



Compiler Technology and its Relationship to DSP Performance

ACE Associated Compiler Experts by

Version: 1.8
Date: May 23, 2003
Status: release
Confidentiality: restricted
Reference: CoSy-8116-dsppformance

Compiler Technology and its Relationship to DSP Performance

by ACE Associated Compiler Experts bv.

© Copyright 2002 by ACE Associated Compiler Experts bv,
Amsterdam, the Netherlands.

All rights reserved. No part of this document may be copied, photocopied, reproduced or translated in any way, without prior written consent of ACE Associated Compiler Experts bv.

Every care has been taken in manufacturing the supplied product and its documentation. ACE Associated Compiler Experts bv will neither assume responsibility for any damages caused by the use of its products, nor accept warranty or update claims, unless stated explicitly otherwise in a special agreement.

This document contains Confidential Information which has been disclosed subject to licensing and/or confidential disclosure conditions as agreed with the authors and shall be treated in the strictest of confidence.

The information contained in this document is subject to change without notice.

Printed in the Netherlands, May 23, 2003.

Many of the designations used by manufacturers and vendors to distinguish their product are trademarks. ACE Associated Compiler Experts bv has made every attempt to supply trademark information of manufacturers and their products mentioned in this document. ACE Associated Compiler Experts bv also recognizes any trademarks used in this document but not mentioned below.

Trademark notices

CoSy[®] is an international trademark of ACE Associated Computer Experts bv.

SuperTest[™] is an international trademark of ACE Associated Computer Experts bv.

HP-UX[®] is an international trademark of Hewlett-Packard Company.

Microsoft[®], Windows[®], and Windows NT[®] are international trademarks
of Microsoft Corporation.

Solaris[®] is an international trademark of Sun Microsystems, Inc.

Linux[®] is an international trademark of Linus Torvalds.

Intel[®], Pentium[®] and Xeon[®] are international trademarks of Intel Corporation.

UNIX[®] is an international trademark of Unix System Laboratories.

Red Hat[®] is an international trademark of Red Hat, Inc.

PostScript[®], Acrobat Reader[®], Acrobat logo[®] and PDF[®] are
international trademarks of Adobe Systems, Inc.

Netscape[®] and Netscape Navigator[®] are international trademarks
of Netscape Communications Corporation.

Motorola[®] is an international trademark of Motorola, Inc.

PowerPC[®] is an international trademark of International Business Machines Corporation.

SPARC[®] is an international trademark of SPARC International, Inc.

Contents

1	Introduction	4
2	Platforms Shorten Compiler Development	4
3	Supporting DSP Features	5
4	Developing Optimisers	9
4.1	Hardware/Software Co-design	10
4.2	Performant Compilers	10
	References	13

1 Introduction

High level languages reduce programming effort, facilitate program maintenance and encourage IP re-use – three factors that are critical to software development success in fast-moving consumer electronics markets. Yet despite this fact, most of the DSP processors at the heart of consumer products such as mobile phones, MP3 players and DVD players are still laboriously programmed in assembly language.

The reason lies in the widespread and previously well founded belief that no compiler can match the ability of a skilled assembly language programmer when it comes to meeting the code size or execution speed requirements of these applications. That belief may still be true, but compiler generated DSP code can now get extremely close to hand-crafted assembler in terms of code size and run-time performance. So close that any marginal loss in performance is far outweighed by the advantages of writing the application in a high level language such as C.

Producing a compiler that can achieve this level of DSP performance is not exclusively the domain of specialized compiler companies. EDA tools for chip design are now so advanced that you can build your own DSP if you choose to, with an architecture and instruction set that is highly specific to a particular application domain. The best group of people to develop a high-level language compiler for such a DSP is often the team that developed its hardware architecture, or assembly language programmers who are skilled at writing applications for it. Both these groups have detailed knowledge of how to squeeze the ultimate performance out of the architecture. Writing the compiler themselves not only leverages this knowledge. It also avoids the need to take a third-party compiler developer up the learning curve.

2 Platforms Shorten Compiler Development

Building compilers from scratch is not a realistic option. Even a relatively straightforward compiler takes many tens of programmer-years to build in this way. You must build a frontend that recognises each construct of your chosen high-level language so that it can turn it into an intermediate representation that reveals the structure of the program. You must then generate optimisers that recognise local and global characteristics of this structure and modify it to achieve the required performance – for example, performing constant propagation, loop unrolling or redundant code elimination. And finally you must write a back-end that allocates registers and maps the program onto the instruction set of your target processor.

