DSP Compilers: Challenges for efficient DSP code generation

Document ID: DSP_COMP_WP01

DACS Software Pvt. Ltd.

Version: 1.0

Date: March 2001

Copyright © 2001 by DACS Software Pvt. Ltd., Bangalore.

All rights reserved. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without prior written permission of DACS Software Pvt. Ltd.
Table of Contents:

1. Abstract ...................................................................................................................... 3
2. Traditional compiler architecture ............................................................................. 3
   2.1 Why is C the most popular choice ................................................................. 3
   2.2 The Intermediate Representation .................................................................. 4
   2.3 Target Independent Optimizations ............................................................... 4
   2.4 Target Dependent Optimizations ................................................................... 5
3. Issues faced by the C compilers ................................................................................ 5
   3.1 What is so different with the DSPs ................................................................... 5
   3.2 How compilers are coping with these DSP needs .......................................... 7
      3.2.1 C Coding Guidelines .............................................................................. 7
      3.2.2 Providing Direct Target Access To The User ......................................... 7
      3.2.3 Language Extensions ............................................................................. 8
      3.2.4 New Optimizations ............................................................................... 8
4. Making the C code compiler friendly ....................................................................... 9
   4.1 Making alias analysis easier ........................................................................... 9
   4.2 Usage of Data Types ..................................................................................... 10
   4.3 Rounding and Saturation operation ................................................................ 11
   4.4 Some general guidelines .............................................................................. 11
5. Typical DSP application development using compilers ........................................ 11
6. Conclusion ................................................................................................................. 12
7. References ................................................................................................................. 13

Table of Figures:

Figure 1. Classical compiler architecture ................................................................. 3
Figure 2. Graph for A = A+B*C ............................................................................ 6
1 Abstract

With the increasing complexity and size of the DSP applications and the DSP architectures getting more and more complex, it is highly desirable that the DSP algorithms are implemented in High Level Languages (HLLs). However, with code efficiency/performance continuing to be the major requirement for DSP based systems, the compiler technology is increasingly falling short of the expectations of the DSP programmer community. Repeated attempts in improving the compilers to match the user requirements have failed, resulting in the usage of compilers getting restricted to the performance non-critical modules, with the performance critical modules still getting coded manually. Here is a study of the challenges faced by the compiler technology due to the peculiarities of DSP architectures and the limitations of HLLs that result in these inefficiencies. the strategies provided by the state of the art compilers and adopted by the programmers to overcome these inefficiencies are studied briefly. Factors that are driving the architectures to get compiler-friendly are also touched upon.

2 Traditional compiler architecture

The compilers are software tools that accept as an input, programs written in a high level language, most popular choice for DSPs being C, and generate the assembly language program for the target processor. The classical compiler architecture is described in figure 1. The input program undergoes various processing phases of syntax and semantic analysis that check the correctness of the input as per the language specification, and generate an intermediate representation (IR), which is an abstract notation of the input program in terms of various low level operators. This is followed by target independent optimizations phase, a code generation phase that emits code in the target assembly, a target dependent optimization phase that is distributed before and after the code generation phase, and a post-assembly optimization phase.

![Figure 1. Classical compiler architecture](image)

2.1 Why is C the most popular choice

Though a variety of HLLs exist, which could be used for DSP coding, C has emerged as the most popular choice over the years. Some of the reasons are:
• A lot of standard committees like ITU, IEC choose to provide the reference implementation in C language. Hence it is very tempting to take this code and directly compile it for the given target.

• Verification of the code independent of the target.

• Ease of development in C

2.2 The Intermediate Representation

This is an intermediate representation between the high level and assembly level. It essentially captures the functionality of the HLL program in a small set of low level operators that have a potential of easy transformation into assembly. The design of an intermediate representation is highly dependent on the type of optimizations one would like to perform. Since the DSP architectures have very special characteristics, many of the important optimizations are target dependent.

2.3 Target Independent Optimizations

Though the traditional compiler theory lists a huge list of target independent optimizations, following are the few ones, which are most popular and are supported by the present day DSP compilers:

• **Branch elimination** branch to a branch instructions are eliminated

• **Common sub-expression elimination** redundant computations of common sub-expressions are removed

• **Constant folding** compile time evaluation of expressions, whose operands are known to be constant

• **Constant propagation** replaces the use of a variable with the constant that was assigned to that variable in a previous assignment.

• **Dead-code elimination** removes code that is either unreachable or has no effects.

