is a method for transforming a large amount of data into relatively small string or number which characterizes the data. This characterization or "fingerprint" is called a *hash* of the data. While hash functions have a variety of uses in all areas of computer science, a large portion of hash function research is in the area of *cryptographic hash functions*. This branch of hash functions is utilized in security applications such as encryption, integrity checking, digital signatures, and authentication. The primary application of such hash functions is as a primitive in protocols like IPSec, SSL, and WAP, which are used for encryption and message authentication. Presently, the two most commonly used hash algorithms are MD5 and SHA-1, though security vulnerabilities have been identified in both. In this section and the one that follows, the MD5 and SHA-1 algorithms are explained in detail in order to enable understanding of optimizations made to their hardware implementations.

Message digest 5 (MD5) was developed by Ron Rivest at MIT in 1991 [26]. This algorithm has found widespread use in integrity checking for Internet downloads and password storage. MD5 transforms a variable-length input message into a 128-bit digest or hash. The algorithm proceeds as follows:

1. The message is padded as follows until its length is divisible by 512 bits:

   (a) A single '1' bit is appended to the message

   (b) '0' bits are appended until the message reaches a length that is 64 bits less than a multiple of 512 bits

   (c) The remaining 64 bits are filled with the 64-bit integer representing the length of the original message in bits

2. A 128-bit state variable is designated that is divisible into four 32-bit words, A, B, C, and D.

3. The state variable is initialized to certain fixed constants

4. The algorithm operates on each 512-bit block in turn and modifies the state variable as follows:

   - The block is operated upon in 64 sub-rounds as described in Figure 2.1.

   - A sub-round is illustrated in Figure 2.2, where f_t denotes a non-linear function described below, M_t denotes a 32-bit block of the message input, S_t denotes a left-shift by an iteration-dependent constant, and K_t denotes a 32-bit constant which is different for each sub-round

   - Each round (or 16 sub-rounds) uses a different non-linear function as described below:
     $$F_{0<=t<16} = (B \wedge C) \vee (\neg B \wedge D)$$
     $$F_{16<=t<32} = (B \wedge D) \vee (C \wedge \neg D)$$
     $$F_{32<=t<48} = B \oplus C \oplus D$$
     $$F_{48<=t<64} = C \oplus (B \vee \neg D)$$

17

```
For t = 0 to 63 do:
        Temp = D;
        D = C;
        C = B;
        B = B + S_t(A + f_t(B,C,D) + W_t + K_t);
        A = Temp;
```

Figure 2.1

MD5 block processing loop
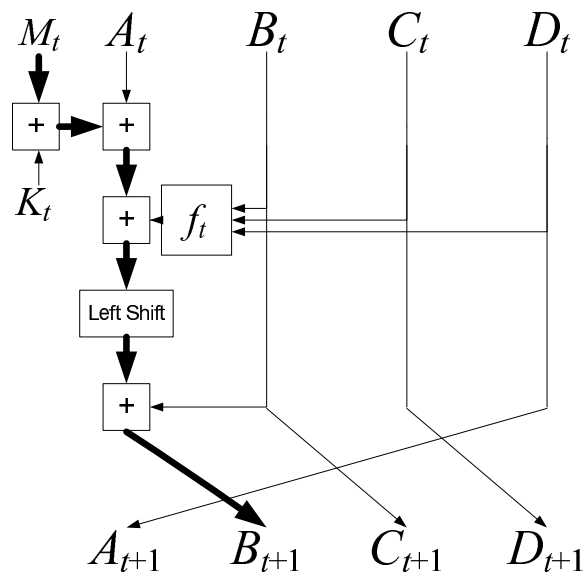
Figure 2.2

Single sub-round of MD5 algorithm

2.2.2 SHA-1

The Secure Hash Algorithm (SHA) designation encompasses five cryptographic hash functions developed by the National Security Agency starting in 1995 as a replacement for MD5 [1]. These algorithms are SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512. The earliest and most widespread of these algorithms, SHA-1, has found use in protocols like SSL, PGP, SSH, and IPSec as well as in integrity checking for Internet downloads and password storage.

SHA-1 encompasses many of the design principles of the MD5 algorithm. Similar to MD5, SHA-1 uses a state variable that is modified as the message body is iterated over; the produced digest is 160-bits. The algorithm proceeds as follows:

1. The message is padded as follows until its length is divisible by 512 bits:

    (a) A single '1' bit is appended to the message
    (b) '0' bits are appended until the message reaches a length that is 64 bits less than a multiple of 512 bits
    (c) The remaining 64 bits are filled with the 64-bit integer representing the length of the original message in bits

2. A 160-bit state variable is designated that is divisible into five 32-bit words, A, B, C, D, and E.

3. The state variable is initialized to certain fixed constants

4. The algorithm operates on each 512-bit block in turn and modifies the state variable as follows:

    - The sixteen 32-bit words are extended into eighty 32-bit words with words 16-79 defined as

$$w[i] = (w[i-3] \oplus w[i-8] \oplus w[i-14] \oplus w[i-16]) << 1$$

- The resulting 80-word array is operated upon in 80 sub-rounds as described in Figure 2.3, where W_t denotes a 32-bit block of the eighty-word array, K_t denotes a 32-bit constant which is different for each sub-round, and the function f_t changes every twenty sub-rounds (or one round) as follows

$$F_{0<=t<20} = (B \wedge C) \vee (\neg B \wedge D)$$
$$F_{20<=t<40} = B \oplus C \oplus D$$
$$F_{40<=t<60} = (B \wedge C) \vee (B \wedge D) \vee (C \wedge D)$$
$$F_{60<=t<80} = B \oplus C \oplus D$$

- The generalized implementation of a sub-round can be found in Figure 2.4

```
For t = 0 to 79 do:
        Temp = S₅(A)+f_t(B,C,D)+E+W_t+K_t;
        E = D;
        D = C;
        C = S₃₀(B);
        B = A;
        A = Temp;
```

Figure 2.3

SHA-1 block processing loop

2.2.3 Hardware Implementations of Hash Functions

Any system intended to facilitate a large amount of secure communication must necessarily have a high-throughput hash implementation. In systems where a high-performance microprocessor is unavailable, hash functions are typically implemented in hardware along with other cryptographic primitives. Consequently, hardware implementations are important to the viability of secure mobile communication, since such applications rarely have
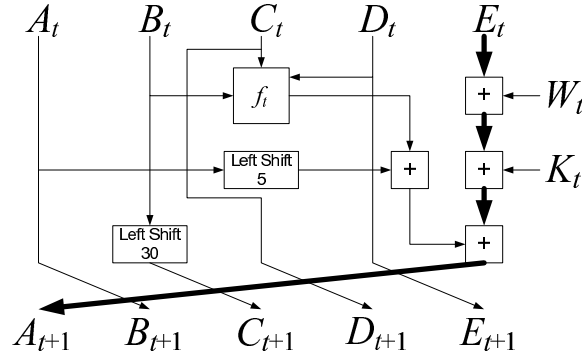
Figure 2.4

Single sub-round of SHA-1 algorithm

the capabilities or power for performing software-based encryption. In addition to performance, another reason for hardware implementations of hash functions is the inherently more secure nature of hardware when compared to software.

