An Optimizing C Compiler for DSP Architectures

by

Jeremy H. Lin

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degrees of Bachelor of Science in Electrical Engineering and Computer Science and Master of Engineering in Electrical Engineering and Computer Science

at the
Massachusetts Institute of Technology

May 21, 1999

© Copyright 1999 Jeremy H. Lin. All rights reserved.

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

Author

Department of Electrical Engineering and Computer Science
May, 1999

Certified by

Martin Rinard
Faculty Supervisor

Accepted by

Arthur C. Smith
Chairman, Department Committee on Graduate Theses

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUL 15 1999
An Optimizing C Compiler for DSP Architectures
by
Jeremy H. Lin

Submitted to the
Department of Electrical Engineering and Computer Science

May 21, 1999

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

ABSTRACT

This is a discussion of an optimizing C compiler for digital signal processors used in real-time embedded systems. A cost-function based design compiler is introduced. A proof of concept is written, and tests are performed to determine the suitability and feasibility of the system for commercial use.

Faculty Supervisor: Martin Rinard
Title: Assistant Professor of Electrical Engineering and Computer Science

Thesis Supervisor: John DeJaco
Title: Director, QUALCOMM, Incorporated
Table of Contents

1 INTRODUCTION ...................................................................................................................................... 7
  1.1 DSP ARCHITECTURE ............................................................................................................................. 8
  1.2 EXISTING DSP APPLICATION DESIGN SOLUTIONS ........................................................................ 8
    1.2.1 Hand-coded DSP Assembly ........................................................................................................ 8
    1.2.2 Existing C Compilers ................................................................................................................. 10
  1.3 GOAL .................................................................................................................................................... 11

2 PROPOSAL ............................................................................................................................................... 12
  2.1 TARGET AUDIENCE .......................................................................................................................... 12
    2.1.1 Intelligent Programmers ............................................................................................................. 12
    2.1.2 Optimized Code Not Required Quickly ...................................................................................... 12
  2.2 HIGH LEVEL DESIGN ....................................................................................................................... 13
    2.2.1 Cost Function – Separation of function and optimization .......................................................... 13
    2.2.2 Instruction Selection .................................................................................................................. 14
    2.2.3 Design Compiler ........................................................................................................................ 16

3 LOW LEVEL DESIGN/IMPLEMENTATION ............................................................................................ 17
  3.1 TARGET ARCHITECTURE – THE QDSP .......................................................................................... 17
  3.2 C CODE ................................................................................................................................................ 18
  3.3 C CODE TO NODE ILR ....................................................................................................................... 20
  3.4 NODE ILR ............................................................................................................................................. 21
  3.5 NODE ILR TO BLOCK ILR ................................................................................................................... 23
    3.5.1 Instruction Selection ..................................................................................................................... 23
    3.5.2 Optimizations ................................................................................................................................ 25
    3.5.3 ISA Dependent/Architecture Implementation Independent .......................................................... 25
    3.5.4 Implementation .......................................................................................................................... 26
  3.6 BLOCK ILR ........................................................................................................................................... 27
  3.7 DSP ASSEMBLY PARALLELLIZATION AND COMPACTION ........................................................... 28
    3.7.1 Purpose ......................................................................................................................................... 28
    3.7.2 Implementation ............................................................................................................................ 29
  3.8 PARALLELIZED BLOCK LEVEL TO ASSEMBLY CODE LEVEL ..................................................... 29
    3.8.1 Purpose ......................................................................................................................................... 29
    3.8.2 Separation of Parallelization from Resource Allocation ............................................................... 29
    3.8.3 Implementation .......................................................................................................................... 30
  3.9 COST FUNCTION ................................................................................................................................... 30
    3.9.1 Implementation ............................................................................................................................ 30

4 PERFORMANCE AND FEASIBILITY ........................................................................................................ 32
  4.1 SETUP ................................................................................................................................................ 32
  4.2 OPTIMIZATION OPTIMALITY ............................................................................................................ 35
  4.3 FEASIBILITY ....................................................................................................................................... 38
    4.3.1 Small Functions ............................................................................................................................ 39
    4.3.2 Large Functions ........................................................................................................................... 39
  4.4 COMPILTE TIME TRADEOFFS ............................................................................................................. 39
    4.4.1 Programmer Directed Optimizations ............................................................................................. 39
    4.4.2 Non-exhaustive Iterative Search .................................................................................................. 40
    4.4.3 Non-iterative Search Techniques .................................................................................................. 41
    4.4.4 Separating Dissimilar Code Segments .......................................................................................... 42

5 CONCLUSION .......................................................................................................................................... 44

6 FUTURE WORK ..................................................................................................................................... 46
List of Figures

FIGURE 1. STEPS OF TRADITIONAL C COMPILERS ....................................................... 16
FIGURE 2. STEPS OF IMPLEMENTED DESIGN COMPILER ................................................ 18
FIGURE 3. FIR CIRCULAR FILTER C CODE ................................................................... 20
FIGURE 4. C++ CODE USED TO CREATE FIR CIRCULAR FILTER NODE LEVEL REPRESENTATION .......................................................... 22
FIGURE 5. FIR CIRCULAR FILTER NODE LEVEL REPRESENTATION ................................ 23
FIGURE 6. SIMPLIFIED CORE LOOP OF IMPLEMENTED DESIGN COMPILER ...................... 27
FIGURE 7. FIR CIRCULAR FILTER BLOCK LEVEL REPRESENTATION ................................ 28
FIGURE 8. FIR CIRCULAR FILTER REPRESENTATION AFTER PARALLELIZATION STEP ..................... 29
FIGURE 9. FIR CIRCULAR FILTER ASSEMBLY CODE ..................................................... 30
FIGURE 10. FIR CIRCULAR FILTER CostInfo Analysis .................................................... 32
FIGURE 11. C++ CODE FOR TEST COST FUNCTIONS .................................................. 33
FIGURE 12. Vector Sum C Code ..................................................................................... 34
FIGURE 13. Loop Unroll C Code ..................................................................................... 35
FIGURE 14. FIR CIRCULAR FILTER OPTIMAL SPEED COST SOLUTION ...................... 37
FIGURE 15. Best Cost (Mix) versus Iteration Number for Combined Function ................... 40
FIGURE 16. Best Cost (numCycles) versus Iteration Number for FIR CIRCULAR FILTER ...... 41
FIGURE 17. Cost (numCycles) versus Iterations for FIR CIRCULAR FILTER .................... 42
FIGURE 18. Cost (numInstBits) versus Iterations for FIR CIRCULAR FILTER .................... 42
List of Tables

