Performance Improvement through Predicated Execution in VLIW Machines

Morteza Biglari-Abhari

Thesis submitted for the degree of

Doctor of Philosophy

Department of Electrical and Electronic Engineering
University of Adelaide
South Australia
5005

February, 2000

Contents

List of Figures

List of Tables

Abstract

State-of-the-art VLSI technology provides the opportunity to implement a complex microarchitecture with many functional units capable of parallel operation. To employ the machine parallelism effectively requires extracting ILP from the application programs as much as possible. Hardware techniques to extract ILP have an impact on cycle time, which may result in lower performance. To achieve higher clock rates, hardware can be used as only the execution engine and ILP extraction is transferred to the compiler. One way to achieve this is through employing the Very Long Instruction Word (VLIW) architecture to implement a wide-issue ILP processor.

One of the major obstacles to the exploitation of ILP is the uncertain change in instruction flow caused by conditional branches. Keeping multiple functional units busy most of the time requires the executing operations from multiple execution paths. Therefore, techniques should be employed to reduce the impact of conditional branches which are less predictable in general-purpose applications. Predicated or guarded execution as an architectural model reduces these limitations. Predicated execution refers to the conditional execution of an operation based on the value of a boolean source operand, referred to as the predicate. Predicated execution is exploited through a structure called the hyperblock which was developed by the IMPACT compiler group.

This thesis investigates techniques to achieve performance improvement in VLIW machines through predicated execution. We describe an experimental processor based on VLIW architecture called EVA. The back-end of an optimising compiler for EVA is implemented in order to investigate the proposed techniques. These include improving the quality of the generated code through estimating the resource usage during the code generation. This is applied at the time of generating the predicated code (through if-conversion) and at the time of prepass scheduling.

VLIW architectures traditionally have not been used extensively for general-purpose ap-

plications. One of the major reasons is lack of object code compatibility among different implementations of the same architecture. This is due to the detailed microarchitectural information employed by the compiler to generate the binary code. When the assumed microarchitectural features are changed, the previously generated code not only may suffer performance loss, but also may generate incorrect results. We propose a new approach to overcome this problem.

To my wife,

my parents, my sons,

and all my teachers from the first day of school.

This work contains no material which has been accepted for the award of any other degree or diploma in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text.

I give consent to this copy of my thesis, when deposited in the University Library, being available for loan and photocopying.

Signed: Date: 17 / 2 / 2000

(Morteza Biglari-Abhari)

Acknowledgements

First, I would like to thank my supervisors Prof. Kamran Eshraghian and Dr. Michael J. Liebelt for their guidance and support throughout my studying.

I would like to thank all staff and postgraduate students in the Department of Electrical and Electronic Engineering for friendly and nice environment to study. I also thank the Computer Science Department for providing access to Silicon Graphics Power Challenge computing system for this research, especially Dr. Francis Vaughan.

I would like to thank my parents for their love, support and encouragement since my first days of school.

Finally and most importantly, I would like to thank my wife and my sons for their love and patience, which gave me the ability and power to work hard in my research.

Chapter 1

Introduction

Performance improvement in high-speed processors relies upon optimising architectural features, implementing in underlying VLSI technology and the ability to extract the available parallelism in programs. CPU time, the main criteria for performance evaluation, as described in [Hennessy and Patterson, 1996] is proportional to three interrelated factors as indicated in Figure 1.1.

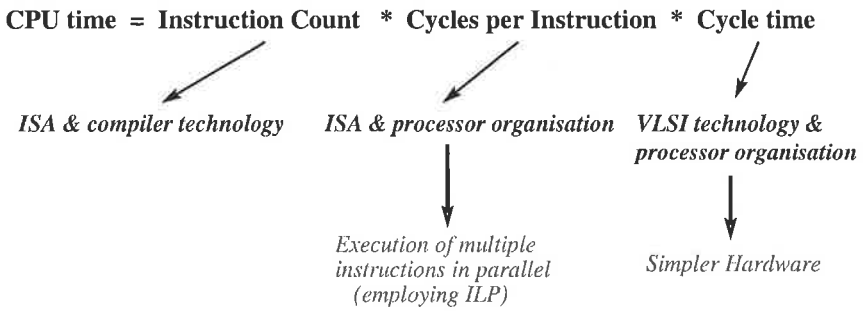


Figure 1.1: Three major factors in computation of CPU time.

Optimisation of one factor independently may have a severe impact on other factors. For example, decreasing cycles per instruction (CPI) through designing specific processor architecture and organisation may increase cycle time. Therefore, it is necessary to evaluate the possible impact of each improvement on other factors.

The research described in this thesis is focused on techniques to reduce cycles per instruction. In the meantime, possible side effects of these improvements which may increase other factors are considered.