In the following sections, common hardware implementations of the MD5 and SHA-1 algorithms are explored in order to enable understanding of the generated generalized hardware optimizations.

The two common physical implementations of the algorithms are single-stage and multiple-stage, with the multiple-stage implementations varying in number of stages [5] [18]. In the single-stage implementation, a single sub-round of the algorithm is generalized and used repeatedly for each sub-round of the main MD5 or SHA-1 loop. This is accomplished as followed:

- multiplexers are used, most notably for selection of the results of the non-linear functions *F*, *G*, *H*, and *I*

- a variable shifter, often a barrel shifter, is used rather than wire transposition to accommodate the varying shift distances at different stages of the algorithm

21

- a state machine with a large number of states is used to correctly control the state of multiplexers, shift-length, and memory addresses

The single-stage design is obviously optimized for area, though its throughput can be comparable to that of some multiple-stage implementations [5] [17].

In the multiple-stage implementation, the sub-rounds of the algorithm are divided into stages with the goal of finding common elements in groups of sub-rounds that will allow a number of sub-rounds to be implemented succintly with a single stage. This can often create a series of stages each of which has a shorter critical path than a single stage implementing a completely generalized iteration. The most obvious example of a multiple-stage implementation is a fully-unrolled implementation. Such an implementation uses different dedicated hardware for each of the 64 or 80 sub-rounds of MD5 or SHA-1, respectively. This has a number of notable advantages, namely:

- multiplexers are not necessary for selecting the appropriate output from F, G, H, and I

- wire transpositions may be used to shift and complicated variable shifters are not needed

- sub-round-specific constants can be hardwired into the design, rather than stored in a lookup table or RAM

- a simpler state machine may be used

These multiple-stage designs are not fully pipelined, as stages do not execute concurrently. This is due to the chaining property of the MD5 and SHA-1 algorithms, which dictates that the next round cannot begin until the previous round has completed. Thus, the multiple-stage design serves only to reduce clock period. However, several efforts have

focused on true pipelining by weaving multiple hash computations in the same multiple-stage system [17]. These systems, apart from having higher resource requirements, require that multiple different hash calculations be requested in order to reach maximum throughput (*i.e.*, they cannot reach maximum throughput for only one computation). Because of this constraint, such solutions are not general-purpose and are considered outside the scope of this work.

2.2.4   Smith-Waterman Algorithm

The Smith-Waterman algorithm is used in the field of bioinformatics for sequence alignment of nucleotides or proteins. This algorithm uses a dynamic programming approach to find locally optimal alignments in two strings P and Q [31]. The main body of the Smith-Waterman algorithm is described in Figure 2.5. In the figure, $M[i,j]$ is a chained variable because the computation of $M[i,j]$ depends on the value of $M[i,j-1]$ computed in the previous iteration. Figure 2.6 illustrates the computations performed in a single Smith-Waterman inner loop. The critical path in this design is highlighted using boldface directional lines.

In the inner loop of the Smith-Waterman algorithm, the value of i is a constant. Also, the array read operations corresponding to $M[i-1,j-1]$, $M[i-1,j]$, $Q[j]$, and $P[i]$ are scheduled as part of the loop indexing scheme, and therefore, these values are available at the beginning of the loop iteration in registers $M_{i-1,j-1}$, $M_{i-1,j}$, q, and p, respectively. Similarly, the functions $\gamma('-',Q[j])$, $\gamma(P[i],'-')$, and $\sigma(P[i],Q[j])$ are also implemented

as table lookups as part of the loop indexing scheme, and therefore, their values are available in registers γ_q, γ_p, and $\sigma_{p,q}$, respectively.

For $i = 1$ to *length*(P) do:
   For $j = 1$ to *length*(Q) do:
     $M[i,j] = max(M[i,j\text{-}1] + \gamma(`\text{-}', Q[j]),$
                $M[i\text{-}1,j] + \gamma(P[i], `\text{-}'),$
                $M[i\text{-}1,j\text{-}1] + \sigma(P[i], Q[j]),$
                $0);$

Figure 2.5

Smith-Waterman block processing loop

While some earlier FPGA implementations of Smith-Waterman utilized single processing elements, such solutions are no longer considered viable. All recent research implements Smith-Waterman as a systolic array of processing elements that implement the block processor in Figure 2.6. Thus far, most research has focused on optimization of the systolic array structure and nearly all implementations have used a naive implementation of the block processing elements [16] [36]. This work focuses on automatic optimization of these block processors, for which there is little basis for comparison.
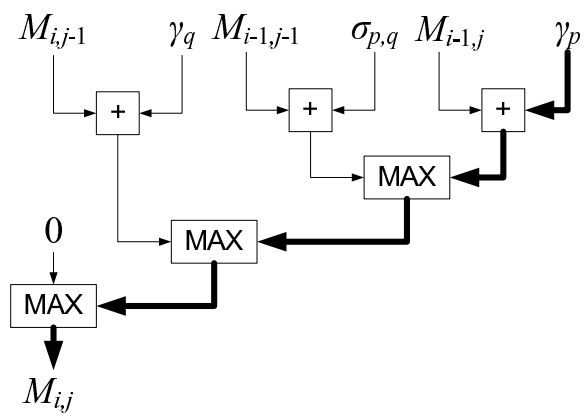
Figure 2.6

Single stage of Smith-Waterman algorithm

CHAPTER 3

METHOD

3.1 Optimizations

In C, loops are used to express repetition. When translated to hardware, they are typically implemented as a single pipeline stage which corresponds to a generalized iteration. This single stage executes a number of times depending on the C loop boundaries.

The following two C-to-HDL optimizations for C loops are introduced:

- partial loop unrolling
- temporal relocation

Partial loop unrolling has been studied in detail for a number of applications and has been automated in a number of tools [25] [15] [23] [33]. The other optimization, temporal relocation, is novel but applicable only to classes of problems in which full concurrent pipelining is impossible or difficult because of variable chaining.

3.1.1 Partial Loop Unrolling

One method for increasing the throughput of some C-based hardware designs is partial loop unrolling. A loop iteration may be described by a long sequence of operations. Loop unrolling enables one or more iterations to be combined into a single stage. This has the

potential for overlapping the operations in the combined iterations. While this technique increases the amount of combinational logic between registers and the critical path length, there is a greater amount of computation per clock period, at the cost of reduced clock frequency.

