Reverse compilation for Digital Signal Processors: a working example

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

We describe the implementation and use of a reverse compiler from Analog Devices 21xx assembler source to ANSI-C (with optional use of the language extensions for the TMS320C6x processors) which has been used to port substantial applications. The main results of this work are that reverse compilation is feasible and that some of the features that make small DSP's hard to compile for actually assist the process of reverse compilation compared to that of a general purpose processor. We present statistics on the occurrence of non-statically visible features of hand-written assembler code and look at the quality of the code generated by an optimising ANSI-C compiler from our reverse compiled source and compare it to code generated from conventionally authored ANSI-C programs.

Keywords: DSP architectures
VLIW architectures
reverse compilation

1 Introduction

This paper describes the design of a reverse compiler which accepts assembler code for a 16-bit Digital Signal Processor (DSP) and outputs ANSI-C with equivalent semantics. The development was motivated by the observation that although most embedded systems based on DSPs have been programmed in assembler, the recent introduction of 'super'-DSPs such as the TMS320C6x which present high levels of instruction level parallelism (ILP) will force a shift to development in a high level language. After approaches from industrial users with a large intellectual property investment locked up in assembler source for the older processors, we applied our existing compiler-compiler tool `rdp` [JS98] and well known control- and data-flow analysis algorithms to the task of constructing a high level language rendering of existing systems for which assembler source is available.

The paper is in three main sections. In Section 2 we describe the trends in the architecture of general-purpose and digital signal processors that indicate that assembler development will no longer be economic for new high-performance DSP systems. In Section 3 we look at the challenges presented by reverse compilation in general. In Section 4 we describe the overall flow in our reverse compiler `asm21toc`, the quality of the results and our current development goals which include assembler-assembler translation as a way of measuring the effectiveness of the high level language compiler's code generator.

2 The retreat from assembly language

Contemporary applications for general purpose processors rarely contain assembly coded routines. The trend over the last twenty years has been towards performing all development in a high level language with the aim of increasing programmer productivity and program portability. The remaining bastions of machine level programming are primarily in embedded systems based on microcontrollers with tightly constrained memory and in Digital Signal Processing where the throughput requirements have continued to justify the costs of developing tightly optimised handcrafted assembler.

2.1 Compilers and humans

High level languages can impose a runtime overhead due to the inefficiencies in translation which arise from the difficulty of mapping high level language semantics onto machine architectures in a way that exploits the special features that assembly language programmers delight in. Traditionally, the runtime cost of high level language development has been justified in terms of programmer productivity but a more fundamental reason for programming at high level is that high performance assembly level programming has become a much more difficult task since the re-introduction of architectures with visible timing constraints.

Many modern general purpose processors feature programmer-visible pipelines and superscalar function units which may contend for register access. Whilst presenting an essentially sequential programming model (in the sense that only a single program counter is used) such architectures blur the boundaries between execution steps and impose constraints in which particular combinations of operations or patterns of data access can give rise to hazards. It has long been known that programmers find programming machines with timing constraints (as opposed to spatial constraints such as limitations in the interconnections between registers and functional units) a special problem. Turing's Pilot ACE computer allowed the user to optimise instruction execution by giving programmer controlled access to the internal sequencing of instruction micro-operations, and Turing expected users to expend considerable effort on scheduling operations using a set of techniques together referred to as 'optimum programming'. Whilst Turing found this hand scheduling natural, other computing pioneers made clear their frustration with the degree of detail that had to be orchestrated to achieve a correct program. In 1954 Christopher Strachey noted as his first design goal for the Ferranti Pegasus computer that optimum programming was to be avoided *'because it tended to become a time-wasting intellectual hobby of programmers'* [Lav80].

2.2 Processors with visible timing

Thirty years after Strachey's pronouncement, low-cost architectures with programmer-visible timing constraints started to appear with the introduction of the MIPS R2000 in 1985 which was based on the Stanford MIPS processor. The R2000 had non-interlocked load delay slots although other more subtle dependencies such as between the multiply/divide unit and the main functional units were hardware interlocked. Even this rather tentative foray into non-interlocked operation was 'corrected' in 1991 with the introduction of the MIPS R4000 which added a hardware interlock on branch delay slots. Most pipelined and superscalar RISC designs do provide interlocks, not so much for the comfort of assembly language programmers as to allow aggressive optimisation by a compiler in the face of non-statically visible dependencies. In the event of the compiler incorrectly predicting execution order, hardware interlocks ensure that high level language program semantics are preserved.

