DISSERTATION

Reverse Compilation Techniques for VLIW Architectures

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines
Doktors der technischen Wissenschaften
unter der Leitung von

ao.Univ.Prof. Dipl.-Ing. Dr. Andreas Krall

E185
Institut für Computersprachen
Technische Universität Wien

eingereicht an der Technischen Universität Wien

Fakultät für Informatik

von

Mag. Nerina Bermudo Pla
0427348
Bräuhausgasse 48/37
1050 Wien

Wien, am 25. Oktober 2005

Abstract

Reverse compilation (or decompilation) is the translation of machine or assembly language into a high-level machine independent language. Decompilation can be used to recover lost or inaccessible source code, to translate code originally written in an obsolete language into a new language or to migrate applications to a new hardware platform.

A digital signal processor (DSP) is a microprocessor designed specifically for the efficient implementation of digital signal processing algorithms. Aiming for high-performance newer DSP processors use design approaches typical of super computer architectures. Very long instruction word (VLIW) architectures consist of multiple execution units, allowing the parallel execution of instructions. Pipelines divide the execution process in several steps in such a way that an instruction can be dispatched before the previous one has finished execution.

Optimizing compilers for modern DSP architectures allow the efficient execution of applications written in high-level languages. Since most available software for older architectures is coded in assembly language, a reverse compiler that translates assembly language code into a high-level language is exceptionally useful.

For both understanding the software and for reverse compiling it to a high-level language, a control flow graph (CFG) is required. However, CFG construction is complicated by architectural features of pipelined VLIW architectures, which include predicated instructions, VLIW parallelism, instructions (in particular branches) with delay slots. This work introduces a new approach for the construction of a CFG, where the parallelism has been eliminated, instruction reordering has been reversed and delay slots have been removed.

Software pipelining is a widely implemented optimization technique used to speed up loop execution in processors that support instruction level parallelism. Software pipelining basically overlaps several loop iterations so that they can be executed concurrently. This causes an increase in the CFG size. Understanding the code becomes very difficult. Software de-pipelining is the reverse of software pipelining. It restores the assembly code of a softwares pipelined loop back to its semantically equivalent sequential form. This thesis presents a de-pipelining algorithm for single loops.

In the course of this thesis a retargetable decompiler framework has been developed which includes entirely new techniques for CFG construction and software de-pipelining. They have been tested on several digital signal processing applications for the TIC62x DSP.

Kurzfassung

Reverse compilation (Rückübersetzung) ist die Übersetzung von Programmen in Maschinen- oder Assemblerspache in eine architekturunabhängige höhere Programmiersprache. Rückübersetzung wird verwendet, um verlorenen oder unzugänglichen Quelltext, der ursprünglich in einer technisch überholten Sprache geschrieben wurde, in eine neue Sprache zu übersetzen oder um Anwendungen zu einer neuen Architektur zu migrieren.

Ein digitaler Signalprozessor (DSP) ist ein Mikroprozessor, der speziell für die effiziente Ausführung von Signalverarbeitungsalgorithmen entwickelt wurde. Neuere DSP verwenden Designlösungen, die typisch für Hochleistungrechner sind: VLIW (*very long instruction word*) Architekturen enthalten mehrere unabhängige Funktionseinheiten und ermöglichen so die parallele Ausführung mehrere Befehle. Pipelines zerlegen die Ausführung von Befehlen in mehrere Teile, so dass der nachfolgende Befehl geladen werden kann, bevor die Abarbeitung des vorherigen Befehls abgeschlossen ist.

Optimierende Übersetzer für moderne DSP Architekturen ermöglichen die effiziente Ausführung von Anwendungen, die in einer höheren Programmiersprache entwickelt wurden. Da ein Großteil der Software für ältere Architekturen in Assemblersprache entwickelt wurde, ist ein Rückübersetzer, der Assemblersprache in eine höhere Programmiersprache übersetzt, sehr wertvoll.

Ein Kontrollflussgraph (CFG) ist notwendig, sowohl um das Programm zu verstehen, als auch für die Rückübersetzung in eine höhere Programmiersprache. Die Konstruktion des CFG wird komplizierter durch Architektureigenschaften, wie bedingt ausgeführte Befehle, VLIW Parallelismus und verzögert ausgeführte Befehle (insbesonders Sprünge). Diese Arbeit stellt eine neue Methode vor, um einen Kontrollflussgraphen zu erstellen, in dem sowohl Parallelismus, als auch die Auswirkungen von verzögert ausgeführten Befehlen entfernt worden sind.

Software pipelining ist eine weit verbreitete Optimierung, um die Abarbeitung von Schleifen bei Prozessoren, die Parallelismus unterstützen, zu beschleunigen. Beim *software pipelining* werden mehrere Iterationen einer Schleife überlappend ausgeführt. Der Kontrolflussgraph ist dann grösser und unübersichtlicher, wobei der generierte Programmcode schwieriger zu verstehen wird. *Software de-pipelining* ist die Umgekehrung der Optimierung, dessen Ziel die Wiederherstellung der sequenziellen Form der Schleife ist, ohne die Sematik des Programmes zu verändern. Diese Arbeit stellt ein Algorithmus für *software de-pipelining* von Schleifen vor.

Im Verlauf dieser Doktorarbeit wurde ein Rückübersetzer entwickelt, der diese neuen Techniken enthält. Mehrere Signalverarbeitungsanwendungen für den TIC62x DSP wurden mit diesem Rückübersetzer evaluiert.

Als meus pares, per tot.

Für dich und die Tiere.

Acknowledgements

I would like to thank several people that have directly or indirectly contributed to this work. Andreas Krall, my supervisor, for his support and dedication. Bogong Su, for the long discussions and his help with software de-pipelining. Everybody at the computer languages department and, in particular, to my colleagues at the CD lab, Gerhart Kobinger, Ivan Prianichnikov, Viera Sipkova, Ulrich Hirnschrott and Christian Thalinger.

I also would like to thank the Kristen family and Harald for helping me in a million ways and making me feel part of their family. Thanks to the Bermudo family for their calls and support and to Piero, for remembering me every time I went to Barcelona. I want to thank my friends that were far away, in particular Jaume for having read this thesis.

Finally, very special thanks to Oliver, Barkley Kingwood, We de Boer, Enju Mabura and Minerva Fernández. Thank you all.

Contents

List of Figures

List of Tables

Chapter 1

Decompilation Techniques

Decompilation is a program transformation which takes machine or assembly language code as input and generates a semantically equivalent high-level language code.

Program compilation translates a high-level program into machine code for a chosen architecture. Due to the major differences between the original and the target language, and considering other restrictions such as performance and code size, several analyses and optimizations must take place to allow a correct and efficient translation. Compiler techniques have been studied by the computer science community for many years and still represent an important research field.

Decompilation is not so different from compilation as it performs the translation of a program from one language to another. Thus, the main structure of a decompiler does not differ significantly from a compiler and many analyses and optimizations are common to both processes. However, the translation from assembly language language into a high-level language opens new issues to take care of.

This chapter introduces the subject of decompilation describing its goals and problems. Section 1.2 presents the main research results in the history of decompilation

1.1 Decompilers

A decompiler is a program that takes machine code as input and translates it into semantically equivalent high-level code. In this sense, a decompiler reverses the compilation process. However, in the compilation process information about the program is lost, such as data structures and function boundaries. Therefore, the answer to the question whether it is possible to decompile a program is not obvious.

Unfortunately, the decompilation problem is theoretically equivalent to the Halting Problem [Dav58]. In practice this means that fully automated decompilation of arbitrary machine-code programs is not possible. Thus, automatic decompilation without user intervention cannot be achieved for all programs.

The question about what proportion of real-world programs can be decompiled to useful source code is still open at this point.

This section introduces the main difficulties of decompilation and describes some possible usages of such a tool.

1.1.1 Decompilation Problems

The following issues represent important complications for decompilers.

Separation of data and code

In machine code, data and program instructions are represented in the same way. This makes it very difficult to distinguish between them. Some decompilers do not take machine code as input, but code written in assembly language, where this problem does not occur. In this work, assembly language language is considered.

Self modifying code

Self modifying code refers to instructions that are modified during the execution of the program. This technique has been used mainly for reducing the code size. Nowadays, since memory is no longer a major limitation, self modifying code is not often used. However, it is used in just-in-time compilers, security applications and virus code.

Types and data structures

In machine or assembly language code types of variables are not specified. Data structures are not explicitly defined. For decompilation it is necessary to recover the type information and to be able to identify variables.

Indirect Jumps

Indirect jumps represent an important obstacle for decompilation. A branch that jumps to an address contained in a register has no explicit target. Complex memory analyses are needed that determine the possible jump-to addresses.

Complex architectures

The evolution in the computer architecture has led to more complex features like parallelism and pipelining. Decompilers need to face the new problems emerging from these new architectures.

1.1.2 Motivation

Decompilers have been written for different purposes since the development of the first compilers. Throughout the last decades, different uses have been given to decompilers. In the 1960s, decompilers were used to aid in the program conversion process from second to third generation computers. In this way, manpower would not be spent in the time consuming task of rewriting programs for the third generation machines. During the 70s and 80s, decompilers were used for translating

code originally written in an obsolete language into a new language, migrating applications to new hardware platforms, documentation, debugging, recovery of lost or inaccessible code, and software maintenance.

In the last years, decompilers have become a reverse engineering tool capable of helping the user with such tasks as

- recovery of lost source code,
- migration of assembly language applications to a new hardware platform,
- translation of code written in obsolete languages no longer supported by compiler tools,
- checking that a compiler generates the right code,
- detection of the existence of viruses or malicious code in the program, and
- understanding of the implementation of a particular library function.

Not all uses of decompilers are legal uses, since computer programs are protected by copyright law. Different countries have different exceptions to the copyright owner's rights so that different uses are allowed by law. However, it is not in the scope of this thesis to discuss the legal and ethical aspects of decompilation.

1.2 History of Decompilation

The first decompilers were developed in the 1960s as a help to programmers in the conversion process of programs from second to third generation computers. Porting these programs automatically to third generation computers would reduce the cost of doing so. Porting programs, together with recovery of lost codes, maintainance and modification of existing binaries and debugging were the main applications of decompilation until the 1990s. Then, decompilers became an interesting reverse engineering tool to help the user check whether the compiler generates the correct code, whether the program contains illegal code such as virus and trojan horses and to translate binary programs from one architecture to another.

The use of decompilers for software piracy is, at least at the moment, not really practicable, since the decompilation problem is not solvable in general. Anyway, we are not going to discuss the legal aspects of decompilation in this work.

Next, the most relevant results in the field of the decompilation since its origin will be presented. A complete history of decompilation up to these days can be found at [Tho].

- **1960 - D-Neliac decompiler** The Donnelly-Neliac Decompiler [Hal62] was developed at the Navy Electronics Laboratory (NEL) in 1960. Neliac is an Algol-type language developed at the NEL in 1955. The D-Neliac decompiler produced Neliac code from machine code programs. This decompiler proved that decompilation is possible. It was also useful for detecting logic errors in the original high-level programs.

- **1966 - W.Sassaman** Sassaman developed a decompiler to aid the conversion of programs from second to third generation computers [Sas66]. This decompiler translated assembler programs for the IBM 7000 series into Fortran. It was the first decompiler to use symbolic assembler instead of binary code.

  With help from the user who, for example, was required to define rules for function recognition, this decompiler was 90% accurate. This work was developed at TRW Inc.

- **1967 - Halstead** An enhanced version of the Neliac decompiler was realized at the Lockheed Missiles and Space Company (LMSC) [Hal67]. The input of this decompiler was source code for the IBM 7094, which was then translated into Neliac code for the Univac1108.

  This decompiler successfully decompiled 90% of the instructions [Hal70]. The other 10% was left for the user to handle. At this time, decompilers focused on straightforward cases. The more complex ones were always left for the programmer to solve, since it had been shown that the time needed to solve this remaining 10% of the instructions was approximately equal to the effort already spent.

- **1967 - IBM Autocoder to Cobol Conversion** Housel at IBM developed a set of decompilers to translate Autocoder programs to Cobol [HH73]. A direct one-to-one mapping was performed and manual optimization was required. There was no type of automatic analysis and optimization, which lead to a rather large and inefficient code.

- **1973 - Hollander syntax-oriented decompiler** Hollander introduced in his PhD dissertation a new approach to decompilation, by means of a formal syntax-oriented metalanguage [Hol73]. The decompiler consisted of 5 processes: initializer, scanner, parser, constructor and generator, each of which were implemented as an interpreter of sets of metarules.

  An experimental version of this decompiler was implemented to translate a subset of IBM's 360 assembler into an Algol-like target language, which worked correctly on the 10 programs it was tested against.

  Basically, this technique can be seen as a kind of pattern-matching of assembler instructions into high-level instructions. This involves a strong limitation on the programs that can be decompiled, since instructions are forced to be in a certain order for patterns to be recognized. Different control flow patterns or optimized code are not allowed.

- **1973 - Housel PhD Thesis** Housel describes in his PhD dissertation [Hou73] an approach for decompilation that uses techniques from the compiler, graph and optimization th theories. A decompiler is divided in three main parts: partial assembly (separates data from instructions, builds the control flow graph generating an intermediate representation of the program), analyzer (analyzes the program in order to detect loops and remove useless instructions)

and code generator (optimizes the translation of arithmetic expressions and generates code for the target language).

Such a decompiler was implemented for translating Knuth's MIX assembler into PL/1 code for the IBM370 machines. In the 6 programs that were tested 88% of the instructions were correctly translated whereas the rest needed manual intervention.

The introduction of an intermediate representation made the decompiler machine independent. However, the chosen architecture provided an assembler that was not general enough.

- **1974 - The Piler System** The Piler System [Bar74] was a first attempt at a general decompiler, able to read machine code of several different machines and able to generate code for different high level languages. It was implemented only for one source architecture (GE/Honeywell 600) and for two target languages (Fortran and Cobol). It used a microform representation for the programs to be decompiled, which was lower-level than an assembler-type representation and made it difficult to be general enough.

- **1974 - Friedman's PhD Thesis** Friedman describes in his PhD thesis a decompiler used for the translation of minicomputer operating systems within machines of the same architectural class [Fri74]. This system consisted of four main phases: pre-processor, decompiler, code generator and compiler. The decompiler used was an adaption of Housel's decompiler [Hou73].

  This is the first attempt to decompile operating systems. However, the preprocessor phase of this decompiler took a too large effort and the final results turned out to be inefficient due to a larger size of the code and longer execution times.

- **1974 - Schneider and Winiger** Schneider and Winiger presented a notation for specifying the compilation and decompilation of high-level languages. A context-free grammar is defined for the compilation process and then, it is shown how this grammar can be inverted to decompile the object code into the original source program [SW74].

  This is another approach of syntax-oriented decompilation [Hol73]. Unfortunately, this only works for a particular compiler and only under certain circumstances, failing in the presence of optimizations, which makes it useless in practice.

- **1978 - Hopwood PhD Thesis** Hopwood describes a 7-step decompiler designed to aid porting and documentation [Hop78]. An experimental decompiler was implemented which translated assembly language into an artificial language called MOL620, featuring machine registers. This choice made the decompiler easy to implement, but the result was not high-level code. Besides, the control flow graph used had one node for each instruction (instead of a node for each basic block), which led to a huge control flow graph in case of large programs.

This decompiler was able to translate one large program successfully.

- **1978 - Workman** Workman introduced a new use of decompilation. The goal of his work was to define a high-level language suitable for real time training device systems using decompilation [Wor78]. In particular, it was implemented for the F4 trainer aircraft. Since the operating system of the F4 was in assembler, this was the input language, rather than machine code. The output language was not determined, as it was the goal of this project to define one.

  However, no code was generated. Only the first parts of the decompiler were implemented. The conclusions of this work were that the high-level language should have the following properties: handle bit strings, support loops and conditional control structures and not require dynamic data structures or recursion.

- **1981 - Zebra** The Zebra decompiler was developed at the Naval Underwater Systems Centre [Bri81]. It should port assembler programs from one platform to another. Thus, the output language was assembler as well.

  This project used available results to develop a decompiler of assembler programs. Although it did not introduce new concepts, it showed that it was hard to capture the semantics of the program to decompile, and that complete decompilation was not economically practical, but could be used as an aid for porting assembler programs.

- **1988 - Decomp** Decomp is a decompiler written by J. Reuter, which took object files of the VAX BSD 4.2 with additional symbolic information and generated C-like programs [Reu88]. This decompiler was exclusively intended to port the Empire game to the VMS environment, as the source code was not available. No data flow analysis was performed and the output still needed significant manual work before it could be recompiled.

- **1990 - exe2C** The exe2c decompiler intended to translate Intel80286/DOS executables into C code [War89]. Basically, the programs were disassembled, then converted into an internal format and finally converted to C. Several machine features such as registers were visible in the output. High level control flow structures such as loops and conditional constructs were recovered up to a certain point.

  However, this project was never completed. The results showed that data flow analysis and heuristics are necessary to produce better C code. It also remarked the importance of detecting library subroutines.

- **1991 - PLM-80 Decompiler** The Information Technology Division of the Australian Department of Defence tried to use decompilation for defence applications, such as maintenance of obsolete code and assessment of systems for hazards to safety and security. This work was described in [Hoo91].

A decompiler for Intel 8085 assembler programs that had been compiled by the PLM-80 compiler was developed, which produced programs in a subset of the C language. It inverted a grammar of the input assembly language. As in the previous attempts to use this kind of approach, optimized code is not supported.

However, this was the first decompiler to include a graphical interface to help the user document the program.

- **1991-1994 - Decompiler compiler** A decompiler compiler is a program that generates a decompiler from a formal specification of the relationship between source code and compiled object code [BBL93a, BB91, BB93, BBL93b, BB92, BB94b, Bow91, Bow93].

  In general, these compiler specifications are not available. Only customized compilers and decompilers can be build using this approach. Since this is a similar idea to inverting the grammar of the input language, optimized code is not supported. Actually, real executable programs cannot be handled. However, this sort of decompiler is useful in the verification of code in safety-critical applications.

- **1991-1993 - C Decompiling System** This work describes an Intel8086/DOS to C decompiler [FZL93, FZ91, HZY91]. It includes a hand-crafted and compiler specific function recognition, which allows the generation of more readable C code, but is rather inefficient in a general case. It contains a more sofisticated data types analysis than all decompilers until this point, since it tries to recognize types of arrays, pointers and structures. However, little detail is given on how this is done.

- **1993 - Source /PROM Comparator** At the Nuclear Electric plc, a tool was developed to demonstrate the correctness of the compilation of source code into Programmable Read-Only Memories (PROMs) in safety-critical systems [PW93].

  This project describes a use of decompilation techniques to help demonstrate the equivalence of high-level and low-level code in a safety-critical system. The output of the decompiler is a special intermediate language that facilitates comparison of the source and decompiled codes. The symbol table from the compiler, as well as specific knowledge of the compiler facilitate the work.

- **1994 - Cifuente's PhD Thesis** Cristina Cifuente's thesis "Reverse Compilation Techniques"[Cif94] sets the standards for the future decompilation research. In her work she showed the usefulness of data flow and control flow analyses in a decompiler. Both are common compiler analyses deeply studied in the computer science community. These techniques were implemented in the research decompiler dcc [Cif], which reads small Intel 80286/DOS programs and generates readable C code.

- **1997-2000 - REC** The Reverse Engineering Compiler is a retargetable decompiler which extends Cifuente's work in several ways. It handles several processors (Intel 386, Motorola 68K...) and multiple input formats (WLF, Windows PE...). It can use debugging information in the input file to name variables and functions. Variable arguments to library functions are handled well. Complex types such as arrays are translated into expressions. Registers and individual instructions semantics are visible in the decompiled output. However, the output is less readable.

  Binary distributions of REC are available from [Cap03].

- **1999 - Mycroft's Type Based Decompilation** In [Myc99] Mycroft describes a system for decompiling Register Transfer Language (RTL) to C. RTL is a common compiler intermediate representation. He uses type inferencing to find types from semantics of machine instructions and is able to generate code with pointers, structures and arrays.

  The solution is not always unique, which requires the user intervention. Since no experimental results are given, the generality and feasibleness of this approach is uncertain.

- **1999 - Ward's FermaT transformation system** Ward's FermaT system is based on formal transformations and is capable of transforming from assembly language to specifications [War99a, War00]. Even though the output is somewhat difficult to read, the system is validated by almost 2000 assembly language files which were translated to C and recompiled without error or warning.

  FermaT is property of Software Migrations Ltd [Ltd01] and is now released under GPL license [War01a].

- **2002-2003 - Boomerang** Boomerang [boo02, Tho] is a general decompiler which, for now can only decompile very small files.

Chapter 2

Hardware and Software Considerations

This work has been developed in the context of a retargetable decompiler for assembler languages with focus on digital signal processors (DSP). Even though the techniques presented here are architecture independent, they have been tested on the TIC62x DSP, which is a core with advanced architecture features. In order to motivate and understand the algorithms that will be explained later several hardware and software issues are discussed in the following sections.

Section 2.1 introduces the concept of parallelism and the hardware approaches to extract as much parallelism as possible from the programs. Digital Signal Processors and their technical evolution are presented in section 2.2. The software approaches to exploit parallelism are described in section 2.3. A description of the TIC62x DSP architecture concludes this chapter.

2.1 Hardware Issues

Parallel computing is the simultaneous execution of several tasks (which may, but do not need to be identical) [Mor98, HP90]. From the performance point of view, the advantages of executing several instructions in parallel is evident.

Thus, parallelism is one of the most interesting ideas in computing. Architectures and compilers have been striving for more than two decades to extract and utilize as much parallelism from the programs as possible in order to speed up computation.

2.1.1 Parallelism

The notion of parallelism is used in two different contexts: the **available** and the **utilized** parallelism. The available parallelism in programs refers to the possibility to perform several operations in parallel. The utilized parallelism is the parallelism occurring during execution, that is, the tasks that the processor does actually execute at the same time.

Types and levels of available parallelism

We can distinguish between two types of parallelism: *functional* and *data parallelism*, where the first one is the parallelism available in a program and data parallelism refers to the data structures (perform a similar computation on many data objects simultaneously). Here we will refer only to the functional parallelism. More information about data parallelism can be found at [Mor98].

Parallelism in a program can be available at different levels

Instruction level

Particular instructions of a program may be executed in parallel.

Loop level

Consecutive loop iterations are candidates for parallel execution.

Procedure level

Parallel executable procedures.

The fact that parallelism is available does not necessarily mean that the processor takes advantage of it when executing the program. Consider the C statements

```
a = a+1;
b = 15;
```

These two statements are completely independent from each other. Whatever program they belong to, it would certainly not affect to the final result if the second statement were executed in the first place. Since the execution order makes no difference, both statements could be executed in parallel. In such a case, instruction level parallelism is *available*.

However, if the architecture where they should be executed has only one execution unit, so that it can only carry out one instruction at a time, these two statements will certainly not be executed in parallel. Thus, the available parallelism cannot be utilized.

Furthermore, even if the architecture were able to execute these operations in parallel, if the compiler cannot detect that these statements can be executed in parallel, the available parallelism will not be utilized either.

Utilization of parallelism

Available parallelism can be utilized by architectures and compilers for speeding up computation. It is quite natural to utilize available parallelism, which is inherent in a conventional sequential program, at the instruction level by executing instructions in parallel, as seen in the example above. For this purpose, architectures are needed, which are capable of executing several instructions at the same time. Such architectures are referred to as **instruction-level parallel architectures** (**ILP-architectures**). Since available instruction-level parallelism is typically implicit in traditional sequential programs, it must be detected before execution. This is done, either by the compiler or by the ILP-architecture itself.

2.1.2 ILP-Architectures

There are two basic ways of exploiting parallelism in ILP-architectures:

- **Pipelining**

  In pipelining a number of functional units are employed in sequence to perform a single computation. These functional units form an assembly line called a *pipeline*. Each functional unit represents a certain stage of the computation and each computation goes through all stages of the pipeline.

- **Replication**

  A natural way of introducing parallelism to a computer is the replication of functional units (for example, processors). Replicated functional units can execute the same operation simultaneously on as many data elements as there are replicated computational resources available.

These two approaches are orthogonal, meaning that they can both be used at the same time in an architecture design.

Pipelined Architectures

The term *pipelining* refers to the temporal overlapping of processing. To achieve this, the job of executing an instruction is divided into several steps or pipeline stages. A basic division would be

1. Fetch

   Take the instruction that will be executed.

2. Decode

   Extract the instruction in order to know which operation needs to be executed.

3. Execute

   Compute the operation.

4. Write

   Write the results of the operation.

Since not all instructions need the same number of stages, there are instructions that need longer to execute than others.

Unfortunately, the pipelining of a functional unit does not ensure that the execution of the program will be ideal. There are several situations, called hazards, that prevent the next instruction in the instruction stream from executing during its designated clock cycle and reduces the performance of the pipeline. Hazards in pipelines can make it necessary to stall the pipeline, that means that some instructions in the pipeline are delayed. The rest is allowed to proceed. As long as the pipeline is stalled, no other instructions can start execution, which obviously slows down the execution of the program. We will not extend the possible causes of the hazards and how to deal with them, since it is out of the scope of this thesis. More information about this topic can be found at [Mor98].

Superscalar and VLIW Architectures

Another way to exploit the instruction level parallelism is using replication. Superscalar and VLIW architectures consist of multiple execution units operating in parallel which are intended to exploit as much as possible the instruction level parallelism of the programs.

Superscalar architectures are able to dispatch a few independent instructions per clock cycle. Instructions are scheduled dynamically, being the processor who is responsible for the order in which they are executed and on which functional unit.

VLIW architectures get their name, Very Long Instruction Word from the way instructions are formulated. The length of the instructions depends on the number of execution units available and the code length required to control each of the units.

Another important feature of the VLIW architectures is the fact that the instructions must be scheduled statically. This means, the compiler is responsible for scheduling the instructions, which considerably reduces the complexity compared to superscalar architectures, because the processor does not need to take care of this. However, since compilers are conservative, superscalar architectures can often extract more parallelism from the code than VLIW.

2.2 DSP processors

A digital signal processor (DSP) is a microprocessor designed specifically for the efficient implementation of digital signal processing algorithms [Dob00]. Like a general-purpose microprocessor, a DSP is a programmable device, with its own native instruction code. DSP chips are capable of carrying out millions of floating point operations per second, and like their better-known general-purpose cousins, faster and more powerful versions are continually being introduced.

DSPs are used, among other fields, in telecommunication (mobile communication, data transport ADSL) and multimedia (modems, MP3 encoders and decoders).

Rather than general computations, DSPs usually have an instruction set (ISA) optimized for the task of rapid signal processing, often using the following techniques:

1. Multiply-accumulate (MAC) operations.

   A MAC operation consist in multiplying two values and adding the result to a third one, the accumulator. This kind of operation is done very often in digital signal processing algorithms. It is convenient for matrix operations, such as convolution for filtering, Dot product, or even polynomial evaluation. Since it is needed so often, a single cycle MAC is an assumption in many DSPs.

2. Pipelining.

3. Saturation arithmetic.

   Operations that produce overflows (that means that the result of the operation is larger than the maximum value that can be represented in the machine),

will accumulate at the maximum (or minimum) values that the register can hold rather than wrapping around (maximum+1 doesn't equal minimum as in many general-purpose CPUs, instead it stays at maximum).

4. Separate program and data memories (Harvard architecture).

5. Fixed-point arithmetic.

   Most DSPs use fixed-point arithmetic, because in real world signal processing, the additional precision and range provided by floating point is not needed, and there is a large speed benefit; however, floating point DSPs are common for scientific and other applications where additional range or precision may be required.

6. Addressing modes.

   Specialized instructions for modulo addressing in ring buffers and bit-reversed addressing mode for FFT cross-referencing.

7. Hardware zero-overhead looping.

   To alleviate the branch impact for execution hi-frequent inner-loops, some processors provide this feature. There are two types of operation: single instruction repeating and multi-instruction loops.

History

The introduction of the microprocessor in the late 1970's and early 1980's made it possible for digital signal processing techniques to be used in a much wider range of applications. However, general-purpose microprocessors such as the Intel x86 family are not ideally suited to the numerically-intensive requirements of digital signal processing, and during the 1980's the increasing importance of DSP led several major electronics manufacturers (such as Texas Instruments, Analog Devices and Motorola) to develop Digital Signal Processor chips.

In 1978, Intel released the 2920 as an "analog signal processor". It had an on-chip ADC/DAC with an internal signal processor, but it did not have a hardware multiplier and was not successful in the market. In 1979, AMI released the S2811. It was designed as a microprocessor peripheral, and it had to be initialized by the host. The S2811 was likewise not successful in the market.

In 1979, Bell Labs introduced the first single chip DSP, the Mac 4 Microprocessor. Then, in 1980 the first stand-alone, complete DSPs – the NEC PD7720 and AT&T DSP1 – were presented at the IEEE International Solid-State Circuits Conference '80. Both processors were inspired by the research in PSTN telecommunications.

The first DSP produced by Texas Instruments (TI), the TMS32010 presented in 1983 was a 16-bit fixed-point accumulator architecture with a specialized instruction set.

The next generation of DSPs appeared around 1987 and were memory architectures with a richer set of addressing modes and hardware loops. Some example of

these DSPs are Motorola DSP56001, AT&T DSP16A, Analog Devices ADSP-2100 and Texas Instruments TMS320C50.

Current DSPs are load/store architectures which have an orthogonal instruction set and efficient code can be generated by a high-level language compiler. Very long instruction word (VLIW) architectures which allow the execution of multiple independent operations per cycle strongly increases performance and are widely used in current DSPs. Notable recent introductions include Texas Instruments TMS320C62xx and Philips Trimedia and Motorola Starcore.

2.3 Software Issues

In the previous section, some advanced hardware features of current DSPs have been presented, which can highly improve the performance of assembler programs. However, it is also necessary that the program is able to take profit of the hardware.

This section introduces two important software techniques that allow the processor to exploit as much instruction level parallelism in the program as possible. These techniques optimize the program to improve the efficiency of its execution. Unfortunately, they are also a considerable hurdle when it comes to decompile it. The work in this thesis is focused on the resolution of the additional problems for decompilation that cause the optimizations that will be presented next.

The aim of this section is not to give details on how to implement these optimizations, but rather describe their purpose and the result of applying them.

2.3.1 Instruction Scheduling

Instruction scheduling is especially useful for VLIW architectures, for which it was actually designed, and is a technique for generating additional parallelism in a program. Basically, it consists in finding a schedule of the instructions in the program such that the semantics are preserved and a maximal number of instructions can be executed in parallel avoiding pipeline stalls.

It can be performed at *block* or *trace* level. Block scheduling reorders the instructions within a basic block. The reordering of each block is independent of the other blocks except, perhaps for information about the values of the registers at the beginning or at the end of blocks.

Trace scheduling reorders the instructions in a simple path of blocks. The paths that are reordered are the most frequently executed paths in the program. Instructions may be moved from one block to another. Since the sequences of instructions are larger than single blocks, there are more opportunities for eliminating stalls than by block scheduling.

For finding a correct order of the instructions the dependencies between them must be taken into consideration. They define a partial order in the instruction set which must be kept. Otherwise, the semantics of the program would change.

Example

In the following example 2-cycle latency for load and 1-cycle latency otherwise is assumed.

```
load r1,  X
load r2, Y
load r4, A
mult r3, r2, r1
mult r5, r4, r3
load r7, B
add r6, r2, r3
mult r8, r5, r7
jmp
```

This code sequence needs 9 cycles to execute.