An example of partial loop unrolling in the SHA-1 algorithm can be found in Figure 3.1. In the original design of Figure 2.4, which is equivalent to the top-half of Figure 3.1, the critical path proceeds through the column of three adders on the right. While this critical path still exists, combining two stages allows the next iteration to begin before the previous iteration has finished. Specifically, the first two additions of the next stage can begin immediately because they do not depend on any calculation from the previous stage.

Partial loop unrolling allows designers the choice of having a given stage implement multiple iterations of a loop instead of just one. While this may decrease performance for pipelined designs with concurrent stage execution, it can often provide a performance boost for non-pipelined designs. Whether or not the application is pipelined, partial loop unrolling is always a valuable tool as it allows designers to experiment with the resource versus performance tradeoff.

3.1.2 Temporal Relocation of Operations

Another method for increasing throughput of some C-based hardware designs is relocation of operations between loop iterations. The principle of this optimization is that
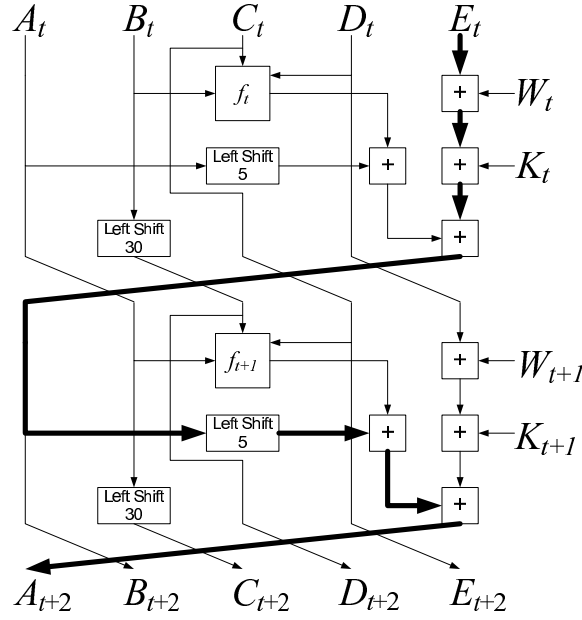
Figure 3.1

Partial loop unrolling on the SHA-1 algorithm

moving a computation to a previous iteration can balance out delay paths in a given iteration, decreasing the critical path length, and consequently increasing clock speed. At the HDL level, this optimization corresponds to rearranging statements within a given stage. However, performance improvement comes at a cost of increased memory, because the results of temporally relocated computations must be stored across stage boundaries until they are required.

An example of temporal relocation in the SHA-1 algorithm can be found in Figure 3.2. In the original design of Figure 2.4, the critical path proceeds through the column of three adders on the right. After temporal relocation of two adders to the previous iteration, the critical path proceeds through the f_t function and the two leftmost adders. Since the f_t

28

functions are simple logical functions and much faster than an addition, temporal reloca-

tion effectively removes an entire adder from the critical path.
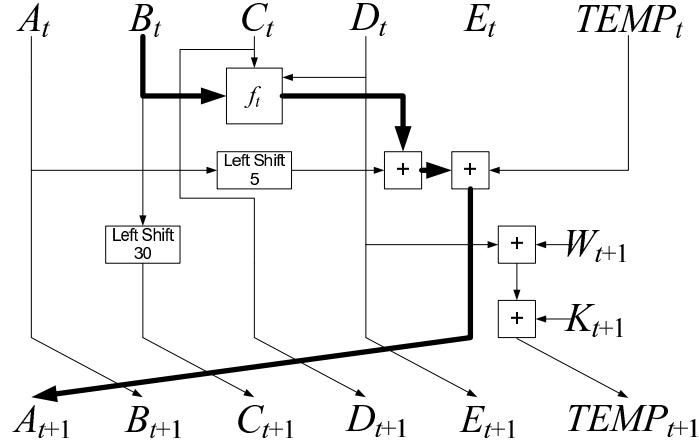
Figure 3.2

Temporal relocation on the SHA-1 algorithm

Care must be taken to preserve functionality during temporal relocation. While this

naturally involves tracing dependencies between operations and ensuring that all are met,

consideration must also be given to start and end loop conditions. In the above example,

the correct functioning of the first loop iteration requires that $TEMP$ be calculated in

the previous clock cycle. This requirement can be satisfied by duplicating the logic for

calculating $TEMP$ in the module's reset block. In many situations, it is also necessary

to ensure that register values upon loop exit hold proper values as opposed to values from

the partial $(n + 1)^{th}$ iteration computed during the terminal n^{th} iteration. This can be en-

sured by registering the results of all relocated operations only when the loop termination condition is not met in the current clock cycle.

In addition to constraints that preserve functionality, the process of selecting operations for temporal relocation that will reduce critical path length is nontrivial; the relocation used in the above example is not the only possible one for SHA-1. One method for identifying the best statements for relocation is through the use of a delay model that specifies delays associated with various operations on a particular technology or platform. Such a model can be used to compute the effects of temporal relocation on critical path length. This allows individual temporal relocations to be applied iteratively using a greedy procedure in order to determine the best set of relocation operations. Any relocations that increase critical path delays are rejected.

The most noteworthy property of the optimization is that its effectiveness decreases as the critical path length approaches the average operation time. This is because the optimization depends on being able to overlap computations with the critical path of the previous stage or iteration. If the critical path length is comparable to that of a single operation, the likelihood of accomplishing this overlap is low. In short, the usefulness of this optimization is limited in highly pipelined systems. However, temporal relocation can provide significant performance benefits for applications with chained variable access.

3.2 Implementation Details

Exploration of optimizations to the C-to-HDL transformation requires an C-to-HDL compiler into which said optimizations can be incorporated. Though some earlier C-based EDA tools are open-source, they lack the functionality necessary for these optimizations and in most cases documentation is poor. Nearly all modern C-based EDA tools are proprietary. Consequently, a prototype platform was developed for testing the temporal relocation optimization.

3.2.1 Operation of Prototype Tool

The basic operation of the prototype tool is described in the following paragraphs. A diagram overview of this tool may be found in Figure 3.3. As is evident from Figure 3.3, the particular HLL and HDL chosen were ANSI C and Verilog respectively. However, it should be noted that the front- and back-ends are implemented as interchangeable modules for which other languages could be substituted. The tool itself is written in C/C++ with some code sections generated by LeX and YACC as detailed in the following sections.

The tool's preprocessor is HLL-specific. In this case, it is a simple ANSI C preprocessor that handles preprocessor directives such as **#include**, **#define**, **#ifdef**, etc. It also substitutes **#define**d values into appropriate places in the code. Lastly, it strips out all comments and unnecessary whitespace. After preprocessing, the code is ready to be parsed and is entirely devoid of "#" directives except for **#pragma**s, which are left to be parsed and passed to the compiler.