TABLE 1. QDSP Instruction Packet Possibilities for L0 = 7 .......................................................... 15
TABLE 2. Internal Memory Structure Descriptions ............................................................................. 18
TABLE 3. Possible C Intrinsic Functions ............................................................................................. 19
TABLE 4. Selected Node Types and Descriptions ............................................................................... 22
TABLE 5. BlockInfoIter Member Functions and Descriptions ......................................................... 26
TABLE 6. CostInfo Member Functions and Descriptions ................................................................. 31
TABLE 7. Example Cost Functions and Descriptions ......................................................................... 33
TABLE 8. FIR Circular Filter Results ................................................................................................. 35
TABLE 9. Vector Sum Function Results .............................................................................................. 36
TABLE 10. Loop Unroll Function Results .......................................................................................... 36
TABLE 11. Compile Time for Example Functions ................................................................................ 38
TABLE 12. Combined FIR Circular Filter and Vector Sum Function Results ..................................... 43
Acknowledgements

I owe everything to God for the wonderful life He has given me. I would like to thank my parents for all the love and support throughout the years. They have always been there for me whenever I needed them. I owe so much to my thesis supervisors, John DeJaco of QUALCOMM Inc. and Martin Rinard of MIT, for all the guidance and help with my thesis. It could not have been done without them. I give great thanks to Rob Gilmore, who helped make QUALCOMM a great place to work and for his support of the VI-A program. I would also like to commend Dawn Grob and the other employees of QUALCOMM who helped make this thesis possible. Finally, I am grateful for all the support and understanding of all my friends. They have made my five years at MIT a wonderful and fun experience.
1 Introduction

The demand for digital signal processing solutions has increased dramatically in recent years. The driver has been the incredible growth of the communications industry. Historically, the ideal choice for signal processing solutions has been hardwired application specific integrated circuits (ASICs). Programmable digital signal processors (DSPs) have been used as coprocessors to offload expensive mathematical operations and signal processing functions. Now, programmers are seeking to increase efficiency and decrease costs by moving even greater portions of their applications into DSP firmware. As competition has increased and development times have shortened, some companies have embraced DSP chips as an alternative to hardwired solutions – moving entire applications onto the DSP and using it more as a general-purpose microprocessor.

For example, early modems used a many chip solution including several complex ASICs with DSP coprocessors. Today, many 56k modems are single chip DSP solutions. The modem is implemented completely in DSP firmware and stored in flash RAM. This allows manufacturers to offer an easy upgrade path to such standards as V.90 and is extremely cost effective.

However, programming a DSP has always been a difficult task. Thus, there is a great need for programming tools that simplify the development process. This thesis focuses on the research of a tool to allow fast design of DSP applications for real-time embedded systems. The target DSP is QUALCOMM’s DSP v2 (QDSP2). Its assembly code and capabilities are similar to many commercial DSP products, like Analog Device’s SHARC DSP.
1.1 **DSP Architecture**

DSPs are explicitly parallel, very long instruction word (VLIW) machines. Most DSPs execute one instruction word per cycle. Each instruction word can contain many separate instruction packets. These instruction packets control independent functional units in the DSP. DSPs do not use any advanced pipelining techniques like out of order execution or register renaming. In order to save time, space, and have the fastest possible processor for the cost, the DSP relies on either the programmer or the compiler to resolve all data hazards and order the packets/instructions at compile time.

The QDSP2 can perform a multiply/accumulate (MAC) operation, an arithmetic logic unit (ALU) operation, 3 load/stores, and 3 address manipulations all in a single cycle. The amount of processing power in these chips is extraordinary considering their power consumption and cost. That makes DSPs an ideal choice for embedded applications like cellular phones, home stereos, and other communication devices.

1.2 **Existing DSP Application Design Solutions**

There are only a few choices available for DSP application designers today. The majority of systems are written in native DSP assembly. Others choose to use C compilers distributed by the DSP manufacturer.

1.2.1 **Hand-coded DSP Assembly**

Hand-coding native DSP assembly is difficult work. The programmer must code and maintain complex low-level assembly. With most DSPs having over five functional units, it is not an easy task to create optimal code for these explicitly parallel machines. Writing highly optimized code requires the programmer to unroll sections of loops, interleave pieces of code that have nothing to do with each other, and create other
complicated hacks to squeeze every usable cycle out of the machine. In addition, DSP assembly programmers have to write code that fits within certain resource usage requirements. All this is difficult, and oftentimes production-level, hand-written assembly applications are not optimal for the task.

Debugging DSP assembly is a nightmare. Highly optimized code is difficult to comprehend. Every instruction contains multiple operations, so the programmer usually must single step through the program in an emulator to discover the complex interrelationships among instructions. Finding a small mistake may take hours in even small applications.

Once the code is written and debugged, it is nearly impossible to maintain. Usually, even a minor change in the system requires the DSP assembly to be rewritten. There is little to no reuse among different versions of the same application because such specialized optimizations have been performed on the code. It is quicker to rewrite the code from scratch than attempt to understand and modify the original code.

As DSP programs get larger, these problems will only grow. Also, these problems do not grow linearly with size. Tradeoffs in one section of the code may drastically affect another section. A two thousand instruction DSP application is much more than two times as difficult to code and maintain than a one thousand instruction DSP application. So, an alternative to hand-coded DSP applications must be used in the future. Hand-coding a large system in DSP assembly is simply not practical.
1.2.2 Existing C Compilers

Current generation DSP manufacturer-supplied C compilers are not well suited for embedded, real-time DSP applications.

This is primarily because different projects have unique resource usage requirements. Most real-time DSP applications need to complete a number of critical calculations within some worst-case time frame. For instance, a certain project may require that a code fragment execute in less than 128 cycles because of real-time processing constraints. However, another project may only have constraints on total instruction RAM (IRAM) or power usage. Code density, execution speed, and total RAM usage are important considerations in evaluating the code. In embedded applications, all resource usage has a cost, and the compiler should take that into account.

DSP C compilers do not allow the designer to adequately describe the real-time constraints of the application. Most compilers only allow the programmer to feed it one bit of optimization information: compile for speed or compile for size. Usually, there is a known execution speed/code size tradeoff in a project, yet compilers ignore this fact.

For this reason, many application designers opt to compile for speed at all times even though that may not be the most efficient use of available resources. Embedded application programmers need the ability to have intimate control of these resources. The result is that a large number of companies still only use hand-coded DSP assembly in their products.
1.3 **Goal**

The existing compilers for embedded DSP programming will not be sufficient for the large DSP applications of the future. This thesis proposes a compiler that overcomes many of the shortcomings of the current generation tools.

The overarching goal is to design a compiler that an embedded system designer would use as an alternative to any of the current generation products. In order to do that, it should satisfy the following objectives:

- Ability to design for embedded system constraints dictated by the programmer
- Code accessibility
- Ease of code maintenance
- Possibility of code reuse
- Ease of debugging
- Output assembly performance comparable to hand-coding
- Limit the use of non-portable inline assembly

If these objectives are met, the result will be shorter product development cycles and better applications at lower cost.
2 Proposal

2.1 Target Audience

This compiler is directed towards intelligent embedded system DSP programmers that are designing real-world products.

