Reverse compilation of Digital Signal Processor assembler source to ANSI-C

Experience and observation paper

Adrian Johnstone
A.Johnstone@rhbnc.ac.uk

Elizabeth Scott
E.Scott@rhbnc.ac.uk

Tim Womack
T.Womack@rhbnc.ac.uk

Department of Computer Science, Royal Holloway, University of London,
Egham, Surrey, TW20 0EX, UK.
Tel: +44 (0) 1784 443425. Fax: +44 (0) 1784 439786

Abstract

Digital Signal Processors (DSPs) are special purpose microprocessors optimised for embedded applications that require high arithmetic rates. These devices are often difficult to compile for: compared to modern general purpose processors DSPs contain unusual hardware features, they often have very small address spaces and they require correct scheduling of operands against pipeline registers to extract the highest levels of available performance. As a result, high level language compilers for these devices generate poor quality code, and are rarely used in practice. Recently, new generation processors have been launched that are very hard to program by hand in assembler because of the complexity of their internal pipelines and arithmetic structures. DSP users are therefore having to migrate to using high level language compilers since this is the only practical development environment. However, there exist large quantities of legacy code written in assembler which represent a significant investment to the user who would like to be able to deploy core algorithms on the new processors without having to re-code from scratch. This paper presents a working report on the development and use of a tool to automatically reverse-compile assembler source for the ADSP-21xx family of DSPs to ANSI-C. We include a discussion of the architectural features of the ADSP-21xx processors and the ways in which they assist the translation process. We also identify a series of translation challenges which, in the limit, can only be handled with manual intervention and give some statistics for the frequency with which these pathological cases appear in real applications.

Keywords: reverse compilation
digital signal processing
low to high level language translation

1 Introduction

Digital Signal Processors are special purpose microprocessors optimised for embedded applications that require high rate fixed and floating point arithmetic. Application areas include mobile telecommunications, video signal modulation and image processing. Historically, technology and cost constraints have stratified the market into two layers:

1. low cost, short word length processors typically supporting 16-bit fixed point arithmetic, and
2. higher cost processors supporting 32-bit fixed point and floating point arithmetic.

Recently, a third class of processors has appeared:

3. processors that provide a high degree of parallelism in the form of multiple, simultaneously active functional units.

In comparison to a general purpose RISC architecture such as MIPS, all of these processors present highly irregular architectures to the programmer, with many restrictions on the combinations of registers and operators that may be executed. In addition, they typically feature special purpose hardware to support indirect addressing and looping that, once initialised, executes in parallel with the main instruction stream allowing repetitive code to execute with no overhead from decrementing and testing induction variables. As in a general purpose superscalar processor, to extract the maximum performance from these devices requires careful scheduling of operations and memory-register transfers.

It is usual for digital signal processors of all classes to support multiple independent memory spaces so that multiple operands can be fetched in parallel. The short word length (class 1) processors have very small address spaces (typically as little as 16K instructions).

These architectural eccentricities present significant challenges to the compiler writer, and in reality the performance of compilers for the class 1 architectures is very poor: a widely used rule of thumb suggests that there is a 20:1 performance ratio between hand-written assembler and compiled code. As a result, the manufacturer supplied C compilers are rarely deployed for real applications and most real programs for both class 1 and class 2 processors are hand coded in assembler.

The class 3 'super-processors' display all of the special features of the class 1 and class 2 processors but in addition provide hardware that can exploit the regularity of many DSP algorithms through large scale data and instruction parallelism. We will describe the differences between general purpose processors and DSPs in some detail in section 2 below. For now we note that the class 3 processors are highly parallel and highly pipelined. It is well known that such processors are extremely hard to program at assembler level because of the requirement to manage the timing relationships between operand usage within the pipeline and across functional units which are interconnected using switching matrices that themselves introduce constraints. The hardware does not guarantee to enforce the correct pipeline data dependencies and so simple programming errors can yield extremely subtle and hard to find bugs. The class 3 processors are supported by ANSI-C compilers with suitable scheduling algorithms that can guarantee to preserve data dependencies, and practical large scale software development for these processors is likely to be performed in a high level language.

Programmers used to developing applications in assembler for the relatively simple traditional fixed and floating point digital signal processors are now having to move to development in a high level language. Usually the language of choice is ANSI-C with some hardware specific language extensions to support particular features.