Instruction Level Parallelism (ILP) is employed to enhance performance through providing a mechanism to overlap different phases of instruction processing (referred to as pipelining), and multiple data paths and functional units. In addition, new compiler transformations are applied to expose parallelism and/or assist the hardware processing engine. ILP processing has been used extensively in recent years. State-of-the-art microprocessors can fetch and complete multiple operations per cycle [Dunn and Hsu, 1996, Yeager, 1996, Papworth, 1996, Kessler, 1999, Horel and Lauterbach, 1999].

There are some limitations in achieving a high level of ILP. Real and synthetic data dependencies exist among the operations, preventing parallel execution of them. Synthetic data dependencies are due to a shortage of resources (such as registers), or conservative decisions made regarding unresolved memory dependencies. Real data dependencies come from the intrinsic order of operations in the program, which are the result of implementing the required algorithm in the program. In addition to data dependencies, research studies indicate that about 20% to 25% of operations in general-purpose applications are conditional branches [Lam and Wilson, 1992]. These are obstacles to reordering operations for parallel execution when the direction and target of branches are not known.

A large amount of research has been done to overcome or reduce these problems. Execution of an operation before knowing that its execution is required is known as speculative execution, and this is employed extensively to reduce control dependencies. Speculative reordering of operations changes the original sequence of operations by predicting the directions of conditional branches. In this way, more operations can be provided for parallel execution to the execution engine. It usually requires special hardware for recovery of mispredictions.

Eliminating the conditional branches is another effective technique to reduce control dependencies. Predicated execution is a technique to remove conditional branches and converting control to data dependencies. In this technique, operations are executed conditionally based on the value of a boolean source operand referred to as the predicate, which is evaluated at run time. [Hsu and Davidson, 1986, Rau et al., 1989]. Compiler optimisations and scheduling are performed on a region of operations. A structure called a *hyperblock*, which was introduced by the IMPACT compiler group [Mahlke et al., 1992b], is employed to make the region exposed to the compiler. Hyperblock is a region of predicated code with single entry and possible multiple exits.

To perform speculative and predicated execution, the direction and target of conditional branches are predicted statically by the compiler or dynamically by the hardware. Static branch prediction techniques are based on heuristics or execution profile information. Although profile information is gathered for some specific program inputs, it was shown that it can be considered valid for almost all inputs [Fisher and Freudenberger, 1992]. In addition, some techniques have been proposed to gather and update profile data at execution time when it is necessary [Conte et al., 1996c].

Regarding data dependency reduction, research has been conducted in two directions, eliminating synthetic (or false) data dependencies and reducing real data dependencies. In static techniques which are applied at compile time, resource renaming (such as register renaming [Cytron and Ferrante, 1987]) removes the false dependencies due to resource limitation. In dynamic techniques at run time, more physical resources are provided to remove dependencies due to the architectural assumptions. For example, for dynamic register renaming the number of physical registers is greater than the number of architectural registers and an architectural register may be mapped to different physical registers at execution time.

Some studies have targeted reduction of real data dependencies. A technique called value prediction makes it possible to overlap execution of operations with real data dependencies [Lipasti and Shen, 1997b]. This technique is based on the concept of value locality, by which is meant the probability of availability of the previous value in a storage location. Also, the dependence prediction technique allows the execution of operations speculatively before their data dependencies are detected [Lipasti and Shen, 1997c]. Value and dependence prediction are applied and recovered at run time using special hardware. For unresolved memory access dependencies, when static memory access disambiguation techniques fail to detect dependency between two memory access operations, they are assumed free of dependency and are reordered speculatively. Later, at run time special hardware checks the correctness of speculation and performs the required recovery [Gallagher et al., 1994].

Two major processing paradigms have been employed for ILP processing. These are Very Long Instruction Word (VLIW) [Fisher, 1983] and superscalar processors [Johnson, 1991]. VLIW processors rely on the compiler to extract and employ ILP, while in superscalar processors this is performed by special hardware at execution time.

This thesis investigates techniques to improve the performance of VLIW processors. This includes employing techniques to expose more ILP through predicated execution and over-

coming one major problem of VLIW architectures, which is lack of object code compatibility for different generations of the same architecture.

1.1 Contribution of the Thesis

The contributions can be summarised as follows:

- An experimental VLIW architecture (called EVA) and its optimising compiler have been designed and implemented. Predicated execution is exploited in the compiler through hyperblock structures.
- A new algorithm is presented to calculate the priority of operations in the hyperblock scheduling process. The priority of an operation is calculated based on its dependency height and resource usage.
- A new algorithm for predicate sensitive register allocation for hyperblocks is proposed.
- A new algorithm is proposed to overcome the performance loss of the original hyperblock formation heuristic for non-uniform processors.
- A technique to provide binary compatibility among different implementations of the same VLIW architecture is presented.

1.2 Outline of the Thesis

The thesis is organised in eight chapters. Chapter 2 presents an overview of the state-of-the-art of compiler techniques in ILP processing. A brief comparison to some related hardware techniques is also presented. This chapter is provided as a general background in this area.

Architectural features of VLIW processors and their problems are described in chapter 3. Description of our experimental VLIW architecture (EVA) is also presented.