2.1.1 Intelligent Programmers

In the worst case, the programmer only needs to know C. It will generate DSP firmware that is equivalent to or better than current generation C compilers.

Yet, the ideal users of this tool are programmers that have the skill and knowledge to write hand-coded assembly for the target DSP. Programmers for consumer electronics cannot be oblivious to the constraints of the system. They can use this tool to generate and maintain applications that are significantly larger than they could otherwise. In other words, programmers that want all the speed and power of assembly, without the disadvantages.

2.1.2 Optimized Code Not Required Quickly

Current generation compilers output the final assembly in seconds or minutes. If required, this compiler also can generate a working version of the firmware quickly. However, in order for this compiler to generate fully optimized assembly, it will take time. It may take hours or even days to generate optimized assembly.

Actually, this should not be much of a problem. While testing and debugging the application, a less optimized version of the firmware can be used. When the high level
code is fully debugged, all the compiler optimizations can then be switched on. Good assembly will take a relatively long time to create compared to current generation compilers. Yet, the slower compile time is a good tradeoff for better resulting assembly. Commercial vendors could just execute the compiler for days on a dedicated machine for the final release version. For embedded applications, compilation time is a one-time cost, whereas better firmware benefits every product manufactured.

2.2 High Level Design

The primary focus of the thesis is to develop ideas and design a possible implementation for future versions of QUALCOMM’s C compiler for its QDSP. By using a cost function based design compiler that employs iterative instruction selection and loop optimizations, excellent code can be generated. The following are descriptions of the important ideas proposed.

2.2.1 Cost Function – Separation of function and optimization

This thesis proposes a cost function based compiler. The cost function is written or supplied by the programmer. In the most general sense, a cost function describes the global resource constraints and costs in the system. It can also be viewed as a value representing how optimized the output code is. Simple cost function examples include the size in bytes of instruction code (optimize for size) or the expected number of cycles a specific program fragment takes to execute (optimize for speed). With high-level code and a cost function, the compiler will attempt to create low-level assembly of minimum cost. Notice that trivial cost functions can be used to emulate existing compiler optimization possibilities.
When comparing two possible assembly variations of the same high-level code, the cost function determines which is better. The better assembly is the one with lower cost. By changing the cost function used, high-level code can be reused for applications with different resource requirements. The same high-level C algorithm might have significantly different low-level assembly depending on the given application. The cost function implicitly selects which optimizations should be used, and unlike hand-coded assembly, these specific optimizations do not disturb the readability of the code.

A cost function allows the programmer to describe non-linear design tradeoffs. This is important from a production standpoint, yet no compiler supports it. For example, the monetary difference in production costs between a 1023 or 1024 byte instruction ROM may be negligible, but going to a 1025 byte ROM might increase costs dramatically because of the cost of additional address lines or other real-world factors. Similarly, code that is even a single cycle over the real-time constraints imposed by the system is useless.

Cost functions are an easy way for programmers to encapsulate the optimization needs of the system. In the remaining part of the thesis, I will refer to optimizations in relation to the cost function.

2.2.2 Instruction Selection
There are usually many sets of low-level instructions that are functionally equivalent in high level code. In DSPs, which combination should be used can only be determined by viewing the surrounding parallel code possibilities and the programmer's cost function needs.
For example, these are some of the ways to load the L0 register with the constant value 7 in the QDSP:

<table>
<thead>
<tr>
<th>#</th>
<th>Packets</th>
<th>Description (Packet Type and Bit Size)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>L0 = 7;</td>
<td>Load the L0 register with 7 (DL40).</td>
</tr>
</tbody>
</table>
| 2 | CLEAR(L0) 
L0 = 7;              | L0 = 0 (ALU8).                                             |
| 3 | L0 = BIT(0); 
L0 = L0 | BIT(1); 
L0 = L0 | BIT(2);       | L0 = 1 (ALU16). 
L0 = L0 | 2 (ALU16). 
L0 = L0 | 4 (ALU16);                 |

Table 1. QDSP Instruction Packet Possibilities for L0 = 7

Each version has its own advantages and disadvantages. The first example generates the L0 value the fastest, but it cannot execute in parallel with any other packets. The second example has a minimum latency of 2 cycles to set the L0 register to 7, but has more parallel combination possibilities. The third example takes a minimum of 3 cycles and has totally different combination possibilities.

For traditional compilers targeted at microprocessors, the obvious choice would be to use the first possibility. However, on DSP architectures, the instruction packet combination possibilities and code space savings must be taken into account. Notice that example 1 does not necessarily generate the fastest code on a DSP. Examples 2 or 3 might marginally take zero cycles because they can be executed in parallel with the surrounding instructions. Sometimes, instruction packets can be executed for “free” because it combines with another parallel instruction and adds no code space.
2.2.3 Design Compiler

Given the cost function and high-level code, there is no easy way to choose the best underlying instruction packet mapping. Instruction packet selection is strongly tied to parallel code generation. As shown above, the packets that initially appear to take the most number of cycles in execution time may actually take the least when combined in parallel with other packets.

This thesis proposes using a computer’s ability to test many instruction packet scenarios to generate code that is as good as or better than hand-coded assembly. The central idea of a design compiler is a large search for the best solution. Its process is similar to chip architecture design: iterate until every solution is tried or you run out of time. This contrasts directly with traditional compiler designs, where there are little or no mechanisms to iterate through different solutions.

Figure 1. Steps of Traditional C compilers
3 Low Level Design/Implementation

3.1 Target Architecture – The QDSP

The QDSP is a 16-bit data, fixed point digital signal processor made by QUALCOMM Incorporated. It is designed for use in digital communication systems. Up to 5 instruction packets can be executed in parallel in a single cycle. The QDSP has variable length instruction words, with widths of 16 bits, 32 bits, and 48 bits. It also features single-cycle instruction execution. The QDSP uses C-like algebraic assembly; so its code is easy to read. Commas delimit instruction packets executed in parallel, while semicolons separate instruction words. The following examples are valid instruction words in QDSP assembly:

- \( \text{LO}=\text{LO}-\text{L1}; \)
- \( \text{LO}=\text{L1}+\text{R0}^\ast\text{R3(SS)}, \quad \text{L2}=\text{RND(LO}<<\text{SR)}, \quad \text{*C0}--=\text{L0}, \quad \text{R0}=\ast\text{A0}++, \quad \text{R1}=\ast\text{B1}++; \)

The amount of work done during an instruction cycle varies greatly with instruction parallelization possibilities.

For this thesis, a complete C compiler for the QDSP was not made. The code written is used solely as proof of concept of the ideas presented. It only deals with the design and analysis of the compiler below the code generation level. The proof of concept code is written in C++. The following is a discussion of the implementation details of the chosen design.