Now, let the architecture where this code is run have 2 execution units. After performing instruction scheduling the schedule shown next is obtained. Now only 6 cycles are needed to execute the code while it remains semantically equivalent to the original.

```
load r1,  X  ||  load r2, Y
load r4, A
mult r3, r2, r1  ||   load r7, B
mult r5, r4, r3
add r6, r2, r3  ||  mult r8, r5, r7
jmp
```

Here, the symbol `||` stands for parallel execution

2.3.2 Software pipelining

Software pipelining [Lam88] is a loop scheduling technique. The goal is to reduce the *initiation interval* of the loop, that is, the number of cycles between the start of one iteration to the start of the next one. In order to achieve this, multiple iterations of the loop are folded upon one another, as shown in figure 2.1. So, different parts of several iterations are executed concurrently. This increases the utilization of hardware functional units and decreases the total execution time of the loop.

The resulting code mimics a hardware pipeline, in which several iterations are in progress at once.

Example

Consider the following example

```
    // do i = 1, N
1 L:   load
```

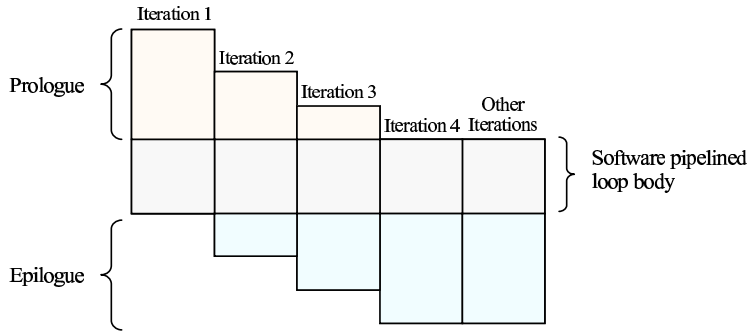

Figure 2.1: Software Pipelining

```
2
3       add
4       store       (cjmp L)
```

where instructions in the same row are executed in parallel and *cjmp L* is the jump to the loop label L. This pseudocode represents a loop that iterates N times and performs a load in the first cycle, an addition in the third and stores some information in the fourth. Each iteration needs 4 cycles to execute, making 4·N cycles necessary for executing the complete loop.

Software pipelining reorders the loop instructions in such a way that a new iteration starts in every cycle (the initiation interval is 1), instead of every 4 cycles. For this purpose, several iterations need to start outside of the loop. The result is shown below.

```
T      i = 1       i=2        i=3          i=4

1      load
2                  load
3      add                    load
4   L: store     add                     load      (cjmp L)
5                     store    add
6                               store     add
7                                            store
```

The instructions executed in 1-3 build the *prolog* whereas instructions in 5-7 build the *epilog*. The loop body of the software pipelined loop is also called *loop kernel*.

After software pipelining, the loop body (L) needs only 1 cycle to execute. If the original was to execute N times, this one will be executed N-3 times and the total number of cycles for executing this loop are N+3.

2.4 TMS320C62x DSP

The TMS320C62x is a fixed-point digital signal processor (DSP) with a high-performance VLIW architecture that belongs to the C6000 DSP family of Texas Instruments. A

traditional VLIW architecture consists of multiple execution units running in parallel, performing multiple instructions during a single clock cycle. Parallelism is the key to extremely high performance, taking these DSPs well beyond the performance capabilities of traditional superscalar designs.

Features of the C6000 devices include:

- Advanced VLIW CPU with eight functional units, including two multipliers and six arithmetic units
- 32 general-purpose registers of 32-bit word length
- Executes up to eight instructions per cycle
- Conditional execution of all instructions
- 8/16/32-bit data support
- 40-bit arithmetic options
- 32-bit integer multiply with 32- or 64-bit result.

The following sections will describe several aspects of this architecture that are interesting for the sections further on. More information is available at [Inc00, Ins02].

2.4.1 Assembler language

An assembler instruction for the C62x DSP has basically three parts:

1. The name of the instruction
2. The unit in which it executes

   As mentioned above, there are 8 execution units: the two multipliers M1 and M2 and six arithmetic-logical units L1, L1, S1, S2, D1 and D2.

3. The operands

   The number of operands depends on the instruction. An operand can be a constant (a number) or a register. This architecture has 32 general purpose registers: A0 - A15 and B0 - B15. The destination register, if any, is placed at the end of the instruction.

2.4.2 Parallel Operations

All instructions executing in parallel constitute an execute packet (or instruction bundle). An execute packet can contain up to eight instructions. Each instruction in an execute packet must use a different functional unit. In assembler, instructions that are to be executed in parallel are connected by the symbol `||`.

2.4.3 Conditional Operations

All instructions can be conditional. The specified condition register is tested before executing the instruction.

Conditional instructions are represented in code by using square brackets, [], surrounding the condition register name.

The following execute packet contains two ADD instructions in parallel. The first ADD is conditional on B0 being nonzero. The second ADD is conditional on B0 being zero. The character `!` indicates the inverse of the condition.

```
[B0] ADD .L1 A1, A2, A3  ||  [!B0] ADD .L2 B1, B2, B3
```

The two conditions shown in this example are the only types of conditions possible.

2.4.4 Pipeline and Delay Slots

The execution of fixed-point instructions can be defined in terms of delay slots. The number of delay slots is equivalent to the number of cycles required after the source operands are read for the result to be available for reading.

The pipeline phases are divided into three stages: fetch, decode and execute. All instructions in the C62x instruction set flow through the three stages of the pipeline. However, the execute stage is not equal for all instructions. Consequently, different types of instruction require a different number of cycles to finish execution, which is also manifest in the number of delay slots.

According to the cycles needed to execute, the instructions can be classified as follows

Single-Cycle Instructions

Most instructions need only one cycle to execute and have therefore no delay slots. Examples of single-cycle instructions are all additions (ADD), substractions (SUB), assignment of a register or a constant to another register (MV, MVK) and logic operations.

Two-Cycle Instructions

All multiplication instructions (MPY) need 2 cycles to execute. The result of the operation is written in the second cycle which generates one delay slot.

Store Instructions

Store instructions (STW, STH) require 3 cycles to finish execution. However, since the address modification is performed in the first cycle, these instructions have no delay slots.

Load Instructions

Data loads (LDW, LDH) require five cycles to complete their operations. Because data is not written to the register until the end, load instructions have

four delay slots. However, there are no delay slots associated with the address modification.

In the following instruction the contents of the memory address given by `A4` are loaded into register `A3` and the value of register `A4` is increased by 1. The new value of `A4` will be available in the next cycle, whereas 4 cycles must pass until the `A3` contains the loaded value.

```
LDW   .D1  *A4++, A3
```

Branch Instructions

Branch instructions (B) need 6 cycles to execute because the target of the branch needs a long time to load. Thus, branches have 5 delay slots.

2.4.5 Resource Constraints

There are several restrictions referring to the registers that can be used in an instruction:

- Two instructions within the same instruction bundle cannot use the same resources.
- Two instructions using the same functional unit cannot be issued in the same execute packet.
- Constraints on Register Reads: More than four reads of the same register cannot occur on the same cycle. Conditional registers are not included in this count.
- Constraints on Register Writes: Two instructions cannot write to the same register on the same cycle. Two instructions with the same destination can be scheduled in parallel as long as they do not write to the destination register on the same cycle. For example, a MPY issued on cycle i followed by an ADD on cycle $i+1$ cannot write to the same register because both instructions write a result on cycle $i+1$. Therefore, the following code sequence is invalid unless a branch occurs after the MPY, causing the ADD not to be issued.

```
MPY .M1 A0, A1, A2
ADD .L1 A4, A5, A2
```

However, this code sequence is valid:

```
MPY .M1 A0, A1, A2  ||  ADD .L1 A4, A5, A2
```

Chapter 3

Decompilation Framework

The context of the work presented in this thesis is the development of a retargetable decompiling tool. This chapter introduces the decompiler.

3.1 The Decompiler

A decompiler is divided into several phases, similar to the phases of a compiler, as shown in figure 3.1. The phases of the decompiler are

- Parsing
- Low level optimizations
- High level optimizations
- **C** Code generation

The first phase, parsing, translates the assembler code into an internal representation of the program (AIR). This AIR is architecture independent which allows the retargetability of the decompiler. Only the parser needs, for obvious reasons, knowledge about the source architecture.

The low-level optimizations are basically intended to create a control flow graph (CFG). Apart from the reconstruction of information which is not available in a low-level language such as data structures, several low-level optimizations need to be undone. This work is focused on the low-level part of the decompiler and, in particular, on the reverse transformation of instruction scheduling and software depipelining.

The high-level optimizations of the decompiler perform several typical high-level analyses and optimizations that also a compiler does.

C code can be more or less generated directly from the IR structure at any time. The quality of the C code will depend on the optimizations that have been performed in the previous phase.

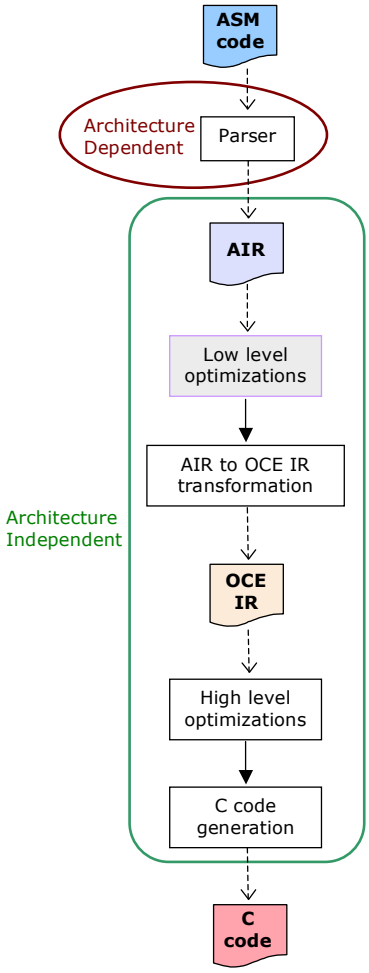

Figure 3.1: The decompiler

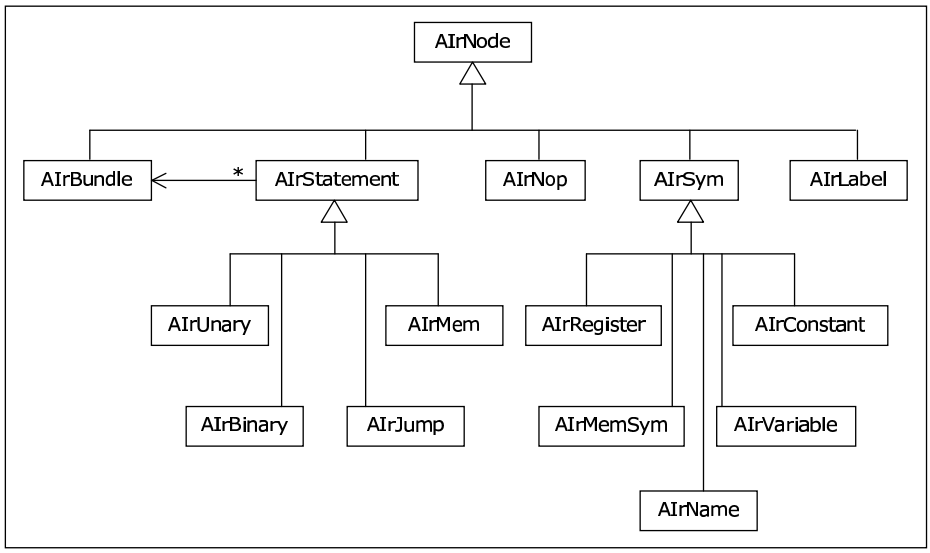

Figure 3.2: Class Diagram of the IR

3.1.1 Parser and Internal Representation

The assembler program file is parsed and an internal representation is created. The program is assumed to be correct and to run without errors on the appropriate architecture.

Parsing an assembler language is simpler than a high-level language due to the shorter, syntactically simpler instructions. However, it is not sufficient to match the instructions with AIR classes and build an intermediate structure. Extra information about the instruction set is also needed. In particular, the size of the registers and the number of delay slots of each instruction is collected in this phase. It is interesting to gather as much information as provided by the instruction set documentation in order to help further decompilation stages. In this case, the parser has been implemented using Flex++ and Bison++. It reads the assembler file and generates a structure in the decompiler's internal representation (IR) implemented in C++.

The internal structure stores the program in a 4 operand form. All operations are implicitly an assignment and are therefore formed by the operand or operands and the destination. A program is stored as a list of nodes. Figure 3.2 shows the class diagram of the structure. There are three types of major nodes that build the program.

- AIrLabel

  Contains a label of the program.

- AIrNop

  Represents a NOP (no operation) instruction.

- AIrBundle

Contains a list of instructions that are to be executed in parallel. Instructions are stored in AIrStatements.

A statement contains the name of the statement it represents and the operands. There are four groups of statements:

- Unary operations (AIrUnary)

  These are the *move* or copy instructions, bit operations like shifts and other typical operations like negation, setting a register to zero or absolute value.

- Binary operations (AIrBinary)

  Binary operations are most arithmetical (*add, sub, mult*) and logical operations (*less, equal, and, or...*).

- Memory operations (AIrMem)

  Refers to loads from and stores to memory.

- Control statements

  Jumps, calls and returns from functions.

This representation has very few expressions, as they are a rather high-level concept and this format was defined to remain close to the assembler language. The only expressions available are

- Constants (AIrConstant)

- Registers (AIrRegister)

- Memory symbols (AIrMemSym)

  These expressions represent an address in memory allowing register modifications (e.g., `*A0`, `*A4++`).

3.1.2 Low Level Optimizations

Once the program has been parsed, the next step is to construct a control flow graph to work with. Whereas the construction of the control flow graph in a compiler is a straightforward task, in a decompiler there are several issues that must be resolved before a complete CFG is generated.

- Functions are not necessarily determined. The program is just a list of statements and it is not known where a function starts and where it ends.

- The boundaries of basic blocks are not obvious.

- Indirect jumps, that are jumps whose target is a register, describe edges that cannot be easily determined.

- There is no type information and data structures are completely hidden.

Instruction Description	**AIR class**	**OCE class**	Comments
Function	AIrFunction	IrFunction	
Basic Blocks		IrBlock	Whenever the basic block information is generated,OCE blocks are built.
Label	AIrLabel	IrLabel	The appropiate basic block is labeled. No label statement is included explicitly in the block.
Branch instruction Call instruction Return instruction	AIrJump	IrJump IrCall IrReturn	Calls and returns from functions are coded as branches in the AIR.
Nop instruction	AIrNop	IrNop	
Load / Store instructions	AIrMem	IrAssign	These instructions are translated into an assignment from memory to register (loads) or from register to memory (stores)
Unary and Binary instructions	AIrUnary AIrBinary	IrAssign	Every unary and binary instruction is implicitly an assignment. The right-hand side of the assignment contains the operation, which is translated as shown in table 3.2

Table 3.1: AIR to OCE Transformation Table

Since it is not possible to solve all problems at the same time, a conservative control flow graph is constructed initially and then improved using the results of further analyses.

The first step is to construct a CFG that solves the problems caused by the different types of instruction scheduling. Here, the program is handled as a whole object; indirect jumps and function calls are ignored.

Function recognition and indirect jump resolution need an extensive memory analysis. Finally, the decompiler tries to detect data structures and other variables in the program.

3.1.3 Transformation to the OCE Representation

After the low-level optimizations phase, the structure is translated into a new representation. This new format is a tree-based high-level intermediate representation part of the Open Compiler Development Environment (OCE). The OCE contains components to analyze and modify the structure performing a large number of compiler optimizations. Since the decompiler was developed in the framework of a project using the OCE, this was chosen as intermediate representation to maintain the consistence of the whole project. Furthermore, this environment allows the reusability of several high-level optimizations which are common to both compiler and decompiler.

The translation from AIR to OCE is rather straightforward: AIR classes are matched to OCE classes. Table 3.1 illustrate the translation of AIR statements into OCE representation.

The targets of the jumps in the program are stored in a jump table. This is implemented as a vector (IrVectorExpr) of addresses of basic blocks. The address

Binary Operation	Opcode	**OCE class**
addition	ADD, ADDU, ADDH, ADDW...	IrAdd
substraction	SUB, SUBAW...	IrSub
multiplication	MPY, MPYH, MPYHL, MPYLH...	IrMult
shift right	SHR	IrShiftRight
shift left	SHL	IrShiftLeft
and	AND	IrAnd
or	OR	IrOr
exclusive or	XOR	IrXor
equal	CMPEQ	IrEqual
less	CMPLT	IrLess
greater	CMPGT	IrGreater
Unary Instruction	**OCE class**	
set to zero	ZERO	IrZero
negate	NEG	IrNeg
absolute value	ABS	IrAbs
move	MV, MVK...	(*)
sign extend	EXT, EXTU...	IrExt

(*) *The semantics of the move (MV) instruction are already expressed in the assignment statement.*

Table 3.2: AIR to OCE Transformation Table (Operations)

of a block is determined by the label associated to it.

The OCE format has one class for each operation. Therefore, the AIR unary and binary classes are divided into several classes according to the opcode of the instruction that was represented by the class. Table 3.2 shows the correspondence between these instructions and OCE classes.

Finally, table 3.3 describes how the AIR expressions are translated.

Once the program is expressed in the OCE internal representation, the high-level analyses and optimizations introduced in next section are performed.

AIR Expression	**OCE class**
AIrConstant	IrNum
AIrRegister	IrReg
AIrMemSym	if (!*offset*) IrRead(*base_address*) else IrAdd(IrRead(*base_address*), offset) (*)

(*) *base_address* is the register containing the base address, *offset* is the offset of the address modification.

Table 3.3: AIR to OCE Transformation Table (Expressions)

3.1.4 High Level Optimizations

Since the focus of the project were the low-level optimizations, only basic high-level optimizations have been implemented. An extensive description of this part of a decompiler, in particular of control flow optimizations such as structuring and goto elimination can be found at [Cif94].

Most high level optimizations require the knowledge about which registers are available at each program point. Therefore, a data flow analysis must be performed in order to collect this information.

Liveness Analysis

Definition: A variable is *live* if it holds a value that will or might be used in the future.

Liveness analysis collects information on the variables (in this case, on the registers), determining which of them are alive at a given program point. This information is necessary for several program optimizations, such as dead code elimination or the determination of the function arguments.

Every use of a variable (an access to the value held by the variable) makes the variable *live*. Every definition of a variable (an assignment to the variable) *kills* it and makes the previous value unavailable.

The representation of the program used for liveness analysis is a control flow graph. For every node in the CFG (representing a basic block), the following sets are defined

Def-set: *def(n)* contains all registers that are defined (killed) in basic block n.

Use-set: *use(n)* contains all registers that are used (generated) in the basic block.

Live In set: *In(n)* is the set of all registers that are live at the entry of block n.

Live Out set: *Out(n)* is the set of all registers that are live at the exit of block n.

The goal of liveness analysis is to compute the sets *In(n)* and *Out(n)* for all blocks in the control flow graph. The Live In and Live Out sets are related to the Use and Def information by the following data flow equations.

$$\begin{aligned} In(n) &= \bigcup_{p \in succs(n)} Out(p) \\ Out(n) &= \big[In(n) - def(n)\big] \bigcup use(n) \\ Out(end) &= \emptyset \end{aligned}$$

where $succs(n)$ is the set of successors of n in the CFG.

These equations can be solved iteratively as shown in the algorithm 3.3.

Our tool performs a simple liveness analysis in order to supply the different optimizations with the necessary information. For each instruction, the definition

```
for n basic block ∈ CFG do
   In(n) = ∅;
   Out(n) = ∅;
od
do
   for n basic block ∈ CFG do
      In'(n) = In(n);
      Out'(n) = Out(n);
      In(n) = use(n) ⋃ (Out(n) - def(n));
      Out(n) = ⋃ In(s), for all s ∈ succ(n)
   od
while ((In'(n) ≠ In(n))  or (Out'(n) ≠ Out(n)))
```

Figure 3.3: Iterative Computation of Liveness

and use information is computed, which is then collected in reverse order to the given by the statements to form the *def* and *use* sets for each block

In the implementation of the algorithm in figure 3.3 the sets above are represented by bit vectors. The union operations are implemented by logical **or**. The equivalence

$$Out(n) - def(n) = Out(n) \wedge \neg def(n),$$

where $\wedge$ is the logical **and** is used to implement of $Out(n) - def(n)$.

The following optimizations are performed using the results of the liveness analysis.

Dead Code Elimination

A register is said to be *dead* if it is defined and not used before being redefined. Since the value given by the definition is useless, the instruction can be eliminated (or changed in order to remove that definition).

Figure 3.4 shows a simple algorithm for removing dead computations. The liveness analysis provides this algorithm with the set `alive` of all live variables at a given point. A variable is dead if it does not belong to `alive`.

The statements in each block are processed in reverse order. In order to remove transitive dead computations, it is required to perform several iterations of this algorithm using the new liveness information.

Register Arguments

The problem targeted here is accurate identification of the arguments to subroutines and functions, since they are usually not specified explicitly in the assembly language code.

The register arguments of a subroutine can be characterized as follows:

A register, which is defined before the call to a subroutine and is then used (without redefinition) inside the subroutine must have been passed to the subroutine as an argument.

```
for all n ∈ N - {start,end} do
   alive = In(n);
   for all statements s ∈ n do
      if s = <v=expr;> then
         if  v ∉ alive then
            remove statement s
         fi
         alive = alive - {v}
      fi
      if statement not removed then
         for all v used in s do
            alive = alive ∪ {v}
         od
      fi
   od
od
```

Figure 3.4: Removing Dead Computations

Information on registers used before being redefined is obtained via intraprocedural live register analysis. It gives, for each basic block, the registers used in that basic block which are live on entry.

Function Return Registers

Functions usually return values in registers, which are subsequently used by the caller. We need to know when a register contains a return value of a function.

Return values of a function can be characterized as follows:

A register defined in the function and used afterwards without being redefined is used to transmit a result out of the function, that is, it contains a return value of the function.

The algorithm looks for registers that are live at the program point immediately following a function call. If there is one such register, it is a plausible candidate to be a function result. This register information is propagated across subroutine boundaries and is solved using a reaching and live register analysis.

Copy Propagation

An instruction is intermediate if it defines a register value that is used by a unique subsequent instruction. In this case intermediate instructions which are simply used for moving information around are removed. For example, if we have an assignment $A0 = B0$, we would like to replace subsequent uses of $A0$ with $B0$, and remove the instruction $A0 = B0$. This can be done when $A0$ is not redefined before the uses we want to replace, and when the value of $B0$ has not been changed either before the uses. In general, that means that the identity $A0 = B0$ is still valid. Similarly,

if an assignment such as $A0 = 23$ is performed, then it may be possible to replace some subsequent uses of $A0$ with the constant 23. Such substitutions often enable additional optimizations to be performed.

In order to solve this problem a new data-flow problem is set up in which *In(n)* is the set of copies $A = B$ such that every path from the initial node to the beginning of n contains the statement $A = B$, and subsequent to the last occurrence of $A = B$ there are no assignments to B. *Out(n)* can be defined correspondingly, but with respect to the end of n.

A copy statement $s : A = B$ is *generated* in block n if s occurs in n and there is no subsequent assignment to B within n. On the other hand, $s : A = B$ is *killed* in$n$ if A or B is assigned there and s is not in n.

Consider the following sets:

Gen set: *gen(n)* contains all copy statements generated in basic block n.

Kill set: *kill(n)* contains all copy statements killed in basic block n.

Then, the data flow equations that model the copy propagation problem are

$$\begin{aligned} In(n) &= \bigcap_{p \in preds(n)} Out(p) \\ Out(n) &= \big[In(n) - kill(n)\big] \bigcup gen(n) \\ Out(start) &= \emptyset \end{aligned}$$

where $predss(n)$ is the set of successors of n in the CFG.

Note that these equations are very similar to the ones solved for the liveness analysis. Whereas this is a forward problem, the liveness analysis is solved backwards. Thus, the algorithm to solve these equations is analogue to 3.3.

Type Analysis

In order to translate assembler code into C code it is necessary to determine the types of the variables. Only a very simple type inference algorithm has been implemented which propagates type information through assignments.

Since not always enough information is available to determine the type of the register variables, two different groups of types are defined. Determined types are full types, whereas undetermined types only contain a part of the type information.

- **Determined types:**
  - short, int, long, longlong
    Numeric types of lengths 8, 16, 32 and 64, respectively.
  - char and string
    Alphanumeric types of length 1 or greater than 1 respectively.
  - bool
    Boolean type.

- **Undetermined types:**
  - numeric
    The register contains a number. The concrete size is not known.
  - alphanum
    Alphanumeric type.
  - unknown
    No information about the type is available.

All registers used in a function are treated like local variables.

Every expression has a type container, which may be a type or a pointer type. The type of an expression is derived from the types of the expressions that define it. Since a register can contain values of any kind of type, we keep track of the actual type of the registers in order to be able to make the correct type inferences.

Thus, every block is annotated with two lists: the type information of all registers at the entry and at the exit of the block. The type information at the exit of a block is inherited by its successor. When a block has several predecessors the different lists must be merged.

The only expressions that define a type directly are IrNum, IrEqual, IrUnequal, IrLess, IrGreater, IrString and IrSymbol. IrNum is a numeric type. Its width determines whether it is a short, int, long or longlong. Logic operations return a bool type. IrString and IrSymbol have type char or string also depending on their length.

The memory addresses involved in load and store instructions determine pointer types. Bit operations like shifts and extensions are also assumed to perform on numbers.

An assignment sets a new type to a register. We assume that arithmetic operations result on numeric types and their operands are also numbers. Whenever an assignment is processed, the destination register's type is updated and it is checked whether the operands have the correct types as well.

The type information is computed iteratively in the style of a fix-point algorithm. The resulting types information can be used in different ways. The final types list determines the types if the registers are considered like local variables of the function. A more accurate approach consists on using the results of the live analysis to define different variables and associate types for the live ranges.

3.1.5 C Code Generation

The last part of the decompilation process from assembler language to C is the actual generation of the C code. Once the structure of the program has been built and all the transformations are completed, the code can be generated. Basically, for all functions in the program, all statements are visited and translated into equivalent C code. The process is relatively straightforward, as shown in the algorithm of figure 3.5.

The function `generate_statement(i)` generates C code for the OCE IR instruction `i`.

```
function
  begin
    for every function f do
      if function has return value r then
        write(r->type);
      else
        write("void");
      fi
      write(f->name);
      if function has parameters then
        write_function_parameters;
      fi
      for every block b in f do
        write("{");
        for every instruction i in b do
          generate$_$statement(i);
        od
        write("}");
      od
    od
  end
endfunction
```

Figure 3.5: C Code Generation Algorithm

The main statements are translated as shown in table 3.4. For registers, labels and symbols their name is written. IrNum classes are translated into their value.

In binary expressions C code for the two operands is generated and the operation, translated according to 3.5, is set between them. By unary expressions the operator is set before the operand.

3.2 Related Work

This section discusses in more detail several interesting decompilation approaches.

Description	**OCE class**	**C statement**
assignment	IrAssign	=
branch	IrJump	goto
conditional	IrIf	if(*cond*){*if-block*} else{*else-block*}
function call	IrCall	*function_type* function_name (*function_arguments*)
return	IrReturn	return (*return_value (optional)*)

Table 3.4: Statements Conversion Table

Operation	OCE class	C statement
and	IrAnd	&
or	IrOr	||
xor	IrXor	^
addition	IrAdd	+
substraction	IrSub	-
multiplication	IrMult	*
comparison: less	IrLess	<
comparison: greater	IrGreater	>
comparison: equal	IrEqual	==
comparison: unequal	IrUnequal	!=
not	IrNegate	!
shift right	IrShiftRight	>>
shift left	IrShiftLeft	<<
extract bits i to j from register *reg*	IrExtract	$reg \;\&\; (2^j + 1) >> i$

Table 3.5: Operations Conversion Table

Cristina Cifuentes - *Decompilation Techniques*

The most important work in this area is the thesis of Cristina Cifuentes [Cif94, CSF98], where a decompiler called `dcc` is introduced [Cif], that translates INTEL80286 binaries compiled for MSDOS back into C.

Cifuentes' work is the most comprehensive description of a decompiler available in the literature treating decompilation similarly to compilation. Thus, the decompilation process is divided into several parts:

- Syntax Analysis
- Sematic Analysis
- Intermediate Code Generation
- Control Flow Graph Generation
- Data Flow Analysis
- Control Flow Analysis
- Code Generation

The decompiler framework introduced in section 3.1 follows the ideas presented in [Cif94]. However, the first three stages are reduced to one parsing stage, since we assume assembler instead of binary code as input.

Although the core of the `dcc` tool is very architecture dependent, Cifuentes introduces a detailed description of data and control flow techniques to recover information and optimize the code. This way, compiler techniques proved to be suitable or adaptable for decompilation.

However, data types and structure recognition are not totally resolved and indirect jumps and function calls are not handled. Besides, the current architectures are more complex than the INTEL80286, leading to new problems that were not discussed in Cifuente's thesis. In particular, these are the problems discussed in this thesis.

FermaT transformation system

The FermaT transformation system [War99b, War01b, War01c, War04] is an industrial-strength formal transformation engine with many applications in program comprehension and language migration.

The FermaT transformation system uses formal proven program transformations, which preserve or refine the semantics of a program while changing its form. The theoretical work on which FermaT is based originated not in software maintenance, but in research on the development of a language in which proofs of equivalence can be performed easily in most cases.

By expressing the program in the WSL (Wide Spectrum Language), which contains low-level as well as high-level constructs, all transformations can be performed without the need of distinguishing between programming and specification language. Thus, the whole transformational development from assembler to high-level language can be performed on a single representation.

There are two types of operations that can be carried out on a program. A *program transformation* is an operation which modifies a program into a different form which has the same external behavior. Since both programs and specifications are part of the same language, transformations can be used to prove that a given program is a correct implementation of a given specification. A *refinement* is an operation which modifies a program to make its behavior more defined or deterministic.

The FermaT workbench is a collection of tools and databases based around the core technology of program transformations in the WSL language. One of its applications is the translation of IBM 370 assembler code to equivalent C code. The assembler to WSL translation works from a listing file (containing, among other information macro expansions and base and index registers for each instruction) rather than a source file. All the information in the list file is necessary for translation.

The first stage in the transformation process is the data translation. Initially all data is accessed directly from memory (represented as a base register and an offset). After the transformation the layout of all data in memory is obtained and variables and data structures are created. However, the details of this transformation are not specified. It is claimed that several reasonable assumptions are performed in this stage, but no concrete information is given in the literature.

The next stage is the control flow restructuring, which consists on eliminating superfluous labels and branches and introducing loops. Then, with the help of data and control flow analysis, procedure boundaries are determined. After control flow restructuring, an extended form of constant propagation, which can propagate return addresses through procedure calls is applied. These last steps are iterated

until no further improvement is possible.

The final step is to generate C code from the structured WSL. This may involve further transformations to eliminate WSL features which cannot be directly implemented in C.

The system has proven to provide good results but unfortunately, due to the industrial nature of the tool, details of how the analysis is performed are not given. However, the tested architecture does not create the more complex problems of a Pipelined VLIW architecture, which is the focus of this work.

asmtoc Decompiler

Johnstone et al. [JSW00a, JSW00b, JSW99] implemented a decompiler from Analog Devices ADSP-21xx assembly language source to ANSI C. The tool also follows the decompilation structure introduced by Cristina Cifuentes' work.

asmtoc focus on the translation of code in an obsolete language into a new one, assuming that the users are familiar with both languages and they have full commented source code available for the entire system.

A particular feature of this tool is that they offer different levels of analysis, from the direct translation of the assembler to a more optimized version of the high-level code.