Extracting maximum performance from these processors still requires extensive analysis and careful scheduling of operations in a manner that would be recognised by Turing as a contemporary form of optimum programming. Turing's typically direct views on the merits of scheduling are illustrated by his 1946 comment on the proposed design for the Cambridge EDSAC computer: *'The "code" which he [Wilkes, the designer of EDSAC] suggests is however very contrary to the line of development here and much more in the American tradition of solving one's difficulties by much equipment rather than by thought'*.

Nowadays, by and large, equipment is cheaper (or at least easier to obtain) than thought. In particular we are able to devote considerable amounts of computing effort to the *production* of programs and so architectures with visible timing which are hard to program at machine level can be made usable at a high level given the availability of an effective optimising and scheduling compiler.

The most extreme manifestation of superscalar processing is represented by the class of processors known as *Very Long Instruction Word* (VLIW) machines. VLIW style architectures were developed by Fisher [Fis83] as a target for his 'trace scheduling' algorithm that allowed multiple heterogeneous functional units to be scheduled for parallel execution under the control of a single stream of instructions with almost no hardware interlocking. A classical VLIW comprises a set of functional units (fixed and floating point arithmetic units, multipliers *etc*) and a set of disjoint memory banks interconnected by a crossbar switch under the control of an instruction word which contains sufficient sub-fields to enable all units to operate independently on each cycle. These instructions have to be very wide, hence the name VLIW. The trace scheduling technique was developed from those used to compact horizontally encoded microarchitectures which VLIWs resemble. One view of contemporary superscalar processors is that they are VLIWs whose long instruction is 'packetized' at run time [LB94].

Early research led to the launch of three commercial general purpose architectures from Culler, Multiflow and Cydrome. All three efforts failed in the marketplace, but the underlying compiler optimisation algorithms have fed into the development of compilers for more mainstream superscalar processors. In addition, Intel and Hewlett Packard have developed a new architecture called IA-64 which is heavily influenced by VLIW ideas and which may displace the Intel x86 processors. Texas Instruments are also shipping a processor which is marketed as a VLIW (the TMS320C6x) although it lacks the generality of Fisher's VLIWs.

2.3 Digital Signal Processors

Digital Signal Processors have evolved as a separate class of microprocessor architecture since the introduction of the first commercially successful DSP, the TMS32010, in 1983. DSP applications often feature small loops containing multiply-and-accumulate operations that are executed on large data sets with very high throughput requirements. Examples of such applications include window-based image processing operations, finite impulse response filters for audio processing, 2-dimensional Fast Fourier Transforms for frequency domain image processing and various well known audio and video compression algorithms.

DSPs show quite a wide variety of features, and some DSP-like techniques are also used in general purpose processors so we do not attempt to give a formal distinction between a general purpose processor and a DSP. However, Digital Signal Processors usually include the following features.

1. Multiple memory banks which may supply operands in parallel. A common technique is to use a Harvard architecture in which instructions and data are kept in separate memories, and to allow the program memory to also supply operands in parallel with the data memory.

2. Zero overhead looping, in which hardware counters and comparators maintain the value of loop induction variables. In such a scheme, the hardware is initialised with a loop count and loop start and finish addresses after which loop bodies are executed in parallel with the update-and-test operations on the induction variable. For the small loops found in DSP algorithms, removing the overhead of incrementing the induction variable, testing its value and then conditionally branching can more than double performance.

3. Conditional instruction execution allowing individual instructions to be enabled on the basis of condition flags or a register's contents. These can be used to efficiently code `if` statements with small blocks in the same way that small loops are assisted by the zero overhead looping hardware.

4. True single-cycle arithmetic. Many general purpose processors trade off the n^2 computational requirements for $n \times n$-bit multiplication and barrel shift operations against time by offering cycle multi-cycle functions that run in parallel with the main functional unit. DSP algorithms are heavily dependent on the performance of the multiplier and so a single cycle instruction is required. In addition, most DSPs provide a single instruction that performs multiply-and-accumulate operations.

5. Hardware index registers and associated address update logic which allow a (possibly circular) buffer of values to be stepped through without the overhead of explicit address update instructions.

6. Hardware stacks which keep subroutine link addresses and loop counts on-chip for high speed access.

7. Extended arithmetic modes such as *saturated arithmetic* in which the result of an operation which overflows is set to the maximum value in the representation rather than just wrapping around as is normal for overflow on general purpose processors.

We should also note here that there is a tension between some of these ideas and the needs of a general purpose processor. The zero-overhead looping hardware and the index registers all add state to a running program and would be expensive to maintain in a multi-processing system. The hardware stacks tend to be very small and unsuitable for holding, for instance, stack frames for high level language functions. On the other hand the ARM general purpose processors do provide conditional program execution and a multiply-and-accumulate instruction.