For a given high-level language, the compiler's frontend and intermediate representation are theoretically independent of the target processor. Only its back-end code generator has to be target specific. However, the more you can tune earlier phases of the compilation process towards specific target features, the better the compiler will be at generating efficient code. For embedded DSPs in particular, the ability to tune the optimisers that operate on the intermediate representation is usually critical to achieving the required run-time performance.

For languages such as ISO C, which has now become a defacto standard for high-level language programming of embedded systems, it is possible to circumvent a large part of the compiler

development process by building your compiler on a platform such as the GNU Compiler Collection (GCC). This approach works well when generating compilers for mainstream microprocessors, but does not work so well for DSPs. Most DSPs have much less regular instruction sets than microprocessors and they often include application-specific hardware acceleration units such as Viterbi decoders. In addition, DSPs often feature a high degree of instruction-level parallelism, requiring the compiler to have instruction scheduling and instruction packing capabilities in its back-end that are either poorly supported or not supported at all by the GCC.

3 Supporting DSP Features

As mentioned earlier, target specific information must be utilized during every phase of the compilation process if you are to create efficient optimising DSP compilers. Because of the wide variety of DSP features, this requires a very open and flexible compiler development platform in which the compiler developer can describe and leverage target architecture features.

Existing compilers for DSPs already employ a range of techniques to indicate DSP features to the compiler front-end. These include programming in C++ with classes, using C plus intrinsics, and using source code annotation or pragmas. However, the problem with all these techniques is that they either impair source code portability or result in less than optimum object code (see reference [1]).

For three generic features of DSPs — fixed point data types, circular pointers and divided memory spaces — ACE Associated Compiler Experts, in collaboration with several lead customers, has tackled this problem by proposing a set of C-language extensions that allow these features to be described in the source code (see figures 1 and 2). ACE has submitted these extensions to the International Standards Organisation (ISO) for inclusion in future versions of the C language, or for publication as an official set of C language extensions. Applications written in the extended language, DSP-C, will therefore be fully portable (see reference [2]).

For programs written in DSP-C, even general optimisations such as constant folding can take these generic DSP features into account. DSP-C itself does not make for an efficient compiler, but the information provided by its C-language extensions is a pre-requisite for a compiler's optimisers to do their job effectively.

Although DSP-C allows programmers to parameterise generic DSP attributes, target specific DSP features still remain. These must also be taken into consideration as early as possible in the compiler if maximum advantage is to be leveraged from code optimisations. For example, it is quite normal for audio DSPs to have non-standard data paths 20 or 24 bits wide. Yet the front-ends of many standard compiler platforms only accommodate the standard 16, 32 or 64-bit data paths used by microprocessors, leaving you with redundant bits in the intermediate representation of the program that must be truncated later on. A compiler development platform for DSPs should let you take these global features of the architecture into account

A typical Finite Impulse Response (FIR) filter written in plain C cannot describe the architectural features of DSPs that speed this type of filter algorithm. As a result, it is difficult for compilers to recognize program structures that will map onto target architecture features. See figure 2 for the same program using DSP-C extensions.

```

/* Typical FIR algorithm written in standard C.
 * Note that this program uses (long) integer arithmetic, but
 * does take care of possible overflow in the result.
 */
#define N 16
extern int x[N], h[N];
static int start;
extern int new_x;

int
fir(void)
{
    int i;
    long sum = 0L;

    for (i=0; i<N; i++) {
        sum += (long)x[start] * (long)h[i];
        if (++start >= N) start = 0;
    }

    x[start] = new_x;
    if (++start >= N) start = 0;

    sum >>= 15;          /* scaling */
    if (sum < INT_MIN) { /* saturation */
        return INT_MIN;
    } else if (sum > INT_MAX) {
        return INT_MAX;
    }

    return (int) sum;
}

```

Figure 1: A typical FIR algorithm written in standard C.

in the compiler front-end. ACE's CoSy-DSP compiler development platform lets you define any data-width you need.

Re-written using ACE's proposed DSP-C extensions, this program is able to use the fixed-point datatype that is being recognized as essential to support embedded processors, plus ACE's proposed 'circ' qualities to describe circular buffers. See figure 3 for the same program in assembly.

```
/* Similar FIR algorithm written in DSP-C
*/
#define N 16
extern circ fixed x[N], h[N];
static circ fixed *xp = x;
static circ fixed *hp = h;
extern fixed      new_x;

fixed
fir(void)
{
    int i;
    accum sum = 0.0a;

    for (i=0; i<N; i++) {
        sum += (accum)*xp++ * (accum)*hp++;
    }
    *xp++ = new_x;

    return (sat fixed)sum;
}
```

Figure 2: FIR algorithm written in DSP-C.

In the same way that global features of your target DSP architecture must be taken into account when the intermediate representation is generated, more detailed architectural features must be taken into consideration by the optimisers that manipulate this intermediate representation.