Figure 2 shows an overview of the steps of the compilation process that are discussed below. Table 2 lists the major data structures used.
Figure 2. Steps of Implemented Design Compiler

<table>
<thead>
<tr>
<th>Intermediate Level Representation Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node</td>
<td>High level DSP operations</td>
</tr>
<tr>
<td>Block</td>
<td>DSP assembly instructions without resource allocation</td>
</tr>
<tr>
<td>Code</td>
<td>DSP assembly instructions with resource allocation</td>
</tr>
</tbody>
</table>

Table 2. Internal Memory Structure Descriptions

3.2 C Code

The most commonly used high level language for DSP programming is C. The C to QDSP compiler should support a superset of the standard C language. DSP-specific extensions are added to support hardware functions whenever possible. Like other platform dependent extensions, these can exist as additional library function calls, macros, or typedefs.
Common DSP functions should be exposed to the programmer for use whenever possible. Some operations that cannot be performed in the standard C language, like OUTPORT, must be exposed in the C environment. Even with these extensions, this code could still be relatively portable across different DSP platforms. If the target architecture does not directly support the extended functions, they can be emulated. Other DSP compilers call these extensions intrinsics. Unlike true inline assembly, intrinsic functions give the compiler the freedom to do useful optimizations with the surrounding code. Later, the low-level representation of these intrinsics will be discussed.

Certain hints or pragma statements may be defined to allow the programmer to direct the compilation process. Examples of these hints include minimum trip counts of loops and most likely taken path of conditional statements. Some extensions of this type already exist in commercial DSP C compilers, like the ones available for Texas Instruments (TI) DSPs. An instruction selection based compiler, like this one, could take full advantage of these hints and produce possibly better code. Further enhancements are discussed in the future work chapter.
Example
Throughout this chapter, an example application will be used to illustrate the
compilation process. A commonly used DSP function is a finite impulse response (FIR)
circular filter. Here is the C code for a 32 tap FIR circular filter with 100 input samples:

```c
void fir(shortCirc *x, shortCirc *h, long *y) {
    short i, j;
    for(j = 0; j < 100; j++) {
        long sum = 0;
        for(i = 0; i < 32; i++) {
            sum += *x++ * *h++;
        }
        *y++ = sum;
        x--;
    }
}
```

Figure 3. FIR Circular Filter C Code

For the QDSP, short integers are defined as 16 bits and long integers as 40 bits.
One thing to notice in the example is that variables x and h are declared as shortCirc,
pointers to circular buffers. In other words, x and h are modulo address registers, a DSP
hardware feature that allows wrap-around addressing of data buffers. This is what
differentiates a circular filter from a normal filter.

3.3 C Code to Node ILR
There is a one-to-one mapping between equivalent C code and Node level ILR
objects. It is the responsibility of the compiler’s front end to convert C code into the
Node intermediate level representation (ILR). The assumption is that traditional
optimizations, like dead-code elimination, constant propagation, and invariant code
motion, are already done before this conversion. These optimizations have been studied thoroughly and many implementations are available.

Also, array references in loops should be converted to incremented pointer form whenever possible. This allows for induction variable removal and more efficient use of the DSP’s address generation unit (AGU) functions. For example, the C code in the form:

```c
for(i = 0; i < 64; i++) {
    a[i] = b[i];
}
```

That should be, in Node ILR form:

```c
LOOP(64) {
    *a++ = *b++;
}
```

The mechanics and implementation of the front end are not exceedingly difficult and are not discussed.

### 3.4 Node ILR

The Node ILR is where the work of this thesis begins. The Node level exists only as in-memory data structures. Table 4 lists some selected Node data structures:
### Table 4. Selected Node Types and Descriptions

<table>
<thead>
<tr>
<th>Node Type</th>
<th>Equivalent C Code or Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assign32CNode(&lt;x::Int32&gt;, &lt;c::constant&gt;)</td>
<td>x = c;</td>
</tr>
<tr>
<td>IndLoadAndInc(&lt;x::Int&gt;, <a href="">ptr::IntPtr</a>)</td>
<td>x = *ptr++;</td>
</tr>
<tr>
<td>IndStoreAndInc(<a href="">ptr::IntPtr</a>, &lt;x::Int&gt;)</td>
<td>*ptr++ = x;</td>
</tr>
<tr>
<td>MAC(&lt;v::Int32&gt;, &lt;x::Int32&gt;, &lt;y::Int16&gt;, &lt;z::Int16&gt;)</td>
<td>v = x + y * z;</td>
</tr>
<tr>
<td>MultiNode(&lt;Node&gt;, &lt;Node&gt;, ...)</td>
<td>Glue for Nodes – makes multiple nodes act as a single node</td>
</tr>
<tr>
<td>LOOPNode(&lt;c::constant&gt;, &lt;n::Node&gt;)</td>
<td>Repeat Node n c number of times</td>
</tr>
<tr>
<td>RTFNode</td>
<td>Return from function</td>
</tr>
</tbody>
</table>

For the purposes of this thesis, the Node level ILR had to be hand-generated.

This is the C++ code used to generate the Node ILR for the filter example.

```c++
NodePtr n1 = new Assign32CNode(sum, 0);
NodePtr n2 = new IndLoadAndInc(temp00, x);
NodePtr n3 = new IndLoadAndInc(temp01, h);
NodePtr n4 = new MAC(sum, sum, temp00, temp01);
NodePtr mn = new MultiNode(*n2, *n3, *n4);
NodePtr fcci = new LOOPNode(32, *mn);
NodePtr n5 = new IndStoreAndInc(y, sum);
NodePtr n6 = new IndLoadAndDec(temp02, x);
NodePtr mn2 = new MultiNode(*n1, *fcci, *n5, *n6);
NodePtr fcci2 = new LOOPNode(100, *mn2);
NodePtr fEnd = new RTFNode;
NodePtr func = new MultiNode(*fcci2, *fEnd);
```

**Figure 4. C++ Code Used to Create FIR Circular Filter Node Level Representation**
Any C function can similarly be defined as a tree of Node objects. Node objects are immutable.

Example

In the Node and Block levels, variable names are bound to their types -- shown in the form [name::type]. Types include 16-bit integers (I16), 16-bit integer pointers (I16Ptr), 40-bit integers (I40), 40-bit integer pointers (I40Ptr), and others. The Nodes of the FIR procedure are shown below, in a readable C-like format.

```
LOOP (100) {
    [sum::I40] = 0;
    LOOP (32) {
        [temp00::I16] = *[x::I16PtrCirc]++;
        [temp01::I16] = *[h::I16PtrCirc]++;
        [sum::I40] = [sum::I40] + [temp00::I16] * [temp01::I16];
    }
    *[y::I40Ptr]++ = [sum::I40];
    *[x::I16PtrCirc]--;
}
RETURN
```

Figure 5. FIR Circular Filter Node Level Representation

3.5 Node ILR to Block ILR

The Node ILR to Block ILR conversion is where most of the work is accomplished. Instruction and optimization selections are performed at this time. There is a one-to-many mapping between the Node level and the Block level. The main search is finding the best Node ILR to Block ILR mapping.