• **Tail-recursion call optimization** replace calls with branches, which not only reduce procedure overhead, but also enable loop optimization

• **Strength reduction** replaces expensive operations, such as multiplication and division by less expensive ones like add or shift

• **Loop invariant code motion** recognizes computations in loops that have the same value on every iteration and moves them out of the loop

• **Loop Unrolling** replaces the body of a loop by several copies which reduces the loop overhead and improves effectiveness of other optimizations like CSE, instruction scheduling, software pipelining.

• **Forward store** allocates register to a variable inside the loop to improve performance.

The sequencing of these optimizations is a major issue, and may result in optimizations canceling benefits due to others. This is termed as the phase ordering problem. If
instruction scheduling is done before register allocation, the live ranges of the variables increase, so optimal register allocation is not possible. On the other hand, is register allocation is done before instruction scheduling, false data dependencies may be created, preventing effective scheduling.

Some recommendations do exist that give a preferred sequence of the optimizations [5], but no ordering can guarantee best results in all cases.

2.4 Target Dependent Optimizations
As the name indicates, these optimizations are specific to the target processor. Following are a few of them:

- **Software pipelining** is an optimization that allows parts of several iterations of a loop to be in process at the same time.
- **Instruction scheduling** is where the instructions are reordered to improve performance
- **Peephole optimizations** include machine idioms and instruction combining to replace single instructions or sequences of instructions by single ones that perform the same task faster.
- **Register allocation** is especially tough in DSPs because of peculiarities in register sets, and is very critical to performance

3 Issues faced by the C compilers
The traditional compiler technology is fairly evolved for the RISC kind of processor architectures. Basically, the resource allocation is a lot easier, since the resources are symmetric in nature. Also, an orthogonal instruction set allows the compiler to choose the best instructions for performing the desired operation. The comparison of the DSP architectures and the RISC architectures for the purpose of code generation has been extensively studied [6], and it has been observed that the generated code has better chances of being efficient when the target is a RISC kind of processor.

3.1 What is so different with the DSPs
DSP architectures evolved over the years to support fast processing of the applications characterized by the Signal Processing domain needs. Some of these needs are mentioned below:

- Requirement of peculiar data types (like fractional, complex and so on)
- Mechanisms to access the required data in specific order for processing (DSP specific addressing modes)
- Hardware acceleration modules for domain specific processing needs and the related higher complexity instructions
- Peculiar kinds of register sets with varying usage
- Better (typically zero overhead) looping mechanisms
Apart from these, other mechanisms which allow faster access of data, including the Harvard / modified Harvard architecture, multiple data and address buses were also introduced into the processors for the purpose of processing the signals faster. As an example, Motorola 56x architectures typically allow two parallel data accesses through X and Y memory banks along with the execution of “mac” like instructions, to ensure that the processor is not starved for data while executing such instructions in loops.

Basic compiler paradigm operates from the assumption that the input to the compiler is a higher level language than the target platform’s native or assembly language. By higher level, it is not only understood that it is easier to program, which is indeed true, but also that the operators have more or at least equivalent complexity as compared to the native operators of the target platform. Also, at a higher level of programming, the programmer is less constrained in terms of resources as compared to the lower level (assembly level) programming. So in the compilation process, these operators are expected to map closely to the basic instruction set of the target processor. Consider the following piece of code in C:

\[
A = A + B \times C
\]

A typical compiler would convert it to the abstract syntax tree as shown in figure 2.

![Figure 2. Graph for A = A+B*C](image)

On the other hand, a programmer, who takes a cursory look, would determine such cases as candidates for the MAC instruction, if it is available on the target processor. However, if adequate care is not taken at the front end of the compiler, such a tree would result, and the optimizer would have to expend extra efforts for operator combining, to generate a MAC instruction out of this sequence. Also, it may not always be possible to combine the operators at a later stage in the overall graph.

Of-course, in cases where the resources on the target are symmetric, everything would be extremely easy to map. By symmetric, we mean equal in terms of their capabilities. In such cases, any of the available resources could be picked up for performing the required task. However, if the resources are non-symmetric, then the compiler would have to choose the most appropriate resource for performing the task. In some cases, this would result in a lot of data movement between various resources. Typical accumulator based DSPs have registers and processing units of non-symmetric nature and this results in inefficient code generation. As a typical example, in some processors, the barrel shifter can operate only on an accumulator register, with the shift count only in the other
accumulator, and not on any general-purpose register. This results in a constraints on the register allocation, as well as a lot of data movement (storing / restoring).