31
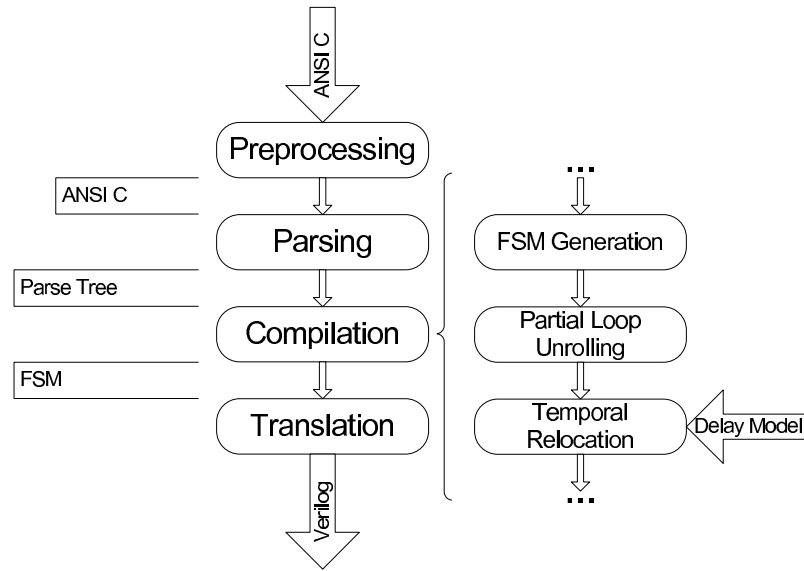
Figure 3.3

Overview of prototype tool

The parser used in the prototype tool was constructed using the LeX lexical analyzer and the YACC parser generator. The LeX-generated lexical analyzer tokenizes the C code, and the YACC-generated parser uses a full ANSI C grammar to parse the tokens. The **#pragma** statements which control compiler options are also tokenized and parsed. The output of this parser is a parse tree, the format of which is similar to that used in traditional compilers. At this point, the parse tree, which is still C-specific, is passed to the compiler.

The compiler is the most complex stage of the prototype tool's operation. The first step of compilation is the conversion of the C to an FSM. While generic C-to-FSM conversion is nontrivial, many non-general or restricted conversions have been studied extensively in

the papers presented in Chapter 2. The prototype tool, like many of the tools of Chapter 2, places restrictions upon the C structures that may be used. Most notably:

- pointers are not allowed

- **struct**s and **union**s are not allowed

- dynamic memory allocation is not allowed

- recursive functions are not allowed

- control structures may not be nested within pipeline blocks

The primary output of FSM generation is an HLL-independent tree-based data structure which is easily translated to an HDL. The secondary output of FSM generation is metadata that identifies the C structures from which FSM elements were generated. It is necessary to preserve this information into the optimization steps, because it allows the optimizer to make assumptions about the behavior of a particular FSM that would be difficult and compute-intensive to prove otherwise.

After FSM generation, the two optimizations are performed, assuming they were requested through the use of **#pragma** statements. The output of the optimizations is in the same format as that generated by FSM generation, namely an HLL-independent tree-based data structure.

Following compilation, the final stage of the tool is translation to an HDL. The simplicity of most HDLs makes this a trivial process, with majority of the effort involving simple aspects such as:

- enumeration of registers in the form of variable declaration

- identification and declaration of inputs and outputs

33

- construction of sensitivity lists

- implementation of the FSM

  - expression of FSM next-state logic
  - expression of FSM output logic

3.2.2 Optimization Implementation

The following sections discuss the implementation of the loop unrolling and temporal relocation optimizations in the prototype tool.

3.2.2.1 Partial Loop Unrolling

Implementation of partial loop unrolling is straightforward. The process begins with the output of FSM generation, in which each C loop corresponds to a single FSM stage that implements a generalized iteration of the loop. It ends with a single FSM stage that implements a generalized number of consecutive iterations of the loop.

The steps for partially unrolling this single-iteration loop stage into an x-iteration loop stage are as follows:

1. duplicate the combinational logic for the stage x times

2. for each duplicated stage:

   (a) instantiate logic to correctly compute the loop-index of the duplicated stage
   (b) replace all uses of the loop-index in the duplicated stage with the computed loop-index
   (c) where appropriate, replace inputs of duplicated stage with outputs from previous stage

3. combine the original stage with all duplicated stages to form a new single FSM stage

In the prototype tool, loop unrolling is allowed on any C loop structure through the use of a **#pragma** statement, specifically **#pragma combine_iterations x** where x is the number of consecutive iterations to combine.

3.2.2.2 Temporal Relocation of Operations

Temporal relocation is more complex than partial loop unrolling. The steps to optimize a stage using temporal relocation are as follows:

1. search the FSM stage for an operation to move to the previous iteration according to the following criteria:

   - relocation of the operation must not alter loop functionality (i.e. operations must not depend upon the results of any previous operations in this stage)
   - relocation of the operation must not lengthen the critical path according to the user-provided delay model

2. if such an operation is found, move it to the previous iteration by placing it at the end of the FSM stage

3. repeat steps 1-2 until an acceptable operation is not found

4. duplicate any moved operations in the reset logic such that the first loop iteration, which is structured on these operations having been completed in the previous iteration, will operate correctly

5. ensure that results of relocated operations are registered only when the loop termination criteria is not met in the current clock cycle

The computationally-intensive portion of this process is location of an appropriate operation for relocation. The dependency-checking step is accomplished by means of internal representation of loop operations used in the prototype tool, namely a dependency graph. In this structure, operations eligible for relocation are those which exhibit no dependencies in the dependency graph.

35

The second step in determining an appropriate operation for relocation is verifying that a candidate for relocation will not increase critical path length. This is accomplished by temporarily making the relocation then computing the new critical path length for comparison to the old critical path length. Given the user-specified delay model, critical path calculation is trivially accomplished with a recursive algorithm operating on the operation dependency graph. If the new critical path is longer than the original critical path, the relocation is rejected; otherwise, it is made permanent.

In the implementation of temporal relocation in the prototype tool, the following simple but effective delay model is used: each operation is assigned a constant delay value expressed in multiples of an inverter-delay according to Table 3.1. These model parameters can be customized for any specific technology platform in order to accommodate devices with different speed grades and resources (*e.g.*, on-chip multipliers, accumulators, and shift registers). Note that this delay model does not include bit-widths of data types; while ignoring width is effective for a system in which majority of operations are of a particular fixed width, a more advanced delay model would account for width differences.