3.5.1 Instruction Selection

Instruction selection is accomplished in the explicit mapping of a Node ILR to Block ILR equivalents.
One example for the QDSP is the LOOP construct. The form is `LOOPNode(C, bodyNode)` where `C` is a constant number representing the number of times the Node `bodyNode` is to be repeated. This directly corresponds to the following C code:

```c
for(i = 0; i < C; i++) {
    <body>
}
```

Special attention has to be made so that no code, except the for statement, depends on the variable `i`. Obviously, other constructs must exist to take care of the remaining for loop possibilities.

For this Node, two obvious lower level representations are available in QDSP assembly. The first possible Block translation is to use the QDSP’s zero overhead loop instruction, `LOOP UNTIL`. One restriction is the constant `C`, the number of loop iterations, must be less than 512. Also, there is a maximum number of nested loop instructions.

So, an alternative solution must exist for those times when the `LOOP UNTIL` instruction cannot be used. The second possibility is the standard way to code a loop: using a loop variable and doing a branch test.

A second Node example is the `if` statement. The Node level construct is `IF(test, conseqNode)` where `test` is a variable and `conseqNode` is the Node to execute if the test is non-zero. The QDSP supports conditional execution of instructions, allowing efficient implementation by conditionally executing the entire body of short if statements. However, not all instructions can be conditionally executed. Also, conditional instructions might cut down the parallel instruction execution opportunities. It may not be advantageous to use conditional execution at all.
In those cases, the traditional test and branch code could be generated. Even for this simple case, there are at least two Block level representations. Which representation is best depends on whether the if body is usually executed or not. Without hints, only the cost function can tell us which one is the better implementation.

In general, instruction selection allows the compiler to use specialized DSP instructions whenever possible. Also, it supports the central idea that the cost function is the ultimate determinant of what code is best.

3.5.2 Optimizations

Ideally, the Node level ILR should only do instruction selection, with another level doing optimizations. However, optimization selections were also done at the Node level because of time constraints.

The greatest speed increase comes from loop optimizations. Several VLIW loop optimization possibilities were implemented, including loop unrolling and loop rotation. The optimizations were exposed in the same way as instruction selection – in the Node to Block level mapping.

3.5.3 ISA Dependent/Architecture Implementation Independent

Instruction selection is instruction set architecture (ISA) dependent. The compiler designer will review the target ISA and create Node constructs that correspond to high level operations the DSP can do.

However, operations at the Node and Block level are DSP architecture implementation independent. In fact, there is no representation of the parallelism of the machine or how many hardware registers it has. Resource allocation and instruction scheduling are reserved for later stages in the compilation process. It is even possible for
the Node level designer to totally ignore issues like instruction cycle count and instruction size. Although, the compiler will run faster if useless representations – ones that give no advantage over the others in all cases – are removed.

3.5.4 Implementation

The most important Node object member function is createIter(). CreateIter is a virtual constructor for type BlockInfoIter. The BlockInfoIter objects expose the iterative selection process to the compiler. It is an iterator object that returns Block ILRs.

<table>
<thead>
<tr>
<th>BlockInfoIter Member Functions</th>
<th>Function Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>first()</td>
<td>Reset to the beginning of iterator</td>
</tr>
<tr>
<td>next()</td>
<td>Move to the next BlockInfo object</td>
</tr>
<tr>
<td>isDone()</td>
<td>True if the end of the iterator is reached</td>
</tr>
<tr>
<td>currentItem()</td>
<td>Return the current BlockInfo object</td>
</tr>
<tr>
<td>clone()</td>
<td>Virtual copy function</td>
</tr>
</tbody>
</table>

Table 5. BlockInfoIter Member Functions and Descriptions

Special care must be taken when writing the BlockInfoIter procedures to insure there are no memory leaks. As these iterators can generate hundreds or thousands of selection possibilities, even small memory leaks can adversely affect compile time performance. To see how what role the iterators play, this is a simplified version of the core loop of the compiler:
/// Main brute-force search of a given Node
for(MultiBlockInfoIter* ni = input->createIter();
    !ni->isDone();
    ni->next()) {

    // Parallelize every Block level possibility
    parallelize(ni->currentItem(l.0), ci);

    // Determine the cost of the parallelized assembly
    bCost = min(cFunc(ci), bCost);
}

Figure 6. Simplified Core Loop of Implemented Design Compiler

The compiler performs a brute-force search through every Block level code possibility. Each one is first parallelized and then sent to the cost function. The best solution is saved.

3.6 Block ILR

Blocks have a one-to-one mapping with QDSP instruction packets. Blocks are in linear, non-parallel form. However, Block ILRs do not have register assignments or memory bank allocations. They are still tagged with variable types. Equivalently, Blocks assume an infinite sized register file and an infinite number of memory ports.

There is a direct translation of intrinsic instructions to Block ILRs. Actually, all Block ILRs can be exposed as intrinsic instructions. The advantage of using intrinsics is that Block ILRs can be optimally parallelized with surrounding instructions, unlike inline assembly.

Example

Figure 7 shows one possible Block level translation of the FIR filter. It takes up the least amount of code space, but also does not have many code parallelization possibilities.
LC = 100
LOOP UNTIL _lab2
[sum::I32] = 0
LC = 32
LOOP UNTIL _lab1
[temp00::I16] = *[x::I16PtrCirc]++
[temp01::I16] = *[h::I16PtrCirc]++
[sum::I40] = [sum::I40] + [temp00::I16] * [temp01::I16] (SS)
_lab1:
[y::I40Ptr]++ = [sum::I40]
[temp02::I16] = *[x::I16PtrCirc]--
_lab2:
RTF

Figure 7. FIR Circular Filter Block Level Representation

The Block ILR is sent directly to the DSP parallelization and compaction process.

3.7 DSP Assembly Parallelization and Compaction

3.7.1 Purpose
This design uses assembly parallelization and compaction as a step in the compilation process. This goal is to rearrange straight-line DSP assembly into maximally parallel code, while preserving instruction selection and data dependencies.

Commercial DSPs, like TI’s TMS320C6x, have assembly parallelizing optimizers that perform simple code compression. Normally, these optimizers are used as an aid for programmers that hand-code DSP assembly. Programmers can simply write in straight-line assembly without regard to the DSP’s parallel combination capabilities.

Yet, this thesis does not use the compaction optimizer as the final step in the code generation process. Instead, it is used as an intermediate step to help determine if an instruction selection was a good one. This allows the compiler to treat the problems of instruction packet selection and instruction packet parallelization separately.