Chapter 4 describes the design and implementation of the VLIW compiler for the EVA. This compiler is based on the **SUIF** infrastructure, which includes *SUIF* [SUIF, 1994] and machine SUIF (*machsuif*) [Smith, 1997] from Stanford and Harvard universities respectively. Our new algorithms are described in detail.

Experimental evaluation techniques and performance assessment of the EVA compiler is presented in chapter 5 with experimental results. SPEC95 integer benchmark programs and some Unix utilities are used as benchmark programs.

Chapter 6 describes a new algorithm which improves the performance of block selection for hyperblock formations.

Our technique to provide object code compatibility in VLIW machines is presented in chapter 7. Our approach is performed with the help of code annotation provided by the compiler to reduce the complexity of the required hardware.

Chapter 8 presents a summary and possible future directions of this research.

Chapter 2

Compiler Techniques in ILP Processing

2.1 Introduction

To achieve high performance processing, more work must be done in a smaller amount of time. This means that some sort of parallelism is employed in the different stages of the processing paradigm. The amount of parallelism depends on the application, and generally the parallelism present in programs can be divided into *coarse-grain* versus *fine-grain.*

Coarse-grain parallelism refers to the parallelism between different sets of instructions such as subroutines in a program. It is usually employed by multiprocessor systems. Fine-grain parallelism indicates the parallelism between individual instructions in the program, and is the so-called *Instruction Level Parallelism (ILP).*

ILP processing involves both compiler and hardware methods. ILP is the result of interaction of the available program parallelism and the machine parallelism capabilities. The amount of work done by each depends on the adopted ILP processing paradigm [Rau and Fisher, 1993].

To extract instructions so that their execution can be overlapped at the same time, the program must be examined by a compiler and/or hardware. For this purpose, a window of instructions is established. An instruction window is the collection of instructions examined for parallelism at a time. Instructions are scheduled considering constraints on ILP and resource limitations.

Compilers utilise appropriate program representation schemes to facilitate ILP extraction. The amount of ILP that can be extracted and exploited is limited by the dependency constraints. Thus, to increase the amount of exploitable ILP these dependencies must be

eliminated or reduced. Figure 2.1 shows an example of the dependencies.

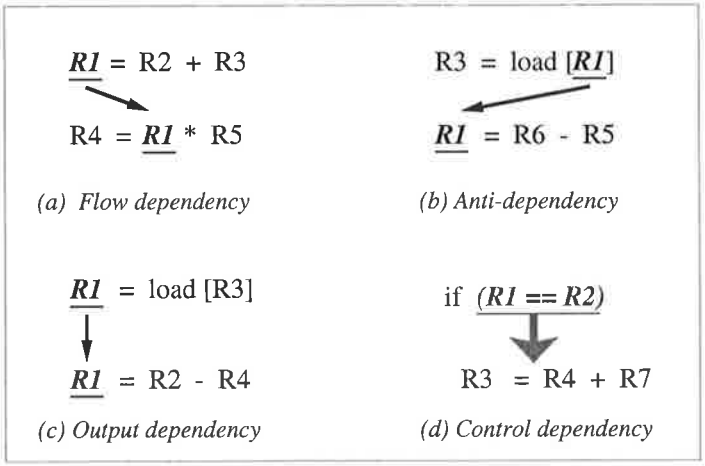


Figure 2.1: Example of data and control dependencies.

The true dependency, which is the main obstacle in parallelising operations is *flow dependency* (or Read-after-Write). *Anti-dependencies* (Write-after-Read), and *output dependencies* (Write-after-Write) are due to storage conflicts, and can be removed by providing more storage capability and using techniques such as *register renaming* [Keller, 1975, Cytron and Ferrante, 1987], and *memory address disambiguation* [Johnson, 1991]. Since two memory operands in a program may have different data dependency characteristics during different periods of execution, determination of their data dependency is more difficult than the determination of data dependency between two register operands. Some techniques have been proposed to reduce the impact of unresolved memory accesses [Chen, 1993, Kathail et al., 1994], and these are reviewed in section 2.3.2.

In addition to data dependencies, control dependencies impose a severe problem and their impact may be reduced through techniques such as *speculative execution*. Control dependencies can be converted to data dependencies and vice versa. Techniques to reduce the number of branches in a program and convert control dependency to data dependency are discussed in section 2.2.2.

In this chapter, basic compiler techniques in ILP processing are reviewed. This is provided as a general background for the rest of the thesis. This includes issues related to techniques for relaxing control and data dependencies and instruction scheduling.

2.2 Control Dependency Reduction

A sequence of instructions with a single entry and a single exit point is referred to as a basic block. Typically, there are fewer than 5 instructions on average in a basic block in non-numeric programs [Lam and Wilson, 1992]. Therefore, there is not enough parallelism within individual basic blocks to achieve significant performance improvements through ILP, and higher levels of ILP may only be obtained through inspection of successive basic blocks. In non-numeric programs, conditional branches are a major obstacle that limit ILP [Smith et al., 1989, Joupi and Wall, 1991]. Therefore, a mechanism is required for parallel execution of operations across basic block boundaries. Two main techniques are used to reduce the impact of conditional branches, which are speculative and predicated execution.