In the prototype tool, temporal relocation is allowed on any C loop structure through the use of a **#pragma** statement, specifically **#pragma optimize_loop**.

Once an operation is found for relocation, the actual relocation in the prototype tool's internal representation is relatively straightforward. The operations in the loop are represented as a tree data structure. Relocation of an operation is represented by relocating a node in the tree (and re-computing the dependency graph to account for the relocation).

Table 3.1

Delay model values used in prototype tool

Operation	Delay Estimation
NOT	1
REGISTER	1
CONSTANT SHIFT	1
AND/OR/XOR	2
MAX/MIN	3
ADDITION/SUBTRACTION	4
MULTIPLICATION	10
VARIABLE SHIFT	10
DIVISION	20

Once optimization of the tree structure is complete, the operation tree and dependency graph are transformed into Verilog.

CHAPTER 4

EVALUATION

In the following sections, the automatically-generated optimized designs are presented.

They are then compared to the architectures of the automatically-generated unoptimized

designs and the hand-coded designs. Specifically, for the selected applications, the effect

of various combinations of temporal relocation and partial loop-unrolling are explored.

Additionally, the performance values of the automatically-generated SHA-1 and MD5 im-

plementations are compared with hand-coded implementations from the literature. This

comparison is not conducted for the Smith-Waterman algorithm, as differing systolic ar-

ray structures between implementations make direct comparison impractical. Despite the

absence of comparison to manual solutions, Smith-Waterman is nonetheless presented as

an example to further examine the effects of temporal relocation in a domain other than

hash functions.

The algorithms are expressed in ANSI C and translated to Verilog with the prototype

tool. All Verilog is synthesized for the Virtex 4 XC4VFX100 FPGA for design space

exploration. The MD5 and SHA-1 implementations are also synthesized for the dominant

research platforms for each, the Virtex 2 XC2V4000 and the Virtex 1 XCV150 FPGA

respectively.

The simple delay model described in Chapter 3 is used for temporal relocation, with the delay values specified in Table 3.1.

4.1 Implementations

The following sections present the automatically-generated optimized implementations produced by the prototype tool with no loop unrolling. A complete enumeration of the designs produced for all possible loop unrolling factors would be of limited usefulness and prohibitive length; consequently, such is not presented.

4.1.1 MD5

The automatically-generated stage with optimized temporal relocation for MD5 can be found in Figure 4.1. The critical path of this implementation is two adders and the f_t function. This is shorter than the critical path of the naive stage implementation of Figure 2.2. This is accomplished by moving two additions to the previous cycle.

4.1.2 SHA-1

The automatically-generated stage with optimized temporal relocation for SHA-1 can be found in Figure 4.2. When compared to the naive stage implementation of Figure 2.4, the most significant alteration due to temporal relocation is the decrease of the critical path from three adders to two adders. This is accomplished by moving to the next iteration the two adders responsible for adding the output of the f_t function to the left-shifted value of A and the sum of $W[t]$ and $K[t]$. A secondary relocation is the relocation of the left-shift of
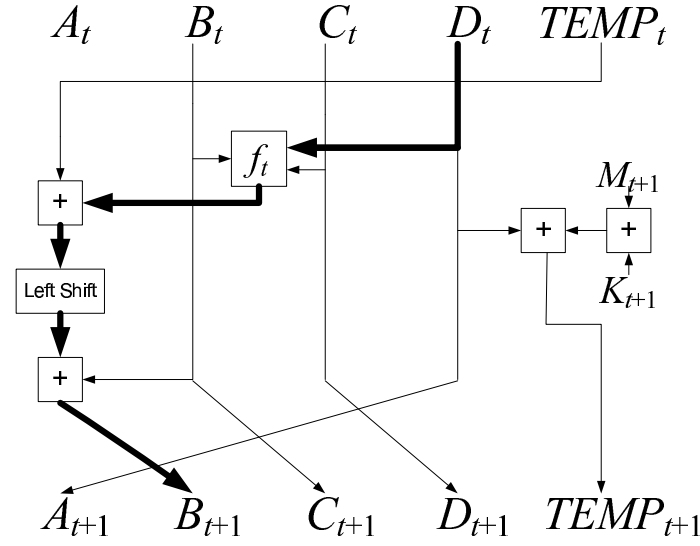
Figure 4.1

Temporal relocation by prototype tool in MD5

A to the next iteration. Though this does not change the critical path (as constant-shifts are implemented as zero-delay wire transpositions), this relocation is made in preparation for the relocation of any operations which depended on this value and has no effect on critical path length. This implementation is better than the example relocation of Figure 3.2, because the critical path length is now two adders instead of two adders in series with the non-linear function f_t.

4.1.3 Smith-Waterman

The automatically-generated stage with optimized temporal relocation for Smith-Waterman can be found in Figure 4.3. The critical path of this implementation is one adder and two MAX functions. This is shorter than the critical path of the naive stage
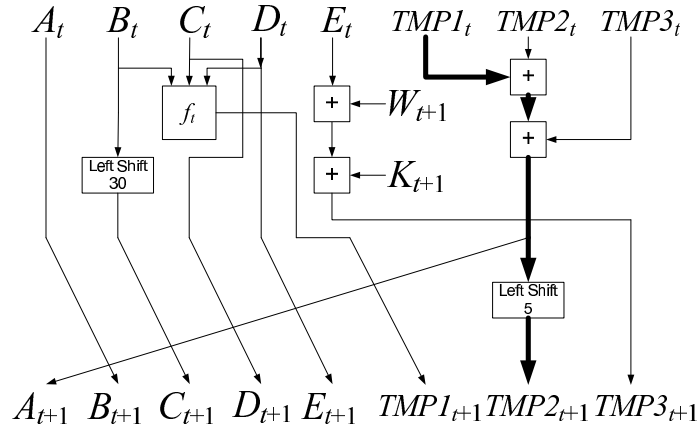
Figure 4.2

Temporal relocation by prototype tool in SHA-1

implementation of Figure 2.6 which is one adder and three MAX functions. This is accomplished by splitting the design such that two additions and a MAX execute in parallel with the later half of the iteration. This results in a slightly unbalanced pipeline stage. However, further improvement is not possible (clearly, evenly dividing an odd number of MAX operations is impractical).