These optimizers do not always generate optimally parallel code.
3.7.2 Implementation

This thesis does not directly deal with the problem of implementing instruction scheduling and parallelization. Tsvetomir Petrov wrote an excellent general purpose code compression and parallelization tool for his M.Eng. thesis. For this portion of the compiler, an early version of his thesis code was used and modified to support QDSP Block ILRs. Several further modifications were made to allow it to function in this multi-pass design compiler.

Example

This is the above Block ILR code after the parallelization and compaction process.

```
LC = 100, LOOP UNTIL _lab2
 [sum::I32] = 0, LC = 32
LOOP UNTIL _lab1
 [temp00::I16] = *[x::I16PtrCirc]++, [temp01::I16] = 
 *[h::I16PtrCirc]++
 [sum::I40] = [sum::I40] + [temp00::I16] * [temp01::I16] (SS)
 _lab1:
 *[y::I40Ptr]++ = [sum::I40], [temp02::I16] = *[x::I16PtrCirc]--
 _lab2:
 RTF
```

Figure 8. FIR Circular Filter Representation after Parallelization Step

3.8 Parallelized Block Level to Assembly Code Level

3.8.1 Purpose

In this step in the compilation process, resource assignment is done. This may require generating register spill code or breaking up parallel instructions because of memory bank conflicts.

3.8.2 Separation of Parallelization from Resource Allocation

Resource assignment is intentionally made a later step in the compilation process. Doing traditional register allocation algorithms before parallelization is incorrect. The register assignments would generate false dependencies, decreasing parallelism. Instead,
it is advantageous to assume an infinite sized register file machine and an infinite number of memory ports for parallelization purposes. After parallelization, analysis can be done to determine which are the best references to spill into memory. Of course, after the spill code is generated, another iteration of parallelization might be worthwhile.

3.8.3 Implementation

In this case, a simplifying assumption of dual-ported RAMs for the QDSP was made. In general, loop unrolling and other optimizations may affect the total number of utilized registers. However, all the optimized examples discussed in the thesis fit into the available number of QDSP registers, so no register allocation algorithm was made. Instead, trivial register assignment was done.

Example

Figure 9 shows one of the final assembly code possibilities for the filter example.

\[
\begin{align*}
LC &= 100, \quad LOOP \ UNTIL \ _lab2 \\
L0 &= 0, \quad LC = 32 \\
\text{LOOP UNTIL } _lab1 \\
R0 &= *A0++, \quad R1 = *A1++ \\
L0 &= L0 + R0 * R1 (SS) \\
\_lab1: \\
\quad *C0++ &= L0, \quad R2 = *A0-- \\
\_lab2: \\
\quad RTF
\end{align*}
\]

Figure 9. FIR Circular Filter Assembly Code

The final assembly is sent to the cost function to determine how good a solution it is.

3.9 Cost Function

3.9.1 Implementation

Cost functions can be implemented in many different ways. There is no specific form or structure to a cost function. It can be the timing results of an external program
like a profiler or chip emulator. If compilation time is no object, system simulation parameters can be run for every code generation possibility. In an actual compiler, the cost function may be written as a TCL or PERL script. The scripts could have predefined interfaces to code analysis tools or functions.

In this case, the cost function is simply a C++ function. For this simple implementation, only two cost analysis parameters are exposed to the programmer through the CostInfo class. These parameters are equivalent to the size and speed of the code.

<table>
<thead>
<tr>
<th>CostInfo Member Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>numInstBits()</td>
<td>Returns the number of bits the final assembly code would take in IRAM. (Code Size)</td>
</tr>
<tr>
<td>numCycles()</td>
<td>Returns the exact number of cycles the assembly code would take to execute, if possible. If it is not possible to determine (because of branches, etc.), it returns an estimate. (Code Speed)</td>
</tr>
</tbody>
</table>

Table 6. CostInfo Member Functions and Descriptions

With these two parameters, it is possible to make a wide variety of useful cost functions. Some cost function examples are given in the performance chapter.
Example

The CostInfo function tags each instruction word with the analysis data in the form \([\text{number of cycles}, \text{number of instruction bits}]\). For example, line 3 of the following example is executed 100 times and takes up 32 bits of IRAM.

\[
\begin{align*}
[1, 48]: & \quad \text{LC} = 100, \text{LOOP UNTIL } _{\text{lab}2} \\
[100, 48]: & \quad \text{L0} = 0, \text{LC} = 32 \\
[100, 32]: & \quad \text{LOOP UNTIL } _{\text{lab}1} \\
[3200, 32]: & \quad \text{R0} = *\text{A0}++ , \text{R1} = *\text{A1}++ \\
[3200, 16]: & \quad \text{L0} = \text{L0} + \text{R0} * \text{R1 (SS)} \\
[0, 0]: & \quad _{\text{lab}1}: \\
[100, 32]: & \quad *\text{C0}++ = \text{L0}, \text{R2} = *\text{A0}-- \\
[0, 0]: & \quad _{\text{lab}2}: \\
[1, 16]: & \quad \text{RTF}
\end{align*}
\]

Figure 10. FIR Circular Filter CostInfo Analysis

So, the above FIR filter assembly takes 224 bits of IRAM and 6702 QDSP cycles to execute the function.

Finally, the process is repeated for every other instruction or optimization selection. The minimum cost selection is saved and output as the final assembly.

4 Performance and Feasibility

4.1 Setup

With the framework in place, several tests were performed to determine the design’s performance and feasibility as a commercial compiler. For each piece of test code, four different cost functions were used. Table 7 gives a description of each of the cost functions used, and Figure 11 shows the C++ code.
Cost Function | Description
---|---
Speed | Select for fastest code, irrelevant of code size.
Size | Select for smallest code, irrelevant of code speed.
Mix | Take both code space and code size into account
Non-Linear | Select for fastest code with a code size of a set number of bits or less.

Table 7. Example Cost Functions and Descriptions

```c
costType cFuncSpeed(CostInfo& ci) {
    return ci.numCycles();
}

costType cFuncSize(CostInfo& ci) {
    return ci.numInstBits();
}

costType cFuncMix(CostInfo& ci) {
    return ci.numCycles() + ci.numInstBits();
}

costType cFuncNonLinear(CostInfo& ci) {
    if(ci.numInstBits() > MAX_COST_BITS) {
        return INFINITE_COST;
    } else {
        return ci.numCycles();
    }
}
```

Figure 11. C++ Code for Test Cost Functions

The "Speed" and "Size" cost functions are obvious. Notice that when compiling for speed, the design compiler will unroll huge loops just to save a cycle. Similarly, when compiling for size, it will choose an extremely slow implementation to save a byte of IRAM.

"Mix" is a more balanced cost function. It will perform cycle saving optimizations if it does not drastically increase code size. The exact tradeoff is a cycle of
execution time for one bit of IRAM. This results in very compact code, while allowing the very beneficial optimizations, like some loop optimizations, to be performed.