2.2.1 Speculative Execution

Execution of an operation before knowing that its execution is needed, is called *speculative execution*. This technique can be employed at compile-time and/or run-time. Run-time speculation is performed through dynamic scheduling [Johnson, 1991]. Dynamic branch prediction is used to select the most likely execution path. There are more opportunities for *speculative code motion* at compile-time in comparison to run-time techniques due to use of a larger instruction window to pick up the data independent operations. However, hardware branch predication techniques are more accurate [Johnson, 1991], so execution of the speculative operation is more likely to be beneficial in this case.

Compile-time speculation can improve performance when the processor resources are utilised efficiently. This means that the amount of speculation should be matched with the resource consumption pattern of the related operations. To reduce the dependency length of more frequently executed paths, the starting operation in this path should be executed as early as possible to overlap its execution with other operations, resulting in a shorter execution time. Research by Lam and Wilson indicated that speculative execution increased the average number of independent operations per cycle in non-numerical applications from 2.1 to 6.8 [Lam and Wilson, 1992]. This clearly indicates the importance of speculative execution.

Speculative code motion must be safe so that the program execution semantics and the processor state are not changed if the result of the branch prediction is not correct. For exam-

ple, a speculated operation should not cause an *exception* that terminates the program. Also, it should not destroy the live registers on the other paths when the branch is mispredicted. This limits the applicability of speculative code motion.

Register renaming can eliminate the problem of overwriting live registers on the other path for the speculative operation.

Architectural Models to Handle Trapping Operations

To handle trapping operations, three architectural models have been studied in the literature [Chang et al., 1995].

Restricted Code Motion - In this model, the operation is moved above the conditional branch if it cannot cause an exception and not overwrite the destination register, which is live on the other path. Without a sophisticated compile-time analysis, the performance of this model is limited.

General Code Motion - To support more speculative code motion, a class of non-trapping counterparts for the trapping operations may be added to the ISA [Colwell et al., 1988]. The non-trapping versions are included for floating-point arithmetic, memory loads and integer divide operations. Speculative operations which may potentially cause an exception are converted into their silent form. Therefore, the possible exceptions are ignored.

The floating-point arithmetic and integer divide functional units have a mechanism to raise the exception flag only for the trapping version of the operation. Also, when an access violation occurs for a non-trapping load, the load is aborted.

When the branch is mispredicted, and there was an exception condition for the speculative operation, the exception is ignored and the program is correct. However, for the correctly predicted branch, not asserting the exception when there is an exception condition may not be acceptable for some special applications such as transaction processing. In this case, the general code motion should be used with some additional hardware and software support for exception handling to get the correct result for the program.

Boosting Code Motion - To relax both restrictions on upward code motion, a technique called *boosting* was introduced [Smith et al., 1990, Smith et al., 1992]. In this model, a speculative operation is moved above a branch with no restriction. The processor state is not changed until the branch commits. Special buffers as shadow structures are provided to save the results before they change the state of the processor. The shadow register file holds

the result of boosted operations which write into registers. The shadow store buffer holds the value of the boosted store operations. When the branch commits, if it was predicted correctly, the contents of the shadow register file and the shadow store buffer are transfered to the sequential register file and the store buffer respectively. Otherwise, the shadow buffers are flushed. Figure 2.2 illustrates the general architecture which can be used for boosting.

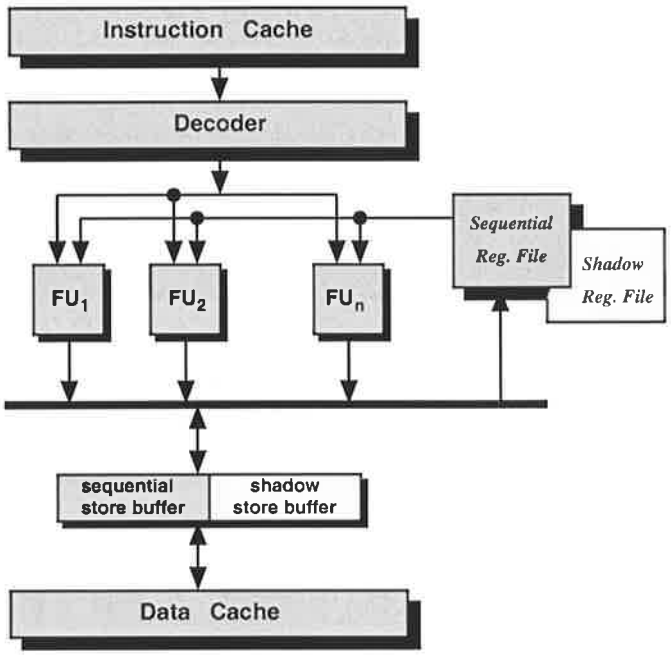


Figure 2.2: General architecture to support boosting code motion.