Apart from these technical tricks, the dominant distinguishing characteristic of DSPs is that they provide *predictable* high performance processing. Most embedded systems are online and have to meet hard realtime constraints. Typical high-end general purpose processors display probabilistic behaviour arising from cache effects and the success rate of speculative execution within the pipeline which make them unsuitable for online digital signal processing. DSP vendors provide true cycle-accurate simulators to allow latencies to be accurately measured.

2.4 Super-DSPs

Apart from significant increases in clock rate, the dominant fixed and floating point DSP architectures have been stable for some years. The small address spaces and irregular architectures of the 16-bit processors has made them difficult compiler targets and as a result most serious application development has been in assembler. In recent years, however, three new types of DSP have been launched:

1. The Analog Devices ADSP-2116x 'Hammerhead SHARC' processor integrates two 32-bit floating point datapaths each providing separate multiplier-accumulator, shifter and arithmetic functional blocks. The 48-bit instruction word does not provide enough bits to allow truly independent scheduling of these blocks. Instead, 2116x operates in a Single Instruction, Multiple Data (SIMD) mode whereby the two datapaths operate in lock step on different data elements.

2. The Texas Instruments TMS320C8x device integrates a 32-bit floating point RISC style control processor and up to four fixed point DSP 'parallel processors' which execute independently and communicate *via* shared memory and dedicated interrupt requests.

3. The Texas Instruments TMS320C6x architecture provides both fixed and floating point variants of a VLIW-like processor with twin datapaths in which the eight functional units may be independently programmed using sub-fields within a 256-bit instruction word. The operations available are RISC-like in the sense that they access two blocks of 16 32-bit data registers which communicate with main memory under the control of load and store instructions. Most of the special DSP-like features noted in the previous section, such as zero-overhead looping, are missing but the high level of available parallelism mitigates this. Saturated arithmetic is provided.

Of these three, the Analog Devices design is the most conservative being effectively a doubling up of its previous floating point processor. Assembly language programmers used to Analog Devices' previous processors find it familiar and tractable.

The TMS320C8x was introduced in 1995 but has not been widely taken up, possibly due to its very complex programming model which includes Multiple Instruction, Multiple Data (MIMD) features, multiple caches and an unusual instruction set. Further developments of this device are not expected.

The TMS320C6x has been well received because of its very high potential throughput. The vendor optimistically refers to the 200MHz device as 1,600 MIPS processor because in principle all eight functional units can perform useful work on each clock cycle. Realistic benchmarks indicate the the at a given clock rate, the TMS320C6x provides about twice the throughput of the ADSP-2116x [BDT99][1]. A future processor from Analog Devices, the TigerSHARC will use VLIW-like techniques to close the performance gap. These developments mark the convergence of the main DSP architectures and general purpose processors (in the form of the IA-64) on VLIW-style machines. It is very hard to produce an efficient schedule for any of these machines by hand.

3 Approaches to reverse compilation

Reverse compilation has been attempted by many and used productively by few. The usual motivations for writing a reverse compiler are:

1. the recovery of intellectual property from 'dusty decks': programs written in obsolete languages,

2. the recovery of intellectual property from binaries for which the source code has been lost, or

3. the acquisition of intellectual property by reverse-engineering binaries for which the source code is not owned by the user.

This last is the most problematic: license agreements for software now typically include clauses forbidding disassembly or reverse compilation of code.

We doubt the usefulness of reverse compilation for either of the last two cases. The only reason for analysing a program in such detail that an equivalent high level program can be produced is surely to aid software maintenance. Programs that have been reverse compiled from binaries that have had debugging information stripped out have to rely on automatically generated variable names and, of course, no comments will survive. Such programs would be hard to read and maintain.

Our reverse compiler `asm21toc` falls into the first class of tools: those that move from an obsolete language to a new one. We have the twin advantages that the users of our tool are intimately familiar with the details of the obsolete language and that we have full, commented source code available for the entire system. A particular feature of `asm21toc` is that it provides translations at four different levels of analysis, from a naïve translation that is very close to the original assembler source in style up to a translation with type information and function parameters that may differ

[1] We provide our static analysis of the processor utilisation on typical TMS320C6x in section 4.2 below.

in style from those intended by the original programmer. It is easy to relate the naïve translation to the original code, and so assembler programmers who are converting their own code can follow the later steps of the translation process (which may involve significant reorganisation of their code) without difficulty.

Before considering the technical details and results of our approach, we look briefly at the spectrum of approaches to reverse compilation.

A typical high performance compiler performs the following steps during translation of a high level language program into scheduled code for a superscalar processor (see for instance the description of DEC's GEM compiling environment [Bli92]).

1. Parse the source into some intermediate form, discarding syntactic sugar and building rudimentary information concerning the semantics of the program, in particular constructing symbol tables that represent the program variables.