The present day DSPs beat the compiler technology in more than one way. Firstly, the signal processing requirements are peculiar in nature, and the DSP architects, for performance reasons, modify the architectures to suit these needs. Secondly, the basic operators in these processors may be of higher complexities then the operators provided by C language and thirdly, the resources on these processors are inherently non-symmetric.

Apart from these newly added issues, the compilers already have a lot of traditional issues to deal with, like the alias analysis, and peephole optimizations.

3.2 How compilers are coping with these DSP needs

The compiler technology has tried to cope up with the requirements of efficient code generation for the DSPs. It has deployed various measures, the main strategies are listed below:

3.2.1 C Coding Guidelines

One of the most popular recommendations by the compiler providers is to write the C code in a way that would aid the compiler to generate better code. These guidelines, though maintain the code portable, are in some sense related to the target in question. What generated an efficient code for a particular machine may not generate efficient code on another. Since this is one of a very widely used method for optimization using the compilers, a complete section (section reference to next section) is dedicated to this topic.

3.2.2 Providing Direct Target Access To The User

At this level the compiler allows the user a direct access of the architecture and it’s resources. It does not put in any intelligence either to optimize the specified code, or to verify anything except the assembly syntax. Following are few examples:

**Assembly Inlining:** This allows direct replacement of the assembly code in the translated source. The assembly code provided by the user is directly inlined amidst the generated code. Here there is no direct mechanism to associate the C variables with the operands of the inlined instructions. Either these operands don’t have to be associated with any C variables (like in case of timer or mode settings), or the user has to take care of associating them with manual efforts.

**Assembly linking:** This provides the facility to link the user coded assembly functions to the C functions. User can manually code the performance critical routines, and, if they comply to the Application Binary Interface, or the calling conventions in short, then the compiler can resolve that function with the generated code.

**Data placement:** This provides the facility to place the data to reduce access overheads. The compilers provide the users with mechanisms to place data in the target memory in ways to reduce the access overheads. For example we could place the data in two different memory banks, or place specific sections in internal or external memory ranges. This is slightly different from the linker’s option of locating a section at a given address.
3.2.3 Language Extensions
As noted in [7], a high level language is characterized by the ease of understanding, naturalness, portability and efficiency of use. Also, some languages can more suitably express problems in specific domains.

Perhaps the biggest problem facing the DSP compilers is that the C language is inadequate to express the DSP application domain. On one hand, it is unable to capture a number of application specific inputs, which are known to the programmer, and on the other hand, it has poor capability of expressing the target’s peculiarities. For example, various data types required by the domain, as well as the addressing mode provided by the target simply have no method of getting captured in the language. This has prompted the compiler technology to provide extensions to the C language. However, there are no standard set of extensions, and they are very much vendor specific. They also result in non-portable code. There have been some attempts to standardize a few extensions specific to the DSP requirements [8]. Following are a few examples of the language extensions typically provided by the DSP compilers.

**Intrinsic functions:** This allows a direct access to the instructions of the target processor. Thus, these are platform specific extensions to the compiler, through which the user can instruct the compiler to emit a particular set of instructions. There are of course mechanisms to specify the operands of the instructions, and the variables that associate with those operands. As an example, one can directly set the carry in the Motorola DSP563xx processor by the following intrinsic function, where \_abs() gets translated directly to the abs instruction of the processor, and the operands to this instruction are specified by the standard convention. The C variable var1 maps to the accumulator required by the instruction

\[ \text{\_abs(var1);} \quad //\text{generate absolute instruction} \]

**New data types:** This incorporates DSP specific data types into the language. The compiler is made generous in accepting the data types that are specific to DSP domain, subject to certain conditions. The related issues like operations / casting / manipulation on these types is defined as per the user requirements. Also various kinds of data addressing modes required by the DSP domain are provided to take advantage of these accesses. As an example, the circular addressing mode can be set for a data_array, with the size of buffer specified. The compiler then takes care of mapping an appropriate addressing mode for the address register allocated to access the elements in that data_array.

**New operators:** New operators are introduced for DSP specific operations. These include features like rounding and saturation, that are an integral part of DSP computations, and can most of the times be done automatically, simply by setting the mode bits in the processor. Executing these operators either simply set the required mode in the processor, or they generate appropriate instructions for the operation.

3.2.4 New Optimizations
For the specific features provided by the DSPs, various target independent optimizations have to be disabled, whereas new target dependent optimizations peep in:
Operator combining: Combining one or more basic level operators to generate higher level operators supported by the processor. This is expected to result in the generation of the higher complexity instructions directly, hence utilizing the architecture capabilities more effectively.