If an operation is to be moved above more than one branch, a separate buffer is required for each branch. Furthermore, to transfer boosting information from the compiler to the hardware, extra bits are added to the instruction words, and one bit is used for speculation over one branch.

Support for Exception Handling

Correct handling of exceptions is the main problem for speculative execution. Accurately detecting and reporting exceptions is necessary to find program errors resulting in an exception. The proposed techniques delay the exception signal of speculative operations. The exception is handled when the outcome of the branch upon which the speculative operation is control

dependent, is known. Several methods relying on the compiler with hardware support have been proposed in the literature to handle this problem.

For Boosting Code Motion, any exception occurrence is marked in the relevant shadow structure for the boosted operation [Smith et al., 1990, Smith et al., 1992]. When the excepting operation is to be committed, the shadow structure is checked to determine if an exception condition exists. The contents of the shadow structure are thrown away for the exception condition and control is transfered to the exception recovery code produced by the compiler. All speculative operations related to the same branch are re-executed sequentially. Since the processor state has not been changed, the exception is raised by re-executing the speculative operations. Then, the exception can be handled through any exception handling technique for pipelined processors. This model provides accurate detection and handling of speculated excepting operations at the expense of a substantial hardware cost.

Write-back suppression [Bringmann et al., 1993], like operation boosting, needs extra hardware to perform exception handling, and this increases significantly if the number of branches on which the operation is speculated increases. In this method, the home block of the potentially excepting operation prior to scheduling is used to suppress updates to the register file by the subsequent operations in the same home block or blocks after it. In this manner, it prevents destroying of live values in source registers for speculative operations, which are needed at the recovery time.

A field in the operation opcode, which is called *speculative distance*, indicates the number of branches above which the operation has been moved. A *check* operation located in the home basic block of the potentially excepting operation reports the exception. At recovery time, a stack of PC values of suppressed operations is used to select the operations to be re-executed.

This technique, like operation boosting, can only handle one execution path between the operation and its origin, and cannot spill registers related to a speculated operation [August et al., 1995]. Spill code refers to the collection of load and store operations inserted due to lack of enough registers to save and restore data.

Another technique referred to as the *sentinel speculation* model [Mahlke et al., 1992a] is based on compiler support and a few architectural changes such as an extra bit (*speculative bit*) for every operation and an *exception tag* for each register, which is at least 1 bit, to handle exceptions efficiently. It requires less hardware than the other techniques mentioned above.

For correct exception recovery, a non-speculative operation in the home basic block of the speculative operation is used as the *check* operation and is called a *sentinel*. This may be a flow dependent operation to the original speculative operation or a newly generated special operation. When execution control reaches the check operation, the exception is signaled.

When an exception occurs for the speculated operation, the current PC is saved in the destination register and the exception tag of the destination register is set. When other flow dependent speculative operations use this register as a source operand, they will propagate its contents and exception tag to their destination register. Once a non-speculative operation encounters a source register with the exception tag, the exception is signaled and the recovery block scheme [August et al., 1995] or in-line recovery [Mahlke et al., 1993] is employed to repair the exception. Sentinel speculation with recovery block is used in the PlayDoh architecture [Schlansker et al., 1997].

2.2.2 Predicated Execution

Control dependencies due to frequent and unpredictable branches largely limit ILP. Speculative execution removes dependencies between operations and branches. However, the branches are still present and appear as a performance bottleneck.

Predicated or guarded execution as an architectural model reduces these limitations. Predicated execution refers to the conditional execution of an operation based on the value of a boolean source operand, referred to as the predicate [Hsu and Davidson, 1986, Rau et al., 1989]. A technique called *if-conversion* [Allen et al., 1983, Park and Schlansker, 1991] is used to remove conditional branches, converting control dependencies to data dependencies. It replaces conditional branches with some comparison operations, which define a predicate. Operations dependent on these branches are converted to predicated operations. The predicated operation is executed if its predicate evaluates as 'true' at run time. Otherwise, it is nullified to keep the processor state unchanged.

Predicated execution is often an efficient method to reduce the impact of control dependencies. It can provide more opportunities for code motion. Figure 2.3 indicates a comparison of predicated and speculative code motion. Predicated operations are free to move below a merge point, while this is not possible through speculative execution. However, a predicated operation is restricted in upward code motion, so that it cannot move above the operation which defines its predicate. Therefore, a combination of speculation and predica-

tion provides more freedom in code motion.