2. Construct function call graphs and basic block graphs that represent the possible execution pathways through the program. Rearrange basic block graphs into structures that map naturally onto the target machine's control flow instructions.

3. Use dataflow analysis techniques to analyse the lifetime and usefulness of variables.

4. Apply standard sequential optimisations such as loop invariant removal and strength reduction.

5. Traverse the intermediate form generating architecture specific code.

6. Apply scheduling algorithms to pack the code down onto the superscalar architecture in an efficient manner.

Interestingly, this series of steps (with the exception of the last, scheduling step which is replaced by a type analysis step) is exactly the sequence followed by `asm21toc`. However, it would be a mistake to deduce that a reverse compiler is simply a lightly modified compiler. The essential difference is that reverse compilation requires a series of *partitioning* decisions to be made. During compilation from a high level to a low level language we move from a syntactically and semantically rich domain to a domain in which all information is represented as bit strings and in which only a few simple operations and datatypes are meaningful. A reverse compiler has to synthesise a high level representation by a series of *partitioning* during the analyses shown in Figure 1.

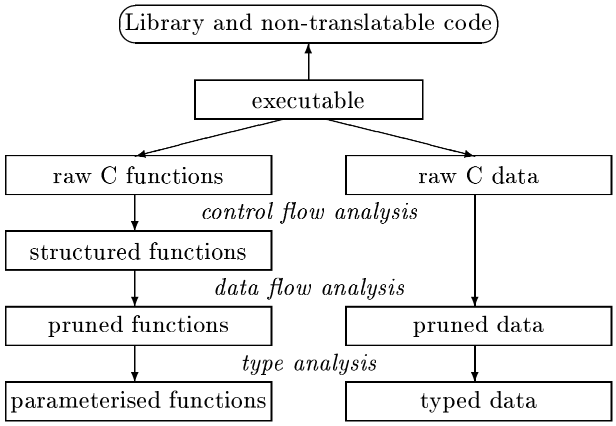

Figure 1. Analysis stages in a reverse compiler

The first task for a reverse compiler is to separate code from data. For a reverse compiler which starts with an executable binary for a Von Neumann computer in which instructions and data share the same memory we can imagine some very extreme situations in which no such separation exists. Consider an executable which is entirely composed of only two kinds of instruction A and B. Instruction A causes an unconditional jump to some other location within the valid address space of the executable. Instruction B overwrites an instruction, randomly selecting an instruction of type A with a new random address or an instruction of type B. The effect of such a program is to continually rewrite itself, all the while jumping around in a random control flow pattern. Over a large number of runs, every location in the executable will be repeatedly overwritten (treated as data) and executed (treated as an instruction).

Fortunately, this kind of pathological behaviour is implausible. Real programs display clear distinctions between code and data and may be statically analysed to find the possible control flow paths through the instructions. The usual heuristic (as exemplified by the highly effective *vest* and *mx* binary translators from VAX and MIPS to Alpha executables [Sit92]) is to start from all of the known entry points to an executable and then statically follow the control flow paths through the executable. Any locations that appear on those control flow paths are instructions and everything else is assumed to be data.

This static heuristic may fail if the executable contains self-modifying code, in which case the full execution path for the executable may only be obtained by actually executing all paths, that is by performing a full dynamic analysis. This is, of course, impractical even for small programs as the designers of viruses know only too well.

Even after separation into instructions and data, some further partitioning may be required to distinguish between library functions and translatable functions. When an executable has been produced by a high level language compiler, as opposed to being written in assembler, there will usually be large amounts of code that have been linked in corresponding to standard library routines, startup and wrap-up code and so on. The user usually does not wish to see this material in the reverse compiled source, and some reverse compilers go to considerable lengths to identify this library code and suppress it from the final translation. The problem with this kind of process is that small changes to the compiler or libraries require changes to the reverse compiler.

Our reverse compiler starts from assembler source and therefore is not faced with the instruction/data separation problem, except for the case of self-modifying code. The ADSP-21xx processor keeps code and data in separate memory spaces although it does allow program space to be written to during program execution so that variables can be kept in program space as well as in data space. It turns out that program memory writes are very rare in real ADSP-21xx programs and so we can at least pinpoint possible instances of self-modifying code with some accuracy. We present statistics on this and some other difficult-to-translate idioms in section 4.1.

In the next section we shall the illustrate the partitioning challenges in later stages of reverse compilation.

4 The `asm21toc` reverse compiler

`asm21toc` is written using our compiler-compiler tool `rdp`. The front end parses ADSP-21xx assembler source and constructs a tree-based intermediate form which combines a parse-tree like representation of the program with a combined call graph and basic block flow graph.

The example C program in Figure 2 scans an array of integers to find the largest element. A hand written version of the program in ADSP-21xx assembler is shown in Figure 3 and the a fragment of the output from the `asm21toc` front end is shown in Figure 4.