In many cases, companies have already invested years of effort in the development of algorithms that they wish to use on the new architecture, but which are coded in assembler for the older style processors. There is essentially no support available for migration between processors even from the same manufacturer.

We provide three levels of translation from assembler to C:

1. low level translation, in which internal register and symbol names are preserved in the ANSI-C code and in which control structures are mostly based on `goto`s,
2. control flow translation in which a structural control flow algorithm is used to map assembly level control flow into high-level ANSI-C control flow statements such as `for` and `switch`,
3. data flow translation in which references to internal data pipeline and address registers are eliminated.

All three levels of analysis are useful when porting an application. It is rare for a complete embedded application to be ported as-is since all embedded systems contain references to hardware which is application specific. Therefore, our reverse compiler is used both for program comprehension and for the production of working code in the new environment. When attempting to understand an assembler program (whose author may not be available) it is very useful to be able to see a low level translation to C in which there is an (approximately) one-to-one correspondence between the assembler code and the lines of translated C. Later stages of the porting process benefit from having the higher level code available.

It turns out that those features of the DSP which make compilation hard (such as zero-overhead looping hardware) assist the reverse compilation process. This is because the special purpose hardware directly implements specific coding idioms (such as nested `for` loops and circular buffer addressing) which can easily be rendered as high level code. From the compiler's point of view, recognition of such idioms is hard, but their presence in our translated code aids human comprehension.

In this paper we present working report on the development and use of a tool to automatically reverse-compile assembler source for the ADSP-21xx [Dev90] family of DSPs to ANSI-C. We include a discussion of the architectural features of the ADSP-21xx processors and the ways in which they assist the translation process. We also identify a series of translation challenges which, in the limit, can only be handled with manual intervention and give some statistics for the frequency with which these pathological cases appear in real applications.

2 DSP architectures

In order to motivate the discussion of our translator, we present here an overview of the architecture of typical Digital Signal Processors. Our aim is to highlight the ways in which DSPs differ from conventional general purpose processors.

DSP architectures are characterised by the presence of some or all of the following features.

Single cycle multiply and shift Traditional single cycle arithmetic operations of addition and subtraction are supplemented by single cycle multiply, multiply-and-accumulate and barrel shifter units.

Parallel operand fetch Multiple operand spaces allowing single-cycle fetching of two operands from memory in parallel with an arithmetic operation.

Extended and saturated arithmetic Extended arithmetic precision allows the accumulation of large results over a sequence of operations without overflow occurring. Saturated arithmetic prevents wrap around of values when overflow occurs: the result instead *saturates* and is forced to the maximum or minimum value that can be represented.

Parallel update of data address pointers Special hardware index registers are provided with associated increment and bounds registers that allow a pointer to be stepped through an in-memory buffer without needing explicit update instructions. The bounds registers allow so-called *circular addressing* where the address pointer resets to the beginning of a buffer when it overflows the buffer.

Conditional instructions DSP algorithms often feature `if` statements with very small blocks of code: sometimes as little as one instruction, In such cases, rather than evaluating a condition and optionally jumping over the block it is more efficient to attach a condition field to each instruction that can be used to disable it. The cost of this approach is in growth of the instruction word.

Zero-overhead looping DSPs provide special hardware counter registers that can be used to control repetition of blocks of code without requiring explicit instructions to update and test the counter. usually, these units require one instruction to initialise and subsequent execution of the loop proceeds with no overhead in loop control instructions.

Many of these features also appear in mainstream processors. Intel processors often have a simple facility to repeat an instruction without overhead, and the ARM architecture allows for conditional instruction execution.

2.1 New generation architectures

The arithmetic rate requirements for realtime digital signal processing are extreme, and it is not uncommon for multiple devices to be used within an application, each handling some specialised part of the overall system. Improvements in semiconductor device processing have led to the recent introduction of a group of new DSP 'super-processors' which provide features that support such parallelism directly. This group (the third class of architectures identified in section 1) is in reality diverse. Just as in the general purpose processor arena, there is little unanimity on the best way to add parallelism to an architecture. Indeed, the three 'super-DSPs' announced so far adopt three radically different approaches.