4.2 Performance

In the following sections, the performance of the various automatically-generated solutions are presented and compared to that of both automatically- and manually-generated designs. For the purposes of this document, throughput is defined for SHA-1 and MD5 as in Equation (4.1).

$$throughput = \frac{\#bits \cdot f_{operation}}{\#operations} \qquad (4.1)$$
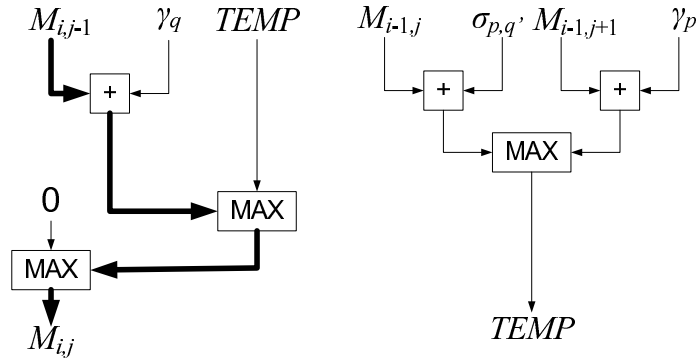
Figure 4.3

Temporal relocation by prototype tool in Smith-Waterman

with $\#bits$ equal to the number of bits in the calculation, $f_{operation}$ equal to the frequency at which operations are performed, and $\#operations$ equal to the number of clock cycles before a full result is calculated. Throughput for Smith-Waterman is defined as in Equation (4.2).

$$throughput = f_{operation} \cdot \frac{cells}{clock} \qquad (4.2)$$

Where $f_{operation}$ is equal to the frequency at which operations are performed and $\frac{cells}{clock}$ is equal to the number of loop iterations which are computed per clock cycle.

4.2.1 MD5

Performance of the automatically-generated MD5 designs with various loop-unrolling factors are listed in Table 4.1. For throughput calculations, $\#bits = 512$ corresponding to the 512-bit block size of MD5 and $\#operations$ varies from 64 to 10 as $\#operations =$

$\frac{64}{L_f} + 2$ where L_f is the loop-unrolling factor and the $+2$ is due to the two additional clock cycles for loading from and storing to the hash register.

Table 4.1

MD5 performance on Virtex 4 XC4VFX100

Loop-Unrolling Factor	Design	Clock Speed (MHz)	Throughput (Mbps)	Improvement (%)
-	unoptimized	105.64	819.51	0.0%
-	optimized	119.15	924.32	12.8%
2	unoptimized	70.76	1065.56	30.0%
2	optimized	76.97	1159.08	41.4%
4	unoptimized	43.50	1237.33	50.9%
4	optimized	45.51	1294.51	58.0%
8	unoptimized	23.98	1227.77	49.8%
8	optimized	24.44	1251.33	52.7%

As expected, the results of Table 4.1 show that the temporally-relocated design invariably outperforms the unoptimized design of the same loop-unrolling factor. Also evident is a decrease in effectiveness as the loop-unrolling factor grows. For comparison purposes, the optimized design with no loop-unrolling was synthesized for the Virtex 2 XC2V4000 FPGA that has been utilized by other researchers in the past. The results from this comparison are listed in Table 4.2.

The optimized implementation outperforms all hand-constructed implementations except for the 725Mbps, 2395Mbps, and 5857Mbps implementations in [17]. Both the 2395Mbps and 5857Mbps implementations use high-throughput techniques such as par-

Table 4.2

MD5 comparison on Virtex 2 XC2V4000

Design	Clock Speed (MHz)	Throughput (Mbps)
Holland	88.3	685
Deepakumara, *et al.* [5]	21.0	165
Deepakumara, *et al.* [5]	71.4	354
Diez, *et al.* [6]	60.2	467
Dominikus [8]	42.9	146
Jarvinen, *et al.* [17]	75.5	586
Jarvinen, *et al.* [17]	78.3	607
Jarvinen, *et al.* [17]	93.4	725
Jarvinen, *et al.* [17]	80.7	2395
Jarvinen, *et al.* [17]	75.5	5857

allel cores and a pipelining organization in which stages operate on different messages concurrently. Both of these techniques are beyond the scope of these research. Of the presented implementations, only the 725Mbps outperforms the optimized implementation using conventional techniques. In particular, the 725Mbps implementation uses a completely unrolled loop (*i.e.*, 64 separate stages, one for each iteration).

4.2.2   SHA-1

Performance of the automatically-generated SHA-1 designs with various loop-unrolling factors are listed in Table 4.3. For throughput calculations, $\#bits = 512$ corresponding to the 512-bit block size of SHA-1 and $\#operations$ varies from 82 to 12 as $\#operations = \frac{80}{L_f} + 2$ where L_f is the loop-unrolling factor and the $+2$ is due to the two additional clock cycles for loading from and storing to the hash register.

Table 4.3

SHA-1 performance on Virtex 4 XC4VFX100

Loop-Unrolling Factor	Design	Clock Speed (MHz)	Throughput (Mbps)	Improvement (%)
-	unoptimized	171.29	1069.52	0.0%
-	optimized	235.41	1469.88	37.4%
2	unoptimized	139.47	1700.20	58.9%
2	optimized	169.35	2064.46	93.0%
4	unoptimized	110.11	2562.56	139.6%
4	optimized	120.71	2809.23	162.7%
8	unoptimized	78.93	3367.68	214.9%
8	optimized	82.99	3540.91	231.1%

As with MD5, the results of Table 4.3 show that the temporally-relocated designs out-perform the unoptimized designs of the same loop-unrolling factor. The same decrease in effectiveness as the loop-unrolling factor grows is also observed. For comparison purposes, the optimized design with no loop-unrolling was synthesized for the Virtex 1 XCV150 FPGA that has been utilized by other researchers in the past. The results from this comparison are listed in Table 4.4.

The optimized implementation outperforms the hand-constructed implementations in [8], [27], [14], [19]. The remaining implementations report a significantly higher through-put than the automatically-generated design because they divide the 80 SHA-1 iterations into four blocks of 20 iterations such that each block is performed in a separate pipeline stage. This way, each of the four stages can operate on different messages concurrently.

Table 4.4

SHA-1 comparison on Virtex 1 XCV150

Design	Clock Speed (MHz)	Throughput (Mbps)
Holland	86.0	537
Dominikus [8]	43.0	119
Selimis, *et al.* [27]	72.0	461
Grembowski, *et al.* [14]	86.0	530
Kang, *et al.* [19]	18.0	114
Sklavos, *et al.* [29]	71.0	1731
Sklavos, *et al.* [28]	72.0	1843
Lien, *et al.* [20]	64.0	1024
Kakarountas, *et al.* [18]	98.7	2527

However, this type of pipeline organization requires complex support logic that multi-plexes blocks from different messages, and therefore, is beyond the scope of this research.

4.2.3 Smith-Waterman

Performance of the automatically-generated Smith-Waterman designs with various loop-unrolling factors are listed in Table 4.5. The bit-width of variables in the Smith-Waterman algorithm is dependent upon the alphabet size of the input sequences. This affects both clock speed and throughput. For these experiments, a bit-width of 32 was chosen to provide sequence alignment for unicode strings, though in bioinformatics the alphabet is typically much smaller and input bit-widths of 4-5 are common with γ and σ widths of 3-4 bits. Since bit-width is varible from application to application in Smith-

Waterman, the throughput is specified in $cells/second$ rather than Mbps, where a cell is the result of one iteration's calculation.