The ADSP assembly language is rather unconventional, being a mix of C-like arithmetic expressions and assembler-like labels and directives. Our parser builds a derivation tree that is then pruned using `rdp`'s node promotion operators to yield a *reduced derivation tree* (RDT). The parser also creates a symbol table for each

```
#include <stdio.h>
#define BUFFER_MAX 3

int elements[BUFFER_MAX] = {1,45, 34};

/* return maximum element */
int max(int left, int right)
{
  if (left < right)
    return right;
  else return left;
}

int largest (int *buffer)
{
  int result = 0, temp;

  /* scan buffer */
  for (temp = 0; temp < BUFFER_MAX; temp++)
    result = max(result, buffer[temp]);

  return result;
}
```

Figure 2. Largest element: hand written C

```
.module example;

.const BUFFER_MAX=3;
.var/dm elements[BUFFER_MAX];
.init elements: 1,45,34;
.entry largest;

max:
    ar = ax0 - ay0;
    if lt ar = pass ax0;
    if ge ar = pass ay0;
    rts;

largest:
    ay0 = 0;
    cntr = BUFFER_MAX;
    i0 = ^elements;
    l0 = 0;
    m0 = 1;
    do check until ce;
    ax0 = dm(i0,m0);
    call max;
check: ay0 = ar;
    ar = pass ay0;  { a redundant move }
    rts;

.endmod;
```

Figure 3. Largest element: hand written assembler

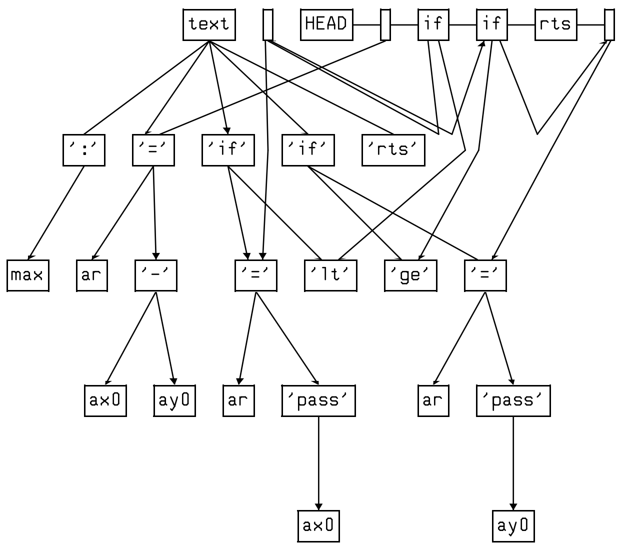

Figure 4. Output from front end for function `max`

module of assembler source. Modules establish scope regions within which `.global`, `.external` and `.entry` directives are used to control external visibility. After all modules have been parsed, the symbol tables are walked to reconcile these directives, effectively simulating the behaviour of a linker.

The RDT is then traversed twice and control flow graph nodes are added. Figure 4 shows the resulting data structure for the `max()` function: the node labeled `text` is the root of the RDT and the node labeled `HEAD` is the root of the control flow graph.

Our control flow graphs fulfill the rôles of both a conventional function-level call graph and a conventional basic block graph. At this stage of the analysis, function boundaries have not yet been established, and so separation into call graph and local basic block graphs is not possible. In fact our control and data flow algorithms are based on this unified data structure, so no subsequent separation is performed.

Identification of functions follows. Any node that is the target of a call instruction will appear as the start of a function in the final translation, but it is possible for that node to be used in other contexts too. The semantics of functions in C (and most other high level languages) are quite limited: functions may only have one entry point and inter-function jumps are not allowed. (The `setjmp/longjmp` mechanism in ANSI-C does support inter-functional jumps, but jump targets must already have been visited at least once during program execution before the jump is taken, so it is not fully general.) Function-level control flow analysis is performed at this stage, and sections of code that are used in more than one context, either as a result of inter-functional jumps or as a result of multi-entry functions are cloned.

At this stage, the naïve translation can be output. At this level, all ADSP-21xx registers are declared as global variables along with any user variables. User constants are mapped to `#define` preprocessor commands. User arrays are initialised *via* an initialiser function. There is an approximately one-to-one correspondence between instructions in the assembler source and lines of code in the naïve translation. This correspondence is very helpful to users since it easy to see how `asm21toc` has performed its translations.

The representation of control structures in the naïve translation is primitive. The existence of `do` and simple `if` instructions in the assembler source enables us to represent these directly in the ANSI-C output, but jumps are only shown as `goto`'s. In addition, even loops using the built-in zero-overhead looping instructions are shown as `do-while` loops in the ANSI-C rather than as `for` loops, which would be a more natural translation.