1. Single Instruction, Multiple Data (SIMD) style extensions add functional units that execute identical operations on multiple data streams under the control of a single instruction stream. SIMD *array* processors have been researched since the 1950's but these special purpose processors have limited applicability. The integration of SIMD style instructions into a conventional single-stream processor is a more recent development: the approach is typified by Analog Device's 210xx SHARC series of DSPs [Dev98]. Similar ideas have been implemented in the general purpose processor world: the Pentium II MMX instructions, the Pentium *III* KNI extensions and AMD's 3D-now! features all add SIMD style instructions to the basic Pentium architecture to provide improved performance on the DSP-type processing required to support multimedia applications.

2. Multiple Instruction, Multiple Data (MIMD) style processing is provided by the Texas Instruments TMS320C8x [Ins99] processor which contains a master processor and up to four parallel processors. The master processor is very similar to a conventional RISC general purpose processor and the parallel processors are similar to conventional fixed point DSPs. The processors are connected to shared memory *via* a crossbar switch.

3. Very Long Instruction Word (VLIW) architectures are represented by the Texas Instruments TMS320C6x [Ins95] DSP. VLIW processors present the programmer with a heteroge-

neous collection of functional units that may in parallel execute different operations whilst all being under the control of a single instruction stream. One way to view VLIW architectures is as a half-way house between SIMD and MIMD processing: there is a single instruction stream as in SIMD, but the data streams each have their own field within the instruction and so may execute different operations. In a true SIMD processor, one instruction is broadcast to all processors limiting applicability to arrays of data that in which all elements must be processed in the same way.

Of these three, the VLIW approach is perhaps the most technically interesting and certainly the least well studied. There have been very few VLIW's built and fewer still sold. However, this situation may be about to change. Much of the original work on VLIW's was performed by Josh Fisher's group at Yale [Ell85] who went on to market a general purpose Unix computer called *Multiflow* based on VLIW techniques. Although this was not a commercial success, the designers have continued their work at Intel and Hewlett-Packard. The forthcoming Merced processor which may finally displace the 80x86 family of processors is based on VLIW principles.

2.2 Programming the new architectures

Perhaps the only common feature of these three diverse processors (the AD2100x SHARC, the TMS320C8x and the TMS320C6x) from the programmer's point of view is that they are all much harder to code for than the class 1 and 2 DSPs. In addition to the system-level parallelism described above, these new processors have programmer-visible pipeline delays which requires the programmer to carefully sequence instructions so as to ensure that an instruction does not attempt to use the result of a previous instruction that is still in the pipeline. In general, hardware interlocks are not provided. A programmer may write a sequence of instructions that appears correct when viewed as purely sequential code but in which the data dependencies are not maintained when executed on a pipeline.

These effects are familiar to any programmer that has written assembler code for pipelined RISC processors such as the original MIPS devices which also lacked hardware interlocks. On the super-DSPs, the pipeline effects coupled with the difficulty of scheduling operations efficiently across multiple functional units present perhaps the most challenging machine level programming environment of any currently available commercial processors. This means that in practice high level, as opposed to assembler level, programming languages will be used.

It is not clear that the users of DSPs are anxious to move to a high level language development environment. There is considerable resistance to the inefficiencies displayed in machine generated code from programmers who have spent much of their careers wringing the last cycle of throughput from the class 1 and 2 processors through the exploitation of subtle programming tricks. We know of one very experienced programmer who spent one week constructing a small program for the TMS320C6x processor and then a further three weeks rescheduling and reworking the code to ensure that no data dependencies were breached. Overall, code production on this project averaged between one and two lines per day.

Super-DSP vendors emphasise the utility of their associated C compilers (in contrast to the compilers for class 1 and 2 processors which are unpopular with users) but provide little in the way of migration aids for assembly code written for those earlier processors. At one level this may seem reasonable: DSPs by their very nature are used almost exclusively in embedded applications where they interact with application specific hardware – large parts of the embedded code are therefore inherently unportable. However, at the heart of each application there is usually a set of reusable functions such as mobile telecommunications protocol handlers, image processing kernels and audio conditioning filters. These functions written in assembler represent significant intellectual property for the developer, and it would be ideal if the code could be moved to the new processors rather than being reimplemented from scratch in a high level language.