Even though asmtoc target a rather modern DSP architecture, it does not handle some critical problems of the decompilation process, like type recovery. Since the ADSP-21xxx has a more conservative design than the TIC62x, its assembler code does not present the issues subject of this thesis.

Breuer and Bowen - *Decompilation: The Enumeration of Types and Grammars*

From the more theoretical point of view, Breuer and Bowen [BB94a, BB93, BB91, BBL93a] proved that, under some restrictions, a decompiler could be constructed if a grammar for the compiler is available.

They set out the method and theory behind a decompiler generator, a utility that can generate a guaranteed decompiler from the specification of relationships between source code and compiled object code. It is likely that in the case of optimizing compilers at least, too much information is lost in compilation to make an automated approach feasible in practice. But in safety-critical systems it is usual to generate non-optimised code.

In their work, Breuer and Bowen show how attribute grammar descriptions (of programming language compilers) may be re-expressed as functional programming code which lists out the intended elements of the grammar and then show how to use this idea to build decompilers from compilers.

A decompiler is seen as a function from object codes to a list of source codes, representing the inverse of the compile relation. Thus, each object code defines a grammar of source codes - precisely those that will compile to it. The grammar of the compiler helps to develop a set of rules which are used to define the decompile relation. It is straightforward to translate an attribute grammar description into functional programming code which enumerates the valid terms of the grammar.

This technique has been applied to generate a decompiler for a small occam-like language obtaining an enumerating decompiler which lists all the possible source codes which compile to a given object code. Since this method only considers compiled code it is not able to handle hand-written programs in general. This technique has not proved to be useful in practice.

Alan Mycroft - *Type-Based Decompilation*

Alan Mycroft [Myc99] describes an application of type inference algorithms for the reconstruction of C code from register transfer language level descriptions of programs.

He describes a system which reverse engineers C programs from target machine code by type-inference techniques. This approach extends recent trends in the process of compiling high-level languages whereby type information is preserved during compilation. The algorithms remain independent of the particular architecture because target instructions are treated as register-transfer specifications.

These algorithms are not implementable in practice either. Type information at assembler level is assumed, which makes it unfeasible for hand-written code. Besides, the problems of VLIW architectures are not treated at all.

Probst et al. - *Register Liveness Analysis for Optimizing Dynamic Binary Translation*

Dynamic binary translators compile machine code from a source architecture to a target architecture at run time. Due to the hard time constraints of just-in-time compilation only highly efficient optimization algorithms can be employed. Common problems are an insufficient number of registers on the target architecture and the different handling of condition codes in source and target architecture. Without optimizations useless stores and computations are generated by the dynamic binary translator and cause significant performance losses. In order to eliminate these useless operations, a very fast liveness analysis is required.

Probst et al. present in [PKS02]a dynamic liveness analysis technique that trades precision for fast execution.

Dynamic liveness analysis is only performed when basic blocks are translated. The liveness information is kept and at the end of execution it is stored in a file. A further run of a program reads the file and refines the information from previous runs. In this setting the propagation of liveness information is performed over several runs of a program. This approach results in a speed-up of 10 to 30 percent depending on the target machine. In addition, the dynamic liveness analysis results are very close to the most precise solution.

Chapter 4

CFG Reconstruction and Sequentialization

One of the main steps in compilation is the construction of the control flow graph (CFG). The CFG is a structure that represents the behavior of the program. Most program analyses and optimizations assume the existence of such a graph which makes it essential for any translation. Usually the CFG is constructed in the frontend of a compiler, where its construction is easy [ASU86].

Due to the analogies between compilation and decompilation, it is obvious that the control flow graph is equally important for decompilation as it is for compilation. Unfortunately, its construction is not as simple.

The result of a reverse compiler is an equivalent program in a high level language. A high level language cannot express instruction level parallelism and has no knowledge about the number of cycles needed to execute each instruction which, if available, are strongly taken into account in assembler programming. This introduces additional problems to the classic control flow graph construction.

This chapter introduces an approach for, given an assembly language program, generating the minimal control flow graph of a semantically equivalent assembly language program where each instruction is assumed to need only one cycle to execute and no parallel execution is allowed.

The first section introduces the problems encountered and how the solution is not always trivial. The remaining of this chapter is dedicated to fully describe the proposed approach.

4.1 Introduction to the Problem

Assembly language or machine code heavily complicate the CFG construction. Instructions are executed in parallel in VLIW architectures. The results of some instructions are available some cycles later than they are executed (delayed instructions). Branch instructions can be executed in the delay slots of other branches. Instructions are executed conditionally (predicated execution). To efficiently utilize the processor techniques like software pipelining are applied and instructions

are reordered. All these factors make the construction of a control flow graph for decompilation a non-trivial problem.

4.1.1 Delayed Instructions

One of the main consequences of pipelined architectures is that the processor does not need to wait until an instruction has finished executing before starting to execute the next one. As seen in section 2.3.1, the programmers take advantage of this by applying instruction scheduling techniques. This scheduling implies a reordering of the instructions that differs from their *original* order, that is, the order they would be written in if the processor had no pipeline.

When writing a program in a high-level language, the programmer does not take care about how the processor will handle its instructions. The logic of the statement sequence assumes that, whenever a statement is executed, all the previous statements in the flow path have been executed in the order in which they are written, taking jumps into account. Therefore, in a high-level language program like

```
b = a;
a = A[i];
c = a+1;
```

the input value of `a` in the statement `c = a+1` is certainly `A[i]`.

Now, let `R` be a register containing the address of `A[i]`. In an assembler language where loads (LDW) need 2 cycles to execute and any other instruction only 1, the previous code would be implemented as

```
MV a, b
LDW *R, a
NOP
ADD a, 1, c
```

where `MV a, b` moves the contents of `a` into `b`, `LDW *R, a` loads the content of the memory location `R` into `a` and `ADD a, 1, c` stores in `c` the value of `a+1`.

This is the unoptimized version of the code, which needs 4 cycles to execute. The instruction scheduled form only needs 3 cycles to execute:

```
LDW *R, a
MV a, b
ADD a, 1, c
```

Since the load instruction needs 2 cycles to execute, the new value of `a` is not available until the addition is performed. Therefore, `b` gets the old value of `a` as planned.

A direct translation of the previous assembler code into high-level language

```
a = *R;
b = a;
c = a+1;
```

gives a wrong result, since the value of `b` is `*R` instead of the old value of `a`, as it should be.

As we have demonstrated above, it can be hard for a human reader to make sense of the code. Perhaps, more importantly, the delay slots present a major complication to tools which attempt to disassemble the code or to construct a control flow graph. In general, the assembler code, as written, shows the instructions in the order in which they are dispatched. However, for the purposes of understanding control flow, we need to see the instructions in the order in which they are completed. It would be desirable, therefore, to remove the effect of delay slots from the code and reverse the instruction scheduling.

4.1.2 Choices for Removing Delay Slots

In the general case, each instruction I would have an associated number of execution cycles I_X.

There are two plausible alternatives for removing the effect of delay slots from the program.

1. Transform the code sequence to an equivalent version where all delay slots are occupied by no-op instructions. For example, a branch instruction which takes six cycles and therefore has five delay slots should appear in the transformed program followed by five `NOP` instructions.
2. Transform the code sequence to a version for an idealized computer which has the same instruction set, but where all instructions complete execution in a single cycle.

We discuss each one in turn.

Filling Delay Slots with Nops

This approach is appealing because the result should be a program which is still executable on the same platform, but where the code is human readable and also easy for tools to process. Unfortunately, it is impossible to complete the transformation without imposing some arbitrary instruction orderings – orderings which are not implied in the original program. For example, consider the following group of three instructions:

```
inst1   ;  has two delay slots
inst2   ;  has one delay slots
inst3   ;  has zero delay slots
```

In this example, all three instructions complete their execution at the same time (as would be possible if they employ different functional units on the processor). If we present the transformed code as follows:

```
inst1   ;  has two delay slots
NOP
NOP
```

```
inst2   ;  has one delay slots
NOP
inst3   ;  has zero delay slots
```

then we have imposed an ordering which was not implied in the original code.

An Ideal Computer with No Delay Slots

Our alternative approach, and the approach adopted in the remainder of this paper, is to transform the code to a version where all instructions are assumed to complete their execution in one cycle and where several instructions may be executed in parallel. For example, the group of three instructions shown earlier may be transformed to the following form:

```
inst1 || inst2 || inst3
```

The new code sequence accurately shows all three instructions as completing simultaneously. However, the code can be misleading or, perhaps, wrong because it also shows the three instructions fetching their operands in the same cycle, whereas the original code sequence showed them being fetched in different cycles. It is easy to construct an example where the timing of an operand fetch is significant. Suppose the original program is as follows where we assume that multiply (MPY) requires two delay slots and loading a constant (MVK) has no delay slots:

```
MPY  R2, R3, R1    ;  R1= R2*R3
MVK  20, R2        ;  R2 = 20
NOP                ;  no delay slots
```

If ordered according to the order in which the instructions complete their execution (and eliminating the NOP), we would have

```
MVK  20, R2
MPY  R2, R3, R1
```

and that is clearly incorrect because R2 is changed before the MPY instruction has read its value. To preserve correctness, we must insert instructions to make copies of operands when needed. A correct solution would be

```
MV   R2, T1
MVK  20, R2
MPY  T1, R3, R1
```

where the idealized architecture is extended with additional temporary registers to hold operand copies.

Thus, when building the CFG of the program, there are two main problems to solve

- Resolve delays taking data dependencies into account
- Sequentialize parallel bundles

First, we will concentrate on the delayed instructions and explain how to build a control flow graph allowing parallel instructions. Later those instruction bundles will be sequentialized.

4.1.3 Delay Slots Resolution

The principle followed when performing the reverse instruction scheduling is to write each instruction at the position in the program in which it finishes execution. For this purpose, each instruction will be assigned a position, which will be the position of the instruction in the program, starting from the beginning. Parallel instructions share the same position.

For a program that executes sequentially the program counter moves one position every cycle. The distance in terms of positions is equivalent to the distance in terms of cycles.

An instruction I located at position p which finishes execution after $n+1$ cycles has n delay slots.

The program counter reaches position p at cycle c. After n cycles, that is at cycle $c+n$ the position of the program counter will be $p+n$. Let I' be an instruction equivalent to I, but needing only 1 cycle to execute. It is semantically equivalent writing I at position p to writing I' at position $p+n$.

In order to remove the delay slots, we need to find out for each instruction with delay, at which position it finishes execution. Then, all delayed instructions I are replaced by instructions I' written where instructions I would finish execution.

Having a program containing no jumps this replacement of instructions is very simple, since each instruction is moved to position $p+n$, being p the original position and n the number of delay slots. The possible data conflicts that appear when relocating an instruction will be discussed later.

4.1.4 Delayed Branches and the Control Flow

The control flow of a program is determined by the branches in it. On some architectures branches are executed with a delay. In particular, in the C62x instruction set, branches need 6 cycles to execute, having 5 delay slots (see section 2.4). Additionally, branches may be declared in the delay of another branch. This is what we call a *delayed branch*. All other branches we call *independent branches*. From a formal point of view, a delayed branch is a branch that needs several cycles to execute. Thus, an independent branch is also a delayed branch. However, it will get the name *independent* in order to express that is has not been declared in the delay slots of any other branch. This fact is relevant for the algorithms presented here.

We have seen before that we need to find out where the instructions with delay finish execution in order to relocate them. Branches are a special case of instructions with delay because they determine the control flow of the program which, as shown in the following example, has a big impact on the final position of all other instruction with delays.

Example

Consider the following example, where load instructions (LDW) and branches need 3 cycles to execute (2 delay slots) and multiplication instructions have 1 delay slot.

```
0       LDW *A4, A1 || LDW *B4, B1
1       LDW *A2, B2
2       NOP
3       CMPGT B1, A1, A0
4       [A0] B jump
5       MVK 15, A5
6       MPY B2, A5, A5
7       ADD A5, B1, B2
jump:
8       ADD A5, A1, B2
```

Considering what has been discussed until now, the final position of the delayed instructions of this code should be

```
LDW *A4, A1          pos 0  →  pos 2
LDW *B4, B1          pos 0  →  pos 2
LDW *A2, B2          pos 1  →  pos 3
[A0] B jump          pos 4  →  pos 6
MPY B2, A5, A5       pos 6  →  pos 7
```

and the delay resolution would give the following result

```
0       NOP
1       NOP
2       LDW *A4, A1 || LDW *B4, B1
3       CMPGT B1, A1, A0 || LDW *A2, B2
4       NOP
5       MVK 15, A5
6       [A0] B jump
7       ADD A5, B1, B2 || MPY B2, A5, A5
jump:
8       ADD A5, A1, B2;
```

In this case the multiplication

```
MPY B2, A5, A5
```

is only executed if the condition

`[A0]` of the branch instruction is not fulfilled. A closer look to the original code shows that this behavior is not correct: when the multiplication starts execution, the jump has not been realized yet. If the branch condition is fulfilled, the next position of the program counter will be 8. Otherwise, it will be 7. In both cases, the multiplication that has started executing will write the result in register A5. It finishes execution either way and for this reason, it should appear in both possible paths described by the branch, and not only in one of them.

This example introduces two important factors for the control flow graph construction:

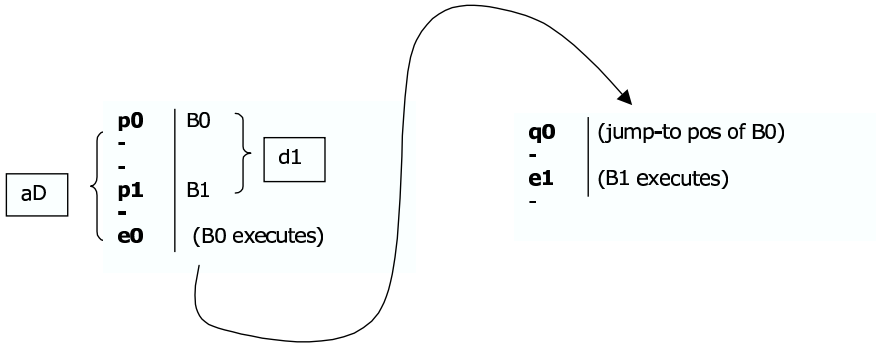

Figure 4.1: One delayed branch

- Knowledge is needed about the control flow **before** the delays are resolved.
- It may be needed to make several copies of the same instruction in some cases.

In practice, this knowledge about the control flow refers to the positions where the branches finish execution. Independent branches are easy to handle: a branch at position p executes at position $p + n$, where n is the number of delayed slots of branch instructions.

However, delayed branches are more difficult to handle.

Delayed Branches

As shown so far, for the delay resolution the basics of the control flow are needed, meaning that we need to find out where jumps are performed, and where they jump to. The target of a branch is assumed to be known, and in the case of independent branches, the position where the jump is realized is the position where it is declared plus the number of delay slots of the branch.

The presence of delayed branches in the program increases the complexity of the problem due to fact that the *place* where a delayed branch finishes execution depends not only on the position where it is declared, but also on the behavior of the previous branch, since it may cause a change in the control flow.

However, the *time* when it will perform is known: let aD be the number of delay slots of branches in the given architecture. This is also the number of cycles the branch has to wait from its declaration in the program until the jump is performed. Thus, if we know how the independent branch behaves, we can find out the behavior of the delayed branch.

In the example in figure 4.1, the independent branch $B0$ is declared at position p_0, executes at position $e_0 = p_0 + aD$, and jumps to position q_0. The delayed branch $B1$ is declared at position p_1. We want to know where it really executes.

Since both branches have the same number of delay slots (aD), $B1$ can not execute before $B0$, which means that it will execute after the jump of $B0$ has taken place, that is, at a position from q_0 or larger. When $B1$ executes, $B0$ still needs to wait a few cycles. In particular, it has to wait as many cycles as have passed from

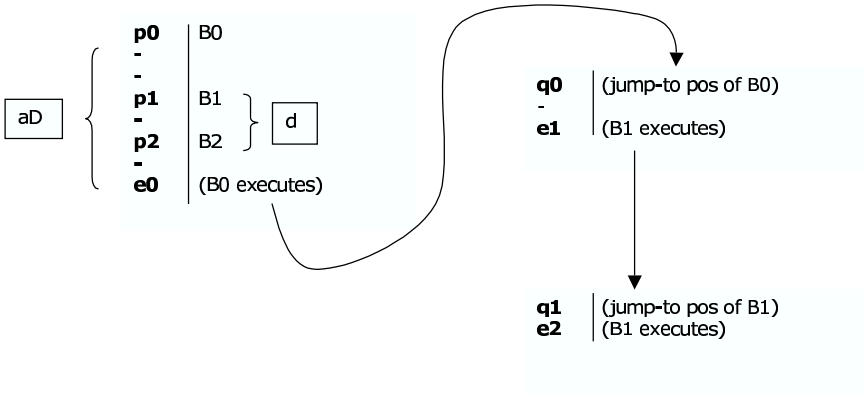

Figure 4.2: Two delayed branches

the declaration of $B0$ until the declaration of $B1$, this is $d_1 = p_1 - p_0$. Thus, $B1$ will execute at position $e_1 = q_0 + (p_1 - p_0) - 1$.

In order to prove this formula, we take into account that the number of cycles that pass from the moment $B1$ is declared until it is executed, must be equal to aD. So,

$$(e_0 - p_1) + (e_1 - q_0 + 1) \quad = \quad aD$$

Since $B0$ has also aD delay slots, we have

$$\begin{aligned} aD &= p_0 - e_0 \\ &= d_1 + (e_0 - p_1) \end{aligned}$$

Considering both equations together, we obtain that

$$\begin{aligned} (e_0 - p_1) + (e_1 - q_0 + 1) &= d_1 + (e_0 - p_1) \\ e_1 - q_0 + 1 &= d_1 \\ e_1 &= d_1 + q_0 - 1 \end{aligned}$$

$d1$ is called the *delay* of $B1$ towards $B0$.

Next, the situation is analyzed in which a branch has more than one delayed branch. For this purpose, consider the example in figure 4.2.

Since $B1$ is the first delayed branch, it behaves as if it were the only delayed branch. $B2$ can impossibly have an impact on any previous branch in the code.

Thus, $B0$ is at position p_0, executes at position $e0$ and jumps to q_0, and $B1$ is declared at p_1, executes at $e_1 = p_1 + d_1 - 1$ and jumps to q_1, as described above.

$B2$ is a delayed branch of $B0$, declared at position $p2$. Now we want to know where it executes. Let d_2 be the distance from $B2$ to $B0$ in terms of positions. Then, we know that $B2$ will execute d_2 cycles after $B0$ executes. Once $B0$ executes, the program counter moves to position q_0. In theory, $B2$ should execute at position $q_0 + d_2 - 1$, the problem is that $B1$ executes before, at position $q_0 + d_1 - 1$ (note that $d_1 < d_2$), and the control flow is changed again. So, $B2$ executes at e_2 such that

$$\begin{aligned} aD &= (e_2 - q_1 + 1) + (e_1 - q_0 + 1) + (e_0 - p_2) \\ e_2 &= aD - (e_0 - p_2) - (e_1 - q_0 + 1) + q_1 - 1 \\ e_2 &= aD - (e_1 - q_0 + 1) - (e_0 - p_2) - (q_1 - 1) \end{aligned}$$

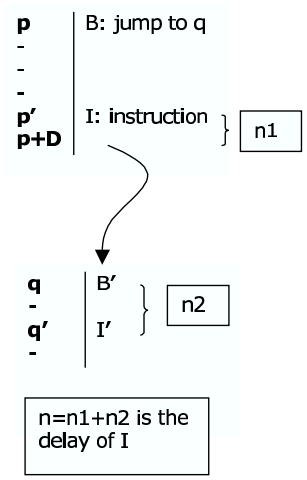

Figure 4.3: Program with one branch

Let $d = p_2 - p_1$ be the distance between the declarations of $B1$ and $B2$. Then,

$$\begin{aligned} e_2 &= aD - (e_1 - q_0 + 1) - (e_0 - d - p_1) - (q_1 - 1) \\ &= aD - (e_1 - j_0 + 1) - (e_0 - p_1) + (q_1 + d - 1) \end{aligned}$$

Since $B1$ has aD delay slots,

$$aD = (e_1 - q_0 + 1) - (e_0 - p_1)$$

and so, e_2 can be expressed as

$$e_2 = q_1 + d - 1 \tag{4.1}$$

This last formula shows that the execution position of $B2$ only depends on the distance between $B2$ and the previous branch $B1$ and the target position of this branch.

In general, this formula is always valid, independently of the nesting depth of the delayed branches.

At this point, delayed branches have been studied, which were all delayed branches of the same independent branch. It is obvious that, in our previous example, $B2$ is, at the same time, delayed branch of $B0$ as well as delayed branch of $B1$. $B0$ is called *parent branch of a nest of delayed branches*.

Delay Resolution Taking the Control Flow into Account

Once the information about the execution position of branches has been collected, it can proceed to relocate the rest of delayed instructions. The basic idea on how to do this is explained in this section.

Assume there is an unconditional branch at position p, executing at position $p + aD$, jumping to position q, as shown in figure 4.3.

Let I be an instruction at position $p' \leq p + aD$ with a delay of n cycles such that $p' + n > p + aD$ and $n_1 = distance(p', p + aD)$. We aim to move this instruction I

to the position where it finishes executing. To find this out, consider the behavior of the program counter.

At a certain cycle c, the program counter (PC) will be at position p', where I is declared. n_1 cycles later, PC will be at $p + aD$ After the next cycle, however, PC will not be at position $p + aD + 1$, but at position q, which is the target of the jump that occurs at position $p + aD$. $n_2 - 1$ cycles later (where $n_2 + distance(q, q')$), I will finish execution. At that moment PC will be at position $q + n_2 - 1$, which is the final destination of I.

In general, I should be moved to the position

$$q + ((p + aD) - p') - 1$$

In case of having a conditional branch, there are two different possible paths. Either the condition is fulfilled and so the branch taken ($taken - path$), or the condition is not fulfilled and the branch ignored ($fall - through - path$). An instruction that is to be moved beyond a conditional branch will be moved following both paths and two copies of it will be inserted.

4.1.5 Parallel Instructions

High level languages in general do not give the possibility to specify which instructions are to be executed in parallel. However, assembly languages for VLIW architectures do (2.1.2). Since there is no way to express parallelism in a high-level language, when decompiling the assembler program, parallel instructions must be rewritten in a sequential form.

Two parallel instructions like

```
ADD A0, A1, A2 || MV B0, B1
```

can be transformed either into

```
ADD A0, A1, A2
MV B0, B1
```

or into

```
MV B0, B1
ADD A0, A1, A2
```

The result will be the same either way, because the two instructions do not cause any conflict with each other. A different situation would be if the instructions were

```
ADD A0, B1, A2 || MV B0, B1
```

Now, the register `B1` is *used* by the first instruction and *defined* by the second. Since they are executed in parallel, the value of $B1$ that uses the addition is the **old** value, that is, the value before the assignment of $B0$ to $B1$.

In this case, the correct sequential form is

```
ADD A0, B1, A2
MV B0, B1
```

Some architectures can execute a large number of instructions in parallel. Then, the sequentialization becomes much more complicated. Section 4.7 discusses different approaches to solve this problem.

4.2 CFG Reconstruction

So far, the major difficulties of reconstructing the control flow graph of an assembler program for decompilation have been described.

The idea of moving instructions in order to get a program whose instructions are in high-level style has been described. This is necessary for any kind of analysis needed to improve the assembler program, and especially necessary for the translation into C.

It has also been proven that instructions cannot be reordered correctly without any knowledge of the program flow. For this reason, the control flow graph is needed, and for the construction of the CFG the behavior of the branches in the program must be analyzed. Although delayed branches are the most difficult ones to handle, it has been shown that it can easily computed where their jumps occur, needing only information about the nearest branch they are delayed branch from.

The following sections present a new approach for constructing the control flow graph and resolving the instructions delay. Basically, the edges of the control flow graph are constructed by studying the behavior of the branches in the program. Then the basic blocks of the graph are built and, finally, the instructions with delay are moved along the graph.

4.2.1 Solution Approach

An algorithm has been developed to construct the control flow graph of an assembly language program including reordering of instructions with delay and sequentialization of parallel instructions. It has been implemented for the assembly language of the TIC62 DSP. However, no architecture specific properties have been used. The algorithm works with an architecture neutral low level intermediate representation, where delay information for each instruction is included.

Basically, this approach is divided in four passes:

1. edge recognition
2. block construction
3. delay resolution
4. sequentialization

The following sections will describe each step in detail.

4.3 Edge Recognition

This section presents the fundamentals of an algorithm for finding the edges of the control flow graph. These edges will connect instructions, since no knowledge about basic blocks is available yet. Later, when blocks have been built the edge information will be updated to basic block level.

The basic idea of this algorithm is to follow the execution of all paths of the program through its branches. Each branch is examined in the order of execution.

Given a branch, the function *handleBranch* shown in figure 4.4 generates one edge. If the branch is conditional, it generates two edges. It follows these edges in order to decide when the next jump will take place.

This is the function responsible for detecting the edges of the control flow graph. Its arguments are

branches: the list of branches in the program

b: the current branch. A branch is defined by a triple (id, p, j), where id is the name of the branch, p its declaration position in the program and j the position of the label it jumps to.

D: a list of pairs (b, d), where b is a delayed branch and d is the distance between b and the previous branch (see section 4.1.4).

epos: is the position where b executes

edges: the list of edges. An edge is a quadruple (e, j, b, D), where e is the origin of the edge and j the target. b is the branch that has caused the edge and D is the list of delayed branches that was valid as the edge was created.

The algorithm for edge constructions starts with the first branch from the entry point of the program. Figure 4.4 shows how each branch is handled.

Note that the program is traversed following all possible paths and whenever a point is reached that was already visited, the exploration of the path stops. Thus, the algorithm will terminate because every branch is handled exactly once.

4.3.1 Selecting new delayed branches

Let B be a branch and D the list of delayed branches. One of the following situations can occur in a program:

1. B is an independent branch (D=$\emptyset$)
2. B is a delayed branch and D=$\emptyset$ (B is the last delayed branch)
3. B is a delayed branch and D$\neq \emptyset$

Note that we cannot have an independent branch and D$\neq \emptyset$, because an independent branch does not have delayed branches and it is impossible that we are handling an independent branch when delayed branches are still pending execution.

```
function handleBranch(branches, (b, pos, jpos), D, epos, edges)
begin
  nextb = -1
  oldD = D
  oldepos = epos
  if !((epos, jpos, b, D) ∈ edges) then
     add (epos, jpos, b, D) to edges
  else
     goto ELSEPATH
  fi
  add all branches in [pos, epos] ⋃ [jpos, jpos] to D
  if (D ≠ ∅ ) then
    (nextb, delta) = first element in D
     epos = jpos + delta
  else
     nextb = next branch from position jpos
     epos = jpos + bd
  fi
  if (nextb ≥ 0) then handleBranch(branches, nextb, D, jpos, epos, edges)
  fi
ELSEPATH:
  if b is conditional branch then
    D = oldD
    epos = oldepos
    nextb = -1
    jpos = epos + 1
    if !((epos, jpos, b, D) ∈ edges) then
       add (epos, jpos, b, D) to edges
    else return
    fi
    add all branches in interval [pos, jpos] to D
    if (D ≠ ∅ ) then
       (nextb, delta) = first element in D
       epos = jpos + delta
    else
       nextb = next branch from position jpos
       epos = jpos + bd
    fi
    if (nextb≥0) then handleBranch(branches, nextb, D, jpos, epos, edges)
    fi
  fi
  return
 end
```

Figure 4.4: CFG Edge Construction Algorithm

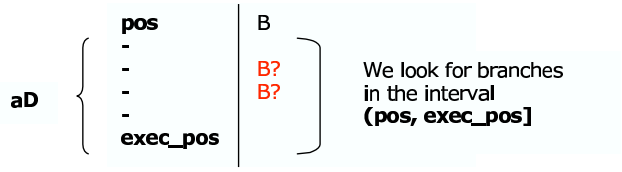

Figure 4.5: B independent and D empty

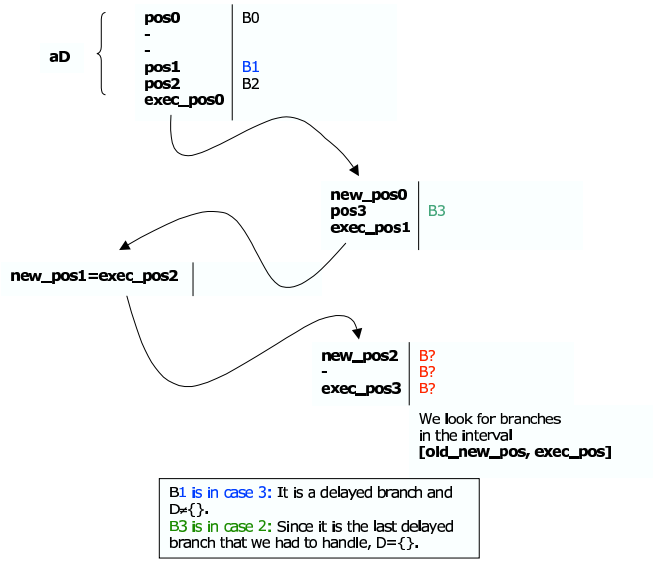

Figure 4.6: B is a delayed branch

B is an independent branch

In this case only one interval has to be taken into account. If B is declared at position *pos*, and aD is the number of delay slots for branches in the given architecture, B will execute at position $exec\_pos = pos + aD$ (see figure 4.5). The delayed branches of B will be all branches declared between *pos* and *exec_pos*. For each delayed branch, its d is computed and the information is added to D. Given a delayed branch, its d is computed relative to the *previous* branch, which may be either B or another delayed branch.

If there are delayed branches for B, the next branch will be the first delayed branch. Otherwise, the next branch following the target position of B is selected.

B is a delayed branch

As figure 4.6 shows, it is necessary to check for delayed branches between the *new_pos* of the previous branch, and the execution position of the present branch.

The branch $B1$ shows the case where D$\neq \emptyset$ and for the branch $B3$, D=$\emptyset$; that is, $B3$ is the last delayed branch.

4.3.2 Example

Through the next example, we will explain in detail how to construct its edges. For simplicity reasons, we will omit the instructions that have no effect on the control flow.

```
jumpA:
     0  inst1
     1  [cond] B jumpB
     2  inst2
     3  [cond] B jumpD
     4  inst3
     5  inst4
     6  B jumpA
     7  B jumpC
     8  inst5
     9  inst6
jumpB:
    10  inst7
    11  B jumpC
jumpC:
    12  inst8
    13  inst9
jumpD:
    14  inst10
    15  inst11
    16  inst12
```

We assume branches to have 3 delay slots. In this piece of code there are 4 labels: *jumpA* at position 0, *jumpB* at position 10, *jumpC* at position 12 and *jumpD* at position 14. There are 2 conditional branches at position 1 and 3, which we will call $B0$ and $B1$ respectively. There are also 3 unconditional branches: $B2$ at position 6, $B3$ at position 7 and $B4$ at position 11.

Let D be the list of pairs (b, d), where b is a delayed branch and d its distance to the closest branch (see 4.1.4). Initially, $D = \emptyset$. We start with the first branch $B0$, which is an independent branch. Therefore, it will execute at position 4. All branches between the declaration of $B0$ and its execution position are delayed branches and so, added to D. In this case, we have one delayed branch: $D = \{(B1, 2)\}$.

Since $B0$ is a conditional branch, we have to handle both possible paths. We start with the taken-path, which means that our new position is the position of $jumpB$. So, we have our first edge: (4, 10).