Table 4.5

Smith-Waterman performance on Virtex 4 XC4VFX100

Loop-Unrolling Factor	Design	Clock Speed (MHz)	Throughput (Mcells/s)	Improvement (%)
-	unoptimized	102.79	102.79	0.0%
-	optimized	134.78	134.78	31.1%
2	unoptimized	62.80	125.60	22.2%
2	optimized	72.76	145.52	41.6%
4	unoptimized	35.29	141.16	37.3%
4	optimized	37.89	151.56	47.4%
8	unoptimized	18.79	150.32	46.2%
8	optimized	19.33	154.64	50.4%

As with MD5 and SHA-1, the results of Table 4.5 show that the temporally-relocated designs outperform the unoptimized designs of the same loop-unrolling factor. The same decrease in effectiveness as the loop-unrolling factor grows is also observed.

4.2.4 Analysis

One noteworthy trend is that the performance increase due to temporal relocation decreases as loop-unrolling factor increases. This is illustrated in Figure 4.4. The explanation for this is that loop-unrolling naturally shifts the critical path to the chained variable paths by exploiting non-chained concurrency; this leaves little concurrency for temporal relocation to exploit.

Nonetheless, temporal relocation improved performance at all loop-unrolling factors for all three algorithms. Furthermore, the optimized versions of MD5 and SHA-1 were shown to perform comparably to hand-coded solutions for these algorithms as hypothesized.
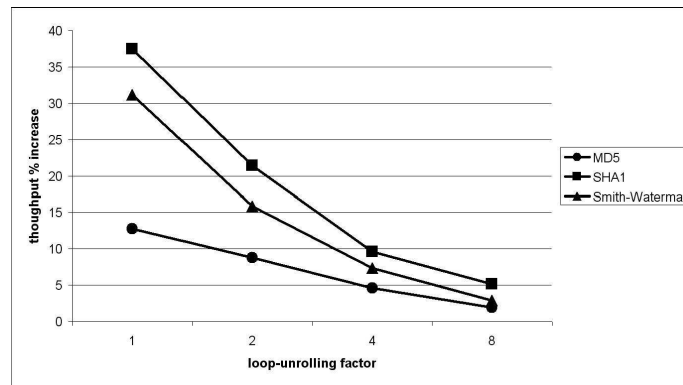
Figure 4.4

Throughput % increase vs. loop-unrolling factor

4.3 Place and Route

To explore resource utilization and performance degradation due to routing, the various MD5 algorithms were placed and routed for the Virtex 4 XC4VFX100. The MD5 algorithm was chosen for its complexity when compared to SHA-1 and Smith-Waterman. The results of these experiments can be found in Table 4.6. When compared to the synthesized performance of MD5, a maximum performance decrease of 16.3% was observed, with the average performance decrease being 11.7%.

Table 4.6

MD5 place and route results on Virtex 4 XC4VFX100

Loop-Unrolling Factor (#)	Design	Clock Speed (MHz)	Throughput (Mbps)	Slices	Flip-Flops
-	unoptimized	100.05	776.15	772	307
-	optimized	111.40	864.19	933	403
2	unoptimized	64.80	975.81	1368	307
2	optimized	69.12	983.04	1514	404
4	unoptimized	37.56	1068.37	2512	307
4	optimized	38.11	1084.02	2523	420
8	unoptimized	20.32	1040.39	5638	308
8	optimized	21.33	1092.10	5446	421

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

5.1 Summary

During the course of this research, a prototype C-to-HDL tool was created which implemented a translation from ANSI C to a Verilog hardware description. This tool implemented a novel automatic temporal relocation optimization which automatically relocates operations to previous stages and iterations to reduce critical path length and increase clock speed and throughput. The prototype tool was used to test temporal relocation on three applications, namely MD5, SHA-1, and Smith-Waterman, each of which exhibits the undesirable property of having chained variable access. For these applications, performance increases of 13%-37% were measured. The effect of loop-unrolling upon temporal relocation was also examined. It was found that the benefit of temporal relocation decreased with high loop-unrolling factors.

5.2 Conclusions

HLL-based EDA tools continue to be a valid and capable design method, as both the optimized and unoptimized automatically-generated implementations of our example applications performed competitively when compared to similar hand-coded solutions.

Additionally, this research has demonstrated that HLL-based EDA tools allow optimizations which would be difficult to implement at the lower HDL level, as current HDL-based tools cannot grasp the overall structure and functionality of a design. Also, while the assertion is difficult to prove, HLL-based design at least arguably reduces the expert digital design knowledge requirement of hardware design and consequently makes the benefits of reconfigurable computing available to a broader community.

Furthermore, temporal relocation is a valid optimization worth integrating into existing HLL-based design tools. In the example applications, performance increased 13%-37% depending upon application. And though this effect decreased with loop-unrolling, such unrolling is not always possible due to resource constraints. Consequently, temporal relocation has its place in the arsenal of automatic optimizations.

5.3 Future Work

Future work includes:

- investigation of the effects of temporal relocation on problems which do not exhibit chained variable access and comparison to hand-coded and automatically-generated pipelined solutions

- investigation of improved delay models for temporal relocation

- investigation of the interaction between temporal relocation and automatic storage access optimization in which an HLL-based tool automatically makes decisions on where to place data (BRAM, registers, etc.), how to access data (BRAMs typically allow a limited number of simultaneous accesses), and how to position data for optimal access (data can be split among multiple BRAM blocks)

- investigation of the effects of temporal relocation on non-loop programming constructs

- comparison of automatically-generated designs with temporal relocation to automatically-generated designs without temporal relocation

REFERENCES

[1] F. P. 180-1, "Secure Hash Standard," National Institute of Standards and Technology (NIST), 1995.

[2] A. Abdallah and J. Hawkins, "Formal Behavioral Synthesis of Handel-C Parallel Hardware Implementations from Functional Specifications," *Proceedings of 36th Annual Hawaii International Conference on System Sciences*, 2003, p. 278.

[3] M. Budiu, P. V. Artigas, and S. C. Goldstein, "Dataflow: A Complement to Super-scalar," *Proceedings of the IEEE Intern. Symp. on Performance Analysis of Systems and Software*, 2005.

[4] M. Budiu and S. C. Goldstein, "Compiling application-specific hardware," *Proceedings FPL, LNCS 2438*, Montpellier, France, 2002, pp. 853–863.

[5] J. Deepakumara, H. Heys, and R. Venkatesan, "FPGA Implementation of MD5 Hash Algorithm," *Proceedings of the Canadian Conference on Electrical and Computer Engineering*, Toronto, Canada, 2001, pp. 919–924.