We apply a control flow analysis stage to extract structured control structures such as `switch`, `if-then-else` and `for` from the raw control flow graph. The result of this phase is a hierarchical graph which represents the nesting of basic blocks within each function. We use the structural control flow analysis algorithm of Sharir [Sha80] with extensions to uncover well formed single-entry functions. A translation output at this stage shows much cleaner control structures but is still based on global variables which explicitly represent registers in the ADSP-21xx processor, as shown in Figure 6.

ADSP-21xx code usually contains a large number of transfers between memory and registers. This is because each functional unit in the ADSP-21xx processor is surrounded by input and output pipeline latches which must be explicitly loaded from data or program memory before an operation can be performed. Hence an operation like $a = b + c$ might be written in ADSP-21xx assembler as

```
ax0 = dm(b);  { Load left source with b }
ay0 = dm(c)   { Load left source with c }
ar = ax0 + ay0;
dm(a) = ar;   { Store ALU result register }
```

In the naïve translation, all of these operations are preserved in the ANSI-C output. In a sense, the underlying architecture of the processor is showing through into the translation. A related issue is that of condition codes: within the ADSP-21xx processor a set

```
/* Level 1 translation from module 'example' in file
  'examp.dsp' by asm21toC V1.02 May 31 1999 11:55:47 */
#include "a2c_main.h"

/* Define manifest constants */
#define buffer_max 3

/* Declare variables local to this module */
long int elements[buffer_max];

/* Declare variable initialisation function */
void asm21_data_initialiser_for_example(void)
{
  elements[0] = 1;
  elements[1] = 45;
  elements[2] = 34;
}

/* Function prototypes */
void largest(void);
void max(void);

/* Function bodies */
void max(void)
{
  ar = ax0 - ay0;
  if (an) ar = ax0;
  if (!an) ar = ay0;
  return;
}

void largest(void)
{
  ay0 = 0;
  cntr = buffer_max;
  i0 = (long int) & elements[0];
  l0 = 0;
  m0 = 1;

  cntr_1 = cntr;
  do   /* check */
  {
    ax0 = *((long int*) i0); DAG_UPDATE(i0, m0);
    max();
    cntr_1--;
    ay0 = ar;
  }
  while(!(cntr_1 <= 0));

  ar = ay0;  /* a redundant move */
  return;
}
/* End of translation from module 'example' */
```

Figure 5. Largest element: naïve translation

```
void max(void)
{
  ar = ax0 - ay0;
  if (an)
    ar = ax0;
  else
    ar =   ay0;
  return;
}
```

Figure 6. Max function after control flow analysis

```
void max(void)
{
  if ((ax0 - ay0) < 0 )
    return ax0;
  else
    return ay0;
}
```

Figure 7. Max function after data flow analysis

of *condition code* bits is modified during each arithmetic operation to indicate whether the last was result was zero, or overflowed and so on in the conventional manner. These bits may then be sampled by the conditional execution unit in the processor to decide whether to execute an individual instruction. The vast majority of values written to the condition codes register are never used, and since there are up to five such writes for each arithmetic operation, we suppress them from the naïve translation because they obscure the underlying behaviour of the program. However, to produce a semantically correct translation we must retain any condition code updates that are used by subsequent conditional instructions.

These redundant condition codes and register accesses are detected in the dataflow analysis phase. We use a structural dataflow analysis framework which exploits the structural control flow graph derived in the previous step to make analysis more efficient. After the analysis we apply some standard sequential optimisations such as constant propagation to further compact the code. The results of this phase are shown in Figure 7.

ANSI-C code produced at this stage has had most references to ADSP-21xx registers removed, but is still composed of parameterless functions with all variables represented as globals. The dataflow analysis can be used to parameterise functions by finding the live ranges of variables. Assuming that redundant variable accesses have already been removed from the global graph, then any variables that are read in a function before they are written must be input parameters and any variables that are written and read outside of the function at later points in the control flow must be output parameters. Other variables used within a function are local. It is our aim to combine this analysis with a type analysis that attempts to partition the set of variables into related subsets of variables that appear in expressions together. In this way we hope to capture both parameterisation and references to structured variables.

4.1 Non-statically visible translations

In a previous paper [JSW99] we have reported on the frequency of non-statically visible translations amongst a large sample of ADSP-21xx assembler source. The sample was made up of code from various publicly available sources and from a large suite of image processing algorithms supplied by one of our industrial partners. The previous results are summarised in Table 1 along with new statistics for a large telecommunications application. The sizes of the two samples are indicated by the total number of instructions in column one. The ADSP-21xx provides a set of *multifunction* instructions which parallelise an arithmetic operation with one or two memory-register transfers. `asm21toc` sequentialises these instructions, inserting new temporary variables where there are co-dependencies within the instruction. Although multifunction instructions occur rarely (just over 1% of the telecoms application, 4.6% of the main sample) they are important in inner loops and can dominate dynamic instruction frequency counts. More than 10% of the instructions in our telecoms application use the *conditional* capability. The remaining columns in Table 1 refer to classes of instructions that are difficult for `asm21toc` to translate because they indicate possible non-statically visible control paths.