Software pipelining, Instruction scheduling are a few other optimizations to utilize the processor pipeline more effectively.

4 Making the C code compiler friendly

The way a piece of C code is written affects the efficiency of the generated target code to a large extent. There are a variety of recommendations that exist as coding guidelines for different compilers, and there is a lot of material available in this regard [1],[2],[4]. Just to take an example, it is extremely difficult for a compiler to disambiguate the memory references through alias analysis. So, unless the compiler has the information to prove that the memory references are non-overlapping, it would not optimize the code to the extent where re-ordering of the pointer references is required [3]. Even if the programmer knows that the pointer references are distinct, the compiler would go in for a conservative code generation in such a case, for the sake of ensuring correctness.

The key in getting efficient code generated through the compiler is to make sure that maximum amount of information that the programmer has is passed on to the compiler. Though different compilers recommend different strategies for rewriting the C code, we will try to capture here some of the most general guidelines.

4.1 Making alias analysis easier

DSP architectures have mechanisms to fetch the data (required for next operations) in parallel to the execution of the current operation(s). This is a particularly beneficial feature for processing data in a loop, with successive data fetched and processed every loop iteration. DSPs provide an extremely tight structure for such loops. However, usage of pointers in the body of such a loop may restrict the compiler from rearranging the data accesses, eventually generating very inefficient code for such loops. In many cases, replacing the pointer accesses with array accesses is a definite help. Once the compiler knows that the accesses are independent of each other, software pipelining is performed, which takes care of fetching the data for the next iteration in the current iteration, improving the performance drastically.

\[
\text{for (I = 0; I < n; I ++)} \\
B += *ptr1++ * *ptr2++; \\
\]

Will tend to generate the sequential code equivalent to:

\[
\text{for (I = 0; I < n; I ++)} \\
\{    \\
\text{Fetch first operand for current iteration} \\
\text{Fetch second operand for current iteration} \\
\}
\]
Fetch B for current iteration
Perform MAC operation
Store B for current iteration
}

whereas if it is made known to the compiler that accesses to B, ptr1 and ptr2 are not aliases (by having the array accesses instead of pointer accesses, or by any other means), it can generate the code equivalent to:

fetch first operand for first iteration
fetch second operand for the first iteration
fetch B for the first iteration
for (I = 0; I < n; I++)
{
    Perform MAC operation
    Fetch first operand for the next iteration
    Fetch second operand for the next iteration
}
Store B

The following things can be noted:

1. “B” need not be stored and loaded and stored in each iteration, since it is known that nothing else accesses it inside the loop.

2. The piece of generated code inside the loop now has better chances of doing parallel data accesses than the previously generated code for a pipelined architecture that supports parallel accesses. All the three operations inside the loop may now be executed in parallel.

4.2 Usage of Data Types

There are two challenges in the usage of data types. The first and foremost is meeting the processing requirements of the applications. Secondly, an important consideration is to write code to match the DSPs capabilities in terms of the data widths for the operations.

For the DSP applications that require processing in fixed point arithmetic, the DSP compilers have provided specific data types, like \texttt{fract}. The appropriate usage of these data types results in a more efficient code generation, since the target also directly supports operations on such data types. These data types also come with the conversion semantics definition, like typecasting, promotion rules, etc., which are vendor-specific and hence result in a non-portable code.

For example, if a DSP has a 16×16-bit multiplier that produces a 32-bit result, and the compiler supports data types 'int' as 16 bits and 'long' as 32 bits, you could write an expression that matches the multiplier as \texttt{long}=(\texttt{long}int1) \ast (\texttt{long}int2). You would write a similar operation quite differently for targets with 32-bit or floating-point arithmetic.
4.3 Rounding and Saturation operation

Overflow is a condition where the width of the storage is insufficient to hold the width of the computation result. Saturation is a process of limiting this result to the nearest extreme that can be represented; so that the error is minimum. The ANSI C definition requires the integers and longs to wrap around on an overflow condition, whereas, any algorithm, that utilizes the output of computation would typically want an output which is saturated.

There is no standard mechanism in ANSI C that would allow the programmer to convey to the compiler that the saturated result is of interest rather than the wrapped around result, though the underlying target does have the capability in most cases to perform the necessary saturation functionality. Compilers either provide vendor specific extensions to C to pass on this additional information, or the user has to explicitly simulate the saturation process in the program, which results in a non-optimum usage of the feature provided by the target processor. The situation with rounding is similar. In most of such cases, the programmer ends up accessing these processor features with the help of intrinsics or assembly inlining, where the compiler does not provide vendor specific extensions for these.