The “Non-linear” cost function is the most interesting. Basically, it assigns an infinite cost to code that use more than a given number of bits of IRAM. Otherwise, the cost is simply the number of cycles it takes to execute. The chosen number of bits is set for each function and is listed in the result tables. This directly corresponds to real world IRAM space constraints. Obviously, real world time constraint cost functions could be similarly written.

To prevent extreme code size explosion, the compiler was set so up to nine levels of loop rotations were performed for the test cases. If this option was not set, the loops would be fully unrolled, and every possible permutation would be tested.

Two commonly used DSP functions were used as example functions. The first is the FIR Circular filter already presented. The second function is a vector sum.

```c
long vector_sum(short *x) {
    long sum = 0;

    for(j = 0; j < 64; j++) {
        sum += *x++;
    }

    return sum;
}
```

*Figure 12. Vector Sum C Code*
The third example is a contrived function of highly parallel loop operations.

```c
long unroll(short *x, long *y) {
  for(j = 0; j < 6; j++) {
    *x++ = 0;
  }
  for(j = 0; j < 5; j++) {
    *y++ = 0;
  }
}
```

**Figure 13. Loop Unroll C Code**

### 4.2 Optimization Optimality

The following tables list the results of the compiler’s output. The cost value of the final assembly was used as the grade of optimality. This is fair comparison, as the speed and size cost functions directly correspond with existing compiler switches.

<table>
<thead>
<tr>
<th>Cost Function Based Design Compiler</th>
<th>Hand-Optimized Assembly (Ideal)</th>
<th>Traditional Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cost Function</strong></td>
<td><strong>Cost</strong></td>
<td><strong>Cycle Count</strong></td>
</tr>
<tr>
<td>Mix</td>
<td>3726</td>
<td>3502</td>
</tr>
<tr>
<td>Speed</td>
<td>3403</td>
<td>3403</td>
</tr>
<tr>
<td>Size</td>
<td>224</td>
<td>6702</td>
</tr>
<tr>
<td>Non-linear (Code Size $\leq$ 256)</td>
<td>3502</td>
<td>3502</td>
</tr>
</tbody>
</table>

**Table 8. FIR Circular Filter Results**
<table>
<thead>
<tr>
<th></th>
<th><strong>Cost Function Based</strong></th>
<th><strong>Hand-Optimized Assembly (Ideal)</strong></th>
<th><strong>Traditional Compiler</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cost</td>
<td>Cycle Size (bits)</td>
<td>Cost</td>
</tr>
<tr>
<td><strong>Cost Function</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mix</td>
<td>226</td>
<td>66</td>
<td>226</td>
</tr>
<tr>
<td>Speed</td>
<td>66</td>
<td>66</td>
<td>66</td>
</tr>
<tr>
<td>Size</td>
<td>144</td>
<td>131</td>
<td>144</td>
</tr>
<tr>
<td>Non-linear (Code Size &lt;= 256)</td>
<td>66</td>
<td>66</td>
<td>66</td>
</tr>
</tbody>
</table>

Table 9. Vector Sum Function Results

<table>
<thead>
<tr>
<th></th>
<th><strong>Cost Function Based</strong></th>
<th><strong>Hand-Optimized Assembly (Ideal)</strong></th>
<th><strong>Traditional Compiler</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cost</td>
<td>Cycle Size (bits)</td>
<td>Cost</td>
</tr>
<tr>
<td><strong>Cost Function</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mix</td>
<td>223</td>
<td>15</td>
<td>223</td>
</tr>
<tr>
<td>Speed</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Size</td>
<td>208</td>
<td>16</td>
<td>208</td>
</tr>
<tr>
<td>Non-linear (Code Size &lt;= 250)</td>
<td>13</td>
<td>13</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 10. Loop Unroll Function Results
The first set of columns in the tables corresponds to the statistics for the solutions provided by the proposed design compiler. The second set of columns corresponds to a hand-coded solution to the given cost function. The final set of columns corresponds to a traditional compiler that does not even consider the cost function as part of the compilation process. This is the result of simply choosing and parallelizing the most obvious assembly. This also happens to be the smallest code in most instances.

The sub-columns show the cost, cycle count, and code size of the assembly outputs. The cycle count is exactly the number of QDSP cycles it takes to execute each function. The code size is the number of bits of IRAM that the function would use.

Interestingly, I was not able to come up with the optimal speed cost function solution for the FIR Circular Filter. The optimal solution is shown below. Instead, I was able to verify that the compiler’s solution was indeed faster than mine was by one cycle. As the code gets larger, it becomes even more difficult for a human to compete with the compiler.

```c
[1, 48]: L0 = 0, R0 = *A0++, R1 = *A1++, LC = 99
[1, 32]: LOOP UNTIL _lab65
[99, 48]: LC = 31, LOOP UNTIL _lab64
[3069, 32]: L0 = L0 + R0 * R1 (SS), R0 = *A0++, R1 = *A1++
[0, 0]: _lab64:
[99, 32]: L0 = L0 + R0 * R1 (SS), R2 = *A0--, R1 = *A1++
[99, 48]: *CO++ = L0, L0 = 0, R0 = *A0++
[0, 0]: _lab65:
[1, 48]: LC = 31, LOOP UNTIL _lab63
[31, 32]: L0 = L0 + R0 * R1 (SS), R0 = *A0++, R1 = *A1++
[0, 0]: _lab63:
[1, 32]: L0 = L0 + R0 * R1 (SS), R2 = *A0--
[1, 16]: *CO++ = L0
[1, 16]: RTF
```

Figure 14. FIR Circular Filter Optimal Speed Cost Solution
For the non-linear cost function, the compiler also did better than me. The main problem with a human assembly programmer is that he or she has no sense of the size of the program in bits until it is compiled. So, finding the optimal solution under a given size is extremely tedious and difficult for a human.

The design compiler found the optimal solution easily. Finding an optimal solution for different maximum IRAM size parameters is as easy as changing a number in the cost function. Humans would more likely hack the original code to fit within the space, probably resulting in non-optimal code.

For these functions, the compiler was able to generate optimal code.

### 4.3 Feasibility

The measurements were performed on a Pentium 166 MMX running under Windows NT.

<table>
<thead>
<tr>
<th>Function</th>
<th>Number of Iterations</th>
<th>Average Time per Iteration</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIR Circular Filter</td>
<td>81</td>
<td>2.75 seconds</td>
<td>223 seconds</td>
</tr>
<tr>
<td>Vector Sum</td>
<td>9</td>
<td>0.88 seconds</td>
<td>8 seconds</td>
</tr>
<tr>
<td>Loop Unroll</td>
<td>120</td>
<td>0.43 seconds</td>
<td>51 seconds</td>
</tr>
<tr>
<td>Combined Vector Sum and FIR Circular Filter</td>
<td>729</td>
<td>5.51 seconds</td>
<td>4023 seconds</td>
</tr>
</tbody>
</table>

Table 11. Compile Time for Example Functions
4.3.1 Small Functions

The compiler generates excellent code. Yet, if the compiler takes weeks or years to compile everything, then it would be a useless tool. As previously mentioned, this design compiler is significantly slower than existing compilers for optimizations. For small functions, the compile time is absolutely reasonable. Actually, the compiler generates optimal code in much less time than a human for these examples.