3 The `asm21toc` translator

`asm21toc` is a translator from `asm21` (the assembly language of the Analog Devices ADSP-21xx series processors) to ANSI-C. The code is intended to be compiled for the TMS320C6x VLIW style super-DSP, and we support optional translations that exploit some of the language extensions provided by Texas Instruments in their C compiler for that processor. Its design goals are as follows.

1. To produce an ANSI-C program with the same semantics as the input `asm21` program.
2. To produce ANSI-C programs which are *readable* and which aid comprehensibility of the original code.

3. To provide copious diagnostic information, particularly concerning variable usage.

4. To identify and report on `asm21` code which is not directly translatable, such as self-modifying code and non-statically visible jumps.

We call `asm21toc` a *reverse* compiler since it produces ANSI-C from hand-written assembler code as opposed to a *de-compiler* which produces C from a previously compiled C. A de-compiler, by its very nature, is unlikely to encounter features in the code to be decompiled that have no counterpart in the high level language. A reverse compiler, on the other hand, must be able to cope at some level with the complete universe of possible assembly language programs. *In extremis* (say when encountering self-modifying code) the reverse compiler should at least report the problem, whereas a decompiler is unlikely to even attempt the necessary analysis.

3.1 The feasibility of translation

There is a very small number of successful reverse- and de-compilers reported in the literature, and so our design goals might be seen as rather ambitious. More than one of our correspondents has asserted that they are completely unachievable, and were we dealing with conversion of binary code to C conversion for a modern, pipelined RISC processor we might be inclined to agree. However, it turns out that many of the features of small DSPs aid the translation process.

1. The fundamental problem of *binary* decompilation is how to separate code and data within a single memory image. We start from assembler source and our processor has (almost) separate code and data spaces. Apart from the potential for self-modifying code (which requires writes to the code space) our program and data are trivial to separate.

2. Our assembler programs are almost exclusively concerned with integer and fixed point operations which are clearly signaled in the instruction stream. As a result, high-level type analysis is not required.

3. The `asm21` assembler allows one dimensional arrays to be declared which significantly eases the dataflow analyser's task and allows us to make assumptions about patterns of data access to variables.

4. The DSP's high level control flow instructions such as the zero-overhead looping unit and the conditional instructions enable us to extract high level `for` loops and simple `if-then` statements trivially.

5. On the ADSP 21xx, indexed indirect access to data and program memory can only be performed *via* special data address generator registers. This significantly simplifies the analysis of addressing modes. Indeed, on the ADSP-21xx data accesses trivially translate into either simple variable accesses or into dereferenced pointer accesses.

6. The use of small hardware stacks restricts the nestability of functions and loops to only 16 and 4 levels respectively for the ADSP-21xx which eases many inter-functional control and data flow analyses.

On the debit side, DSPs do present some special challenges of their own. On the simple fixed-point processors, division and floating point operations are often only partially supported. On the ADSP-21xx, signed and unsigned 16-bit divisions are constructed from a sequence of 16 division substep instructions. There are many ways to optimise or distribute these substep operations for short word length arithmetic, and we have seen examples of code in which the substep instructions have been used creatively for purposes not related to division, such as packing the result of a series of threshold operations into a single word. These possibilities make it quite difficult to identify all of the possible assembly language idioms that might represent a valid division. Similar considerations apply to the block floating point instructions on the ADSP-21xx.

A particular difficulty arises from the many processor *modes* that these devices provide. On the ADSP-21xx, the detailed operation of the arithmetic units, the active register bank and the multiplier rounding mode (amongst other things) are specified using control bits in a global status register. As long as this register is updated in a statically visible way, the translator can generate semantically equivalent code in a straightforward fashion. If however, the status register is modified within, say, an `if` statement whose control expression is data-dependent then its value is only run-time, or dynamically, visible. In this case we must either report failure or simulate the behaviour of the processor in our generated ANSI-C code by providing a status bit which is tested during arithmetic evaluation.