As noted above, writes to program memory may indicate that self-modifying code is in use, or may simply be write accesses to variables held in program space. ADSP-21xx programmers rarely use variables (as opposed to read-only constants) from program space as the contention between instruction fetch and variable write introduces a performance penalty. Even in our large sample we found only 16 instances, and of those about half were variable accesses. The remaining ones relate to a particularly tricky piece of code which saves machine cycles by directly twiddling an instruction bit rather than conditionally selecting between two buffer addresses. Even this situation could in principle be detected by `asm21toc` since in this case the program addresses being manipulated were statically visible. However, we believe that self-modifying code is so rare in practice that we propose to merely flag each instance of a write to program space and ask the user to review the code manually. As may be seen from the table, our telecoms application does not contain any such references.

Indirect calls and *indirect jumps* refer to procedure call and jump instructions where the destination is held in a register. Since the contents of registers are not in general statically visible, it can become impossible to perform control and data flow analysis around such points. In practice, the main use of this facility is to implement switch statements and function dispatch tables in which the range of values that may appear in the register is statically visible. This is the case for the majority of the small number of indirect control transfers noted in Table 1. Again, we flag these occurrences and ask the user to review them manually.

The ADSP-21xx processor supports a number of arithmetic modes which may be changed by modifying global control bits. Available modes include saturation arithmetic; sticky overflow (in which the overflow condition code bit stays set until explicitly reset); bit reverse mode which reflects the address bits from one of the indexing units (which is useful when implementing the Fast Fourier Transform); and MAC placement which selects between fractional and integer output formats from the multiplier-accumulator. The translation of arithmetic operations varies with these modes, and non-statically visible changes to the bits therefore cause translation problems. The majority of mode changes are statically visible, since they tend to occur during initialisation. When `asm21toc` encounters a non-statically visible mode change it inserts code that simulates the behaviour of the real processor by creating a global status bit in the translated ANSI-C program and selecting between alternatives at run-time. This is very undesirable because of the large run-time overhead, so we expect that user will wish to restructure their assembler code to eliminate such mode changes. They are thankfully rare.

4.2 Interaction with the TMS320C6x software development environment

A critical issue in the design of our translator is how well the generated ANSI-C code interacts with the native TMS320C6x development tools. At present we have performed static analysis of the degree of parallelisation achieved by the C compiler on hand written code and code generated by `asm21toc`. The results are summarised in Table 2 which shows the code produced

	total instructions	multifunction instructions	conditional instructions	program writes	indirect calls	indirect jumps	mode changes
PD&IP	120195	5517	4303	16	79	77	280
Telecoms	2784	33	289	0	1	0	1

Table 1. Characteristics of hand written assembler code

by the compiler alone and and Table 3 which shows the effect of running the code through Texas Instrument's assembler level scheduler.

We used four applications in the test.

`asm21toc` is the source code for the reverse compiler itself. This is perhaps an odd choice since it is most unlikely that an embedded processor would ever run a code development tool. However, `asm21toc` is interesting because it makes very heavy use of dynamic memory and very little use of static arrays. In his thesis on the first well-documented VLIW compiler [Ell85] John Ellis expresses doubts concerning the effectiveness of VLIW architectures at dealing with such unstructured programs, although later experiments with the Multiflow processor found large degrees of parallelism in the Unix kernel. Our results bear this out, showing the parallelisation performance whilst not being as good as for DSP-like code still leaves only 52% of the code in purely sequential form.

`qm min` is a Quine-McCluskey boolean equation function minimiser. It makes heavy use of arrays, but also performs a lot of bit-level masking. This program shows the greatest proportion of purely sequential code after optimisation, but the compiler did manage some six-way parallelism in an inner loop.

`image processing` is a set of 3×3 window operators which perform Sobel edge enhancement, mean filter noise removal and histogram collection within a 256×256 buffer of integer pixels. Several versions of the filters were written and the performance of the compiler was found to be very sensitive to the details of expression layout in the source code. The compiler performed extremely well on this code, leaving only 21% of the instructions in sequential form and even finding some eight-way parallelism.