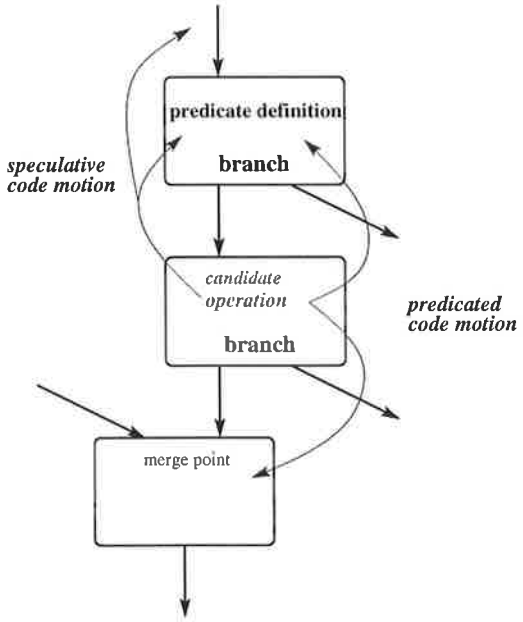


Figure 2.3: Comparison of speculative and predicated code motion opportunities.

Predicated code can be used for efficient speculation. As an example, Ando and his colleagues proposed predicated state buffering to reduce restrictions which limit speculation [Ando et al., 1995]. This simplifies the handling of side effects caused by speculative execution.

In this technique, the selected region is if-converted. The predicate for each operation is defined based on its control dependencies. The predicate of the operation is evaluated at the issue time. If it is 'true', the operation is executed. If it is 'false', the operation is squashed. When the value of the predicate is unspecified due to an undetermined condition, the operation is executed and its destination is marked as speculative. At write-back time, the speculative operations label the result with the predicate for later commit.

As architectural support, each entry in the register file and store buffer is augmented by an additional shadow storage, a predicate and three extra bits. One bit is used to indicate if the result is in the shadow storage or in the register (or store buffer entry). At commit time, it is only required to invert this bit to move the speculative result to the register. Another bit shows a speculative state, and the third bit indicates a pending exception, which is handled at commit time, when the associated predicate value is known. Each operation is encoded to indicate if the source operands are in the shadow storage or the actual one.

In this scheme, the processor operates in normal (referred to as sequential) or speculative mode. In each cycle the processor evaluates the predicate associated with each register (or a store buffer entry) using the conditional code register (CCR). The CCR has one bit for each branch in the region considered for if-conversion. If the predicate is 'true', the result is committed and a pending exception is handled. If it is 'false', it is squashed.

This method reduces the pressure on register allocation connected to the techniques described in the previous section for exception handling and it facilitates exception recovery. When an exception is signaled, the speculative state is invalidated. In this manner, operations semantically after the excepting operation which depend upon it are re-executed at recovery time. The CCR is not updated and a future conditional code register (CCR) keeps the conditions at the commit point. In the normal mode, upon entry into the region the value of the PC for the top operation is saved into a special register referred to as the RPC. In the recovery mode, the control is transferred to top of the region using the RPC. In this mode, those operations which have a 'true' or 'false' predicate referring to the CCR are squashed. Other operations are re-executed. If their predicates evaluate to 'true' by referring to the future CCR, the pending exception is handled. The recovery mode terminates when it reaches the original speculative exception commit point.

Another form of combined predication and speculation can be achieved through *predicate promotion* [Lin, 1992, Mahlke, 1996]. In this technique, the predicate of the operation is promoted to another predicate with fewer constraints such as the predicate of the operation which defined this predicate, or to 'true' value. Hardware support for speculative execution is necessary to handle the side effects of this technique too.

2.3 Data Dependency Reduction

The following techniques have been proposed in the literature to improve data dependency restrictions.

2.3.1 Register Renaming

Artificial data dependencies may be introduced by reusing registers. Software register renaming tries to assign a unique architectural register to each variable definition where appropriate to reduce the amount of output and anti-dependencies [Cytron and Ferrante, 1987]. It

has been used as a major technique in ILP compilation.

2.3.2 Memory Access Disambiguation

Load operations are often on the critical long latency paths in programs. Scheduling them as early as possible, so that other dependent operations can be executed earlier, increases the ILP. In addition to control dependencies, unresolved output and anti-dependencies between memory accesses are obstacles to this process. When two memory access operations refer to the same memory address, it is said they are aliases of each other [Aho et al., 1986]. To identify the potential aliases between loads and preceding stores, or between different stores, compilers attempt to disambiguate memory references. If compile-time memory disambiguation cannot prove that two memory accesses are to different memory locations, it is forced to schedule them conservatively. This may result in a large reduction in the ILP.