We consider several related problems and give some statistics on the frequency of occurrence in real code of these difficulties in section 6 below.

```
.module/ram dump_mod;

.external txhex4;
.var/dm count;      { Declare count variable }
.init count: 5;     { Initialise count to 5 }

.entry dumper;      { Dumper is function }

dumper: i4=0; m4=1; l4=0;
cntr=dm(count);

DO dumploop UNTIL ce;
call txhex4;
dumploop: pm(i4,m4)=ar; {get word}

rts;

.endmod;
```

Figure 1: `asm21` example source code

4 `asm21` language structure

The `asm21` assembler for the ADSP-21xx processors recognises a language that is an unusual hybrid of a conventional line-oriented assembler and a free-format high level language. Common arithmetic instructions, for instance, are represented by algebraic expressions in `asm21` rather than the more usual assembler convention of a mnemonic opcode followed by a list of parameters. Figure 1 shows an example of a self contained code module.

This example defines a single function `dumper` that updates a block of memory with values delivered by an external function `txhex4`. The statement `DO dumploop UNTIL ce;` initialises the zero-overhead loop control hardware on the termination condition `ce` (counter empty). The statement `pm(i4,m4)=ar;` write the contents of the arithmetic result (`ar`) register to the program memory address pointed to by index register `i4` and then increments `i4` by the value of modifier register `m4`.

5 Translation levels

`asm21toc` can produce three different levels of translation. Level 1 translations are designed to be as similar to th original assembler source code as possible, with essentially a one-to-one correspondence between the lines of code. The aim is to allow a programmer who is familiar with the assembler program to see the manner in which `asm21toc` converts symbols and instruction expressions into ANSI-C variables and functions. Some high level analysis is performed: zero-overhead loop constructs are converted into `do-while` statements and indirect accesses *via* the Data Address Generators are converted to pointer dereference operations with optional pointer update. A level 1 translation of the example code is shown in Figure 2.

Level 1 translations also provide extensive diagnostics concerning variable usage, and a simplified text-based control flow diagram which we call a control-flow sketch. The level 1 translator also constructs a hybrid call-graph/basic block graph which is passed to later translator stages.

Level 2 translations extend the control flow analysis by removing explicit `goto` instructions from the code wherever possible. Our algorithm is based on the structural flow analysis of Sharir [Sha80] with extensions to cope with the (possibly ill-formed) call graphs arising from hand-written code. The output of a level 2 translation retains explicit references to condition code bits and machine registers but displays high level control structures, converting counter based loops to ANSI-C `for` loops, for instance. The effect of this conversion on function `dumper` is shown in Figure 3.

Level 3 translation is based on extensive structural data flow analysis. We exploit the information provided by the control flow analysis from level 2 to enable high-level dataflow equations to be solved. This approach, sometimes called *syntactic* dataflow analysis because it effectively exploits the syntactic grouping in high-level control constructs, is in contrast to the more conventional use of simple interval-based dataflow analysis followed by dominator-based control flow analysis. The purpose of the level 3 translation is to remove references to redundant intermediate variables and unused condition codes. The effect is to remove the majority of register references, yielding a program that is mainly composed of expressions containing variables declared using the `.var` directive in the original `asm21` source code. This is particularly important when translating DSP code because the arithmetic units are usually surrounded by pipeline registers with computation being a three stage process: (i) load input pipeline register from memory; (ii) compute function and load result to output pipeline register; (iii) write output pipeline register to memory. The level 3 analysis effectively removes steps (i) and (iii) from the code, leading to significant simplification.

As part of future work we plan to extend the dataflow analysis to the extraction of parameters for functions.

```
/* Translated from module 'dump_mod'
   in file 'dsp2c.dsp'
   by asm21toC V1.01 Mar 05 1999 07:43:05 */

#include "a2c_main.h"

/* Declare variables local to this module */

static long int count;

/* Declare variable initialisation function */

void asm21_data_initialiser_for_dump_mod(void)
{
  count = 5 ;
}

/* Function prototypes */

void dumper(void);
extern void txhex4(void);

/* Function bodies */

void dumper(void)
{
  i4 = 0 ;
  cntr = count;
  do
  {
    txhex4();
    *((long int*) i4++) = ar;  /* get word */
  }
  while (--cntr > 0);
  }

/* End of translation from module 'dump_mod' */
```

Figure 2: `asm21toc` level 1 translation

```
void dumper(void)
{
  i4 = 0 ;
  for (cntr = count; cntr > 0; cntr--)
  {
    txhex4();
    *((long int*) i4++) = ar;  /* get word */
  }
```

Figure 3: `asm21toc` level 1 translation

In the present translator, all registers and variables are represented by global variables, and all functions have the signature void (void). By analysing the usage of variables within functions we can find variables that are used before definition (which are either programming errors or input parameters) and variables that are defined and not used (which are either programming errors or output parameters).