Since D is not empty, the next branch to execute will be the first delayed branch: $B1$. According to the formulas developed in 4.1.4, $B1$ will execute at position 11. In order to update D, we have to look for delayed branches between the declaration of $B1$ and the execution position of $B0$, and between the target of $B0$ and the execution position of $B1$. We find one branch. Thus, $D = \{(B4, 3)\}$.

$B1$ is a conditional branch as well. We take the taken-path and jump to position 14. The new edge (11, 14) is added to the list of edges.

Again, since D is not empty, the next branch will be the delayed branch $B4$, which executes at position 16. No delayed branches are found. The new edge is (16, 12). Since no branches are declared from position 12 until the end of the program, the path we have taken ends here. (16, 12) is a final edge.

Now, we go back to take the else path of branch $B1$. The new position is 12. Recall that $D = \{(B4, 3)\}$ and therefore, the next branch to execute will be $B4$. It will execute at position 14, and will build the edge (14, 12) which, like before, is a final edge.

The next step is to handle the else path of branch $B0$. At this point, $D = \{(B1, 2)\}$. The new position is 5. This describes the edge (4, 5).

$B1$ will be the next branch, executing at position 6. The new delayed branches list is $D = \{(B2, 3)\}$. Since $B1$ is conditional, we will follow both possible paths.

The taken-path goes to position 14, following edge (6, 14). The next branch is $B2$, which will be execute at position 16, having no delayed branches and describing the edge (16, 0). At position 0, the next branch will be $B0$ as independent branch. This is a situation in which we have already been before (at the beginning of the analysis). Since we have found a cycle, the analysis of this path stops here.

The else path of branch $B1$ describes the edge (6, 7). Delayed branch $B2$ is the next one to execute. In this case, it has one delayed branch. $D = \{(B3, 1)\}$. $B2$ describes edge (9, 0). Branch $B3$ executes at position 0 without delayed branches. Jumps to position 12 building the edge (0, 12). There are no more branches to handle. The algorithm stops here.

4.4 Minimal Number of Edges

A problem that is most certain to arise when trying to recover the control flow graph of such an assembler program is that conditional delayed branches may lead to an important explosion in the number of edges. The fact of having to move branches along both paths of a conditional branch can make the number of copies of branches to be very high.

The example in figure 4.7 shows a small piece of code with three conditional branches. $B1$ and $B2$ are delayed branches. Since both possible paths need to be followed in every case, these 3 branches generate 12 edges.

However, all these conditional branches share the same condition in such a way that, if one branch is not taken, its delayed branches will not be taken either, assuming that the register involved in the condition is not changed between delayed branches. This observation provides the opportunity to simply ignore some branches in some paths. Figure 4.8 shows the result of such an optimization.

A value analysis for the conditional registers is needed, which can help to remove useless edges in the control flow graph. Instead of generating the CFG and later performing an analysis to cut paths, the analysis is included as part of the edge construction algorithm. By doing this, the cost of the computation is reduced.

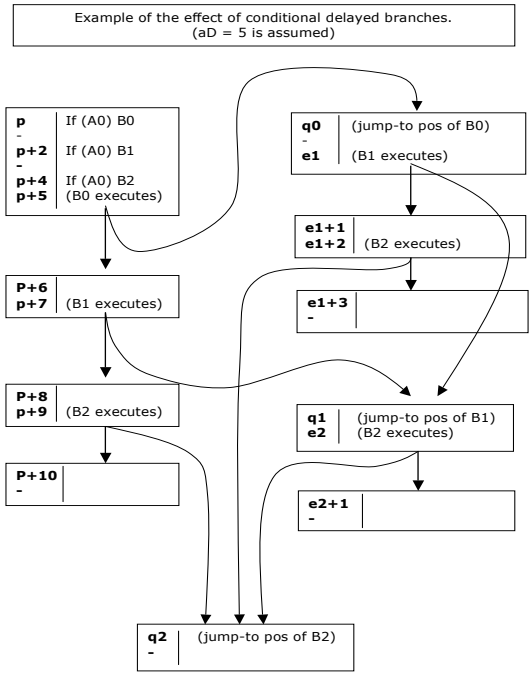

Figure 4.7: Explosion of branches

Unfortunately, no traditional method for value or range analysis can be used at that point because the CFG, which is required by any of these techniques has not been constructed yet. The proposed approach performs an analysis of the registers involved in the branch conditions in parallel to the edge construction.

Since the edge recognition algorithm follows the program flow from jump to jump, the information on the registers contained in the code between jumps must be collected, together with the information that gives us the fact whether a certain (conditional) branch is or not taken.

This analysis can be more or less accurate, depending on how much information we want to collect. Of course, the more accurate the analysis, the more time consuming it will be.

We have implemented two versions of this analysis. The first one considers only the conditional registers and the second one takes into account the effect of all program registers. Next, we will explain with more detail how the analysis is done.

4.4.1 Algorithm

This algorithm is an extended version of the edges construction algorithm presented in section 4.3, in which an environment for a certain number of registers has been defined, that contains information on the value of these registers at each point of the program. Then, before taking a path in the edges construction algorithm, first we look whether the condition (if any), is fulfilled. And in case it is not, the path is not taken.

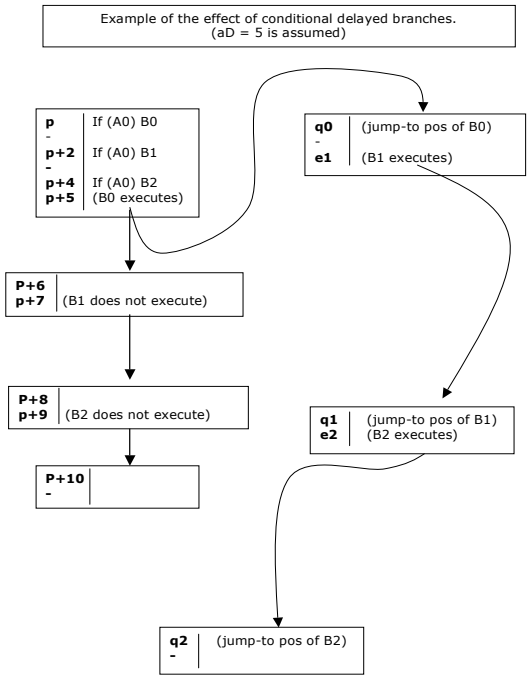

Figure 4.8: Effect of Conditional Register Analysis

The register environment **regEnv** is a list of triples `(reg, state, value)`, where

reg is a register,

state is the state of the register at that point. A register can be in one of the following states:

strue, if $reg \neq 0$

sfalse, if $reg = 0$, and

sunk if we have no information about *reg*, or not enough to say anything else about it.

value is the concrete value of *reg*, if it known.

Having only this information about the registers would allow to extract information almost only from direct assignments. So, an assignment like $R0 = R1 - 1$, where the state of $R1$ is **strue**, would lead $R0$ to be **sunk**. However, we know that if the value of $R1$ is 1, $R0$ will have the state **sfalse**, and if $R1 \neq 1$, the state of $R0$ will be **strue**. For this reason, also the value of the register, if available, is stored.

The register environment information is used to dismiss certain paths and branches in the edge reconstruction phase. When, according to the environment, the condition of a conditional branch is not fulfilled, the jump it describes can be ignored and only the fall-through path will be followed. In the same way, if the condition of a

```
function handleBranch(branches,(b, pos, jpos),D,epos,edges,regEnv)
  nextb = -1
  oldD = D
  oldepos = epos
  oldregEnv = regEnv
  if !((epos, jpos, b, D) ∈ edges) then
     add (epos, jpos, b, D) to edges
  else goto ELSEPATH
  fi
  add all branches in [pos, epos] ⋃ [jpos, jpos] to D
  update environment regEnv
  if (D ≠ ∅ ) then
    (nextb, delta) = first element in D
    epos = jpos + delta
  else
    nextb = next branch from position jpos
    epos = jpos + bd
  fi
  if (nextb ≥ 0) then
    if ((b is conditional and
         b's condition is coherent with regEnv) or
         b is unconditional) then
      set condition of b to strue if available
      handleBranch(branches, nextb, D, jpos, epos, edges)
    fi
  fi
ELSEPATH:
  if b is conditional branch then
    D = oldD
    epos = oldepos
    regEnv = oldregEnv
    nextb = -1
    jpos = epos + 1
    if !((epos, jpos, b, D) ∈ edges) then
      add (epos, jpos, b, D) to edges
    else return
    fi
    add all branches in interval [pos, jpos] to D
    update environment regEnv
    if (D ≠ ∅ ) then
      (nextb, delta) = first element in D
      epos = jpos + delta
    else
      nextb = next branch from position jpos
      epos = jpos + bd
    fi
    if (nextb ≥ 0) then
      if (b's condition is coherent with regEnv) then
        set condition of b to sfalse
        handleBranch(branches, nextb, D, jpos, epos, edges)
      fi
    fi
  fi
```

Figure 4.9: CFG Edge Construction Algorithm With Register Analysis

branch will certainly be fulfilled, only the if-path is taken into consideration. Figure 4.9 shows the revised algorithm. As an extension of 4.4, it maintains the same structure. There is only one additional argument: *regEnv*, the register environment.

This method makes it possible to remove several edges from the CFG even before they are built. How accurate the result is, and how close to the actual minimal CFG, depends on the analyzed program and on the accuracy of the value analysis for the registers.

Again, the existence of delayed branches makes the process of searching for changes on the registers, more complicated. Next, we will give the position intervals in which we are to look for redefinitions of the conditional registers.

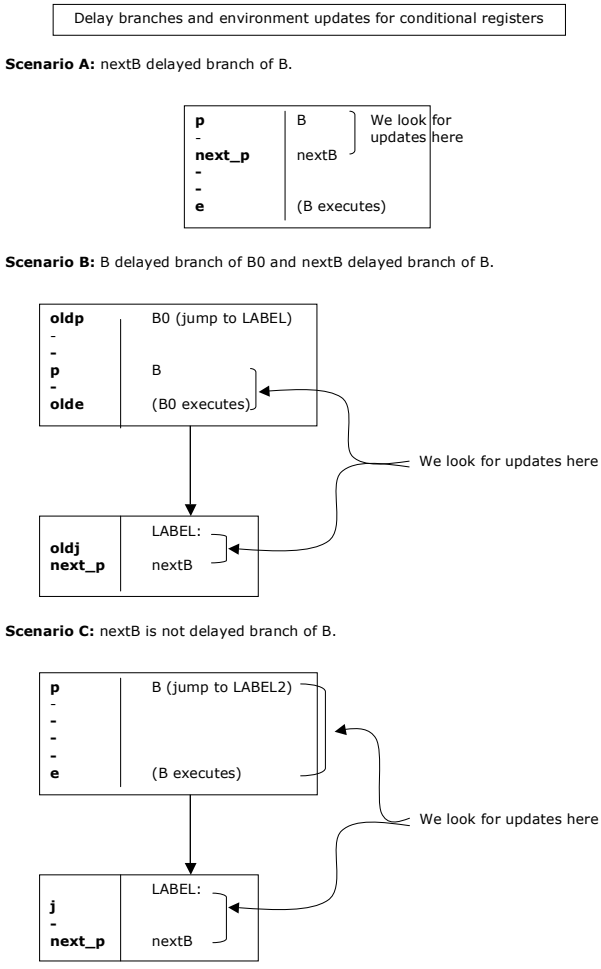

Figure 4.10: Redefinition Intervals

4.4.2 How to Update the Environment

Let B be the current branch and $nextB$ the branch that is to be executed next. B is declared at position p, executed at position e and jumps to position q. In case B

is a delayed branch, its previous branch would be $B0$, declared at old_p, executed at old_e and jumped to old_q. $nextB$ is declared at position $next\_p$. There are 3 possible scenarios depending on the location of $nextB$.

- **Scenario A:** $nextB$ is a delayed branch of branch B and B is either independent or a delayed branch of $B0$, where $nextB$ is also a delayed branch of $B0$.
- **Scenario B:** B is a delayed branch of $B0$ and $nextB$ is delayed branch of B but not of $B0$.
- **Scenario C:** $nextB$ is an independent branch.

These situations are shown in figure 4.10, and lead to the following intervals to be analyzed.

- If $next\_p \in [p, old\_p]$ look for redefinitions in $[p, next\_p)$
- Else if $next\_p \in [old\_q, e]$ look for redefinitions in $[p, old\_e] \cup [old\_q, next\_p)$
- Else, look for redefinitions in $[p, old\_e] \cup [old\_q, e] \cup [q, next\_p)$

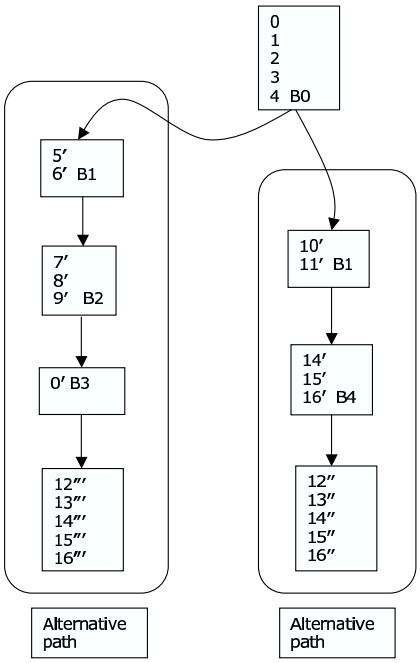

Figure 4.11: CFG with conditions analysis

4.4.3 Example

Consider a simple example to show the important effect of this optimization for the quality of the control flow graph. Recall the code of section 4.3.2. Positions 1-3

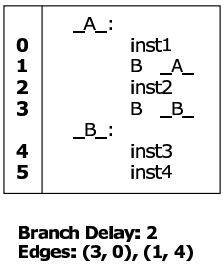

Figure 4.12: BB Construction: example code

have been modified adding concrete instructions.

```
jumpA:
     0  inst1
     1  if(A0=0) B jumpB
     2  A1=A1+1
     3  if(A0=0) B jumpD
     4  inst3
     5  inst4
     6  B jumpA
     7  B jumpC
     8  inst5
     9  inst6
jumpB:
    10  inst7
    11  B jumpC
jumpC:
    12  inst8
    13  inst9
jumpD:
    14  inst10
    15  inst11
    16  inst12
```

In this example there are two conditional branches and both have the same condition: $if(A0 = 0)$. Note that from declaration of $B0$ at position 1 until declaration of $B1$ at position 3, there are no updates of register $A0$. Therefore, if $B0$ is taken, then $B1$ will also be taken, and if $B0$ is not taken, then $B1$ will not be taken either. Thus, the extended edge construction algorithm will generate the following edges:

$$(4, 10), (11, 4), (16, 12), (4, 5), (6, 7), (9, 0), (0, 12)$$

The corresponding control flow graph is shown in Figure 4.11. Note that this new graph has 8 nodes and 7 edges, whereas the unoptimized graph had 10 nodes and 11 edges. This graph is clearly a subgraph of the original one.

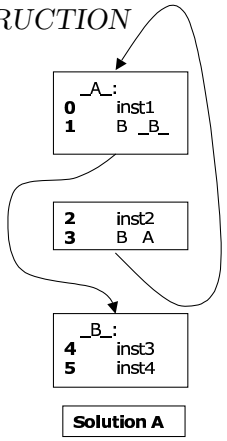

Figure 4.13: BB Construction: direct solution

4.5 Basic Blocks Construction

Cooper et al. [CHW02] proposed an algorithm for computing the control flow graph of an assembly language program with delayed branches. Their algorithm solves a different problem as the delayed branches remain in the program. The program is only partitioned into basic blocks and the necessary control flow edges are inserted.

In general, the presence of delayed branches introduces changes in the control flow, which must be carefully treated.

Let us consider the code in figure 4.12. This code has two branches in positions 1 and 3. A branch delay of 2 is assumed. Thus, the second branch is a delayed branch of the branch to $\_A\_$, since it is declared in the second delay slot of the first branch. The first branch executes at position 3 and jumps to $\_A\_$, then the second branch executes at position 1 and jumps to label $\_B\_$. Therefore, the edges of the control flow graph will be (3, 0) and (1, 4). The intuitive approach to construct

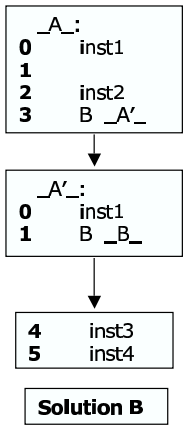

Figure 4.14: BB Construction: correct solution

the CFG would be to extract the block boundaries directly from the edges. There is an edge from position 3 to position 0, ending a block at position 3 and starting one at position 0. From the second edge it can be concluded that some block ends

at position 1 and eventually another one will start at position 4. The expected but wrong CFG would have three blocks as shown in figure 4.13. Branches have been moved to the positions where the jumps are performed.

The correct solution (see figure 4.14), is obtained by considering, not only the edges, but also some path information. Here, $inst1$ and $inst2$ are executed before the first jump is performed. The first branch, instead of jumping to the original label $\_A\_$, jumps to a copy of it; $inst1$ is executed again, and then the second jump to $inst3$ and $inst4$ is performed. For a correct solution copies of some instructions must be added to the code.

4.5.1 Definitions

Next, a few terms will be defined that will be used throughout the following sections.

- **edge caused by another edge**
  An edge e is *caused by* an edge e' if the branch that defined e was a delayed branch of the branch B' that defined e' and was not a delayed branch of any other branch, or when e was caused by an edge that was caused by e'. In the same way e is also *caused by* B'.

- **normal edges**
  A normal edge is an edge that is not caused by any other edge in the program. Normal edges correspond to branches that are no delayed branches.

- **alternative edges**
  An alternative edge is an edge that is *caused by* some edge in the program. Alternative edges correspond to delayed branches.

- **normal path**
  The normal path in a program is given by all the normal edges in the program.

- **alternative path**
  An alternative path is built by all the alternative edges that are caused by the same edge.

4.5.2 Algorithm

In order to create the basic blocks, normal and alternative edges are handled separately. First, the basic blocks of the *normal path* are created. The *target* and *origin* of normal edges define the start and the end of a block. The order of the edges is, in this case, irrelevant.

Edges are updated to block level, except for those that are a *parent* of some alternative path. These edges are connection edges between the normal and an alternative path. If there is a branch that causes the jump at the end of a block, it is moved there.

Finally the alternative paths are constructed. Alternative edges of the same parent edge are handled in the order in which they have been generated. Starting

from the target of the parent edge, new blocks are created. The start of a block is given by the target of an edge, and the end of a block is determined by the origin of the next alternative edge. The branch and label information is adapted to the alternative path, in order to avoid repeated labels and incoherent jumps.

As mentioned before, the parent edge connects the normal path with the alternative path.

The end of an alternative path

Alternative paths were introduced in the previous section and defined as *the path built by all alternative edges caused by the same parent edge*. Alternative paths implicitly define new basic blocks whose instructions are copies of the original instructions of the program. They are connected to the normal path through the parent edges.

The edge algorithm stops in two situations: when there are no more branches left to handle in the path and when there is an edge that is already in the edges list.

The edge recognition algorithm follows all execution paths. It stops either at the end of a path (in the traditional sense), or at a control flow join in the program. *Final* and *join* edges determine the exit points of an alternative path. Note that,

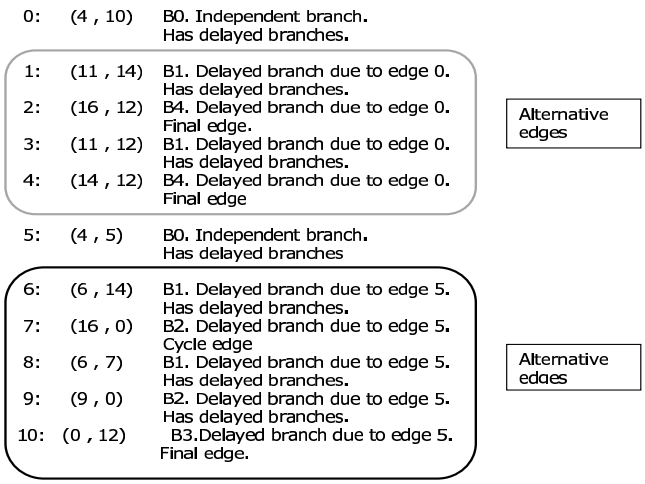

Figure 4.15: Result of the edges construction algorithm

although the current last edge of the set will be either a final or join edge, we may have these kind of edges earlier, due to the effect of conditional branches.

- **Join Edges**

  A join edge connects an alternative path to the normal path. Since this edge connects two paths, there exists a block in the normal path, which starts at the target position of the join edge.

- **Final Edges**

  In principle, join edges connect the alternative path to the normal path and final edges do not. For each final edge a new block is constructed which

contains instructions from the target of the edge, to the last instruction in the program. If such a block is already available in the normal path, the alternative path is connected with this block and returns here to the normal path as well.

4.5.3 Example

This section presents an example of the basic blocks construction. Figure 4.15 shows the result obtained by the edge construction algorithm for the program analyzed in the previous section. Figure 4.11 shows the control flow graph generated using the approach presented above.

Note that, at this point, no more knowledge about the original program is available than shown here. The edges are divided in three groups: the normal edges,

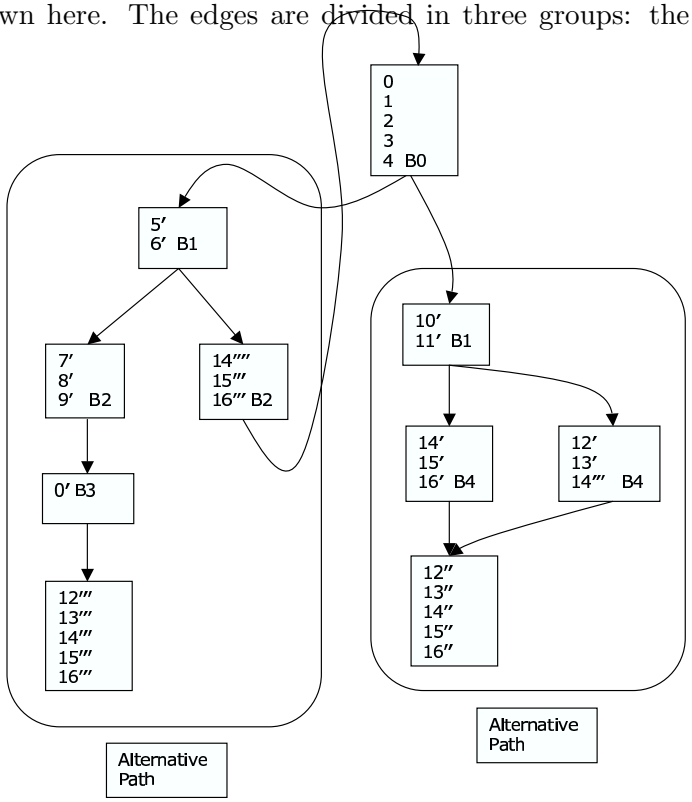

Figure 4.16: CFG

the alternative edges with parent edge (4, 10) and the alternative edges with parent edge (4, 5).

The paths are built as follows:

- **Normal Path**

  The normal edges are: (4, 10), (4, 5). Thus, a block ends at positions 4, and 9, and a block starts at positions 5, and 10. Thus, the following blocks are defined: {0, 1, 2, 3, 4}, {5, 6, 7, 8, 9} and {10, 11, 12, 13, 14, 15, 16}.

In general, these blocks would then be connected by the corresponding edges. However, since the only normal edges are (4, 10) and (4, 5) and they have delayed branches, they are parent edges of alternative sets. Therefore, they are used as normal branches to build the blocks, but not updated to block level edges.

- **Alternative Path (4, 10)**

  This path contains only four edges: (11, 14), (16, 12), (11, 12) and (14, 12). Its parent edge is (4 ,10). Therefore, the first block of the alternative path will be {10', 11'}. The edge (4, 10) connects the normal block {0, 1, 2, 3, 4} to the alternative block {10', 11'}.

  The next edge is (16, 12). Therefore, we have to build the copy block {14', 15, '16'}. Previous edge (11, 14) connects the two new blocks. Since the alternative edge (16, 12) is a *final edge*, it should connect the block {14', 15', 16'} to a new block {12", 13", 14", 15", 16"}. Before constructing it, we check whether there is an equivalent block {12, 13, 14, 15, 16} which is also final in the normal path. Since there is no such block, we build the copy and connect it to the path via the edge (16, 12).

  The next edge is (11, 12). Since the previous edge in the path was a final edge, this edge must connect an earlier block. The first thing to do is to look for this block. For this purpose, we look for an alternative block that ends at instruction 11 and whose branch is the one that created this edge. Block {10', 11'} fulfils these properties. Thus, the edge (11, 12) is an edge that connects the block {10', 11'} with another alternative block, which has not been built yet.

  The last edge in this path is (14, 12). The origin of this edge determines the last instruction of a new block, which starts at the target instruction of the previous edge, (11, 12). The new copy block {12', 13', 14"'} is built.

  The alternative edge (14, 12) is a *final edge*. Therefore, it should connect the block {12', 13', 14"'} to a new block {12"', 13"', 14"', 15"', 16"'}. However, since there is an equivalent block {12", 13", 14", 15", 16"} in this path, which is also final block, there is no need to construct a new one.

- **Alternative Path (4, 5)**

  This path is formed by five edges: (6, 14), (16, 0), (6, 7), (9, 0), (0, 12). The target of the parent edge is at position 5, and the origin of the first alternative edge is 6. So, the first block {5', 6'}, is connected to the normal path via the parent edge. The target of the first edge determines the beginning of the second block, which ends at the position given by the origin of the second edge (16, 0), and so the block {14"", 15"', 16"'} is built. Since the edge (16, 0) is a *join edge*, it connects the alternative path with the normal path. In particular, it connects block {14"", 15"', 16"'} with block {0, 1, 2, 3, 4}.

  The origin block of edge (6, 7) in this alternative path is {5', 6'}. This edge, together with (9, 0), defines the new block {7', 8', 9'}. Edge (9, 0) connects

this new block with block {0'}. Final edge (0, 12) connects this last block with a new final block {12"', 13"', 14"', 15"', 16"'}. Actually, an equivalent block of the other alternative path could be used, but this would add some exploration time to the algorithm.

This example has shown all the possible situations while constructing the basic blocks of an assembly program. Note that the delayed branches are the cause for the need of introducing the alternative paths and so the copies of basic blocks.

This approach makes it possible to construct a control flow graph from which it is easy to reproduce the execution of the program, and so is suitable for decompilation purposes.

4.6 Delay Resolution

Once the control flow graph has been created, it is relatively simple to resolve any other instructions which have delay slots. Nevertheless it is necessary to move instructions across CFG edges and to duplicate code. (Similar problems arise in global instruction scheduling and have been described before [Gup98]).

Consider instruction I in block b having d delay slots. Let p be the position of I inside b (starting from 0) and s the size of the block. As discussed in section 4.1.4, I should be moved to position $p + d$.

If $p + d < s$, I is removed from its current position and inserted at position $p + d$ in parallel to the existing instructions. If I is a single instruction at p, it is replaced by a NOP instruction, in order to maintain the instruction slot structure of the program. These NOP instructions are removed once the reordering is finished.

If $p + d >= s$, I is moved across the basic block boundary to all successors of b. For each successor b_k of b, instruction I will be inserted at position $p + d - s$ of b_k. If this position is outside of b_k again, instruction movement to all successors of b_k is repeated, until the final destination of I has been reached.

In case basic block b_k has other predecessors than b in the CFG, it is necessary to create a copy of b_k and insert the instruction there in order not to affect other paths in the program. If the new position occurs within the same basic block, then the moved instruction is inserted at its new position in parallel with any other instructions located at that position. If the new position does not occur within the same basic block, then I must be inserted in each successor block at the appropriate position.

If the number of delay slots is sufficiently large and the length of a successor block sufficiently short, then I might have to be propagated to successors of the successor, and so on. Whenever an instruction should be moved to a block with several predecessors, this block has to be duplicated and the instruction is moved to the copy. Furthermore, whenever a instruction I traverses a block with several predecessors before reaching its final destination, all blocks from the one having more than one predecessor until the destination of I must be copied. The next example will show delay resolution which needs basic block duplication.

In the following piece of code we assume that branches and loads have 2 delay slots.

```
0   [cond]  B label1
1       NOP
2       LDW  *A4++, A0
3       inst1
4       inst2
5       inst3
6       inst4
label1:
7       MV B0, A0
8       NOP
9       MV A0, A1
```

The branch at position 0 executes at position 2. Since it is a conditional branch the CFG contains two edges (2, 3) and (2, 7) and three basic blocks {0, 2}, {3, 6} and {7, 9}.

The load instruction at position $p = 2$ has $d = 2$ delay slots. As explained above, it should be moved to position $p + d = 4$. Since the destination position is outside the block {0, 2}, it is moved following the control flow. The successors of the current block are {3, 6} and {7, 9}.The relative position in the destination blocks is $p' = p + d - s$, where s is the size of block {0, 2} resulting in $p' = 1$.

Assuming that there are no other instructions with delays and ignoring at the moment other predecessor blocks the resolved code would be:

```
0       NOP
1       NOP
2  [cond]  B  label1
3       inst1
4       LDW  *A4++, A0 || inst2
5       inst3
6       inst4
label1:
7       MV B0, A0
8       LDW  *A4++, A0
9       MV  A0, A1
```

In case the condition *cond* is not fulfilled, register $A0$ is loaded at position 4. Several instructions later at position 8, register $A0$ is loaded again which did not happen in the original code. Therefore, a copy of block {7, 9} is needed, because there are other predecessor blocks.

The correct solution after delay resolution is:

```
0       NOP
1       NOP
2  [cond]  B  label1'
3       inst1
4       LDW  *A4++, A0 || inst2
5       inst3
```

```
6       inst4
label1:
7       MV B0, A0
8       NOP
9       MV A0, A1
10      B END
label1':
7'      MV B0, A0
8'      LDW  *A4++, A0
9'      MV A0, A1
END:
11
```

```
function handleDataConflict(reg, <J,d,P>)
begin
  t = getNewTemporaryRegister();
  m = instruction "MOV reg,t";
  add m to bundle at position P in new CFG;
  if reg is postincremented then
    a = instruction "ADD reg,1,reg";
    add a to bundle at position
      P in new CFG;
    remove postincrementing from J;
  fi;
  Handle other addressing modes similarly
  replace reg by t in J;
end
```

Figure 4.17: Handling Of Data Conflicts

4.6.1 Data Conflict Resolution

When moving an instruction to a different position in the program, it is important to take the possible data dependence conflicts into account and resolve them to maintain the semantic meaning of the original program.

Let I be an instruction at position p with a delay $d > 0$. For now, we assume the destination of I after delay resolution to be in the same block as I. Moving I from p to $p + d$ will cause a conflict if in the range $[p \dots p + d]$ there is any instruction which completes a redefinition of any of the operands of I. Consider the following example where we assume that multiplication (opcode MPY) has one delay slot and the load instruction (opcode LDW) has two.

```
0       MPY  B7, B9, B13  // 1 delay slot
     || ADD  A14, B4, B7
1       ADD  A10, A7, A13
     || LDW *A0++, B9     // 2 delay slots
2       NOP
3       NOP
```

b: actual block containing bundles in positions [start, end]
s: actual statement
d: delay of s
pos: position of s org˙pos: original position of s
V: list of variables causing a conflict

```
function moveInstruction(b, s, org_pos, pos, d, V)
begin
  if (d>0) then
    dest = pos+d; // destination of s
    rel_pos= pos-start; // relative position in block b
    rel_dest = dest-start; // relative destination in b
    if (dest <= end) then
      a = check_if_auxvar_needed(b, s, rel_pos, rel_dest, V);
      if (a) then
        handleDataConflict(b, s, org_pos, V);
      fi
      insert_statement(s, dest);
    fi
  else
    a = check_if_auxvar_needed(b, s, rel_pos, end, V)
    if (a) then
      handleDataConflict(b, s, org_pos, V);
    fi
    for all successors b  of b do
      if (b  has more than 1 predecessor) then
        b  =copy(b ); move_instruction(b  , s, org_pos, 0, d-(endpos+ 1));
      else move_instruction(b , s, org_pos, 0, d-(end-pos+1));
      fi
    od
  fi
end
```

Figure 4.18: Delay resolution algorithm

The new location for the MPY instruction will be position 1. This instruction defines register B13 and uses registers B7 and B9. We find the instruction `ADD A14,B4,B7` at position 0, which redefines B7. In order to preserve the result of the multiplication, we invent a new temporary register *temp0*, then we add an instruction initializing *temp0* at position 0 (the old position of MPY), and we modify the MPY instruction to be `MPY temp0,B9,B13`. Note that LDW instruction at position 1, which redefines the register B9, does not create a conflict because it does not change that register until two cycles later.