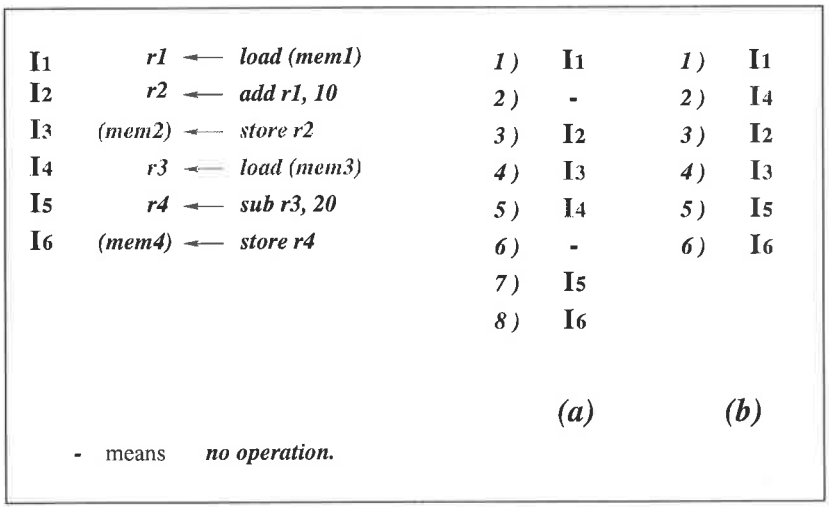


Figure 2.4: Scheduling example of potential conflicting memory accesses in a single issue processor. (a) Conservative scheduling. (b) Aggressive scheduling.

Compile-time memory disambiguation is a difficult problem and often it cannot produce the complete result. This is worse, especially for languages such as C, which extensively use pointers. Most current techniques are useful for vector access disambiguation. Figure 2.4 illustrates an example of how conservative scheduling may decrease performance. Operations I3 and I4 have different memory addresses. If the compiler is not able to prove this, it

schedules the code conservatively as indicated in Figure 2.4 (a), while it can be scheduled with two cycles less as shown in Figure 2.4 (b). In this example, a single issue processor is assumed. All operations have one cycle latency, except load which has two cycles latency.

Superscalar processors employ run-time techniques to find aliases of the same memory location. Due to the availability of the required information at the execution time, these techniques are more successful at resolving this issue. However, their lookahead scope to evaluate the potentially conflicting memory accesses is less than that of compiler methods. ILP processors which only rely on compile-time scheduling can also employ special hardware to increase the opportunities for memory access disambiguation. A mechanism referred to as the *memory conflict buffer* was proposed for this purpose [Chen, 1993].

A few architectural extensions are required to support this mechanism. Load operations are divided into two different forms, one which performs the load operation irrespective of the possible aliasing with the preceding stores and one which is used to verify the correctness of the first load at run-time to find if there is a conflict and fix it. These two different forms of loads are respectively called *pre_load* and *check* operations in [Chen, 1993], or *data speculative load* and *data verify load* in PlayDoh architecture [Kathail et al., 1994].

If it is intended to schedule the operations dependent upon the load, another form of check operation is required in addition to a compiler generated compensation code to impose the correct execution order, when conflict occurs. This check operation, which we call *check_branch*, transfers the control to the compiler generated compensation code if an alias is detected. In this case, the issues related to speculative code motion discussed in section 2.2 should be considered too. Figures 2.5 and 2.6 show examples of using pre_load and check operations.

A structure which may be called the Data Speculative Load Record (DSLR) is used by the memory conflict buffer to keep an execution record of *pre_load* operations. Figure 2.7 displays its general structure. The number of DSLR entries and the required logic is implementation dependent based on the number of architectural registers. In [Chen, 1993] a number of implementations ranging from a full-associative or set-associative address comparison to a simpler address hashing method was proposed.

The semantics of these new operations are as follows:

pre_load does the normal load operation. It invalidates any entry in the DSLR whose target register field is the same as the destination register of the *pre_load*. Depending on the

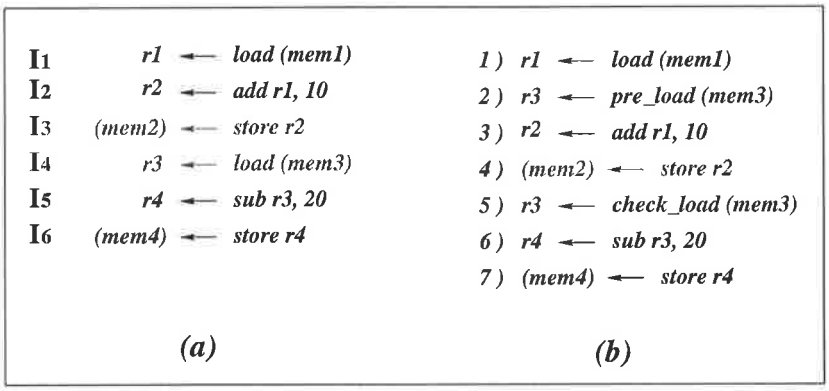


Figure 2.5: Example of increased scheduling opportunity of potential conflicting memory accesses in a single issue processor. (a) Original unscheduled code (b) Scheduled code using *pre_load* and *check_load* operations.