5.1 `asm21toc` implementation

`asm21toc` is written in `rdp`, our compiler generator tool [JS98]. Reverse compilation is not fundamentally different from normal forward compilation in that a source language must be parsed into an intermediate form, and control and dataflow analyses must be performed to support a walk of the intermediate form that outputs the translated code. `rdp` is a general purpose translator generator which provides support for language specification using an attributed extended BNF notation. `rdp` has built-in lexical analysis and symbol table support and a rich library of functions to manipulate sets, symbol tables and generalised graphs. The symbol table handler handles nested scope rules automatically, and we make use of this to maintain independent sub-tables for each module in the application during linking, as described below. A feature of `rdp` is that dynamic data structures (including trees and completely unrestricted graphs) built by `rdp` generated compilers may be rendered as text files suitable for input to Georg Sander's VCG (Visualisation of Compiler Graphs) visualisation tool [San95].

The translator front end parses `asm21` code into a tree based intermediate form. `rdp` can automatically build abstract syntax tree-based intermediate forms using a set of *promotion* operators that convert the full derivation tree into a form that we call a *Reduced Derivation Tree*. An example RDT, visualised using the VCG tool, is shown in Figure 4.

During parsing, information from `asm21` declarations

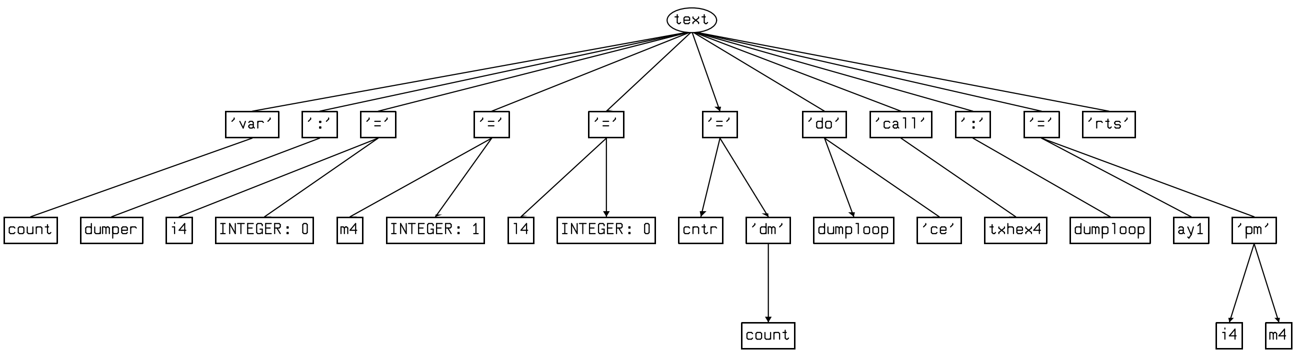

Figure 4: Reduced Derivation Tree for example code

is loaded into the main symbol table. Symbols are updated whenever they are used, so that by the end of the parse a complete map of symbols that are the targets of jump instructions, call instructions (function instantiation) and `do` loop initialisers can be constructed. Usually, DSP applications are written as a series of separately assembled modules which must be linked together, and complete usage information can therefore only be obtained by reading all modules. `asm21toc` therefore must be supplied with the names of all files in the application and, whilst parsing this set of files, performs the functions of the linker in resolving cross references.

The result of the parse, then, is a forest of RDT's, one for each module in the application, along with a symbol table listing all symbols in the application. Subsequent tree-walk phases traverse the forest of RDT's constructing a hybrid control flow graph which represents both call graph style information and traditional basic block control flow. We use this hybrid because assembly language programmers can and do write code which can not be rendered as a set of single entry functions in a straightforward way. It is possible for a label to appear both as a jump target and as a call target, and fatal error handlers in particular often take the form of basic block with one entry and no exits (which we call a 'black-hole' block) that is called from a large number of places.