The LDW instruction does, however, have a side-effect other than loading a value into register B9. In this particular case, register A0 is post-decremented and that side-effect occurs in the same cycle as when the instruction is initiated. If we wish to move the LDW instruction two positions later, we should decrement the A0

register immediately (in case other instructions use A0 as an input operand) while using the old value for the LDW instruction. We can achieve the desired effect with a second new temporary register, *temp1*. The final version of the code sequence where no instructions have any delay slots is therefore as follows.

```
0       MV   B7, temp0
     || ADD  A14, B4, B7
1       MPY  temp0, B9, B13
     || ADD  A10, A7, A13
     || MV  A0, temp1
     || ADD  A0, 1, A0
2       NOP
3       LDW  *temp1,  B9
```

We propose a simple approach where we make copies of *all* input register operands used by the delayed instructions which cause conflicts. Later passes in the reverse compiler should include copy elimination to remove the redundant copy operations.

With our simplification, resolution of the data conflicts can be conveniently combined with moving instructions to their final positions in the delay-free version of the code. It traverses the CFG as produced by the combined edge detection and basic block construction algorithm, and creates a new CFG.

Figure 4.18 shows the main aspects of the algorithm for moving an instruction.

4.7 Sequentialization

After eliminating delays and removing data conflicts, we have a form of program where several instructions may occupy the same execution slot. That is, the instructions are organized into bundles. In order to remove this instruction level parallelism, the instruction bundles need to be rewritten in a sequential form. However we need to be careful because one instruction in the bundle can define a new value for a register while another instruction uses the same register as an input operand.

A simple approach to sequentialize the bundle is to write the instructions in the same order as they appear in the bundle, but modifying the instructions to use copies of input registers whenever needed to avoid a conflict. The function *sequentializeBundle* shown in Figure 4.19 implements the simple sequentialization.

4.8 Experimental Results

This section presents some results obtained by the implementation of the algorithms explained in this chapter. These evaluations have been performed on hand-written DSP code.

Table 4.1 shows the results of the CFG construction algorithm on several test programs. All the programs used contain at least one branch. Column 1 shows the total number of branches, and the number of conditional branches in parentheses. Columns 2 and 3 give the number of loops in the program and their lengths. Column 4 shows the number of basic blocks in the constructed control flow graph, and

```
function sequentializeBundle(bun)
begin
  foreach register r do
    registerMap[r] = r;
  od;
  S = ∅;
  defRegs = ∅;
  foreach instruction I in bun do
    inRegs = { r | I uses register r };
    foreach r ∈ inRegs ∩ defRegs do
      // resolve the conflict
      if registerMap[r] = r then
         t = getNewTemporaryRegister();
         m = instruction "MOV r,t";
         registerMap[r] = t;
         S = m + S;
      fi;
      replace occurrences of r in I used as
         an input register by registerMap[r];
    od;
    outRegs = { r | I defines register r };
    defRegs = defRegs ∪ outRegs;
    S = S + I;
  od;
  return S
end
```

Figure 4.19: Sequentialization Algorithm

Program	Branches (Cond)	Loops	Loop length	Blocks	Block copies	Edges	Bundles original	Bundles CFG	Stmts Original	Stmts CFG	Temp registers
autcor	2 (1)	1	8	4	1	5	30	38	116	211	13
bitrev	1 (1)	1	7	4	1	5	25	32	76	129	9
blk_move	3 (3)	1	2	7	4	7	33	33	46	51	0
dotprod	6 (5)	1	1	9	6	13	18	25	81	128	0
gouraud	2 (2)	1	4	5	2	5	29	29	97	99	3
idct	2 (2)	1	11	7	2	10	53	75	202	384	11
iir	2 (2)	1	5	6	3	7	37	42	119	179	12
iircas4	2 (1)	1	4	6	3	7	23	27	84	126	3
latanal	2 (1)	1	3	6	3	7	21	24	83	121	5
latsynth	3 (1)	1	2	5	2	6	14	17	52	55	0
max	2 (2)	1	3	6	3	7	25	28	92	145	11
minerror	1 (1)	1	9	4	1	5	27	36	99	206	36
vecsumsq	6 (6)	1	1	9	6	13	20	28	86	133	0
w_vec	3 (3)	1	2	7	4	7	32	35	88	101	5
dct	2 (2)	2	11	7	2	10	48	70	194	368	25
fir4	3 (3)	2	16	10	4	14	34	56	130	282	18
radix2	3 (3)	2	19	10	4	14	41	67	176	355	13

Table 4.1: Experimental Results CFG Construction

column 5 states how many of those blocks were caused by instruction copying while creating the basic blocks of the CFG or while moving delayed instructions. Column 6 gives the number of edges in the CFG. Finally, columns 7 to 11 provide a code size comparison between the original assembler code and the code described by the CFG, as well as the number of variables that have been added in order to resolve data conflicts while removing the delays.

The following section introduces a concrete example showing the control flow graph obtained using the techniques presented in this paper as well as a simple form of C translation of the code.

4.8.1 Example

Figure 4.20 shows the assembler code of a lattice filter for the TIMS320C60x architecture. To simplify the presentation, the code omits execution unit information. This example contains 3 delayed branches, all of which jump to label *LOOP*. The CFG generated by our program is shown in figure 4.21. We can observe that the second and third blocks of the graph contain the two first iterations of the loop. They are originated by the first two branches in the assembler code. The fourth block contains the actual loop body. Because of the delays associated with the multiplication and load instructions, the two first iterations of the loop are not identical to the remaining iterations. Due to a data conflict between instructions `MV A3, A0` and `MPYHL A4, A0, A2`, temporary variables have been added when moving the latter instruction.

Finally, figure 4.22 shows the C code that we would obtain from our CFG after sequentialization of the instruction bundles using a simple direct translation from our IR into C. (An optimizing compiler should find many opportunities for simplifying the code.)

4.9 Related Work

4.9.1 CFG Construction

Computing the CFG in a compiler from a function's intermediate representation is straightforward and described in any compiler book [ASU86, App98]. Computed branches and mixing of code and data make CFG construction more difficult or even impossible for binary programs or programs written in assembly language.

This section introduces several approaches that handle the different problems discussed in this chapter.

Cristina Cifuentes - *Decompilation Techniques*

Cristina Cifuentes [Cif94] describes in detail CFG construction in her thesis about reverse compilation. Since the considered architecture is the INTEL 80286, the CFG construction is relative easy. No delay slots make the process equivalent to the CFG construction in a high-level program.

```
latsynth:
         B     LOOP
||       ZERO  A7
||       ZERO  B7
         MVK   7, A1
||       ADD   B4, B4, B1
         B     LOOP
||       ADD   B1, A6, A6
||       ADD   B1, B6, B6
         ADD   3, B4, B0
LOOP:
   [B0]  B     LOOP
||       SHR   A2, 16, A5
||       MPY   1, B1, B5
||       MPYHL A4, A0, A2
||       MV    A3, A0
||       LDH   *--A6, A3
||       LDH   *--B6, B7
   [A1]  ADD   -1, A1, A1
|| [B0]  ADD   -1, B0, B0
|| [!A1] STH   B4, *+B6[7]
||       ADD   A5, B5, B4
|| [B0]  SUB   A4, A7, A4
||       MPY   1, B7, B1
||       MPY   B7, A3, A7
         SHR   A4, 16, A6
         STH   A6, *+B6[6]
```

Figure 4.20: Example Assembler Code

Cooper et al. - *Building a Control Flow Graph from Scheduled Assembly Code*

As far as we know, no other algorithm for constructing the CFG of programs with delayed branches which removes delay slots has been proposed before. The work closest to the one presented here is the CFG construction algorithm by Cooper et al. [CHW02]. They present an algorithm for building a correct CFG from scheduled assembly code that includes branches in branch delay slots.

The algorithm works by building an approximate CFG and then refining it to reflect the actions of delayed branches.

In general, CFG construction takes three steps. The first step partitions the code into a set of basic blocks (maximal length sequences of straight-line code). These blocks are defined by what we have called *independent* branches. The code is scanned and for each independent branch, a counter is set, which will be decremented when moving to the next instruction. When either the counter is 0 or a label is found, a block ends. If the current block ends before the counter reaches zero, the counter is discarded without adding edges to the branchs targets.These blocks become the nodes in the cfg.

The second step looks at the branches in the code and fills in the cfgs edges to represent the flow of control. Here, only the branches that have defined the blocks are considered.

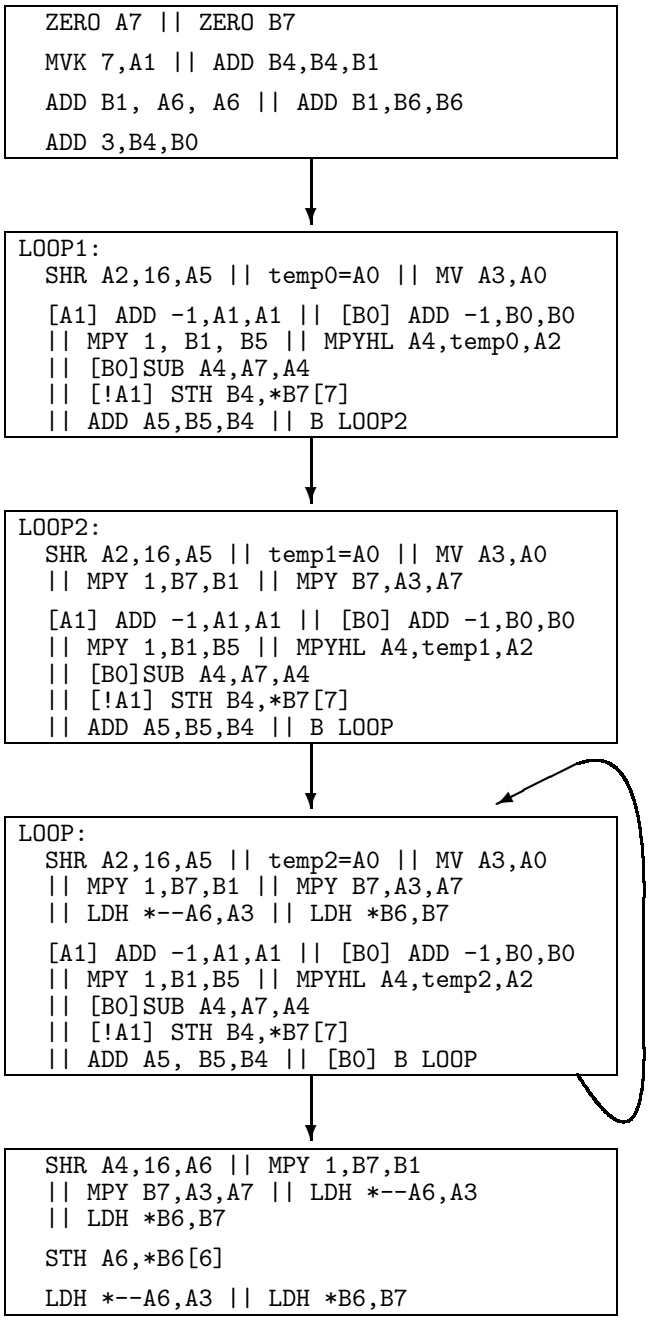

Figure 4.21: CFG for Figure 4.20 after Delay Resolution

```
        A7 = 0;
        B7 = 0;
        A1 = 7;
        B1 = B4+B4;
        B0 = 3+B4;
        temp0 = 0;
        A5 = A2>>16;
        A0 = A3;
        if (A1) A1 = (-1)+A1;
        if (B0) B0 = (-1)+B0;
        B5 = 1*B1;
        A2 = (A4>>16)*(temp0 & 0xFFFF);
        if (B0) A4 = A4-A7;
        if (!A1) *(B7+7) = B4;
        B4 = A5+B5;
        goto LOOP2;
LOOP2:
        temp1 = 0;
        A5 = A2>>16;
        A0 = A3;
        B1 = 1*B7;
        A7 = B7*A3;
        if (A1) A1 = (-1)+A1;
        if (B0) B0 = (-1)+B0;
        B5 = 1*B1;
        A2 = (A4>>16)*(temp1 & 0xFFFF);
        if (B0) A4 = A4-A7;
        if (!A1) *(B7+7) = B4;
        B4 = A5+B5;
        goto LOOP;
LOOP:
        temp2 = 0;
        A5 = A2>>16;
        A0 = A3;
        B1 = 1*B7;
        A7 = B7*A3;
        A3 = *(--A6);
        B7 = *B6;
        if (A1) A1 = (-1)+A1;
        if (B0) B0 = (-1)+B0;
        B5 = 1*B1;
        A2 = (A4>>16)*(temp2 & 0xFFFF);
        if (B0) A4 = A4-A7;
        if (!A1) *(B7+7) = B4;
        B4 = A5+B5;
        if (B0) goto LOOP;
        A6 = A4>>16;
        B1 = 1*B7;
        A7 = B7*A3;
        A3 = *(--A6);
        B7 = *B6;
        *(B6+6) = A6;
        A3 = *(--A6);
        B7 = *B6;
```

Figure 4.22: C Code for Figure 4.20

The third step consists on traversing the CFG maintaining a list of pending control-flow instructions with a countdown timer for each that shows when it will activate. When a counter reaches zero, it breaks the block at that point, adds an edge from the shortened block to the remainder of the block, and adds an edge from the shortened block to each of the targets of the activated branch. The algorithm continues in this way, processing blocks until no block has a new branch counter.

The main difference of this edge construction method to the one described in this chapter is the order in which the tasks are done. We consider all branches from the beginning, whereas Cooper et al. use only independent branches. This difference affects the further CFG construction.

The fact that their algorithm was not particularly intended for decompilation causes that they do not replace delayed branches but just insert all control flow graph edges to the existing instructions. This algorithm is helpful for program understanding. However, if the CFG is used as the basis for further analyses, the delay slots of instructions may have to be taken into account.

Debray et al. - Unpredication, Unscheduling, Unspeculation: Reverse Engineering Itanium Executables

Debray et al. [SD05] techniques to reverse engineer optimized Itanium binaries. EPIC (Explicitly Parallel Instruction Computing) architectures, exemplified by the Intel Itanium, support a number of advanced architectural features. Thus, compilers perform several low-level optimizations to take full advantage of the architecture. In particular, there are three optimizations that can significantly improve program performance: instruction scheduling, predication (refers to the selective elimination of conditional branches in favor of predicated instructions), and speculation (refers to the early execution of memory operations in order to hide their latency).

This paper describes techniques to undo some of the effects of such optimizations and thereby improve the quality of reverse engineering such executables. In particular, and being the most relevant deoptimization in this context, the goal of unscheduling is to group together related instructions that may have been separated during scheduling (it is possible for this to also group together code fragments that had been separate in the original program).

Most of the work is focused on unpredication and does not consider branches or delay slots.

Sutter et al. - *On the Static Analysis of Indirect Control Transfers in Binaries*

Sutter et al. [SBB+00] present techniques for improving analysis of indirect branches

The reconstruction of a CFG is very hard when indirect control transfers occur (control transfers through a memory or register operand as used when calling a procedure through a procedure pointer, or when transferring control through an address table).

In order to reconstruct the CFG, a careful analysis is needed to find out what the targets of the indirect control transfers are. However, the necessary analyses, including constant propagation, require an CFG themselves, which leads to a phase

ordering problem. The solution proposed by Sutter et al. is to start with a very conservative CFG, where every indirect control transfer instruction is replaced by a control transfer to an artificial node called the *hell node*.

After constant propagation, some of the edges to hell nodes (hereafter called hell edges) can be replaced by edges to regular nodes, allowing for a more detailed constant propagation, which in turn might eliminate additional hell edges, etc. After a few iterations, this process con- verges to a stable CFG with fewer hell edges.

They show how a hell node can be used to model unknown control flow behavior in a conservative approximation of the CFG and how the CFG can be refined during the analysis and transformation of the binary.

The construction of the CFG in our decompiler is also performed in different stages. A memory analysis helps improving the quality of the control flow graph constructed as explained in the following sections. Thus, indirect jumps are treated later and not in the scope of this thesis.

4.9.2 Basic Block Construction

Ammarguellat - *A Control-Flow Normalization Algorithm and its Complexity*

Basic block duplication also occurs at control flow graph normalization. CFG normalization facilitates program transformations, program analysis and automatic parallelization.

Ammarguellat [Amm92] presents a method for normalizing the control-flow of programs to facilitate program transformations, program analysis, and automatic parallelization. The constructed CFG is normalized into single-entry, single-exit while loops, and all goto's are removed. This method avoids problems of code replication that are characteristic of node-splitting techniques.

Erosa - *Taming Control Flow: A Structured Approach to Eliminating goto Statements*

Goto statements can be eliminated and transformed to structured control flow.

Erosa [EH94] describes an algorithm to structure C programs by eliminating all goto statements. The method works directly on a high-level abstract syntax tree (AST) representation of the program and could easily be integrated into any compiler that uses an AST-based intermediate representation. The actual algorithm proceeds by eliminating each goto by first applying a sequence of goto-movement transformations followed by the appropriate goto-elimination transformation. Different goto structures are classified. A goto-elimination transformation replaces a goto structure by another structure without the goto statement . A goto movement transformation replaces a goto structure with another goto structure.

Janssen - *Controlled Node Splitting*

Janssen and Corporaal minimize the number of duplicated nodes with controlled node splitting [JC96].

To exploit instruction level parallelism in programs over multiple basic blocks, programs should have reducible control flow graphs. However not all programs satisfy this property.

Janssen and Corporaal present a new method, called Controlled Node Splitting (CNS), for transforming irreducible control flow graphs to reducible control flow graphs. CNS duplicates nodes of the control flow graph to obtain reducible control flow graphs performing a minimum number of splits and a minimum number of duplicates.

4.9.3 Delay Resolution Problems

N. Ramsey and C. Cifuentes - *A Transformational Approach to Binary Translation of Delayed Branches*

Cifuentes and Ramsey [RC03] develop a method for identifying problematic cases and translate them into not problematic cases.

A very interpreter for the source machine is written, which specifies at the register-transfer level, how the machine executes instructions, including delayed branches. Then, the interpreter is transformed into an interpreter for a target machine without delayed branches. The transformation of the instructions becomes the algorithm for binary translation.

This approach has been developed for the translation from SPARC binary code to Pentium. Since all transformations are realized at machine code level, they model the behavior of the program counter PC in the presence of delayed branches. Branches can have here only one delay slot in which a branch can also be declared. There is no indication on how to apply these transformations in a decompiler to high level language and no other delayed instructions are taken into account.

Gupta et al. - *A Code Motion Framework For Global Instruction Scheduling*

Code motion across basic block boundaries leads to code duplication. This problem is common in compilers and well studied. Gupta [Gup98] presents a code motion framework for global instruction scheduling which eliminates delay slots.

The scheduling through delay elimination is as follows: First, a simple local basic block scheduling is performed and then all delay slots remaining in the code are targeted for elimination through global code motion.

The elimination of a delay slot is carried out in two steps: a goal oriented search which identifies a global code motion or a cascade of code motions that eliminate the delay without introducing additional delay slots in critical areas of the program; and a transformation step in which the code motion is performed along with compensation code placement and application of code optimizations enabled by code motion.

Krall et al. - *Ultra Fast Cycle-accurate Compiled Emulation of Inorder Pipelined Architectures*

Krall et al. [KFH05] describe an ultra fast compiled emulator for an architecture with delayed instructions. Emulation of delayed instructions leads to the execution of parts of an instruction in succeeding basic blocks. To keep code duplication small a combination of basic block duplication and conditional execution is used.

Chapter 5

Software De-pipelining

As introduced in section 2.3.2, software pipelining [Lam88] is a loop optimization technique used to speed up loop execution. It is widely implemented in optimizing compilers for VLIW and superscalar processors that support instruction level parallelism. Software de-pipelining is the inverse of software pipelining. The goal of software de-pipelining is to rewrite a software pipelined loop in the original sequential form. Section 5.1 describes why software pipelining represents a problem in the context of decompilation. An approach for software de-pipelining simple loops using the control flow graph of the program is described in section 5.2.

Additional optimizations are performed on software pipelined loops when not only the speed but also the code size is critical, as it is in the case of digital signal processors. Section 5.3 introduces a widely used technique for code reduction of software pipelined loops: prolog and epilog collapsing. This optimization requires an alternative approach for de-pipelining. The general algorithm is presented in section 5.5. Sections 5.6 and 5.7 discuss software de-pipelining techniques for other types of loops and for loops that have been optimized in different ways as well. The experimental results obtained with the software de-pipelining approach are shown in section 5.8.

5.1 Introduction

When constructing the control flow graph of an assembler program for decompilation, the presence of software pipelined loops may significantly increase the size of the graph. Furthermore, the decompilation process becomes more complex due to the larger size of the CFG and the resulting code is difficult to understand.

Software pipelined loops with small loop bodies written in a language where branches have a large delay need many branches to be implemented. Furthermore, since the goal of software pipelining is to minimize the initiation interval (and so, the length of the loop kernel), it is usual to have loops with small bodies.

5.1.1 Example

Consider the following C code for the fixed-point dot product of two arrays.

```
int dotp(short a[], short b[])
{
   int i, sum=0;
   for(i=0; i<100; i++)
       sum += a[i] * b[i];
   return(sum);
}
```

The sequential assembly code for this function is shown in figure 5.1. A parallel version of this code can be found in 5.2.

Note that, although both versions of the dot product loop in 5.1 and 5.2 are not software pipelined, the goal of software de-pipelining is to generate code in the form shown in figure 5.1, where there are no parallel instructions. This sequential form is more convenient for decompilation, because all parallel instructions in the program will have to be sequentialized anyway before translation into a high-level language.

```
0      MVK .S1 100, A1 ; set up loop counter
1      ZERO .L1 A7 ; zero out accumulator
LOOP:
2      LDH .D1 *A4++,A2 ; load ai from memory
3      LDH .D1 *A3++,A5 ; load bi from memory
4-7    NOP 4 ; delay slots for LDH
8      MPY .M1 A2,A5,A6 ; ai * bi
9      NOP ; delay slot for MPY
10     ADD .L1 A6,A7,A7 ; sum += (ai * bi)
11     SUB .S1 A1,1,A1 ; decrement loop counter
12   [A1]B .S2 LOOP ; branch to loop
13-17  NOP 5 ; delay slots for branch
;  Branch occurs here
```

Figure 5.1: Non-Parallel Assembler Code for Fixed-Point Dot Product

Figure 5.3 shows the basic structure of the control flow graph for the code in 5.1 after delay resolution. This graph has 3 blocks and 3 edges and the loop in the code is recognized easily.

The code of the dotprod algorithm after software pipelining is shown in figure 5.4. Due to the complexity of this code, it has been written in form of a table, where all instructions in a row are executed in parallel. This format will be kept for the rest of the chapter.

Note that temporary variables that may have been introduced to remove data dependence conflicts while resolving the delays, are ignored. It is necessary to consider the original register names in order to recognize instructions that are equal.

The software pipelined code has a large number of branches. Therefore, the control flow graph after delay resolution, with 13 basic blocks and 6 edges is much more complex than the one for the non-pipelined version (see figure 5.5). Here, although the CFG clearly shows a loop around block `B6`, it is not obvious that blocks `B1-B5` are also part of it.

```
0      MVK .S1 50,A1 ; set up loop counter
    || ZERO .L1 A7 ; zero out sum0 accumulator
    || ZERO .L2 B7 ; zero out sum1 accumulator
LOOP:
1      LDW .D1 *A4++,A2 ; load ai & ai+1 from memory
    || LDW .D2 *B4++,B2 ; load bi & bi+1 from memory
2      SUB .S1 A1,1,A1 ; decrement loop counter
3    [A1] B .S1 LOOP ; branch to loop
4-5    NOP 2
6      MPY .M1X A2,B2,A6 ; ai * bi
    || MPYH .M2X A2,B2,B6 ; ai+1 * bi+1
7      NOP
8      ADD .L1 A6,A7,A7 ; sum0+= (ai * bi)
    || ADD .L2 B6,B7,B7 ; sum1+= (ai+1 * bi+1)
; Branch occurs here
9      ADD .L1X A7,B7,A4 ; sum = sum0 + sum1
```

Figure 5.2: Parallel Assembler Code for Fixed-Point Dot Product

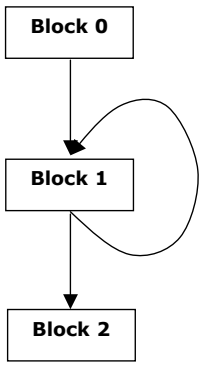

Figure 5.3: Control Flow Graph of Dot Product Sequential Version

Since the CFG is the basis for the translation into the high-level language, the more complex it is at the time of code generation, the more difficult to read the C code will be. Therefore, it is desirable to have sequential loops rather than software pipelined loops in the program. Software de-pipelining replaces software pipelined loops with their sequential form.

5.1.2 Limitations

The algorithms described in this work have been developed for software de-pipelining single loops or the innermost loop of a group of nested loops. In addition, following assumptions about the loops are made:

1. Only natural loops are considered.

2. The number of iterations of the loop is explicitly specified.

0	MVK 43,A1	LDW *A4++,A2	LDW *B4++,B2	ZERO A7	ZERO B7			
1	[A1]SUB A1,1,A1	LDW *A4++,A2	LDW *B4++,B2					
2	[A1]SUB A1,1,A1	LDW *A4++,A2	LDW *B4++,B2	[A1]B LOOP				
3	[A1]SUB A1,1,A1	LDW *A4++,A2	LDW *B4++,B2	[A1]B LOOP				
4	[A1]SUB A1,1,A1	LDW *A4++,A2	LDW *B4++,B2	[A1]B LOOP				
5	[A1]SUB A1,1,A1	LDW *A4++,A2	LDW *B4++,B2	[A1]B LOOP	MPY A2,B2,A6	MPYH A2,B2,B6		
6	[A1]SUB A1,1,A1	LDW *A4++,A2	LDW *B4++,B2	[A1]B LOOP	MPY A2,B2,A6	MPYH A2,B2,B6		
7 LOOP	[A1]SUB A1,1,A1	LDW *A4++,A2	LDW *B4++,B2	[A1]B LOOP	MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
8					MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
9					MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
10					MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
11					MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
12					MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
13							ADD A6,A7,A7	ADD B6,B7,B7
14							ADD A6,A7,A7	ADD B6,B7,B7
15								ADD A7,B7,A4

Prolog
Kernel
Epilog

Figure 5.4: Dot Product Assembler

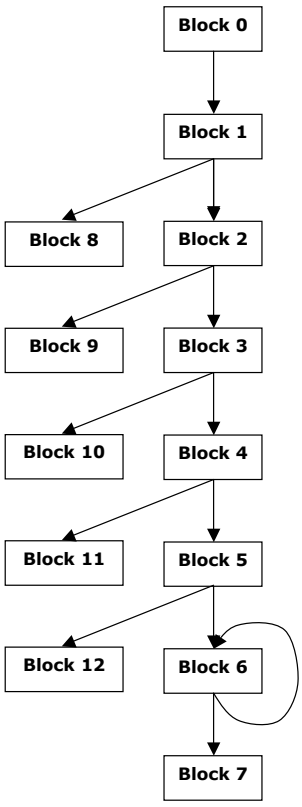

Figure 5.5: Control Flow Graph of Dot Product Software Pipelined Version

The last assumption is made for clarity reasons. Section 5.7 discusses how to handle not counted loops.

5.1.3 Reversing Software Pipelining: Problems

In order to recover the original sequential form of a software pipelined loop, it is necessary to examine its different parts and their relations to the original loop.

As introduced in section 2.3.2, a software pipelined loop has 3 parts:

- **Prolog:** Contains the initial instructions of several iterations.
- **Loop Kernel:** Contains all instructions of the loop body.
- **Epilog:** Contains the last instructions of several iterations.

Let us consider the example in 5.4. The parts described above are marked in different colors in the figure.

Note that not all instructions before the loop kernel belong to the prolog, as well as not all instructions after the kernel belong to the epilog. It is necessary to determine which instructions belong to the software pipelined loop in order to reverse the optimization. Thus, prolog and epilog need to be located in the code.

Loop Schedule

Since the loop kernel of the software pipelined loop is built from pieces of different iterations of the original loop, all loop instructions are found there. However, they are not necessarily written in the same order as they would be in a sequential version of the loop. Thus, in order to software de-pipeline a loop, the correct order of the kernel instructions must be determined, which preserves the semantics of the loop despite sequentialization.

Loop Counter

After software pipelining, several instructions start outside of the loop, which eventually makes the initial value of the loop counter change. In the dot product example register `A1` contains the loop counter. In the sequential version (see figure 5.1), `A1` is initialized to 50, whereas in the software pipelined version (see figure 5.4) this initial value is 43.

The sequentialization of the software pipelined loop requires the recovery of the initial value of the loop counter. However, not all loops have a loop counter that is increased or decreased by a constant every iteration. How to handle other types of loops will be discussed later (see section 5.7).

5.2 Software De-pipelining Approach

This section presents an approach for performing the software de-pipelining of pipelined loops. As discussed in the previous section, there are basically three subproblems to be considered in the reconstruction of a software pipelined loop:

1. Locate prolog and epilog instructions.

2. Schedule loop instructions.

3. Compute the new initial value of the loop counter.

Once these problems are solved, prolog and epilog instructions must be removed from the code, the loop kernel is substituted by the new loop following the schedule and the initial value of the loop counter is changed. Then, the deoptimization is finalized.

5.2.1 Notation and Concepts

In this section, several concepts are defined, which will be used in the following sections.

Prephase and Prelude

In the CFG after delay resolution, the prolog instructions of the software pipelined loop may be distributed in several blocks.

The prelude blocks are determined by the branch instructions from the prolog. Basically, each basic block in the prelude corresponds to a loop iteration that starts outside the loop kernel.

The prephase contains those prolog instructions whose position after delay resolution was located above the loop entry.

Postlude

Is built by the set of blocks that contain the epilog of the software pipelined loop. The instructions in these blocks are the finishing instructions of the last iterations of the loop.

Note that prelude and postlude are alternative names for prolog and epilog and can be found in the literature representing the same concepts. However, in this work the different names are used in different contexts intentionally, in order to avoid confusions.

Thus, *prelude* and *postlude* will be used when referring to the control flow graph whereas *prolog* and *epilog* will allude to the assembler code.

Example

Figures 5.6 and 5.7 show the contents of the blocks of the control flow graph of figure 5.5. The instructions of prolog, kernel and epilog are marked in different colors. The following sections will refer to these figures.