Most DSPs do not employ the orthogonal register-to-register architectures of RISC microprocessors. Their architectures are much more complex, with registers spread throughout the data path — situated between sequential ALUs such as a multiplier/accumulator and barrel shifter, or embedded within individual ALUs. In addition, DSPs frequently have separate register sets dedicated to their address calculation units, as well as special addressing modes designed to speed the processing of data arrays.

Few DSPs support offset addressing, for example, but many of them have complex post-increment or post-decrement addressing modes that automatically adjust register pointers after load/store operations. If optimisers that work on the intermediate representation know

Armed with this information, a compiler is able to map the loop onto a target DSP's hardware-loop instructions (target example: Texas Instruments' TMS320C54x). See figure 1 for the corresponding C-code.

```

.
.
.
; AR2 contains xp
; AR3 contains hp
; AR5 points to new_x
STM #N      , BK      ; block size of the circular buffer
SSBX 1      , 6       ; setup fractional mode (set FRCT bit)
STM #1      , ARO     ; increment value for circular addressing

RPTZ B      , #N-1    ; for (i=0; i<N; i++) { B=0;
LMS *AR3+0% , *AR2+0% ; B += h[i] * x[i]; }
MVDD *AR5   , *AR2+0% ; write the new input value to delay line of x
SAT B      ; saturate the result
.
.
.

```

Figure 3: FIR algorithm for Texas Instruments' TMS320C54x.

about these post-increment/decrement operations, they can use them to eliminate the lengthy code sequences needed to turn offset addresses into absolute addresses.

However, this requires the post-increment/decrement instructions to be associated with the previous load/store operation on a particular register rather than the current one. As a result, the optimisation of one piece of code may involve modifications to code that is some distance from it. Such optimisations would be impossible in the code generator because it has too narrow a view of the program, but pre-configuring them in the intermediate representation allows the compiler back-end to generate code that is considerably more efficient in terms of size.

Although the C language itself does not support parallelism, a well defined intermediate representation also gives you an early opportunity to start matching the source code to the parallelism inherent in many DSP architectures. For example, it makes it easy to recognise and vectorise data arrays that can be handled through the SIMD (Single Instruction Multiple Data) memory-to-memory architectures of certain DSPs. Manipulating the intermediate representation of the program to create data structures that maximise the use of SIMD operations can dramatically increase program execution speed.

While large-scale parallelism needs to be handled in the intermediate representation, the fine-

grain parallelism typical of many DSP architectures is best exploited in the code generator. Its detailed knowledge of the target DSP's instruction set allows it to schedule and pack instructions so that all of the DSP's ALUs are kept maximally busy. This may be as simple as mapping sequential multiply and accumulate instructions onto a single instruction for a DSP's MAC unit. However, it is also possible for more sophisticated optimisation engines to assemble VLIW instructions that control multiple ALUs while taking into account data dependencies between parallel computations. Even the seemingly simple process of recognising multiply/accumulate operations and setting up the appropriate registers to perform a MAC instruction is extremely difficult for a GNU based compiler.

4 Developing Optimisers

Most of the high-volume applications in which DSPs are currently used, such as personal audio players and mobile phones, impose severe design constraints in terms of system cost and power consumption. This means that DSPs are likely to remain highly tuned to particular application areas, and will therefore require compilers that are equally well tuned.

To generate such highly tuned compilers it is probable that you will want to add your own optimisers. To make this possible, a compiler development platform must fulfil two requirements. Firstly, its intermediate representation must be entirely visible to the compiler developer. Secondly, the intermediate representation must be unambiguously defined. Unless both these requirements are fulfilled, it will be impossible to generate optimisers that can examine every aspect of the code, or to guarantee the validity of the code after these optimisers have been run.

One of CoSy-DSP's key strengths is that it allows you to write optimisers that work at a much higher level than assembler. The advantages that apply to high-level language programming then also apply to the optimisers themselves, in the form of shorter development times, better maintainability and easier validation. Optimisers that are written and that operate at a higher level are also easier to parameterise. This makes them more applicable across a range of DSP architectures.

Many DSPs, for example, support hardware loops, but the way in which these loops are handled is often very target specific. Some DSPs limit the number of instructions that you can execute in the loop, the range of the loop counter, or the conditional statements that can be included in the loop. CoSy-DSP's standard hardware loop optimiser allows you to specify these constraints so that it can more accurately identify loops that are suitable for mapping onto the target's hardware loop support functions.