6 Difficult translations and their frequency

In this section we consider a set of difficult-to-translate structures along with our proposed solutions and some initial statistics on their occurrence in real code. The measurements were obtained by analysing over 600 `asm21` source files amounting to over 300,000 lines of code. The majority of the code deals with image processing and real time image processing functions supplied by one of our industrial collaborators. Also included are some protocol converters and the complete source code from Analog Device's Web site which hosts an extensive library of example files. The image processing code was considered by its authors to contain instances of difficult programming, in particular the use of self modifying code, non-standard control structures and unusual function call conventions. This is in contrast to the example code from Analog Devices which is intended to be straightforward to understand and therefore is much more well behaved. In spite of this emphasis on 'trick' code, we found the number of instances of truly difficult code to be very low.

The total number of machine instructions represented by our sample set was 120,195. The number of assembler directives was 18,376 of which 3,712 were variable declarations. We were surprised to find that only 4.59% of the instructions were multifunction (the ADSP-21xx term for instructions that parallelise an ALU operation and one or two memory transfers). A further 3.58% of the instructions used the conditional execution capability. Multifunction instructions do not allow conditional execution (due to a lack of bits in the instruction word), and so a total of 91.56% of the instructions were straightforward sequential, unconditional operations.

6.1 Processor modes

The ADSP-21xx processors have a set of processor modalities that modify the behaviour of instructions. Modes include the following.

Bit reverse mode which, when set, bitwise reverses the address generated by one of the Data Address Generators. This assists the implementation of the Fast Fourier Transform.

AR saturation mode which sets the output of the ALU to a full scale value in the event of an overflow rather than wrapping the value round as on a conventional processor.

ALU overflow latch which causes the overflow condition code bit to stay set after an overflow rather than being reset on subsequent arithmetic operations. This allows an overflow within a long sequence of operations to be checked for at the end rather than wasting an instruction after each intermediate result.

Multiplier accumulator placement which selects between fractional and integer output representations.

If a processor mode is changed in a non-statically visible manner then we cannot statically write the corresponding ANSI-C code. In this case we must provide a status flag which is checked during execution, effectively simulating the operation of the original processor. In reality, it is most unlikely that a user would wish to preserve this behaviour in the ported application, preferring instead to rewrite the code in a clearer fashion. We found only 280 mode changes in our sample of 120,195 instructions. Of these, less than 40 are potentially non-statically visible, since the usual convention is to set processor modes during system initialisation. The majority of the difficult cases involve the multiplier-accumulator placement mode which is has to be switched to distinguish between integer and non-integer multiplies. We believe that we can use type information gathered from the contexts in which the source operands are later used to infer the correct multiplier mode in most cases, and are able to flag the remaining cases for human intervention.

6.2 Indirect jump and call

Most processors (including DSPs) have a facility to execute a call or jump *via* the contents of a register rather than to an absolute address. Such *indirect* control flow switches have few counterparts in high-level languages: the computed-goto statement in FORTRAN is one example and some compilers use indirect jumps in association with a table of addresses to implement `switch` and `case` statements. Our approach to translating indirect jumps and calls is to use program slicing to detect the range of addresses which could be reached.

Indirect jumps and calls are even rarer than mode switches: we found 79 indirect calls and 77 indirect jumps in our sample. More than half of these instances were in only two modules which contained large scale dispatch tables. Since these instructions are so rare, we do not feel at present that we have fully explored their possible uses. We are actively seeking new examples.

6.3 Self modifying code

DSP programmers occasionally resort to self modifying code in an effort to save critical cycles in an inner loop. One example we have contains two memory buffers held at identical addresses but in different memory spaces. By masking a single bit in a memory reference instruction, the active buffer could be switched between the two address spaces.

Direct translation of self modifying code is not feasible, given the strict separation into static code and dynamic data in most high level programming languages. A more pertinent issue is whether such code can be *detected*. General purpose Von Neumann processors do not lend themselves to such analysis, but the modified-Harvard architecture ADSP-21xx with its separate program and data memories at least allows us to narrow down the possible instances of self modifying code. In detail, the ADSP-21xx does allow data to be held in program space, but the capabilities of the processor are such that normal practice is to store only read-only constants in program memory and place read/write data in data memory. In our sample of 120,195 instructions we found only 16 writes to program memory, about half of which were accounted for by genuinely self-modifying code. This very encouraging result has led us to adopt a strategy of issuing a warning message for every program memory write we encounter and requiring the user to manually check the validity of the access.