4.4 Some general guidelines

Points below, when applied to DSP programs, generally tend to generate good code in most cases-

- Initialization of global variables with constants at load time, which would prevent them getting computed at run-time.
- Adjust the typecasts as per the underlying target’s convenience, so that the compiler can directly map the operations onto the target processors instructions. This may not always be possible though, due to the bit-exactness requirements for the algorithms.
- Memory mapped register should be accessed with constant pointers rather than pointer variables, which will make alias analysis for the compiler easier.
- Global / local decision for the variable in some cases can be done based on the knowledge of the access mechanisms provided by the architectures.
- Wherever possible, use constant loop counts, allowing the compiler to map them efficiently to the looping mechanisms provided by the architectures, as well as helping to make decisions on software pipelining and loop unrolling.
- The code inside the loop can be more expressive instead of being a tight C code.
- Program sequencing changes inside the loops should be avoided (eg. Function calls, branches out of and in to the loops).

5 Typical DSP application development using compilers

As seen above, the compilers perform very bad in terms of the generated code efficiency for straight compilation on a DSP, and are no good for straight code generation, where MIPS and memory requirements are stringent. On the other side, the processors are
evolving with more complex architectures and pipeline structures, thereby making it prohibitively difficult for the users the program in assembly directly for the entire application. At the same time, the applications are expected to reach to the market faster than ever before.

The application development is approached in the following manner:

**Step 1:** Compilation and verification of the code using the compiler and toolchain. In this step, the entire application in question is ensured to run correctly, without any modifications, or optimizations.

**Step 2:** Modification of the application to make it compiler friendly. Here the C code is modified with the vendor specific intrinsics and data types. Also the code is re-arranged at places such that compiler finds it easy to generate an efficient code for the application. Optimization switches are activated to get the best possible performance with the compiler.

**Step 3:** Application profiling on the target platform. The compiled and correctly executing code is profiled to determine the hot-spots (or the performance critical modules) in the code. These typically are the modules that follow the 80-20 rule, where 80 percent of the computation happens in 20% of the code, and vice-versa.

**Step 4:** Coding performance critical modules in assembly. The performance critical modules are then coded in the assembly of the target platform. These modules are typically the tight loop computation modules. Once coded, these modules are linked to the generated modules to build the application.

Step 3 and Step 4 are repeated till the desired performance for the application is achieved. This way it is ensured that the code is mostly portable in the 80% modules, and only the 20% code has to be completely re-written for every new platform. This way about 30% to 40% reduction in the development time is achieved as compared to the assembly development time, at an expense of approximately 20% degradation in the overall application efficiency, in general.

## 6 Conclusion

The DSP compiler technology has evolved since the introduction of the DSPs. However, the DSP programmers still continue programming in the assembly language of the target for achieving efficiency. We have here studied a variety of issues that make it difficult to for the DSP compilers to generate good tight code. The primary of the reasons include the inadequacy of the C language to capture the information that the programmer has about the code as well as inability of the compiler to take advantage of the features that the target provides to accelerate the processing of DSP algorithms. The non-symmetric nature of the DSP architectures in terms of the provided resources compounds the problem for the compiler multifold.

Even with the various mechanisms that the compiler vendors provide to overcome these limitations of the compiler by providing a few non-standard extensions to the C has not gone a long way in terms of increasing the efficiency of the generated code. Many of the domain specific features still remain to be captured in the extensions. Though the
modifications in the C code yield some improvement in the generated code, and retain the code portable, they are not completely independent of the target, and may not yield good code on other targets. And the programmers are still not relieved of coding in assembly for the performance critical modules.

Further improvements in the generated code performance may possible with a better domain specific language for DSP applications and better mechanisms in the compiler technology to extract the DSP specific features of the target platform.

7 References
1. Guidelines for efficient C code generation in accumulator based DSPs – the C54x as an example, Karen Baldwin et.al., Texas Instruments Inc., ESC 99
2. Trends in application programming for digital signal processing, Robert Jan Ridder, Tasking Inc., ESC 99
3. C compilers for DSPs flex their muscles, Markus Levy, EDN Magazine, June 97
4. Tips for writing more efficient DSP C code, Alan Davis, Senior Member Technical Staff, TI's Software Development Systems Semiconductor Group, EDN Magazine, June 97
7. Principles of compiler design, Alfred V. Aho, Jeffrey D. Ulman, Addison-Wesley/Narosa, 1999