Block 0

0	MVK 43,A1			ZERO A7	ZERO B7			
1	[A1]SUB A1,1,A1							
2	[A1]SUB A1,1,A1							
3	[A1]SUB A1,1,A1							
4	[A1]SUB A1,1,A1	LDW *A4++,A2	LDW *B4++,B2					
5	[A1]SUB A1,1,A1	LDW *A4++,A2	LDW *B4++,B2					
6	[A1]SUB A1,1,A1	LDW *A4++,A2	LDW *B4++,B2		MPY A2,B2,A6	MPYH A2,B2,B6		

Block 1

LOOP	[A1]SUB A1,1,A1	LDW *A4++,A2	LDW *B4++,B2	[A1]B LOOP1 [!A1]B lab0	MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7

Block 2

LOOP1	[A1]SUB A1,1,A1	LDW *A4++,A2	LDW *B4++,B2	[A1]B LOOP2 [!A1]B lab1	MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7

Block 3

LOOP2	[A1]SUB A1,1,A1	LDW *A4++,A2	LDW *B4++,B2	[A1]B LOOP3 [!A1]B lab2	MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7

Block 4

LOOP3	[A1]SUB A1,1,A1	LDW *A4++,A2	LDW *B4++,B2	[A1]B LOOP4 [!A1]B lab3	MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7

Block 5

LOOP4	[A1]SUB A1,1,A1	LDW *A4++,A2	LDW *B4++,B2	[A1]B LOOP5 [!A1]B lab4	MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7

Block 6

LOOP5	[A1]SUB A1,1,A1	LDW *A4++,A2	LDW *B4++,B2	[A1]B LOOP6 [!A1]B lab5	MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7

Block 7

lab0		LDW *A4++,A2	LDW *B4++,B2		MPY A2,B2,A6	MPYH A2,B2,B6		
		LDW *A4++,A2	LDW *B4++,B2		MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
		LDW *A4++,A2	LDW *B4++,B2		MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
		LDW *A4++,A2	LDW *B4++,B2		MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
					MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
					MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
							ADD A6,A7,A7	ADD B6,B7,B7
							ADD A6,A7,A7	ADD B6,B7,B7
								ADD A7,B7,A4

Figure 5.6: Control Flow Graph of Dot Product Sequential Version

Block 8

lab1		LDW *A4++,A2	LDW *B4++,B2		MPY A2,B2,A6	MPYH A2,B2,B6		
		LDW *A4++,A2	LDW *B4++,B2		MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
		LDW *A4++,A2	LDW *B4++,B2		MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
		LDW *A4++,A2	LDW *B4++,B2		MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
					MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
					MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
							ADD A6,A7,A7	ADD B6,B7,B7
							ADD A6,A7,A7	ADD B6,B7,B7
								ADD A7,B7,A4

Block 9

lab2		LDW *A4++,A2	LDW *B4++,B2		MPY A2,B2,A6	MPYH A2,B2,B6		
		LDW *A4++,A2	LDW *B4++,B2		MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
		LDW *A4++,A2	LDW *B4++,B2		MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
		LDW *A4++,A2	LDW *B4++,B2		MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
					MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
					MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
							ADD A6,A7,A7	ADD B6,B7,B7
							ADD A6,A7,A7	ADD B6,B7,B7
								ADD A7,B7,A4

Block 10

lab3		LDW *A4++,A2	LDW *B4++,B2		MPY A2,B2,A6	MPYH A2,B2,B6		
		LDW *A4++,A2	LDW *B4++,B2		MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
		LDW *A4++,A2	LDW *B4++,B2		MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
		LDW *A4++,A2	LDW *B4++,B2		MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
					MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
					MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
							ADD A6,A7,A7	ADD B6,B7,B7
							ADD A6,A7,A7	ADD B6,B7,B7
								ADD A7,B7,A4

Block 11

lab4		LDW *A4++,A2	LDW *B4++,B2		MPY A2,B2,A6	MPYH A2,B2,B6		
		LDW *A4++,A2	LDW *B4++,B2		MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
		LDW *A4++,A2	LDW *B4++,B2		MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
		LDW *A4++,A2	LDW *B4++,B2		MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
					MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
					MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
							ADD A6,A7,A7	ADD B6,B7,B7
							ADD A6,A7,A7	ADD B6,B7,B7
								ADD A7,B7,A4

Block 12

lab5		LDW *A4++,A2	LDW *B4++,B2		MPY A2,B2,A6	MPYH A2,B2,B6		
		LDW *A4++,A2	LDW *B4++,B2		MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
		LDW *A4++,A2	LDW *B4++,B2		MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
		LDW *A4++,A2	LDW *B4++,B2		MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
					MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
					MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
							ADD A6,A7,A7	ADD B6,B7,B7
							ADD A6,A7,A7	ADD B6,B7,B7
								ADD A7,B7,A4

Figure 5.7: Control Flow Graph of Dot Product Sequential Version (cont.)

5.2.2 Locate Prephase, Prelude and Postlude Blocks

Finding the kernel of a software pipelined loop in the control flow graph is easy, because there is a backedge defining the loop body. Determining which other blocks are also a part of the software pipelined loop requires some more work.

The following observations will motivate the algorithm described at the end of this section.

Prelude

The prelude contains several initial iterations of the loop. Therefore, the prelude blocks must be predecessors of the loop kernel block. Furthermore, since prelude and kernel all belong to one single loop, there may not be any other blocks in the path that connects the first prelude block with the kernel block (see figure 5.5).

Thus, the last prelude block B_n is the predecessor of the kernel block. The next prelude block, B_{n-1} is the predecessor of B_n, B_{n-2} is the predecessor of B_{n-1}, and so on. It is obvious, that if a block is found by these method having more than one predecessors, it does not belong to the prelude. However, this condition does not guarantee that all found blocks do really belong to the prelude.

One approach to know when all prelude blocks have been found is to inspect the contents of the blocks and to look for branches at the end of them that were originally to the loop start. Since each prelude block corresponds to a loop iteration, at the end of each of them there must be a branch to the beginning of the loop. The labels have been modified in the construction process of the CFG, to avoid redundancies. However, it is possible to recover the original position the branch jumped to. Then, whenever a block is found which does not contain such a branch, the set of prelude blocks is complete.

An alternative approach for computing the number of blocks in the prelude (`preludeLength`) consists in using the length of the loop kernel (`kernelLength`) and the number of delay slots of branches. The implementation of loops in languages where branches have delay slots is subject to certain limitations. For example, consider the assembler language of the TIC62 DSP. Here, branches have 5 delay slots. Using only one branch it is impossible to implement a loop of length 3:

Example

Let `B` be the branch, which is located at position `p` in the program.

```
p         B
p+1
p+2
  LOOP:
p+3
p+4
p+5       ;loop ends here
```

Since the branches in the TIC62 DSP have 5 delay slots, `B` finishes execution at position `p` + 5. Thus, the loop ends at position `p` + 5 as well. Since we want the loop to have length 3, its entry must be at position `p` + 3. The branch `B` is then outside the loop body, so that it will only be executed once. Therefore, it does not implement a loop.

Now, let us consider a second branch `B'` at position `p` + 3.

```
p         B
p+1
p+2
  LOOP:
p+3       B'
p+4
p+5       ;loop ends here
```

The distance between `B` and `B'` is 3. As discussed in section 4.1.4, this means that `B'` finishes execution 3 cycles later than `B`. Since the jump target of `B` is `LOOP`, `B'` will perform its jump at position `p` + 5. Thus, `B'` describes the same loop as `B`. Furthermore, since the distance between the first and the second execution of `B'` is 3 as well, it correctly describes the loop.

The example above shows that there is a relationship between delay of branches, length of the loop and number of branches necessary to implement it.

In general, a loop of length l smaller than the number of delay slots of the branch instructions aD, requires more than one branch to implement it. The number of branches nB can be computed by the formula

$$nB = \left\lceil \frac{aD}{l} \right\rceil$$

Note that the problem is equivalent to determining how many branches can be distributed in an interval of length aD when there must be a distance of l between consecutive branches.

A loop such that $l \geq aD$ cannot be implemented with delayed branches. Here, one branch is enough because it can be placed inside the loop body at a distance of aD from the loop end. Furthermore, if there were more branches inside the loop (delayed or not), there would be incorrect jumps. However, the given formula is correct in any case.

Inside the loop kernel there can only be one branch instruction. Note that we have proven that the distance between two consecutive branches must be equal to the length of the kernel. Thus, having more than one branch inside the loop is not possible.

Since there is one prelude block for each branch instruction in the prolog of the software pipelined loop, and we know that only one of the branches can

be in the kernel, the following formula describes the number of prelude blocks nP.

$$nP = \left\lceil \frac{aD}{l} \right\rceil - 1$$

Considering integer division, we have $nP = \frac{aD}{l} - 1$.

Since aD is a known constant in the decompiler and the loop kernel is also available, this formula provides a method to compute the number of prelude blocks without needing to inspect the block instructions looking for branches.

Prephase

Once the prelude has been defined, the predecessor of the first prelude block contains the prephase.

The first prelude block could only have another predecessor if there had been a branch jumping to a target in the middle of the prolog of the software pipelined loop. If this happens, the second predecessor of the prelude must contain the same loop instructions as the first one. Otherwise the software pipelining would not work.

In this case one of the predecessors can be used for de-pipelining. Once the loop has been sequentialized, the second block can be updated so that the same instructions are removed as in the first block.

Postlude

The postlude is made up of the blocks that are executed once the loop is exited. Therefore, the successor of the loop kernel belongs to the postlude. In case the branches in the prelude blocks are conditional, the loop can be exited before reaching the loop kernel. Then, the prelude blocks (or some of them) will have two successors. One of them will be another prelude block or the kernel, and the second one will be the exit block. These blocks also belong to the postlude.

Following these ideas, figure 5.8 presents the algorithms to locate prephase, prelude and postlude. The block containing the loop kernel (`loopKernel`) is assumed to be available. Other arguments of the functions `findPreludeAndPrephase` are

- *CFG* is the control flow graph of the assembler program, and
- aD is the number of delay slots of branch instructions in the given architecture. In this particular case, it is 5.

5.2.3 Schedule Loop Instructions

The main part of the software de-pipelining technique determines the order of the loop instructions.

```
function findPreludeAndPrephase (CFG, loopKernel, aD)
  begin
    Prelude = ∅
    kernelLength = length of the loop kernel (in bundles)
    number of blocks in the prelude = aD / kernelLength
    b = loopKernel
    for 0 to number of blocks in the prelude do
       b = predecessor of b
       add b to Prelude
    od
    Prephase = predecessor of b
  end
endfunction

function findPostlude (CFG, loopKernel, Prelude)
  begin
    Postlude = ∅
    b = successor of loopKernel different from loopKernel
    add b to Postlude
    b = first block from Prelude
    for 0 to number of blocks in the prelude do
       for all successors of b in CFG do
          succ = successor of b
          if succ ∉ Prelude then
             add succ to Postlude
          fi
       od
    od
  end
endfunction
```

Figure 5.8: Prephase, Prelude and Postlude Location

Since the kernel of the software pipelined loop has been created by overlapping several iterations, it contains pieces of original loop iterations that are in a different order than in the original loop. As mentioned in previous sections, all loop instructions are contained in the loop kernel, so the problem consists on scheduling the kernel instructions.

In order to determine the correct schedule, we observe the *first* iteration of the original loop. Due to the software pipelining, this iteration starts in the prephase or prelude and starts *before* the next iteration. Therefore, the first loop instruction that is found in the prelude must belong to the first iteration of the loop. Following the same principle, we conclude that for all loop instructions, their first copy belongs to the first iteration of the original loop.

Once the prelude has been analyzed, we look for the remaining loop instructions in the kernel. The algorithm for scheduling the loop instructions is shown in figure 5.9. Whenever several instructions of the same iteration are executed in parallel, they have to be sequentialized as done after the CFG construction. Therefore, we

```
function ScheduleLoopInstructions (CFG, loopKernel, Prephase, Prelude)
  NewLoop = create new loop body
  b = Prephase
  while b do
     for each instruction bundle I in b do
        for each instruction s in I do
          if (s  ∈  loopKernel AND s has not been scheduled yet) then
            insert s at the end of NewLoop
          fi
        od
     od
     if (b = Prephase) then
   b = first block in the prelude
     else
       b = next block in the prelude
     fi
   od
endfunction
```

Figure 5.9: Loop scheduling algorithm

have to check for data dependency conflicts as explained in section 4.7 and, in case there are any, solve them following the same ideas explained in 4.6.1.

5.2.4 Compute the New Initial Value of the Loop Counter

When software pipelining a loop, several iterations are *moved outside* of it. Therefore, as discussed in section 5.1.3, the number of loop iterations is changed.

In order to restore the original value of the loop counter, we need to compute how often the condition register is updated before entering a loop iteration. By simply scanning the code, the conditional register can be determined. This is the register involved in the branch condition.

There are several factors involved in the recovery of the original loop counter initial value (CV). This section discusses them separately and finalizes giving a formula that takes all of them into account.

It is assumed that the instruction that modifies the condition register has one of the following forms:

```
ADD R, n, R
SUB R, m, R
```

where `R` is the condition register and `n`, `m` are constant values.

It is also assumed that no other modification on the register is performed from its initialization until its last use by loop instructions.

Other cases will be discussed at the end of the section.

Number of Condition Register Modifications

The condition register is modified every iteration. This means, that for every branch instruction there must be an counter update instruction. Otherwise, some iterations would not be counted.

Since `ADD` and `SUB` instructions have no delay, after delay resolution they must be placed in the prephase. Thus, if nB is the number of branches originally in the prolog of the software pipelined loop (that is, the number of prelude blocks) and nU is the number of condition register update instructions in the prephase, nU must be at least nB. The difference $nB - nU$ is the number of iterations that are performed but not counted. Thus, $nB - nU$ must be added to CV.

A particular case are unconditional branches, since the iterations corresponding to these branches are not explicitly counted either. Since after sequentializaton all branches will be conditional, the iterations defined by unconditional branches need to be added to the loop counter.

When the Register is Modified

Another important aspect to take into account is the relative position of the update instructions towards the branch inside the loop. When sequentializing the loop, the branch instruction will become the last instruction of the loop. Since delays have been removed, any instruction placed after the branch will not belong to the loop. In particular, the instruction that modifies the loop counter will be placed before the branch, independently of its relative position towards the branch instruction before de-pipelining.

Consider the following situation:

```
        MVK 2, A1
LOOP:
        SUB A1, 1, A1
        [A1] B LOOP
        ; the branch executes here
```

In this case the branch will be taken only once, since in the second iteration `A1` will be 0.

On the other hand, if the loop is like

```
       MVK 2, A1
LOOP:
       [A1] B LOOP || SUB A1, 1, A1
       ; the branch executes here
```

the branch will be taken twice.

Thus, if the update instruction is in parallel with the branch instruction in the loop kernel, placing it before the branch in the sequentialized version makes the loop iterate one time less. Therefore, the initial value must be increased by 1.

However, if the branch and the update instruction are in parallel in the software pipelined loop, sequentializing the code writing the branch in the last position does not preserve the semantics of the code.

If the conditional register is used after exiting the loop (that is, if the register is not dead), its value will not be correct. In order to solve this, a temporary variable is introduced that holds the *old* value of the condition register. This variable is then used in the condition of the branch.

Thus, in the previous example, the resulting code would be as follows:

```
        MVK 2, A1
LOOP:
        MV A1, temp
        SUB A1, 1, A1
        [temp] B LOOP
        ; the branch executes here
```

Unconditional Branches

Unconditional branches may be used to implement the loop. The iterations corresponding to these branches are not explicitly counted. However, after sequentializaton all branches will be conditional. Therefore, they need to be added to the loop counter.

Recovery Formula

Taking the aspects described above into consideration, the new initial value of the loop counter NCV is given by the following formula:

$$NCV = CV + s \cdot nB + \epsilon,$$

where nB the number uncounted branches (conditional as well as unconditional), CV is the initial value of the register in the software pipelined loop, s is the stride in which the conditional register is modified (the sign of s corresponds to the kind of operation) and

$$\epsilon = \begin{cases} 1, & \text{if the branch instruction and the conditional register update} \\ & \text{instruction are in parallel in the loop kernel} \\ 0, & \text{otherwise} \end{cases}$$

Other Update Instructions

The assumptions on the instruction that modifies the condition register can be little loosen.

Instructions like

```
ADD R, A, R
SUB R, B, R
```

where `R` is the condition register and `A`, `B` are registers, can only be handled if a data dependence analysis proves that `A` and `B` are not modified in the software pipelined code. Then, the initial value of the loop counter would be replaced by a more complex expression like `R =` CV `+ A` $\cdot nB + \epsilon$.

If the condition register is updated by another type of expression, is modified in another way or depends on a register that contains no constant value, CV cannot be computed and the loop cannot be software de-pipelined.

5.2.5 Rewrite Loop

Once all the necessary information about the original loop has been collected and computed, the software pipelined code must be replaced by the sequential version. The following steps need to be performed

- Remove loop instructions from the prephase
- Remove loop body instructions from the postlude
- Delete prelude blocks
- Delete loop kernel
- Set new initial value counter
- Set `NewLoop` as successor of prephase and predecessor of postlude block

Where `NewLoop` is the sequential version of the loop.

5.2.6 Dot Product Example

This section shows how, parting from the code in 5.5, the code in 5.1 is recovered.

Loop Kernel

The CFG in figure 5.5 contains one backedge from and to block 6. Therefore, block 6 contains the loop kernel of the software pipelined loop.

Prelude and Postlude Detection

The length of the loop body is 1. Considering that branches have 5 delay slots, 5 branches are needed outside the loop kernel in order to implement the loop. Therefore, the prelude has 5 blocks. The last block of the prelude is the predecessor of block 6: block 5. So, the prelude contains the blocks 1-5. Block 0 contains the prephase of this loop.

The successors of the prelude blocks, together with the successor of the loop kernel constitute the postlude. Thus, the postlude holds blocks 7-12.

Loop Counter Detection

The loop branch is defined by the instruction

```
[A1] B  LOOP
```

Thus, the condition for the branch to execute is `A1=0`. The register `A1` is updated in each iteration by the instruction `[A1] SUB A1,1,A1` and not changed anywhere else. Thus, `A1` is the loop counter, which is initialized by the instruction `MVK 43, A1` in the prephase.

Instruction Schedule

The instructions on the loop kernel are

```
MPY A2,B2,A6
MPYH A2,B2,B6
[A1] SUB A1,1,A1
ADD A6,A7,A7
ADD B6,B7,B7
LDW *A4++,A2
LDW *B4++,B2
[A1] B LOOP
```

The first loop instruction appearing in the prephase is `[A1] SUB A1,1,A1`. Thus, it will be scheduled as the first instruction in the loop. Then, the two instructions

```
LDW  *A4++,A2     LDW  *B4++,B2
```

are displayed in parallel. Since it is aimed for a sequential loop, these parallel instructions need to be sequentialized. However, there is no conflict between them (none of them uses a register that the other defines), and the order is irrelevant.

To this point, the schedule looks like

```
[A1] SUB A1,1,A1
LDW *A4++,A2
LDW *B4++,B2
```

The next not scheduled instructions in the prephase are

```
MPY A2,B2,A6     MPYH A2,B2,B6.
```

Since they do not cause any data dependence conflicts either, they can be added to the schedule in any order. Now, the prephase has been analyzed and contains no more kernel instructions. Next, the first block of the prelude is considered.

Here, instructions

```
ADD A6,A7,A7      ADD B6,B7,B7
```

belong to the kernel and are scheduled next.

Now all kernel instructions have been scheduled. The branch is scheduled at the end of the loop, for the jump to be performed correctly. The new loop body is

```
[A1] SUB A1,1,A1
LDW *A4++,A2
LDW *B4++,B2
MPY A2,B2,A6
MPYH A2,B2,B6
ADD A6,A7,A7
ADD B6,B7,B7
[A1] B LOOP
```

New Initial Value of Loop Counter

The initial value of the loop counter is 43. In the prephase, it is modified by the instruction `[A1] SUB A1,1,A1` 6 times. In the loop kernel, the update instruction of the loop counter is parallel to the branch, which adds 1 to the counter. Therefore, the new initial value of the loop counter is 50.

Rewrite the Loop

The next step would be to remove the prelude blocks in the control flow graph as well as all postlude blocks except for the successor of the loop kernel. In the prephase, all loop instructions must be removed and the initialization of the loop counter must be replaced by the instruction `MVK 50, A1`. Finally, all loop instructions also must be removed from the remaining block in the postlude.

Then, the software de-pipelining of the dot product example results in

```
        MVK 50, A1
        ZERO A7
        ZERO B7
LOOP:
        [A1] SUB A1,1,A1
        LDW *A4++,A2
        LDW *B4++,B2
        MPY A2,B2,A6
        MPYH A2,B2,B6
        ADD A6,A7,A7
        ADD B6,B7,B7
        [A1] B LOOP
        ADD A7, B7, B4
```

5.3 Prolog / Epilog Collapsing

Software pipelining results in code replication. The kernel has approximately the size of the loop, but the prolog and epilog increase the number of instructions. Code

size is an issue especially for embedded architectures. Therefore, several techniques have been developed for controlling the code size of software pipelined loops, such as prolog and epilog collapsing [GSS+].

5.3.1 Epilog Collapsing

Consider the following loop code

```
LOOP:
   inst1
   inst2 || SUB n, 1, n
   inst3 || [n] B LOOP                          ; branch to loop if n>0
```

After software pipelining it becomes

```
   SUB n, 2, n
   inst1                                                ; prolog state 1
   inst2 || inst1 || SUB n, 1, n                        ; prolog state 2
LOOP:
   inst3 || inst2 || inst1 || [n] SUB n, 1, n || [n] B LOOP
             inst3 || inst2                             ; epilog state 1
                      inst3                             ; epilog state 2
```

The only difference between the loop kernel and the first stage in the epilog is that inst1 is executed in the kernel, but not in the epilog. Suppose it were safe to execute inst1 an extra time. That means, that the execution of inst1 has no effect on the further instructions in the code. Then, we could simply execute the kernel one extra time and remove the first stage of the epilog. So, the first epilog state can be *collapsed* back into the kernel. The loop counter needs to be modified to make sure that the loop realized one additional iteration. The resulting loop would be as follows.

```
   SUB n, 1, n                                  ; exec. kernel n-2+1 times
   inst1                                                ; prolog state 1
   inst2 || inst1 || SUB n, 1, n                        ; prolog state 2
LOOP:
   inst3 || inst2 || inst1 || [n] SUB n, 1, n || [n] B LOOP
                     inst3                              ; epilog state 2
```

Now, suppose the same process can be applied to epilog stage 2 and eliminate all epilog stages.

In this case, ignoring loop control instructions, there are two instructions of the loop kernel which do not appear in the epilog: `inst1` and `inst2`. Assume that `inst1` can be safely executed an extra time, but `inst2` cannot. Then, to collapse this second epilog stage, it will be necessary to add a condition to `inst2`. Before loop execution begins, the new predicate register for `inst2` is initialized to one less than the trip counter, so that `inst2` is not executed during the last iteration of the kernel.

The final version of the loop is

0	MVK 57,A1	ZERO A6	ZERO B6	ZERO A7	ZERO B7			
1	[A1]SUB A1,1,A1	ZERO A2	ZERO B2					
2	[A1]SUB A1,1,A1			[A1]B LOOP				
3	[A1]SUB A1,1,A1			[A1]B LOOP				
4	[A1]SUB A1,1,A1			[A1]B LOOP				
5	[A1]SUB A1,1,A1			[A1]B LOOP				
6	[A1]SUB A1,1,A1			[A1]B LOOP				
7 LOOP	[A1]SUB A1,1,A1	LDW *A4++,A2	LDW *B4++,B2	[A1]B LOOP	MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7
8								ADD A7, B7, A4

Figure 5.10: Dot Product with prolog and epilog collapsed

```
                                                    ; exec. kernel n-2+2 times
   SUB n, 1, p                                                    ; p = n-1
   inst1                                                  ; prolog state 1
   inst2 || inst1 || SUB n, 1, n                          ; prolog state 2
LOOP:
   inst3 || [p] inst2 || inst1 || [p] SUB p, 1, p || [n] SUB n, 1, n
   || [n] B LOOP
```

5.3.2 Prolog Collapsing

Prologs can be collapsed in a similar way, except that it must determine whether it is safe to execute an instruction additional times before the loop. In the previous example, in order to collapse the prolog, the loop must be transformed in such a way, that instructions ommited from the prolog (`inst3` during prolog stage 1 and `inst2` and `inst3` during prolog stage 1) can be safely overexecuted.

Assume that it is safe to overexecute `inst2` before the loop, but not `inst3`. Thus, the compiler must guard `inst3` to collapse the prolog. The fully collapsed loop is

```
   ADD n, 2, n                              ; exec. kernel n-2+2 +2 times
   SUB n, 1, p                                                    ; p = n-1
   MV 2, q                                                          ; q = 2
LOOP:
   [!q] inst3 || [p] inst2 || inst1 || [p] SUB p, 1, p || [q] SUB q, 1, q
   || [n] SUB n, 1, n || [n] B LOOP
```

However, it is not necessary to always fully collapse prolog and epilog.

Next, the problems that collapsing introduces for software de-pipelining will be discussed based on the dot product example which has been used throughout the whole chapter.

5.3.3 Dot Product Example

Let us recall the dot product example introduced in section 5.1. After complete prelude and postlude collapsing, the resulting assembler code is shown in figure 5.10.

After delay resolution, the control flow graph corresponding to this code has the same structure as the not-collapsed version (see figure 5.5). However, the contents of the blocks are different. Figures 5.11 and 5.12 show the contents of the blocks

in detail. The prelude blocks, which in 5.6 contained whole iteration code are not complete iterations anymore. Following the idea behind prolog collapsing, several operations are performed that do not affect the result of the algorithm. Registers `A2`, `B2`, `A6`, `B6`, `A7` and `B7` are initialized to zero in the prephase, in order to allow these operations to be performed without having any effect on the program.

Thus, both instructions `ADD A6,A7,A7` and `ADD B6,B7,B7` in block 1 add two registers that have the value 0. In block 1, the same happens with the additions and the multiplications `MPY A2,B2,A6` and `MPYH A2,B2,B6` . In block 5, finally the registers `A2` and `B2` are loaded and the loop iterations realize complete computations.

With regard to the postlude, the situation is similar. All instructions in the postlude have no effect, since the registers that are loaded or the values that are computed are never used in this example.

Loop Instructions Schedule

The presence of these *useless* instructions make the original approach for scheduling the loop inappropriate. Even though considering the new order given in the prephase and prelude would not necessarily produce a wrong result, the software pipelining would not be reversed and the question on how many times should the loop iterate would be much more complex to answer.

Observe that, applying the former algorithm, the schedule of the loop body would be

```
[A1]  SUB A1,1,A1
      ADD A6,A7,A7
      ADD B6,B7,B7
      MPY A2,B2,A6
      MPYH A2,B2,B6
      LDW *A4++,A2
      LDW *B4++,B2
[A1]  B LOOP
```

However, the first two iterations of this loop will not be complete. In the first one, registers `A2` and `B2` are loaded. The additions and multiplications have no effect. In the second iteration the multiplications `MPY A2,B2,A6` and `MPYH A2,B2,A6` do modify registers `A6` and `B6`, but it is not until the third iteration that the additions `ADD A6,A7,A7` and `ADD B6,B7,B7` do add non-zero values.

This behaviour does not correspond to the original non-software-pipelined loop, since it implies an overlapping of the loop iterations. Therefore, an alternative approach is needed, to recover the correct order of instructions. As the prelude does not give any further information, we need to study the loop kernel in order to produce the correct schedule.

The basic principle that our algorithm needs to fulfil is that the definition of each register that is used in the loop must be placed before every use. In other words, every value must be available when needed.

Block 0

0	MVK 57,A1	ZERO A6	ZERO B6	ZERO A7	ZERO B7			
1	[A1]SUB A1,1,A1	ZERO A2	ZERO B2					
2	[A1]SUB A1,1,A1							
3	[A1]SUB A1,1,A1							
4	[A1]SUB A1,1,A1							
5	[A1]SUB A1,1,A1							
6	[A1]SUB A1,1,A1							

Block 1

LOOP 7	[A1]SUB A1,1,A1			[A1]B LOOP1 [!A1]B lab0			ADD A6,A7,A7	ADD B6,B7,B7

Block 2

LOOP1 7'	[A1]SUB A1,1,A1			[A1]B LOOP2 [!A1]B lab1	MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7

Block 3

LOOP2 7''	[A1]SUB A1,1,A1			[A1]B LOOP3 [!A1]B lab2	MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7

Block 4

LOOP3 7'''	[A1]SUB A1,1,A1			[A1]B LOOP4 [!A1]B lab3	MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7

Block 5

LOOP4 7''''	[A1]SUB A1,1,A1	LDW *A4++,A2	LDW *B4++,B2	[A1]B LOOP5 [!A1]B lab4	MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7

Block 6

LOOP5 7'''''	[A1]SUB A1,1,A1	LDW *A4++,A2	LDW *B4++,B2	[A1]B LOOP6 [!A1]B lab5	MPY A2,B2,A6	MPYH A2,B2,B6	ADD A6,A7,A7	ADD B6,B7,B7

Block 7

lab0					MPY A2,B2,A6	MPYH A2,B2,B6		ADD A7, B7, A4
		LDW *A4++,A2	LDW *B4++,B2					

Block 8

lab1					MPY A2,B2,A6	MPYH A2,B2,B6		ADD A7, B7, A4
		LDW *A4++,A2	LDW *B4++,B2					
		LDW *A4++,A2	LDW *B4++,B2					

Figure 5.11: Blocks of the dotprod collapsed CFG

Block 9

lab2					MPY A2,B2,A6	MPYH A2,B2,B6		ADD A7, B7, A4
		LDW *A4++,A2	LDW *B4++,B2					
		LDW *A4++,A2	LDW *B4++,B2					
		LDW *A4++,A2	LDW *B4++,B2					

Block 10

lab3		LDW *A4++,A2	LDW *B4++,B2		MPY A2,B2,A6	MPYH A2,B2,B6		ADD A7, B7, A4
		LDW *A4++,A2	LDW *B4++,B2					
		LDW *A4++,A2	LDW *B4++,B2					
		LDW *A4++,A2	LDW *B4++,B2					

Block 11

lab4		LDW *A4++,A2	LDW *B4++,B2		MPY A2,B2,A6	MPYH A2,B2,B6		ADD A7, B7, A4
		LDW *A4++,A2	LDW *B4++,B2					
		LDW *A4++,A2	LDW *B4++,B2					
		LDW *A4++,A2	LDW *B4++,B2					

Block 12

lab5		LDW *A4++,A2	LDW *B4++,B2		MPY A2,B2,A6	MPYH A2,B2,B6		ADD A7, B7, A4
		LDW *A4++,A2	LDW *B4++,B2					
		LDW *A4++,A2	LDW *B4++,B2					
		LDW *A4++,A2	LDW *B4++,B2					

Figure 5.12: Blocks of the dotprod collapsed CFG (continued)

Initial Value of Loop Counter

Since prolog and epilog collapsing insert *dummy* iterations in the loop, the number of times it iterates is increased. In the dot product example, the new initial value of conditional register `A1` is 57, whereas the real number of iterations was 50. Note that, although software pipelining reduces the number of times the loop kernel is executed, collapsing has the opposite effect. In the following sections a way to recover the initial value of the loop counter will be described.

5.4 Software De-pipelining Formal Description

This section introduces several key concepts for the general software de-pipelining algorithm. The basis of the loop body scheduling is presented in form of different theorems.

Dependence

Let I_1 and I_2 be two instructions in the flow graph. The instruction I_2 is dependent on I_1 if and only if there is a path in the control flow graph that connects I_1 with I_2 and the instructions may reference the same memory location (or register). There are several types of dependencies:

- *True dependence:* Occurs when I_1 writes on a memory location that is read by I_2. In terms of registers, I_1 defines a register that is used by I_2.