7 Related work

The most well known work in this area is that of Cifuentes whose `dcc` tool translates Intel 80286 binaries into C. The tool is well documented in her thesis [Cif94] and in some related papers [CG95]. Although `dcc` is specific to the 80286, some claims are made for the generality of the main decompilation engine, and later papers discuss their extension to cover some aspects of RISC architectures. Cifuentes uses traditional interval-based dataflow analysis followed by structural control flow analysis. `dcc` expends considerable effort on detecting function signatures and can perform some useful type analysis.

Fuan and Zongtang have reported on an 8086 decompiler [FZL93] which is restricted to a particular combination of compiler and memory model. In this tool, much effort is invested in recognising standard C library functions, and hence the restriction to a particular compiler/memory model combination.

A variety of unsophisticated partial decompilers circulate on the Internet mostly, it seems, in response to attempts by games players to investigate the internal workings of commercial programs. These tools are characterised by a lack of control and dataflow analysis and they typically produce output that is similar to our level 1 translation.

Breuer and Bowen investigated the feasibility of a decompiler-decompiler as part of the ReDo project [BB94]. In this work, decompilers are generated from the attribute grammar that provides the compiler specification. The work is theoretically interesting but not directly applicable since attribute grammars for real production quality compilers are not available.

A wider overview of previous academic work in this area is given by Cifuentes in her thesis. There are also a variety of commercial concerns that offer decompilation services and assistance in the porting of applications between architectures. Usually, the techniques employed are not reported in the literature, and we are not aware of any commercial translation services for DSP code.

8 Conclusions

Our translator from `asm21` to ANSI-C is fully operational to Level 1 (naïve translation) and partially operational to Level 2 (control flow analysis) and Level 3 (data flow analysis). Our statistical analysis of over 300,000 lines of `asm21` code indicates that the difficult translation issues we have identified are sufficiently rare that it is appropriate to flag them for the user to review manually. We have found, in general, that the restricted scope of fixed point Digital Signal Processors compared to modern general purpose processors make reverse compilation practical and useful, both for automatic porting and for general program comprehension purposes.

We intend to extend the work to provide better dataflow analysisof parameter usage. We also expect to add a back end that can directly emit sequential assembler code for the TMS320C6x family of devices. We will then be able to directly compare the efficiency of ADSP-21xx programs that have been converted to C and thn compiled down to TMS320C6x code with the efficiency of programs that have been translated directly from assembler.

Acknowledgements

The authors would like to acknowledge the support of the directors of Image Industries Ltd who provided much of the source code used in our statistical sample and Georg Sander, the author of VCG, for permission to include his tool in the distribution package for our `rdp` compiler generator.

References

[BB94] Peter T. Breuer and Jonathan P. Bowen. Decompilation: the enumeration of types and grammars. *Transactions on Programming Languages and Systems*, 16(5):1613–1648, September 1994.

[CG95] C. Cifuentes and K. J. Gough. Decompilation of binary programs. *Software — Practice and Experience*, 25(7):811–829, July 1995.

[Cif94] Cristina Cifuentes. *Reverse compilation techniques.* PhD thesis, Queensland University of Technology, July 1994.

[Dev90] Analog Devices. *ADSP 2101 User's manual (architecture).* 1990.

[Dev98] Analog Devices. *ADSP 2106x SHARC DSP microcomputer family.* 1998.

[Ell85] John R. Ellis. *Bulldog: a compiler for VLIW architectures.* MIT Press, 1985.

[FZL93] C Fuan, L. Zongtian, and L. Li. Design and implementation techniques of the 8086 c decompiling system. *Mini-micro systems*, 14(4):10–18, 1993.

[Ins95] Texas Instruments. *TMS320C8x system level synopsis.* 1995.

[Ins99] Texas Instruments. *TMS320C6000 CPU and instruction reference guide.* 1999.

[JS98] Adrian Johnstone and Elizabeth Scott. `rdp` – an iterator based recursive descent parser generator with tree promotion operators. *SIGPLAN notices*, 33(9), September 1998.

[San95] Georg Sander. *VCG Visualisation of Compiler Graphs.* Universität des Saarlandes, 66041 Saarbrücken, Germany, February 1995.

[Sha80] Micha Sharir. Structural analysis: a new approach to flow analysis in optimising compilers. *Computer Languages*, 5(3/4):141–153, 1980.