4.3.2 Large Functions

The problem with this design compiler framework is that compile time grows exponentially with the number of selection possibilities. A large function was created by combining the FIR Circular Filter and Vector Sum code into a single function. This is not an extremely large function, but it already takes over an hour to compile.

This may not be acceptable for commercial use. If an optimal solution is required, then an exhaustive iterative search is the only solution. It will take a long time. Block parallelization is the most time consuming step. Efforts should be focused on making that step run faster.

Other search possibilities were investigated and discussed below.

4.4 Compile Time Tradeoffs

4.4.1 Programmer Directed Optimizations

What if an optimal solution is not required? The programmer can easily direct the compiler to perform less instruction and optimization selection. If the programmer has an intuition of what instruction or optimization possibilities will never used, then the compile time will decrease markedly.
A graphical user interface could allow skilled programmers to interactively direct the compiler to perform specific optimizations. The tedious unrolling of loops or loop rotations could be done instantaneously, without human error. This could be an excellent next generation compilation tool for DSP assembly programmers. Instant cost function analysis of the code selection or optimization could be shown onscreen. Using this tool, programmers scroll through the possible optimizations easily and quickly.

4.4.2 Non-exhaustive Iterative Search

Good solutions can be found without an exhaustive search. The following graphs show best cost versus iteration number. In most cases, optimal solutions were found very early in the search process. The search could be terminated early in the process with little degradation of optimality.

![Figure 15. Best Cost (Mix) versus Iteration Number for Combined Function](image)
4.4.3 Non-iterative Search Techniques

Another possibility is to use non-iterative search techniques. By analyzing the structure of the following graphs, several possibilities look promising. One is to perform a randomized, non-iterative search. Another is to use an artificial intelligence search heuristic like hill-climbing. Both would significantly decrease compile times. From the data collected, they also appear to have a good probability of finding an excellent solution.
4.4.4 Separating Dissimilar Code Segments

Finally, dissimilar code segments can be separately optimized with only a little loss. This is perhaps the most useful discovery. Unless extreme optimization is required,
it is generally good to split code into relatively non-interacting segments. Then, independently optimize each segment and merge the segments at the end.

<table>
<thead>
<tr>
<th>Cost Function</th>
<th>Cost of Ideal Code</th>
<th>Cost of Merged Code</th>
<th>Percentage Cost Loss</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mix</td>
<td>3935</td>
<td>3936</td>
<td>0.02%</td>
</tr>
<tr>
<td>Speed</td>
<td>3467</td>
<td>3468</td>
<td>0.03%</td>
</tr>
<tr>
<td>Size</td>
<td>352</td>
<td>352</td>
<td>0%</td>
</tr>
<tr>
<td>Non-linear</td>
<td>3530</td>
<td>3567</td>
<td>1%</td>
</tr>
</tbody>
</table>

Table 12. Combined FIR Circular Filter and Vector Sum Function Results

As shown above, this technique works extremely well for linear cost functions. However, it does not immediately work for non-linear cost functions. To get the merged cost, each of the separate functions were optimized for a non-linear cost function with size <= 256 bits. Then the merged function is guaranteed to be <= 512 bits. However, this will not work in all circumstances and creates code that is not very well optimized.

The advantage of this is a speed-up of compile time. The comparison is dramatic. The time to find the separate best-cost solutions is 231 seconds versus 4,023 seconds to find the optimal solution. Notice that the compile time is constant over all the given cost functions. This is because the instruction selection search space is always the same and does not depend on the selected cost function.
5 Conclusion

A cost function based compiler is simply a generalized version of the compile-test-redesign phases of traditional code design. The described system has been designed and built. By writing some proof of concept code, the system was tested for performance and feasibility. These are some of the observations:

1. The brute-force iterative method for finding the optimal solution was implemented.
   The instruction selection problem is NP complete, so this probably is the only way to find the optimal solution. Yet, given the nature of the search, other heuristics may find good solutions in much less time.

2. For small functions and sections of code, it produces optimal solutions (according to the cost function) at faster times than a human.

3. For large functions, it also produces optimal solutions. However, the time needed to arrive at the optimal solution may be prohibitively long.

4. Breaking up relatively independent sections of code will result in much faster solutions with only a small degradation of optimality.

5. The bottleneck of the compilation process is the parallelization step. If solution memoization or other techniques were employed, the compile process could be sped up tremendously

6. With some modifications, this design compiler method appears to be feasible for commercial use. Many heuristics appear to be available to shorten the compile time
significantly. It would be especially useful for experienced DSP assembly programmers.

Ideally, DSP code compilation should be a highly interactive process. A programmer’s experience could be used to speed up the process dramatically. Possible optimizations could be proposed by the compiler and tested against the current solution in the background. Optimization transformations could be done on the fly with the click of a mouse button. This would allow the programmer to focus on the important global aspects and reduce the number of errors.
6 Future work

The proof of concept code is obviously deficient. Many forms of compile time savings could be implemented, like multi-level cost functions or using branch-and-bound techniques to eliminate code possibilities earlier in the process. More programmer interaction should be allowed through hints or pragma statements to produce better code in a shorter amount of time. There is support for this in the proof of concept code, but the advantages of it were not investigated.

This work can also be extended in many ways. A cost-function based compiler is perfectly suited to be directly interfaced with many areas of cutting edge research. Examples include trace scheduling, profile driven optimizations, and power usage optimizations.

By exposing hints in the Node layer, trace information could be used to direct the search to the most likely optimal solution. Code could be optimized to make an extremely fast common case execution. A cost function model could be built that used event trace information to determine the best implementation.

Profile-driven optimizations are already being researched heavily. For consumer products, large amounts of system simulation test data are available for use. By tying the profiler or simulator’s results directly in with the compile process, ideal code can be generated for many applications without human intervention.
Low energy consuming code is a big requirement in modern day embedded systems. If good tools can be developed to simulate or measure power consumption, then this would be the ideal framework for a low power coding platform.
References

[1] 1,600-Mips DSP shifts design focus toward software

[2] A VLIW Architecture for a Trace Scheduling Compiler


[5] Code Scheduling for VLIW/Superscalar Processors with Limited Register Files


[7] Evaluation of the Performance of the TMS320C6201 and Compiler V1.1

[8] Motorola recharges the VLIW DSP debate

[9] Programming Tips for the TMS320C6x Family