  If a true dependence exists, I_1 must be executed before I_2.

- *Antidependence:* The dependence is an antidependence if I_1 reads a memory location that is written by I_2. Again, I_1 must be executed before I_2; otherwise I_2 might destroy the value needed by I_1 before it is used.

- *Output dependence:* Occurs when both instructions write the same memory location. Again, I_1 must be executed before I_2 because it must be ensured that the correct value is in memory for later references.

- *Input dependence:* If both instructions read from the same memory location, the dependence is called an input dependence. In this case the relative order of these instructions is irrelevant, because no value can be changed before it is needed.

Data Dependence Graph

The data dependence graph (DDG) is defined as (O, E), where O is the set of instructions and E the set of edges. There is an edge between two instructions I_1 and I_2 if there is a true, anti- or output dependence between I_1 and I_2.

Loop Data Dependence Graph

Given a loop, the loop data dependence graph (LDDG) is defined as a weighted data dependence graph $G = (O, E, d)$, where O is the set of loop instructions, E is the set of dependence edges and d is the dependence distance. Thus, an edge $e = (I1, I2) \in E$ expresses that instruction $I1$ must be executed before instruction $I2$. Let the distance associated to this edge be $d(e)$. Then, $I1$ must be executed $d(e)$ iterations before $I2$. If $d(e) = 0$, the dependence between the two operations is *loop independent*, since they are executed in the same iteration. A distance $d(e) > 0$ caracterizes a *loop-carried* dependence.

Loop Schedule

A loop schedule of a given loop is a mapping σ defined by

$$\begin{aligned} \sigma : O \times N &\longrightarrow N \\ (I, i) &\longmapsto c \end{aligned}$$

where (O, E, d) is the LDDG of the loop, N is the set of natural numbers. σ defines the cycle number c in which the instance of instruction I of the i^{th} iteration is issued for execution.

The mapping σ defines a valid schedule for the loop if the following conditions are fulfilled.

1. *Resource constraint:*

   In each cycle there is no hardware resource conflict, meaning that there are enough registers and execution units to perform all necessary computations.

2. *Data dependence constraint:*

   $\forall e(I1, I2) \in E$, $\forall j > 0$,

   $$\sigma(I1, j) \leq \sigma(I2, j + d(e))$$

   This relation simply illustrates the meaning of the dependence distance: $I1$ must execute $d(e)$ iterations earlier than $I2$. Any schedule for the loop which preserves its semantics must fulfil this condition as well.

3. *Cycle constraint:*

   The schedule defined by σ must represent a loop, otherwise it would be no loop schedule. Formally, this condition is expressed by the following condition:

   $\exists II \in N$ such that $\forall I \in O$, $\forall j > 1, j \in N$,

   $$\sigma(I, j) = \sigma(I, j - 1) + II \cdot (j - 1)$$

   This means that the same operation is executed periodically. II corresponds to the initiation interval of the software pipelined loop.

In this context, software pipelining consists in finding a valid loop schedule which has the smallest initiation interval II possible.

Next, the task of software de-pipelining will be defined with the tools introduced above.

Software De-pipelining

Definition

Given a software pipelined loop and its $LDDG(O, E, d)$, *software de-pipelining* aims to find a valid loop schedule dp such that

- For all $I \in O$ and $\forall j > 0, j \in N$,

  $$dp(I, j) \leq dp(I, j + 1)$$

  , and
- For all $I1, I2 \in O$ and $\forall j > 0, j \in N$,

  $$dp(I, j) \leq dp(I2, j + 1)$$

In a software pipelined loop instructions from different iterations are overlapped for execution. On the contrary, in a de-pipelined loop, instructions from different iterations must not overlap and the loop-carried dependence is automatically satisfied. All dependencies in a de-pipelined loop are independent, which motivates the following definition:

Definition

Given a loop and its $LDDG(O, E, d)$, the *loop body data dependence graph* (LBDDG) is defined by (O, E_0), where E_0 is a subset of E containing only its loop-independent edges.

Then, a valid schedule for the LBDDG solves the software de-pipelining problem.

Theorem 1 Given the body of a software pipelined loop and its prelude, the LBDDG can be constructed by scanning the prelude and the pipelined loop body if the prelude is not collapsed.

This corresponds to the scheduling of the loop instructions when the prelude is not collapsed. All loop instructions corresponding to the first loop iteration are considered. Because all these instructions come from the same iteration, the data dependence edges of the algorithm only cover all the loop-independent dependence edges of the original sequential loop, defining the LBDDG.

Theorem 2

Given a loop and its $LBDDG(O, E_0)$, a loop schedule σ is valid if the following conditions are satisfied:

1. For all $I \in O$ and $\forall j > 0, j \in N$,$dp(I, j) \leq dp(I, j + 1)$, and
2. For all $I1, I2 \in O$ and $\forall j > 0, j \in N$,$dp(I1, j) \leq dp(I2, j + 1)$
3. *Resource constraint.*
4. *Cycle constraint.*
5. For all $e = (I1, I2) \in E_0$, $\sigma(I1, j) \leq \sigma(I2, j)$

The proof is straightforward because the loop-carried dependences are automatically satisfied if the two conditions that define a valid schedule are satisfied. Given a software-pipelined loop, Theorem 2 provides the basis to generate the de-pipelined loop. That is, any list scheduling algorithm can be applied to schedule the loop under the constraints of hardware resource and the $LBDDG$.

5.5 General Software De-pipelining Approach

In this section an approach for handling software pipelined loops whose prolog and/or epilog may have been collapsed is described.

The main algorithm for software de-pipelining is the same as described in section 5.2. Scheduling the loop instructions and recovering the initial value of the loop counter is handled in a different way when collapsing is allowed.

Another important issue to take into account is the removal of the postlude instructions. In a collapsed loop there may be instructions in the postlude that define values that are not used by other loop instructions. Removing them after de-pipelining as it is done in the not collapsed case may change the semantics of the

program. A memory analysis is then necessary, in order to make sure that those values are not used further in the program and that those instructions do not cause memory faults. Otherwise, they cannot be removed and are maintained in the exit block of the de-pipelined loop.

The following sections explain how to schedule the loop instructions in a general case and how to recompute the initial value of the loop counter. Section 5.5.3 discusses how to determine whether the prolog and the epilog have been collapsed. Finally, the programs that have been tested for de-pipelining are introduced in section 5.8.

5.5.1 Scheduling the Loop Instructions

One of the problems that appear when considering loops where collapsing has been performed is that the former approach for scheduling the instructions in the new sequential loop by taking them in the order they appear in the prephase and prelude does not work anymore (see 5.3.3).

Thus, a new approach is needed that takes into account the semantics of the loop instructions to place them in an appropriate order that makes the different variable values available when they are used. For this purpose, the data dependency graph of the loop body (LBDDG) is constructed. The LBDDG implicitly defines a schedule (see section 5.4), which preserves the semantics of the loop.

Example

Considering the dot product example, let E be the set of edges of the LBDDG. And let nodes correspond to loop body instructions as follows:

```
I1 → [A1] SUB .S1 A1,1,A1
I2 → ADD .L1 A6,A7,A7
I3 → ADD .L2 B6,B7,B7
I4 → MPY .M1X A2,B2,A6
I5 → MPYH .M2X A2,B2,B6
I6 → LDW .D1 *A4++,A2
I7 → LDW .D2 *B4++,B2
I8 → [A1] B .S2 LOOP
```

Instruction `I1` defines register `A1`, which is used by `I8`. Therefore, edge $(I8, I1) \in E$. Registers `A6` and `B6` are defined respectively by `I4` and `I5` and used by `I2` and `I3`. Instructions `I6` and `I7` load registers `A2` and `B2`, which are needed by instruction `I4` and `I5`. Thus, the final set of edges is
$E = \{(I8, I1), (I4, I6), (I5, I6), (I4, I7), (I5, I7), (I2, I4), (I3, I5)\}$. Figure 5.13 shows the data dependence graph for the instructions in the loop body of the dotprod algorithm.

The order defined by the edges of the LBDDG, produces a correct loop schedule as defined in section 5.4. Figure 5.14 shows the scheduling algorithm for software pipelined loops with collapsed prelude.

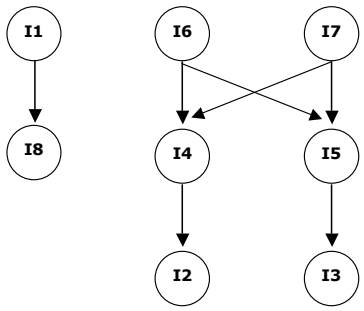

Figure 5.13: Data dependence graph of dotprod's Loop Body

```
function ScheduleWithCollapsing (CFG, loopKernel)
  NewLoop = create new loop body
  DDG = data dependence graph of LoopKernel
  Schedule the instructions following the DDG
     In case of conflict, resolve data dependence conflict.
endfunction
```

Figure 5.14: Loop Scheduling with Prelude Collapsing

In general, the presence of cycles in the LBDDG correspond to data conflicts in the program. This case should be treated like discussed in section 4.7.

Another main issue when trying to reconstruct the original sequential loop of a given software pipelined code is to recover the original value of the loop counter. The next section discusses how collapsing affects to this value.

The strategy will be as follows: consider several implementations of the dot product algorithm and force the loop kernel to execute only once. Then, observe how many times the full set of loop instructions are executed.

5.5.2 Restore the Initial Value of the Loop Counter

In a software pipelined loop, some values are computed before the loop (in the prelude) and some after the loop (postlude). On the other hand, in a sequential loop, all computation work is done inside of it. Therefore, when sequentializing the code, the loop counter must be modified in order to reflect the actual number of iterations that are executed.

Prelude and postlude collapsing do also have an effect on the loop counter. Next, we will see how to modify it, depending on the situation. The following examples show different versions of the dot product algorithm written in the TIC62 assembler language. For each example, two tables are shown: one containing the original code and the second one presenting the program after delay resolution.

The loop kernel is assumed to execute only once. Observing how many times will the instructions of the loop body actually execute, relevant conclusions can be extracted about the effect of the collapsing and not-collapsing on the initial value

of the loop counter.

Prolog and Epilog not Collapsed

Figure 5.15 shows a software pipelined loop with prelude and postlude. There are 6 conditional branches. Since the loop kernel is to be executed only once, and the loop counter is decreased by 1 for each branch, the loop counter must be at least 7. However, since it is updated once before the first branch, it should have a value of 8. Up to this point the effect of the prelude and postlude have been ignored. The delay resolution will give more information about it.

	LDW *A4++,A0	LDW *B4++,B0	MPY A5,0,A5	MPY B5,0,B5	MV 8,A2	ZERO A1	ZERO B1	
	LDW *A4++,A0	LDW *B4++,B0			SUB A2,1,A2			
	LDW *A4++,A0	LDW *B4++,B0			[A2] SUB A2,1,A2			[A2] B LOOP
	LDW *A4++,A0	LDW *B4++,B0			[A2] SUB A2,1,A2			[A2] B LOOP
	LDW *A4++,A0	LDW *B4++,B0			[A2] SUB A2,1,A2			[A2] B LOOP
	LDW *A4++,A0	LDW *B4++,B0	MPY A0,B0,A1	MPYH A0,B0,B1	[A2] SUB A2,1,A2			[A2] B LOOP
	LDW *A4++,A0	LDW *B4++,B0	MPY A0,B0,A1	MPYH A0,B0,B1	[A2] SUB A2,1,A2			[A2] B LOOP
L:	LDW *A4++,A0	LDW *B4++,B0	MPY A0,B0,A1	MPYH A0,B0,B1	[A2] SUB A2,1,A2	ADD A1,A5,A5	ADD B1,B5,B5	[A2] B LOOP
			MPY A0,B0,A1	MPYH A0,B0,B1		ADD A1,A5,A5	ADD B1,B5,B5	
			MPY A0,B0,A1	MPYH A0,B0,B1		ADD A1,A5,A5	ADD B1,B5,B5	
			MPY A0,B0,A1	MPYH A0,B0,B1		ADD A1,A5,A5	ADD B1,B5,B5	
			MPY A0,B0,A1	MPYH A0,B0,B1		ADD A1,A5,A5	ADD B1,B5,B5	
			MPY A0,B0,A1	MPYH A0,B0,B1		ADD A1,A5,A5	ADD B1,B5,B5	
						ADD A1,A5,A5	ADD B1,B5,B5	
						ADD A1,A5,A5	ADD B1,B5,B5	
						ADD A5,B5,A4		

Original assembler

			MPY A5,0,A5	MPY B5,0,B5	MV 8,A2	ZERO A1	ZERO B1	
					SUB A2,1,A2			
					[A2] SUB A2,1,A2			
					[A2] SUB A2,1,A2			
	LDW *A4++,A0	LDW *B4++,B0			[A2] SUB A2,1,A2			
	LDW *A4++,A0	LDW *B4++,B0			[A2] SUB A2,1,A2			
	LDW *A4++,A0	LDW *B4++,B0	MPY A0,B0,A1	MPYH A0,B0,B1	[A2] SUB A2,1,A2			
L:	LDW *A4++,A0	LDW *B4++,B0	MPY A0,B0,A1	MPYH A0,B0,B1	[A2] SUB A2,1,A2	ADD A1,A5,A5	ADD B1,B5,B5	[A2] B LOOP
L:	LDW *A4++,A0	LDW *B4++,B0	MPY A0,B0,A1	MPYH A0,B0,B1	[A2] SUB A2,1,A2	ADD A1,A5,A5	ADD B1,B5,B5	[A2] B LOOP
L:	LDW *A4++,A0	LDW *B4++,B0	MPY A0,B0,A1	MPYH A0,B0,B1	[A2] SUB A2,1,A2	ADD A1,A5,A5	ADD B1,B5,B5	[A2] B LOOP
L:	LDW *A4++,A0	LDW *B4++,B0	MPY A0,B0,A1	MPYH A0,B0,B1	[A2] SUB A2,1,A2	ADD A1,A5,A5	ADD B1,B5,B5	[A2] B LOOP
L:	LDW *A4++,A0	LDW *B4++,B0	MPY A0,B0,A1	MPYH A0,B0,B1	[A2] SUB A2,1,A2	ADD A1,A5,A5	ADD B1,B5,B5	[A2] B LOOP
L:	LDW *A4++,A0	LDW *B4++,B0	MPY A0,B0,A1	MPYH A0,B0,B1	[A2] SUB A2,1,A2	ADD A1,A5,A5	ADD B1,B5,B5	[A2] B LOOP
	LDW *A4++,A0	LDW *B4++,B0	MPY A0,B0,A1	MPYH A0,B0,B1		ADD A1,A5,A5	ADD B1,B5,B5	
	LDW *A4++,A0	LDW *B4++,B0	MPY A0,B0,A1	MPYH A0,B0,B1		ADD A1,A5,A5	ADD B1,B5,B5	
	LDW *A4++,A0	LDW *B4++,B0	MPY A0,B0,A1	MPYH A0,B0,B1		ADD A1,A5,A5	ADD B1,B5,B5	
	LDW *A4++,A0	LDW *B4++,B0	MPY A0,B0,A1	MPYH A0,B0,B1		ADD A1,A5,A5	ADD B1,B5,B5	
			MPY A0,B0,A1	MPYH A0,B0,B1		ADD A1,A5,A5	ADD B1,B5,B5	
			MPY A0,B0,A1	MPYH A0,B0,B1		ADD A1,A5,A5	ADD B1,B5,B5	
						ADD A1,A5,A5	ADD B1,B5,B5	
						ADD A5,B5,A4		

Assembler code after delay resolution.

Figure 5.15: Prolog and Epilog not collapsed

After delay resolution, it can be seen that exactly 13 full iterations of the loop have been executed: the 6 iterations given by the 5 branches, and the 7 iterations finished in the postlude. The instructions in the prelude have ensured that when the first jump is performed, all necessary information was available and a complete

iteration could be executed. The postlude finishes iterations that started previously due to the latency of the load instructions. Thus, considering that the loop counter is decremented in the prelude once before the first branch, the initial value of the loop counter for the sequentialized version of the loop must be 14, since this is the real number of iterations that are executed.

Prolog not Collapsed, Epilog Collapsed

We have seen the effect of having both, a prelude and a postlude. The following example has only a prelude (see figure 5.16). The postlude has been collapsed. As in the previous case, the counter is set to 8. After delay resolution, we see that

	LDW *A4++,A0	LDW *B4++,B0	MPY A5,0,A5	MPY B5,0,B5	MV 8,A2	ZERO A1	ZERO B1	
	LDW *A4++,A0	LDW *B4++,B0			SUB A2,1,A2			
	LDW *A4++,A0	LDW *B4++,B0			[A2] SUB A2,1,A2			[A2] B LOOP
	LDW *A4++,A0	LDW *B4++,B0			[A2] SUB A2,1,A2			[A2] B LOOP
	LDW *A4++,A0	LDW *B4++,B0			[A2] SUB A2,1,A2			[A2] B LOOP
	LDW *A4++,A0	LDW *B4++,B0	MPY A0,B0,A1	MPYH A0,B0,B1	[A2] SUB A2,1,A2			[A2] B LOOP
	LDW *A4++,A0	LDW *B4++,B0	MPY A0,B0,A1	MPYH A0,B0,B1	[A2] SUB A2,1,A2			[A2] B LOOP
L:	LDW *A4++,A0	LDW *B4++,B0	MPY A0,B0,A1	MPYH A0,B0,B1	[A2] SUB A2,1,A2	ADD A1,A5,A5	ADD B1,B5,B5	[A2] B LOOP
						ADD A5,B5,A4		

Original assembler

			MPY A5,0,A5	MPY B5,0,B5	MV 8,A2	ZERO A1	ZERO B1	
					SUB A2,1,A2			
					[A2] SUB A2,1,A2			
					[A2] SUB A2,1,A2			
	LDW *A4++,A0	LDW *B4++,B0			[A2] SUB A2,1,A2			
	LDW *A4++,A0	LDW *B4++,B0			[A2] SUB A2,1,A2			
	LDW *A4++,A0	LDW *B4++,B0	MPY A0,B0,A1	MPYH A0,B0,B1	[A2] SUB A2,1,A2			
L:	LDW *A4++,A0	LDW *B4++,B0	MPY A0,B0,A1	MPYH A0,B0,B1	[A2] SUB A2,1,A2	ADD A1,A5,A5	ADD B1,B5,B5	[A2] B LOOP
L:	LDW *A4++,A0	LDW *B4++,B0	MPY A0,B0,A1	MPYH A0,B0,B1	[A2] SUB A2,1,A2	ADD A1,A5,A5	ADD B1,B5,B5	[A2] B LOOP
L:	LDW *A4++,A0	LDW *B4++,B0	MPY A0,B0,A1	MPYH A0,B0,B1	[A2] SUB A2,1,A2	ADD A1,A5,A5	ADD B1,B5,B5	[A2] B LOOP
L:	LDW *A4++,A0	LDW *B4++,B0	MPY A0,B0,A1	MPYH A0,B0,B1	[A2] SUB A2,1,A2	ADD A1,A5,A5	ADD B1,B5,B5	[A2] B LOOP
L:	LDW *A4++,A0	LDW *B4++,B0	MPY A0,B0,A1	MPYH A0,B0,B1	[A2] SUB A2,1,A2	ADD A1,A5,A5	ADD B1,B5,B5	[A2] B LOOP
L:	LDW *A4++,A0	LDW *B4++,B0	MPY A0,B0,A1	MPYH A0,B0,B1	[A2] SUB A2,1,A2	ADD A1,A5,A5	ADD B1,B5,B5	[A2] B LOOP
	LDW *A4++,A0	LDW *B4++,B0	MPY A0,B0,A1	MPYH A0,B0,B1		ADD A5,B5,A4		
	LDW *A4++,A0	LDW *B4++,B0						
	LDW *A4++,A0	LDW *B4++,B0						
	LDW *A4++,A0	LDW *B4++,B0						

Assembler code after delay resolution

Figure 5.16: Prolog not collapsed, Epilog collapsed

6 complete iterations of the loop are executed, as many as branches in the code. The fact that the postlude is collapsed, makes the partial iterations that have not finished executing when leaving the loop to be useless.

The sequentialized version of this loop will have an initial value of 8. Thus, a collapsed postlude has no effect on the initial value of the loop counter.

Prolog and Epilog Collapsed

The example in figure 5.17 shows the dotprod code with collapsed prolog and epilog.

After delay resolution, taking the instruction dependencies into consideration, only 1 complete iteration of the loop is executed. We have seen in the previous

			MPY A5,0,A5	MPY B5,0,B5	MV 8,A2	ZERO A1	ZERO B1	ZERO B0
					SUB A2,1,A2	ZERO A0		
					[A2] SUB A2,1,A2			[A2] B LOOP
					[A2] SUB A2,1,A2			[A2] B LOOP
					[A2] SUB A2,1,A2			[A2] B LOOP
					[A2] SUB A2,1,A2			[A2] B LOOP
					[A2] SUB A2,1,A2			[A2] B LOOP
L:	LDW *A4++,A0	LDW *B4++,B0	MPY A0,B0,A1	MPYH A0,B0,B1	[A2] SUB A2,1,A2	ADD A1,A5,A5	ADD B1,B5,B5	[A2] B LOOP
						ADD A5,B5,A4		

Original Assembler

			MPY A5,0,A5	MPY B5,0,B5	MV 7,A2	ZERO A1	ZERO B1	ZERO B0
					SUB A2,1,A2			
					[A2] SUB A2,1,A2			
					[A2] SUB A2,1,A2			
					[A2] SUB A2,1,A2			
					[A2] SUB A2,1,A2			
					[A2] SUB A2,1,A2			
L:					[A2] SUB A2,1,A2	ADD A1,A5,A5	ADD B1,B5,B5	[A2] B LOOP
L:			MPY A0,B0,A1	MPYH A0,B0,B1	[A2] SUB A2,1,A2	ADD A1,A5,A5	ADD B1,B5,B5	[A2] B LOOP
L:			MPY A0,B0,A1	MPYH A0,B0,B1	[A2] SUB A2,1,A2	ADD A1,A5,A5	ADD B1,B5,B5	[A2] B LOOP
L:			MPY A0,B0,A1	MPYH A0,B0,B1	[A2] SUB A2,1,A2	ADD A1,A5,A5	ADD B1,B5,B5	[A2] B LOOP
L:	LDW *A4++,A0	LDW *B4++,B0	MPY A0,B0,A1	MPYH A0,B0,B1	[A2] SUB A2,1,A2	ADD A1,A5,A5	ADD B1,B5,B5	[A2] B LOOP
L:	LDW *A4++,A0	LDW *B4++,B0	MPY A0,B0,A1	MPYH A0,B0,B1	[A2] SUB A2,1,A2	ADD A1,A5,A5	ADD B1,B5,B5	[A2] B LOOP
	LDW *A4++,A0	LDW *B4++,B0	MPY A0,B0,A1	MPYH A0,B0,B1		ADD A5,B5,A4		
	LDW *A4++,A0	LDW *B4++,B0						
	LDW *A4++,A0	LDW *B4++,B0						
	LDW *A4++,A0	LDW *B4++,B0						

Assembler after delay resolution

Figure 5.17: Prolog and Epilog collapsed

example that the fact that the postlude is collapsed has no effect on the number of iterations, since it only causes not-complete iterations. However, we see that a collapsed prelude does have an effect on it. Since the loop instructions with delay do not start execution outside of the loop, the information they compute is not available for the other loop instructions the first time they are reached.

Therefore, loop iteration time is used for this purpose and so the first iterations of the loop are not complete. An equivalent sequential loop code would have an initial value of 2.

Another Example of Prolog Collapsing

In the example in figure 5.18 there is one unconditional branch and 5 conditional branches. The initial value of the loop counter must be 6, since the first update of the loop counter is not done before the first conditional branch. After delay resolution, we see that the first iteration block does not contain all loop body instructions. The second iteration block, although it contains all instructions, is not a complete iteration, because of the data dependencies between the instructions. Therefore, there are only 4 complete iterations executed. The sequentialized loop code, which will contain a conditional branch, must have an initial value of 5.

Conclusions

Through the previous examples we have seen that

	LDW *A4++,A0	LDW *B4++,B0	MPY A5,0,A5	MPY B5,0,B5	MV 6,A2	ZERO A1	ZERO B1	B LOOP
	LDW *A4++,A0	LDW *B4++,B0			SUB A2,1,A2			[A2] B LOOP
	LDW *A4++,A0	LDW *B4++,B0			[A2] SUB A2,1,A2			[A2] B LOOP
	LDW *A4++,A0	LDW *B4++,B0			[A2] SUB A2,1,A2			[A2] B LOOP
	LDW *A4++,A0	LDW *B4++,B0			[A2] SUB A2,1,A2			[A2] B LOOP
L:	LDW *A4++,A0	LDW *B4++,B0	MPY A0,B0,A1	MPYH A0,B0,B1	[A2] SUB A2,1,A2	ADD A1,A5,A5	ADD B1,B5,B5	[A2] B LOOP
						ADD A5,B5,A4		

Original Assembler

					MV 6,A2	ZERO A1	ZERO B1	
			MPY A5,0,A5	MPY B5,0,B5	SUB A2,1,A2			
					[A2] SUB A2,1,A2			
					[A2] SUB A2,1,A2			
	LDW *A4++,A0	LDW *B4++,B0			[A2] SUB A2,1,A2			
L:	LDW *A4++,A0	LDW *B4++,B0			[A2] SUB A2,1,A2	ADD A1,A5,A5	ADD B1,B5,B5	B LOOP
L:	LDW *A4++,A0	LDW *B4++,B0	MPY A0,B0,A1	MPYH A0,B0,B1	[A2] SUB A2,1,A2	ADD A1,A5,A5	ADD B1,B5,B5	[A2] B LOOP
L:	LDW *A4++,A0	LDW *B4++,B0	MPY A0,B0,A1	MPYH A0,B0,B1	[A2] SUB A2,1,A2	ADD A1,A5,A5	ADD B1,B5,B5	[A2] B LOOP
L:	LDW *A4++,A0	LDW *B4++,B0	MPY A0,B0,A1	MPYH A0,B0,B1	[A2] SUB A2,1,A2	ADD A1,A5,A5	ADD B1,B5,B5	[A2] B LOOP
L:	LDW *A4++,A0	LDW *B4++,B0	MPY A0,B0,A1	MPYH A0,B0,B1	[A2] SUB A2,1,A2	ADD A1,A5,A5	ADD B1,B5,B5	[A2] B LOOP
L:	LDW *A4++,A0	LDW *B4++,B0	MPY A0,B0,A1	MPYH A0,B0,B1	[A2] SUB A2,1,A2	ADD A1,A5,A5	ADD B1,B5,B5	[A2] B LOOP
	LDW *A4++,A0	LDW *B4++,B0	MPY A0,B0,A1	MPYH A0,B0,B1		ADD A5,B5,A4		
	LDW *A4++,A0	LDW *B4++,B0						
	LDW *A4++,A0	LDW *B4++,B0						
	LDW *A4++,A0	LDW *B4++,B0						

Assembler after delay resolution

Figure 5.18: Another example of Prolog collapsing

1. Epilog collapsing does not change the initial value of the loop counter.
2. An epilog increases the initial value of the loop counter in the sequential code.
3. A collapsed prolog reduces the initial value of the loop counter in the sequential code.

Next section will discuss how to determine whether prolog and the epilog collapsing have been applied to the software pipelined loop.

5.5.3 Criterion for Prolog / Epilog Collapsing

A still open issue is how to determine whether the prolog and/or the epilog of the loop have been collapsed. If so, it is also necessary to determine how many stages have been involved.

Prolog

The prolog of a software pipelined loop is intended to perform the necessary computations that allow a complete iteration execution when entering the loop. For this purpose, dependencies between instructions as well as their delays are taken into account. Let us consider a loop containing 4 instructions such that its data dependence graph is as shown in figure 5.19 and the instructions have originally the following delays: $d(\texttt{I1}) = 1$, $d(\texttt{I2}) = 3$, $d(\texttt{I3}) = 1$ and $d(\texttt{I4}) = 0$.

According to the DDG, instruction `I1` must be executed first, then `I2` followed by `I3` and finally `I4`. Considering these dependencies and the delays of the instructions, `I4` is executed 8 cycles after `I1` was dispatched.

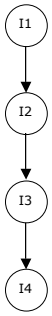

Figure 5.19: DDG example

Now, assume this loop has been software pipelined with an initiation interval $II = 1$. This means, that in each cycle a new iteration is started. Thus, from the moment the first iteration starts executing until it finishes executing 9 cycles later, 8 new iterations have started execution themselves.

In general, a DDG may have several paths and several connex components. The loop is assumed to be scheduled in such a way that connex components are executed in parallel. In case this would not be possible due to resource constraints, the longest path would be defined by concatenating conflicting components.

Let $P = \{\mathtt{I}_1, ..., \mathtt{I}_n\}$ be the longest path in the DDG and $d(\mathtt{I})$ be the delay of instruction $\mathtt{I}$. Then, the number of iterations NP that start in a not collapsed prolog are given by the formula

$$NP = \frac{\sum_{k=1}^{n-1} d(\mathtt{I}_k) + 1}{II}$$

where II is the initiation interval of the loop, or the length of the loop kernel.

The number of iterations that start in a not collapsed prolog is equivalent to the number of iterations that finish in a not collapsed epilog.

After Delay Resolution

After delay resolution several prolog instructions are moved to prelude blocks while other may remain in the prephase block. The prolog is not collapsed if and only if

1. the first iteration block in the prelude is complete, and

2. all necessary values are available (the data dependencies are fulfilled).

It is easy to check whether the first prelude block is a complete iteration: it must contain all loop kernel instructions.

In order to find out whether all necessary values have been computed, the DDG is considered. Since the delays have been resolved, the relative distances between the instructions have changed. Consider the previous example in figure 5.19. Assuming one iteration starts at cycle 0, the instructions before and after delay resolution are

scheduled as follows:

	With delays	No delays
0	I_1	
1		I_1
2	I_2	
3		
4		
5		I_2
6	I_3	
7		I_3
8	I_4	I_4

Whereas before delay resolution the distance between `I1` and `I2` was given by the number of delay slots of `I1`, afterwards it is the former number of delay slots of `I2`. In general, given an edge $e = (\mathtt{I}_j, \mathtt{I}_k)$ from the DDG, the distance between both instructions is

$$dist(I_j, I_k) = \begin{cases} d(I_j), & \text{if delays have not been resolved} \\ d(I_k), & \text{if delays have been resolved} \end{cases} \tag{5.1}$$

Therefore, after delay resolution, the number of cycles between the execution of `I1` and `I4` is 3. Following the same reasoning as in section 5.5.3, this means that, if the prolog has not been collapsed, 3 iterations must start before entering the loop (that is, in the prephase).

In general, the number of iterations that start execution in the prephase once the delay slots have been resolved is

$$NP_{ad} = \frac{\sum_{k=2}^{n} d(\mathtt{I}_k) + 1}{II}$$

Let `I1` be the first instruction in the DDG path. Then, in our example, if 3 copies of this instruction are found in the prephase, we can assume that the prolog is not collapsed. Otherwise, it is collapsed. Thus, we derive the following result

Criterion for the existence of a collapsed prolog

Let I_{first} be the first instruction in the LBDDG path. Let n_{first} be the number of copies of I_{first} in the prephase. Then, if

$$n_{first} < NP_{ad}$$

, the prolog has been collapsed.

The number of collapsed stages cs_{pro} is given by the following formula:

$$cs_{pro} = (NP_{ad} - n_{first}) + miss_{first}$$

, where $miss_{first}$ is the number of prelude blocks where instruction I_{first} is missing.