`telecoms` is the telecoms suite introduced in the last section after translation into ANSI-C by `asm21toc`. It is clear that the compiler performs relatively well on the kind of code produced by `asm21toc`. We have not been able to perform a dynamic analysis, but on the strength of this static analysis and other available benchmarks we believe that a 200MHz TMS320C6x will run this translated application around six times faster than a 75MHz ADSP-21xx running the original hand written assembler code.

5 Related work

The most well developed work in this area is the `dcc` tool described by Cifuentes in her thesis [Cif94] and a related paper [CG95]. `dcc` translates from Intel 80286 binaries compiled for MS-DOS back to C. Cifuentes makes some claims for generality in `dcc` and we looked at using the core of the tool for our reverse compiler but found that the intermediate representation was very Intel specific. Cifuentes has looked at the issues involved in extending `dcc` to handle more RISC oriented instruction sets.

Another Intel specific compiler [FZL93] is restricted to a particular combination of compiler and memory model because it specialises in recognising standard C library functions and suppressing them from the translated output.

Two projects have taken a more theoretically well found approach to decompilation. Breuer and Bowen [BB94] showed that a decompiler can be constructed clerically under some circumstances given the availability of an attribute grammar describing the compiler. This is interesting but not practically helpful because attribute grammars specifying real compilers and their associated optimisers and schedulers are not available. A recent paper by Alan Mycroft [Myc99] describes a fascinating application of type inference algorithms to the reconstruction of C code from register transfer language level descriptions of programs.

6 Conclusions

We have described our approach to reverse compilation and motivated it with a description of trends in the embedded DSP market. Preliminary results show that users of our `asm21toc` tool will be able to realise significant speed-ups of their code if they migrate to one of the new processors by translating to ANSI-C. We

application	instruction count	percentage instructions in block size							
		1	2	3	4	5	6	7	8
asm21toc	38821	64.46	11.65	3.35	0.51	0.01	0.01	0.00	0.00
qm min	3830	75.46	10.23	0.97	0.26	0.03	0.00	0.00	0.00
image processing	1965	60.36	8.70	2.90	1.37	1.12	0.41	0.00	0.00
telecoms	4436	70.33	14.83	0.00	0.00	0.00	0.00	0.00	0.00

Table 2. Parallelisation by TMS320C6x ANSI-C compiler: minimal optimisation

application	instruction count	percentage instructions in block size							
		1	2	3	4	5	6	7	8
asm21toc	36318	52.28	13.40	4.68	1.45	0.17	0.04	0.00	0.00
qm min	3329	59.33	14.39	2.61	0.66	0.21	0.06	0.00	0.00
image processing	1629	20.81	9.09	5.89	4.05	2.58	1.66	0.18	0.37
telecoms	5904	46.21	16.87	4.69	1.13	0.22	0.05	0.00	0.00

Table 3. Parallelisation by TMS320C6x ANSI-C compiler: full optimisation

have found that the restrictions of Digital Signal Processors in terms of depths of stacks, separation of code and data spaces, explicit declaration of vectors and the availability of high level flow control instructions assist the initial stages of our translation.

At present we are investigating type inference techniques by which we can construct structured data types from hand written assembler. We are also experimenting with direct assembler-assembler translation by moving directly to TMS320C6x code rather than translating to C first. Preliminary results indicate that the code produced *via* the C route is better because the Texas Instruments C compiler is able to perform some analyses and leave hints for the assembler level optimiser.

Acknowledgements

We would like to thank the directors of Image Industries Ltd who provided much of the source code used to generate the statistics for ADSP-21xx assembler source reported in section 4.1 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.

[BDT99] *Buyer's guide to DSP processors.* Berkley Design Technology Inc, 1999.

[Bli92] David S Blickstein. The gem optimizing compiler system. *Digital Technical Journal*, 4(4):121–136, 1992.

[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.

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

[Fis83] J.A. Fisher. Very long instruction word architectures and the ELI-512. In *Proc. Tenth Annual Internat. Symbp on Computer Architecture*, pages 140–150, june 1983.

[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.

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

[JSW99] Adrian Johnstone, Elizabeth Scott, and Tim Womack. Reverse compilation of Digital Signal Processor assembler source to ANSI-C. In

Proc Internat. Conference on Software Maintenance. IEEE, 1999.

[Lav80] Simon Lavington. *Early British Computers*. Digital Press, Bedford Ma., 1980.

[LB94] David J Lilja and Peter L Bird, editors. *The interaction of compilation technology and computer architecture*. Kluwer Academic Publishers, 1994.

[Myc99] Alan Mycroft. Type-based decompilation. In S Doaitse Swierstra, editor, *Proc. 8th European Symposium on programming (ESOP'99), Lecture notes in Computer Science 1383*, pages 208–223, Berlin, 1999. Springer.

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

[Sit92] Richard L Sites. Binary translation. *Digital Technical Journal*, 4(4):137–152, 1992.