[6] J. M. Diez, S. Bojanic, L. Stanimirovice, C. Carreras, and O. Nieto-Taladriz, "Hash Algorithms for Cryptographic Protocols: FPGA Implementations," *Proceedings of the 10th Telecommunications Forum*, Belgrade, Yugoslavia, 2002.

[7] P. Diniz, M. Hall, J. Park, B. So, and H. Ziegler, "Bridging the gap between compilation and synthesis in the DEFACTO system," *Proceedings of the 14th Workshop Languages and Compilers for Parallel Computing*, 2001.

[8] S. Dominikus, "A Hardware Implementation of MD4 Family Hash Algorithms," *International Conference on Electronics, Circuits and Systems*. IEEE, 2002, pp. 1143–1146.

[9] S. A. Edwards, "The Challenges of Hardware Synthesis from C-like Languages," *Proceedings of Design, Automation, and Testing*, 2005.

[10] T. K. et al, "A C-based synthesis system, Bach, and its application," *Proceedings ASP-DAC*, Yokohama, Japan, 2001, pp. 151–155.

[11] J. Frigo, M. Gokhale, and D. Lavenier, "Evaluation of the Streams-C C-to-FPGA Compiler: An Application Perspective," *Proceedings of ACM/SIGDA International Symposium on FPGAs*, Monterey, CA, 2001, pp. 134–140.

[12] D. Galloway, "The Transmogrifier C Hardware Description Language and Compiler for FPGAs," *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 1995.

[13] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski, "Stream-Oriented FPGA Computing in the Streams-C High Level Language," *In. IEEE Symp. on Field-Programmable Custom Computing Machines*, Napa Valley, CA, 2000, pp. 49–56.

[14] T. Grembowski, R. Lien, K. Gaj, N. Nguyen, P. Bellows, J. Flidr, T. Lehman, and B. Schott, "Comparative Analysis of the Hardware Implementations of Hash Functions SHA-1 and SHA-512," *Information Security Conference*, Heidelberg, 2002, pp. 75–89.

[15] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S. Liao, E. Bugnion, and M. Lam, "Maximizing Multiprocessor Performance with the SUIF Compiler," *IEEE Computer*, 1996.

[16] D. T. Hoang, "Searching genetic databases on Splash 2," *IEEE Workshop on FPGAs for Custom Computing Machines*, 1993, pp. 185–191.

[17] K. Jarvinen, M. Tommiska, and J. Skytta, "Hardware Implementation Analysis of the MD5 Hash Algorithm," *Proceedings of the 38th Hawaii International Conference on System Sciences*, 2005.

[18] A. P. Kakarountas, H. Michail, A. Milidonis, C. E. Goutis, and G. Theodoridis, "High-Speed FPGA Implementation of Secure Hash Algorithm for IPSec and VPN Applications," *The Journal of Supercomputing*, 2006, pp. 179–195, Volume 37.

[19] Y. K. Kang, D. W. Kim, T. W. Kwon, and J. R. Choi, "An Efficient Implementation of Hash Function Processor for IPSEC," *Asic-Pacific Conference on ASIC*. IEEE, 2002.

[20] R. Lien, T. Grembowski, and K. Gaj, "A 1 Gbit/s Partially Unrolled Architecture of Hash Functions SHA-1 and SHA-512," *Cryptographers Track at RSA Conference*, Berlin, 2004, Springer-Verlag, pp. 324–338.

[21] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," *Proceedings of the 25th International Symposium on Microarchitecture*, 1992, pp. 45–54.

[22] G. D. Micheli, "Hardware synthesis from C/C++ models," *Proceedings of DATE*, Munich, Germany, 1999, pp. 382–383.

[23] G. D. Micheli, D. Ku, F. Mailhot, and T. Truong, "The Olympus Synthesis System," *IEEE Design and Test of Computers*, 1990, pp. 37–37.

[24] K. Morriz, "Cray Goes FPGA: Algorithm Acceleration in the New XD1," *FPGA and Programmable Logic Journal*, 2005.

[25] D. Pellerin and S. Thibault, "Practical FPGA Programming in C," Upper Saddle River, New Jersey, 2005, Prentice Hall.

[26] R. Rivest, "The MD5 Message-Digest Algorithm," *RFC 1321*. MIT LCS and RSA Data Security, Inc., 1992.

[27] G. Selimis, N. Sklavos, and O. Koufopavlou, "VLSI Implementation of the Keyed-Hash Message Authentication Code for the Wireless Application Protocol," *International Conference on Electronics, Circuits and Systems*. IEEE, 2003, pp. 24–27.

[28] N. Sklavos, E. Alexopoulos, and O. Koufopavlou, "Networking Data Integrity: High Speed Architectures and Hardware Implementations," *IAJIT*, 2003, pp. 54–59.

[29] N. Sklavos, P. Kitsos, E. Alexopoulos, and O. Koufopavlou, "Open Mobile Alliance (OMA) Security Layer: Architecture, Implementation, and Performance Evaluation of the Integrity Unit," *Computing Paradigms and Computational Intelligence*. Springer-Verlag, 2004.

[30] A. Smith, M. Wazlowski, L. Agarwal, E. L. T. Lee, P. Athans, H. Silverman, and S. Ghosh, "PRISM II Compiler and Architecture," *Proceedings of the IEEE Workshop on FPGA-based Custom Computing Machines*, Napa, CA, 1993, pp. 9–16.

[31] T. Smith and M. Waterman, "Identification of Common Molecular Subsequences," *Journal of Molecular Biology*, 1981, pp. 198–197.

[32] B. So, H. Ziegler, and M. Hall, "A compiler approach for custom data layout," *Proceedings of the 15th Workshop Languages and Compilers for Parallel Computing*, 2002.

[33] D. Soderman and Y. Panchul, "Implementing C Algorithms in Reconfigurable Hardware using C2Verilog," *Proceedings of FCCM*, 1998, pp. 339–342.

[34] SystemC, "SystemC:Welcome," Internet: http://www.systemc.org/, May 2007, Jul. 4, 2007.

[35] D. W. Wall, "Limits of instruction-level parallelism," *Proceedings ASPLOS*, New York, NY, 1991, pp. 176–189.

[36] Y. Yamaguchi, T. Maruyama, and A. Konagaya, "High Speed Homology Search with FPGAs," *Proceedings of the Pacific Symposium on Biocomputing*, 2002, pp. 271–282.

[37] H. Ziegler, M. Hall, and B. So, "Search Space Properties for Mapping Pipelined FPGA Applications," *Proceedings of the 16th Workshop Languages and Compilers for Parallel Computing*, 2003.