Epilog

The epilog of a software pipelined loop is intended to finish the execution of iterations that started in the loop.

An iteration finishes executing if the last instructions are executed. In the DDG, the last instructions are those that no other instruction depends on. In example 5.19, the last instruction is `I4`. Since a new iteration starts in every cycle, a new iteration finishes in every cycle.

When the epilog is not collapsed, all iterations that start executing also finish executing. In particular, all iterations that start executing in the epilog finish executing. Thus, knowing that the distance between I_{first} and I_{last} is $d = NP_{ad} \cdot II$, there must be a copy of I_{last} at a distance d of the last instance of I_{first}.

So, the number of collapsed stages is given by the number of missing I_{last} instructions.

5.5.4 General Algorithm to Compute the New Loop Counter Value

Given a software pipelined loop after delay resolution, the new initial value of the loop counter CV can be computed as described in figure 5.20, where NP and the collapsed stages are computed as described in section 5.5.3, s is the update step of the conditional register and ϵ is as defined in 5.2.4.

```
function computeCounterInitialValue (CV, NP, s, nb, CFG)
  if (epilog not fully collapsed) then
    n = NP - (number of collapsed stages)
    CV = CV + s· n
  fi
  if (prelude collapsed) then
    m = number of collapsed stages
    CV = CV − s· m

  fi
  nb = number of not conditional loop branches

  CV = CV + nb + ε
endfunction
```

Figure 5.20: Algorithm for Computing the Initial Value of the Loop Counter

where `CV` is the initial value of the counter in the software pipelined loop.

Note that this new algorithm, although different to the one used in the case without collapsing, does not contradict it. In a software pipelined loop without collapsing, there are no incomplete iterations. Every iteration that starts, will be fully executed. Therefore, counting how many iterations finish execution is equivalent to counting how many iterations start execution.

5.6 Other Loop Optimizations

This section discusses the effect of other usual loop optimizations on the software de-pipelining techniques presented here.

5.6.1 Loop Unrolling

Loop unrolling is a transformation applied when the body of the loop is too small to use all of the execution units efficiently.

Unrolling the loop is a simple transformation. It consists on making the appropriate number of copies of the loop body and adjust de loop index accordingly.

The more difficult issue with loop unrolling is to decide how much to unroll a loop. Some loops, such as those with conditional branching and procedure calls do not benefit from unrolling. Thus, the typical loop to unroll consists of a body that has sequential statements in it without branching.

There is another form of loop unrolling that applies even when the loop is not a counting loop. This technique has benefits when dealing with WHILE loops in which later instruction scheduling is done globally.

Loop unrolling is often performed while software pipelining a loop in order to get a more efficient result.

5.6.2 Modulo Variable Renaming

An unrolled loop contains several copies of an original loop iteration in its body. In order to avoid conflicts with the definitions and uses of the variables in the loop, variable renaming may be needed. A simple approach consists in considering all temporaries that are evaluated inside the loop: $T_1, ..., T_k$. Assuming the loop has been unrolled S times, S different copies of the registers are generated: one for each iteration of the loop in the unrolled loop. The temporaries are modified in the formed loop so that all temporaries that apply to one iteration of the original loop use the same set of temporaries.

5.6.3 Loop Peeling

Loop Peeling is a form of loop unrolling where the first iterations from a loop are unrolled. It removes `n` iterations from the beginning of a loop and adds `n` copies of the loop body directly before the start of the loop.

5.6.4 Loop Optimizations and De-pipelining

A software pipelined loop whose body has been unrolled is treated by the software de-pipelining algorithm as a normal loop. In most cases, modulo variable renaming has also been applied. This makes reversing the unrolling a much harder task. Consider the code in figure 5.21, which implements a function that searches the maximal element in an array `a`. When software pipelining, the loop has been unrolled 6 times and registers have been renamed.

```
                ADD     .L2X    2, A4, B4       ; copy a
||              MVK     .S1     -32768, A5      ; max[j] = -32768
||              MVK     .S2     -32768, B5      ; max[j] = -32768
||              MV      .L1X    B3, A0          ; move return address
||              SUB     .D2     B4, 6, B0       ; i--
||              MV      .D1     A4, A7          ; copy a

                LDH     .D1     *A7++[2], A8    ;** x[j] = a[i + j]
||              LDH     .D2     *B4++[2], B8    ;** x[j] = a[i + j]

                MVK     .S2     -32768, B6      ; max[j] = -32768
||              MVK     .S1     -32768, A6      ; max[j] = -32768
||              LDH     .D1     *A7++[2], A9    ;** x[j] = a[i + j]
||              LDH     .D2     *B4++[2], B9    ;** x[j] = a[i + j]

                LDH     .D1     *A7++[2], A3    ;** x[j] = a[i + j]
||              LDH     .D2     *B4++[2], B3    ;** x[j] = a[i + j]

        [B0]    B       .S1     LOOP            ; for
||      [B0]    SUB             B0, 6, B0       ; i--
||              LDH     .D1     *A7++[2], A8    ;** x[j] = a[i + j]
||              LDH     .D2     *B4++[2], B8    ;** x[j] = a[i + j]

                LDH     .D1     *A7++[2], A9    ;** x[j] = a[i + j]
||              LDH     .D2     *B4++[2], B9    ;** x[j] = a[i + j]
||              MVK     .S1     -32768, A4      ; max[j] = -32768
||              MVK     .S2     -32768, B7      ; max[j] = -32768

                CMPLT   .L1     A5, A8, A1      ;* t[j] = max[j] < x[j]
||              CMPLT   .L2     B5, B8, B1      ;* t[j] = max[j] < x[j]
||              LDH     .D1     *A7++[2], A3    ;** x[j] = a[i + j]
||              LDH     .D2     *B4++[2], B3    ;** x[j] = a[i + j]
LOOP:
        [B0]    B       .S1     LOOP            ; for
||      [B0]    SUB             B0, 6, B0       ; i--
||              CMPLT   .L1     A6, A9, A2      ; t[j] = max[j] < x[j]
||              CMPLT   .L2     B6, B9, B2      ; t[j] = max[j] < x[j]
||      [A1]    MPY     .M1     1, A8, A5       ; if (t[j]) max[j] = x[j]
||      [B1]    MPY     .M2     1, B8, B5       ; if (t[j]) max[j] = x[j]
||              LDH     .D1     *A7++[2], A8    ;** x[j] = a[i + j]
||              LDH     .D2     *B4++[2], B8    ;** x[j] = a[i + j]

                CMPLT   .L1     A4, A3, A2      ; t[j] = max[j] < x[j]
||              CMPLT   .L2     B7, B3, B2      ; t[j] = max[j] < x[j]
||      [A2]    MPY     .M1      1, A9, A6      ; if (t[j]) max[j] = x[j]
||      [B2]    MPY     .M2      1, B9, B6      ; if (t[j]) max[j] = x[j]
||              LDH     .D1     *A7++[2], A9    ;** x[j] = a[i + j]
||              LDH     .D2     *B4++[2], B9    ;** x[j] = a[i + j]

        [A2]    MPY     .M1      1, A3, A4      ; if (t[j]) max[j] = x[j]
||      [B2]    MPY     .M2      1, B3, B7      ; if (t[j]) max[j] = x[j]
||              CMPLT   .L1     A5, A8, A1      ;* t[j] = max[j] < x[j]
||              CMPLT   .L2     B5, B8, B1      ;* t[j] = max[j] < x[j]
||              LDH     .D1     *A7++[2], A3    ;** x[j] = a[i + j]
||              LDH     .D2     *B4++[2], B3    ;** x[j] = a[i + j]
; branch occurs here
                CMPLT   .L1     A5, A6, A1      ; t[0] = max[0] < max[2]
||              CMPLT   .L2     B5, B6, B1      ; t[1] = max[1] < max[3]

                CMPLT   .L1X    A4, B7, A2      ; t[4] = max[4] < max[5]
||      [A1]    MV      .S1     A6, A5          ; if (t[0]) max[0] = max[2]
||      [B1]    MV      .L2     B6, B5          ; if (t[1]) max[1] = max[3]

        [A2]    MV      .L1X    B7, A4          ; if (t[4]) max[4] = max[5]
||              CMPLT   .L2X    B5, A5, B1      ; t[1] = max[1] < max[0]

        [B1]    MV      .L2X    A5, B5          ; if (t[1]) max[1] = max[0]
                CMPLT   .L1X    A4, B5, A2      ; t[4] = max[4] < max[1]
        [A2]    MV      .L1X    B5, A4          ; if (t[4]) max[4] = max[1]
```

Figure 5.21: Assembler Code for Max Function

For a human eye, it is obvious that several operations are equivalent, except they are using different register names. The copies of the iterations can be distinguished after some observation. However, to do this automatically is not trivial. The lack of knowledge about the intention of the programmer makes it in general impossible to decide whether two instructions that are equivalent except for the operand names are actually performing the same task.

Therefore, the software de-pipelining algorithm ignores this and generates an unrolled sequentialized version of the loop.

5.7 Other Types Of Loops

So far, a method for software de-pipelining so called *counted loops*, that is, loops for which the number of iterations is explicitly given, has been presented.

However, not all loops have this property. The initial value of the loop counter may not be known and just represented by a variable. Furthermore, we may have a **while** loop whose condition register is not updated by a linear instruction, but holds the result of a much different and complex condition like a length comparison or the contents of a pointer or memory location.

Since these kind of loops can also be software pipelined, it is necessary to consider them for software de-pipelining.

5.7.1 Variable Counter

The number of times a loop executes is not always given explicitly. The dot product algorithm introduced in section 5.1.1, is a concrete implementation for multiplying arrays of a given size. Thus, there is an explicit loop counter initialized to a concrete value. A general version of this code should accept arrays of any size. This could be given as a parameter to the function, which in C would then have the following form:

```
int dotp(int N, short a[], short b[]){
    int i, sum = 0;
    for(i=0; i<N; i++)
        sum += a[i] * b[i];
    return(sum);
}
```

In the assembler implementation the instruction `MVK A1, 50` is substituted by the load of a parameter. Thus, the software pipelined version of this algorithm

5.7.2 Not Counted Loops

Figure 5.22 shows the software pipelined version of the loop in the following function

```
int strlen (char *c){
  int length = 0;
  while (*c != 0){
```

```
0      LDW *B2++, B1
1      MVK 0, A0
2      LDW *B2++, B1
3      NOP
4      LDW *B2++, B1
5      CMPEQ B1, 0, A1
6      LDW *B2++, B1   ||   [A1] B LOOP
7      [A1] CMPEQ B1, 0, A1   ||   [A1] ADD A0, 1, A0
8      LDW *B2++, B1   ||   [A1] B LOOP
9      [A1] CMPEQ B1, 0, A1   ||   [A1] ADD A0, 1, A0
LOOP:
10     LDW *B2++, B1   ||   [A1] B LOOP
11     [A1] CMPEQ B1, 0, A1   ||   [A1] ADD A0, 1, A0
;  Branch occurs here
```

Figure 5.22: Strlen Software Pipelined

```
    c++;
    length++;
  }
  return(length);
 }
```

which computes the length of a string.

The epilog of the software pipelined loop has been collapsed. Here, it cannot be assumed that the iterations outside of the loop will be fully executed. Therefore, predication must be applied.

In this sense, the loop code is prepared to find out when the loop needs to stop, without the need of an explicit counter. Since the software pipelining does not care about this, neither does the software de-pipelining. Thus, no initial counter value needs to be recovered when software de-pipelining this code.

The sequentialized version of the loop after delay resolution is presented in figure 5.23.

```
0      MVK 0, A0
LOOP:
1      LDW *B2++, B1
2      CMPEQ B1, 0, A1
3      [A1] ADD A0, 1, A0
4      [A1] B LOOP
;  Branch occurs here
```

Figure 5.23: Strlen Sequentialized

Program	Instructions Pipelined	Instructions Sequential	Branches	Optimizations
blk_move	26	13	3	unrolling 4 times epilog partially collapsed
dotprod	35	14	6	unrolling once
gouraud	63	49	2	unrolling 2 times epilog collapsed
iir	93	50	2	inner loop completely unrolled
latanal	47	35	2	unrolled once epilog collapsed
latsynth	27	25	3	prolog and epilog collapsing
max	57	31	2	unrolled 6 times
vecsumsq	39	20	6	epilog collapsed
w_vec	50	24	3	epilog collapsed

Table 5.1: Experimental Results Software De-Pipelining

5.8 Experimental Results

The software de-pipelining algorithms described in this chapter have been tested for several assembler loops written for the TIC62 DSP. Table 5.1 introduces the analyzed codes.

The first and second comlumns show the number of instructions before and after de-pipelining. The third column shows the number of branches that implement the software pipelined loop. The second column describes the additional optimizations that have been performed on the programs. Note that in those loops where collapsing tehcniques have been applied, the instruction difference is smaller than in examples with prolog and epilog.

5.9 Related Work

This section introduces interesting results from the field of software pipelining as well as de-pipelining. Section 5.9.1 presents further techniques for optimizing software pipelined loops. Section 5.9.2 describes the current software de-pipelining approaches.

5.9.1 Software Pipelining

Zhuge et al. - *Optimal Code Size Reduction for Software-Pipelined Loops in DSP Applications*

Code size expansion of software pipelined loops is a critical problem for DSP systems with strict code size constraints.

While software pipelining helps to achieve compact schedules, its disadvantage is the introduction of prologue and epilogue sections which cause code size expansion. Previous work on code size reduction of software pipelined loop such as code collapsing has the drawback that the quality of the code cannot be guaranteed.

Zhuge et al. [ZSS02] establish a theory and technique of code size reduction based on the retiming concept [Pap90, CS97]. The input code is transformed into a data flow graph whose nodes represent the operations to be performed, and whose edges illustrate precedence relations. A loop pipeline is depicted as a three-component object: prolog, repeating schedule and epilog to reflect the actual repeating pattern of the pipeline schedule.

The retiming model of software pipelining process makes that the prolog, new loop body and epilog can be derived directly from retiming functions.

A flexible scheduling technique presented by Chao and Sha, Rotation Scheduling [CLS93] is used to generate the pipelined schedule. In each rotation phase, it implicitly applies retiming operations on a set of nodes, which are then rescheduled to obtain a software pipelined schedule.

Three types of applications of this technique are presented:

1. Total code size reduction, totally removes the prolog and epilog assuming that there is enough conditional registers.

2. Partial code size reduction adjusts the number of prolog and epilog stages that are removed to the number of registers available.

3. Prolog or Epilog only code size reduction, removes only either the prolog or the epilog of the software pipelined loop

This technique is claimed to be a generalization of prolog/epilog collapsing [GSS$^+$] presented in section 5.3. Although there is no real comparison between both methods, this technique allows more control on how many stages are to be removed.

However, the idea of code size reduction is the same as in the collapsing approach, meaning that in both cases prolog and epilog stages are folded back into the loop kernel. Thus, the resulting software pipelined loop will have the same structure as considered by our software de-pipelining approach, which could then be applied without modifications.

Granston - *Software Pipelining Irregular Loops On The TMS320C6000 VLIW DSP Architecture*

Traditionally, software pipelining has been restricted to *regular* (FOR) loops. More recently, software pipelining has been extended to *irregular* loops, but only on architectures that provide special-purpose hardware.

Granston et al. describe their experience extending a production compiler for the TIC6x family to software pipeline irregular loops [GSZ01]. They preprocess irregular loops so that they can be handled by the existing software pipeliner.

The challenge with pipelining these kind of loops is that there is no adwance warning when the loop execution is nearing completion. Hence, there is no way to know when to enter the epilog and begin draining the pipeline. Therefore, to consider an irregular loop as a softwre pipelining candidate, the compiler must preprocess the loop so that each instruction can be safely over-executed. Then, there is no need

for a pipe-drain phase (epilog). Execution simply halts when the last valid iteration completes.

Basically, the preprocessing consists on introducing a new register which stores the result of the condition that guards the branch. This new register becomes the new guard for the branch as well as it guardes the loop instructions to guarantee safe over-execution.

This approach provides good results when automatically software pipelining irregular loops for the TIC6x architecture. Speedups of up to 9.00 are achieved for the tested benchmarks.

5.9.2 Software De-ipelining

B. Su et al. - *Software De-pipelining*

The only approach for software de-pipelining presented to date was proposed by Su et al. in [SWHM04]. Whereas Su de-pipelines loops directly from the assembler code, the approach presented in this thesis uses the control flow graph as working structure. Since, in compilation all optimizations are performed on a CFG, it is more intuitive to perform this reverse optimization on the CFG as well. In addition, the lack of delayed instructions make the process simpler in several ways.

An additional difference is the fact that our approach has been integrated in a decompilation framework, while Su et al. presented a rather formal result.

B. Su et al. - *Software De-pipelining for Nested Loops*

In [BKS05] Su et al. propose an approach for software de-pipelining nested loops. Nested loops of depth 2 are considered. Due to the high difficulty of software pipelining nested loops [WS98], there is no compiler that can perform this optimization automatically and the manual implementation consists in general on overlapping only two iterations of the outer loop.

In order to sequentialize a nested loop, the outer loop is de-pipelined first. Then, the inner loop is de-pipelined and the sequential version is obtained as explained in [SWHM04]. Finally, the code of the outer loop is scheduled.

Chapter 6

Conclusions and Further Work

This thesis has presented reverse compilation techniques to handle the specific problems originated by modern approaches in the digital signal processor design. Latest trends in DSP include pipelines and a VLIW architecture in order to fulfil the performance demands of current digital signal processing algorithms.

Software optimizations such as instruction scheduling and software pipelining intend to take the most advantage of these hardware features. Unfortunately, they introduce extra difficulties when it comes to decompile the assembler code into a high-level language like C: parallel instructions, delay slots, scheduling of instructions and software pipelined code become important problems for a reverse compiler.

This thesis proposes several techniques to deal with these problems. The delay resolution and with it the reversing of the instruction scheduling is performed together with the construction of a control flow graph for the program. An analysis of the branches in the program allow to define the edges of the CFG. The edge information is then used to determine the boundaries of the basic blocks. In a third step, the delays of the instructions are resolved by placing the instructions in the positions where they finish executing instead of where they start execution, as is in the assembler code.

At the same time, this approach takes care to maintain the semantics of the program by considering possible data conflicts that can arise when moving instructions around. Temporary variables for storing intermediate values are used whenever an input variable of an instruction is redefined between its original position and its destination.

Assembly language code for a VLIW architecture can explicitly express parallelism. Since high-level languages do not have this property, the groups of parallel instructions need to be rewritten in a sequential form. Here, an algorithm has been described to solve this problem, which also takes the possible data dependences between the instructions into account.

In general, the first contribution of this thesis is an approach for constructing a CFG of an assembly language program that is suitable for decompilation removing all VLIW specific characteristics.

A particular problem is introduced by software pipelining loops. Although the CFG construction approach can deal with software pipelining and will generate a

correct control flow graph without delays, it will be rather complex. In addition, the C code that can be generated from this CFG will be very hard to understand. A technique for recognizing a software pipelined loop in the CFG and recovering its original sequential form is another contribution of this thesis. The proposed algorithm can handle software pipelined loops whose prolog or epilog have been collapsed to reduce the code size.

All techniques presented here have been tested in the framework of a decompiler for the TIC62x DSP. However, they are completely architecture independent.

Further Work

In the context of our reverse compilation framework, a PhD thesis in progress by Ivan Prianichnikov has developed an approach that detects program functions in general, when no explicit call and return instructions are available. Whereas the general approach for function recognition is based on the semantics of the assembler file, this work describes a more independent technique that uses memory analysis to determine the targets of indirect jumps. Assuming that function names are known, this allows to detect calls and returns. In addition, this analysis helps resolving other indirect jumps providing an approximation of the possible jump targets.

Furthermore, data structures are detected in memory. Although no work has been done yet to define those data structures, this provides a good starting point for an accurate type analysis.

Determining the types of variables is one of the main open issues in decompilation. In particular for DSP applications, that perform a large number of array operations, it would be very interesting to be able to detect vectors and other structures. Compiler generated assembler usually contains additional information that could eventually help to solve this problem, but a general approach that can handle hand-written code does not seem feasible in general.

In the control flow graph construction for reverse compilation introduced in this thesis, more accurate data dependence information could be used in the sequentialization and delay resolution for better results.

Software de-pipelining is a rather new reverse optimization. The technique presented in this work handles only simple loops. Although other loop optimizations have been discussed, there are still open problems, like the effect of loop compaction, that should be handled.

An approach that is able to de-pipeline all types of nested loops needs some research effort. There are other techniques that also overlap iterations of nested loops, like conditionally executing the code of the outer loop inside of the inner loop. Reversing these optimizations requires an important knowledge of the semantics of the program, which makes an automatic solution particularly challenging. The effects of other optimizations of software pipelined loops such as code compaction is also an open problem.

Bibliography

[Amm92] Zahira Ammarguellat. A control-flow normalization algorithm and its complexity. *IEEE Transactions on Software Engineering*, 18(3):237–251, March 1992.

[App98] A.W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Bar74] P. Barbe. The piler system of computer program translation. September 1974.

[BB91] J. Bowen and P. Breuer. Decompilation techniques. 1991.

[BB92] P.T. Breuer and J.P. Bowen. Decompilation: The enumeration of types and grammars. 1992.

[BB93] P.T. Breuer and J.P. Bowen. Decompilation: the enumeration of types and grammars. In *Transaction of Programming Languages and Systems*, 1993.

[BB94a] Peter T. Breuer and Jonathan P. Bowen. Decompilation: The enumeration of types and grammars. *ACM Transactions on Programming Languages and Systems*, 16(5):1613–1647, September 1994.

[BB94b] P.T. Breuer and J.P. Bowen. Generating decompilers. 1994.

[BBL93a] J. Bowen, P. Breuer, and K. Lano. A compendium of formal techniques for software maintenance. In *Software Engineering Journal*, pages pages 253–262, 1993.

[BBL93b] J. Bowen, P. Breuer, and K. Lano. A compendium of formal techniques for software maintenance. In *Software Engineering Journal*, pages pages 253–262, 1993.

[BKH05] Nerina Bermudo, Andreas Krall, and Nigel Horspool. Control flow graph reconstruction for assembly language programs with delayed instructions. In *Proceedings SCAM05*, 2005.

[BKS05] Nerina Bermudo, Andreas Krall, and Bogong Su. Software de-pipelining for nested loops. Technical report, 2005.

[boo02] Boomerang web page., 2002.

[Bow91] J. Bowen. From programs to object code using logic and logic programming. In *In R. Giegerich and S.L. Graham, editors, Code Generation - Concepts, Tools, Techniques, Workshops in Computing*, pages pages 173–192, 1991.

[Bow93] J. Bowen. From programs to object code and back again using logic programming: Compilation and decompilation. In *Journal of Software Maintenance: Research and Practice*, pages 5(4):205–234, 1993.

[Bri81] D.L. Brinkley. Intercomputer transportation of assembly language software through decompilation. October 1981.

[Cap03] G. Caprino. Rec - reverse engineering compiler, 2003.

[CHW02] Keith D. Cooper, Timothy J. Harvey, and Todd Waterman. Building a control flow graph from scheduled assembly code. Technical Report TR02-399, Rice University, June 2002.

[Cif] C. Cifuentes. The dcc decompiler.

[Cif94] Cristina Cifuentes. *Reverse Compilation Techniques.* PhD thesis, Queensland University of Technology, School of Computing Science, July 1994.

[CLS93] Liang-Fang Chao, Andrea S. LaPaugh, and Edwin Hsing-Mean Sha. Rotation scheduling: A loop pipelining algorithm. In *DAC*, pages 566–572, 1993.

[CS97] Liang-Fang Chao and Edwin Hsing-Mean Sha. Scheduling data-flow graphs via retiming and unfolding. *IEEE Trans. Parallel Distrib. Syst.*, 8(12):1259–1267, 1997.

[CSF98] Cristina Cifuentes, Doug Simon, and Antoine Fraboulet. Assembly to high-level language translation. In *ICSM*, pages 228–237, 1998.

[Dav58] M. Davis. *Computability and Unsolvability.* McGraw-Hill, 1958.

[Dob00] Gerhard Doblinger. *Signalprozessoren.* J. Schlembach Fachverlag, 2000.

[EH94] Ana M. Erosa and Laurie. J. Hendren. Taming control flow: A structured approach to eliminating goto statements. In *Proceedings: 5th International Conference on Computer Languages*, pages 229–240. IEEE Computer Society Press, May 1994.

[Fri74] F.L. Friedman. *Decompilation and the Transfer of Mini-Computer Operating Systems*. PhD thesis, Purdue University, Computer Science, August 1974.

[FZ91] C. Fuan and L. Zongtian. C function recognition technique and its implementation in 8086 c decompiling system. 1991.

[FZL93] C. Fuan, L. Zongtian, and L. Li. Design and implementation techniques of the 8086 c decompiling system. 1993.

[GSS+] E. Granston, R. Scales, E. Stotzer, A. Ward, and J. Zbiciak. Controlling code size of software-pipelined loops on the tms320c6000 vliw dsp architecture.

[GSZ01] Elana D. Granston, Eric Stotzer, and Joe Zbiciak. Software pipelining irregular loops on the tms320c6000 vliw dsp architecture. In *LCTES/OM*, pages 138–144, 2001.

[Gup98] Rajiv Gupta. A code motion framework for global instruction scheduling. In *International Conference on Compiler Construction*, LNCS 1383, pages 219–233, Lisbon, Portugal, 1998. Springer Verlag.

[Hal62] M.H. Halstead. *Machine-independent computer programming*. Spartan Book, 1962.

[Hal67] M.H. Halstead. Machine independence and third generation computers. In *In Proceedings SJCC (Sprint Joint Computer Conference)*, pages pages 587–592, 1967.

[Hal70] M.H. Halstead. Using the computer for program conversion. In *Datamation*, pages pages 125–129, 1970.

[HH73] B.C. Housel and M.H. Halstead. A methodology for machine language decompilation. December 1973.

[Hol73] C.R. Hollander. *Decompilation of Object Programs*. PhD thesis, Stanford University, Computer Science, January 1973.

[Hoo91] S.T. Hood. Decompiling with definite clause grammars. 1991.

[Hop78] G.L. Hopwood. *Decompilation*. PhD thesis, University of California, Irvine, Computer Science, 1978.

[Hou73] B.C. Housel. *A Study of Decompiling Machine Languages into High-Level Machine Independent Languages*. PhD thesis, Purdue University, August 1973.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, 1990.

[HZY91] L. Hungmong, L. Zongtian, and Z. Yife. Design and implementation of the intermediate language in a pc decompiler system. Mini-Micro Systems, 1991.

[Inc00] Texas Instruments Inc. Tms320c6000 cpu and instruction set reference guide, 2000.

[Ins02] Texas Instruments. Tms320c6000 programmerś guide, spru198g, 2002.

[JC96] Johan Janssen and Henk Corporaal. Controlled node splitting. In Tibor Gyimóthy, editor, *Compiler Construction, 6th International Conference*, volume 1060 of *Lecture Notes in Computer Science*, pages 44–58, Linköping, Sweden, April 1996. Springer.

[JSW99] Adrian Johnstone, Elizabeth Scott, and Tim Womack. Experience paper: Reverse compilation of digital signal processor assembler source to ansi-c. In *ICSM*, pages 316–325, 1999.

[JSW00a] Adrian Johnstone, Elizabeth Scott, and Tim Womack. Reverse compilation for digital signal processors: A working example. In *33rd Hawaii International Conference on System Sciences-Volume 8*, 2000.

[JSW00b] Adrian Johnstone, Elizabeth Scott, and Tim Womack. What assembly language programmers get up to: Control flow challenges in reverse compilation. In *CSMR*, pages 83–92, 2000.

[KFH05] Andreas Krall, Stefan Farfeleder, and Nigel Horspool. Ultra fast cycle-accurate compiled emulation of inorder pipelined architectures. In Jarmo Takala, editor, *SAMOS 2005*, LNCS 3553, pages 222–231, Samos, July 2005. Springer.

[Lam88] Monica Lam. Software pipelining: An effective scheduling technique for vliw machines. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, 1988.

[Ltd01] Software Migrations Ltd. The assembler comprehension and migration specialists, 2001.

[Mor98] Robert Morgan. *Building an Optimizing Compiler*. Butterwort-Heinemann, 1998.

[Myc99] Alan Mycroft. Type-based decompilation. *Lecture Notes in Computer Science*, 1576:208–??, 1999.

[Pap90] M. C. Papaefthymiou. ON RETIMING SYNCHRONOUS CIRCUITRY AND MIXED-INTEGER OPTIMIZATION. Technical Report MIT/LCS/TR-486, 1990.

[PKS02] Mark Probst, Andreas Krall, and Bernhard Scholz. Register liveness analysis for optimizing binary translation. In Elizabeth Burd and Arie van Deursen, editors, *Working Conference on Reverse Engineering*, Richmond, October 2002. IEEE.

[PW93] D.J. Pavey and L.A. Winsborrow. Demonstrating equivalence of source code and prom contents. In *The Computer Language*, pages 36(7):654–667, 1993.

[RC03] Norman Ramsey and Cristina Cifuentes. A transformational approach to binary translation of delayed branches. *ACM Trans. Program. Lang. Syst.*, 25(2):210–224, 2003.

[Reu88] J. Reuter. Downloadable from the decompdecompiler page, 1988.

[Sas66] W.A. Sassaman. A computer program to translate machine language into fortran. In *In Proceedings SJCC*, pages pages 235–239, 1966.

[SBB+00] Bjorn De Sutter, Bruno De Bus, Koenraad De Bosschere, Peter Keyngnaert, and Bart Demoen. On the static analysis of indirect control transfers in binaries. In Hamid R. Arabnia, editor, *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)*, pages 1013–119, Las Vegas, USA, June 2000. CSREA Press.

[SD05] N. Snavely and S. Debray. Unpredication, unscheduling, unspeculation: Reverse engineering itanium executables. In *IEEE transactions on Software Engineering, 31(2)*, 2005.

[SW74] V. Schneider and G. Winiger. Translation grammars for compilation and decompilation. pages 14:78–86, 1974.

[SWHM04] B. Su, J. Wang, E. Hu, and J. Manzano. Software de-pipelining technique. In *Proceedings Of SCAM2004*, 2004.

[Tho] Peter Thoeny. The software transformation wiki.

[War89] M. Ward. *Proving Program Refinements and Transformations.* PhD thesis, Oxford University, 1989.

[War99a] M. P. Ward. Assembler to c migration using the fermat transformation system. In *Proceedings of ACSM'99*, pages pages 67–76, 1999.

[War99b] Martin P. Ward. Assembler to c migration using the fermat transformation system. In *ICSM*, pages 67–76, 1999.

[War00] M. P. Ward. Reverse engineering from assembler to formal specifications via program transformations. In *Proceedings of th 7th Working Conference on Reverse Engineering*, 2000.

[War01a] M. P. Ward. The fermat transformation system, 2001.

[War01b] Martin P. Ward. The fermat assembler re-engineering workbench. In *ICSM*, pages 659–662, 2001.

[War01c] Martin P. Ward. The fermat assembler re-engineering workbench. In *ICSM*, pages 659–662, 2001.

[War04] Martin P. Ward. Pigs from sausages? reengineering from assembler to c via fermat transformations. *Sci. Comput. Program.*, 52:213–255, 2004.

[Wor78] D.A. Workman. Language design using decompilation. December 1978.

[WS98] J. Wang and B. Su. Software pipelining of nested loops for real-time dsp applications. In *Proceedings ICASSP*, 1998.

[ZSS02] Qingfeng Zhuge, Zili Shao, and Edwin Hsing-Mean Sha. Optimal code size reduction for software-pipelined loops on dsp applications. In *ICPP*, pages 613–620, 2002.