CoSy-DSP also has another very useful attribute when it comes to developing optimisation strategies. Each of its optimisers is capable of running independently — examining the intermediate representation of the program, modifying it, and returning its results in the form of new intermediate representation code. This mutual independence allows you to run optimisers in any order (see figure 4). It also allows you to validate the optimised code against the source code at any point in the process. In optimising compilers for DSPs this is a very important capability, because changing the order in which optimisers run can have

a significant effect on code efficiency. This ability to run optimisers in any order was one of the original design goals for CoSy, so that ACE's own engineers could evaluate the effects of phase ordering.

The independence of optimisers has another important advantage. It allows new optimisers to be added without affecting those that are already there. DSP architectures and applications are so varied that no 'out-of-the-box' compiler development platform can possibly contain all the optimisers that may be needed. However, if it is easy to insert new ones, compiler developers can use their own intimate knowledge of the target architecture to create optimisers that leverage particular architectural features of the DSP or particular run-time requirements of the application.

In this respect, developers of custom compilers are often in a better position than developers producing compilers for standard DSP parts, because they know about both the target DSP architecture and the target application. Being able to match optimization strategies to applications is one of the most powerful tools in achieving efficient code. It mimics the skills of an experienced assembly language programmer.

4.1 Hardware/Software Co-design

Even today, compiler development for DSPs is typically only started after the DSP's hardware architecture has been fully defined. However, a compiler development platform that allows new compilers to be generated and evaluated extremely quickly can also provide you with the opportunity to influence the hardware architecture design.

ACE's tool-set, for example, includes a CoSy-DSP simulator that allows you to emulate programs with 'bit-true' accuracy on standard workstations. As a result, you can functionally test your application before the target hardware is available. CoSy also allows you to generate new compiler variants in a matter of days, making the combined tool set an ideal environment for architecture exploration. With the acceptance that high-level language programming is the way forward for DSPs, and the consequent need for highly efficient optimising compilers, the co-design of hardware architectures and software compilers will become increasingly important.

4.2 Performant Compilers

If you are already an expert at programming DSPs in assembler, developing a compiler will allow you to embody that knowledge into a powerful software development tool that others can use. Equipped with the right compiler development platform, generating such a compiler is not that difficult. Many tens of years of programming effort will be provided by the platform as it comes 'out-of-the-box', and many generic optimisers will already be included.

ACE's CoSy-DSP compiler development platform ships with over 100 standard optimisers and over 45 programmer/years of expert compiler knowledge built into it, which means that even out-of-the-box you can use it to create optimising compilers that produce very efficient code. When it comes to adding your own optimisers, a complete framework and tool-set for

Changing the order in which optimisers run can have a significant effect on code efficiency. CoSy's Engine Description Language makes the ordering of optimizers easy.

```
//
// Top of the compiler.
//
ENGINE CLASS compiler (IN IR) [top]
{
REGION u: mirUnit;
PIPELINE          // Defines compiler flow:-
  frontend (u)      // translation to intermediate representation,
  optimizer (u)     // optimisation of intermediate representation ,
  codegenerator (u) // mapping to target DSP assembler
}

//
// Top Level Optimizer: Decompose unit into
// procedures and run Procedure Level Optimizers.
//
ENGINE CLASS optimizer(IN u: mirUnit)
{
REGION p{}: mirProcGlobal;
PIPELINE
  unit2proc (u, p{}) // Decomposes unit into individual procedures
  optim_proc (p)     // Runs procedure sequence defined by optim_proc
}

//
// Procedure Level Optimizers: Iterate the
// following optimizers until fixpoint
//
ENGINE CLASS optim_proc(IN p: mirProcGlobal)
{
LOOP
  unsetchange (p) // This list defines the order in which
  constprop (p)   // optimizers run on the procedures. The LOOP
  copyprop (p)    // statement allows iteration until
  deadcode (p)    // optimization leads to no further changes
  constfold (p)   // in the intermediate representation
  EXIT unchanged (p)
}
}
```

Figure 4: Flexible optimization strategies.

generating, evaluating and integrating them into your compiler is already there. As a result, you can tune compilers to particular target architectures and run-time requirements, focusing on code size or execution speed, while making the most of target-specific DSP features. CoSy-DSP also includes a complete set of validation suites so that you can produce rigorously tested ‘production-quality’ compilers.

As the case studies included in this white paper illustrate, highly efficient optimising compilers for a range of custom DSPs have been constructed by taking CoSy out-of-the-box and adding relatively little engineering effort. In every case, the code efficiency produced by these compilers has exceeded expectations.

References

- [1] ACE Associated Computer Experts bv. Programming DSPs in High-Level Languages. 2000.
- [2] ACE Associated Computer Experts bv. Proposals for Extensions for the programming language C to support embedded processors. <http://std.dkuug.dk/JTC1/SC22/WG14/www/docs/n907.pdf>, 2000.