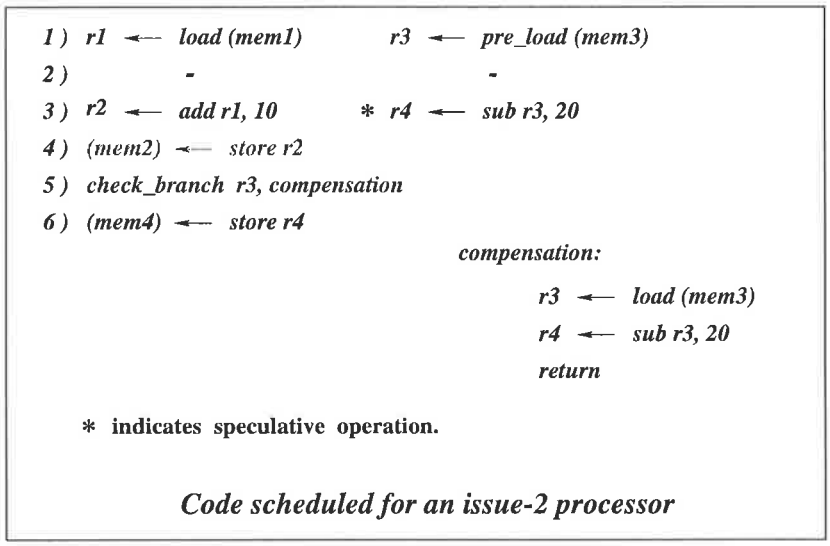


Figure 2.6: Scheduling example of a load and operations dependent upon it above a potential conflicting store.

implementation, a new entry may be assigned in the DSLR for this *pre_load*. At execution time, the memory addresses of store operations are checked with valid entries in the DSLR. The entry is invalidated if a conflict was detected. The *check_load* operation accesses the DSLR to find a valid entry with the same target register as its destination register. If one is found, this valid entry is invalidated and the *check_load* is nullified. Otherwise, it performs the normal load operation. The *check_branch* behaves like a *check_load* except that if the valid entry is not found it branches to the specified compensation routine.

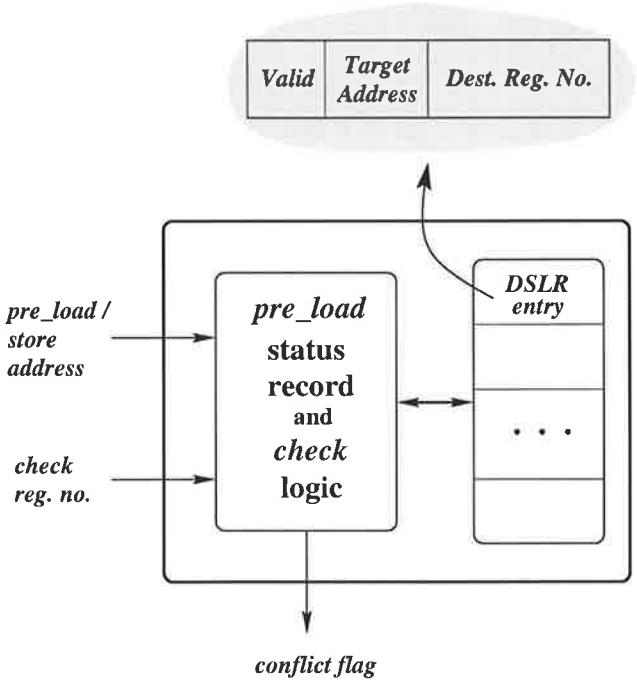


Figure 2.7: General structure to record and check the status of pre_load operations.

Some heuristics or profile information are required to decide which loads can be replaced by pre_load operations. This is because if conflict occurs frequently, performance loss due to the overhead of the memory conflict buffer degrades overall performance. Thus, pre_loads are used when there is less chance of conflicts at run-time. There exists the possibility of lower latency of the load verify (or check) operation [Kathail et al., 1994]. Since it is expected not to face conflicts, a load verify operation need only check the possibility of conflict by accessing the information in the load execution record hardware, which is faster than a normal load operation.

2.3.3 Operand Value and Dependency Prediction

Most studies on improving data dependency restrictions have focused on false dependencies (WAR and WAW). A recent study by Lipasti and his colleagues revealed that many opportunities exist in programs so that operand values can be predicted [Lipasti et al., 1996, Lipasti and Shen, 1997b]. Also, it was shown that dependence relationships between operations are predictable [Lipasti and Shen, 1997c]. To keep the execution semantics of a program, it is not necessary to detect operand dependencies first and then apply them at the start of execution. Dependencies can be predicted and if a recovery mechanism exists to handle mispredictions the program correctness is assured. In this way, operation execution is decoupled from dependence checking. This leads to a higher number of instructions executed per cycle (IPC) [Lipasti and Shen, 1997a].

2.4 Static Instruction Scheduling

Instruction scheduling is the process of rearranging the sequence of operations so that the execution of the longest sequence can be started as soon as possible, given dependencies and resource conflicts. In this way, the number of concurrent operations for execution can be increased.

Scheduling algorithms are based on the nature of the control flow graph that can be scheduled by them [Rau and Fisher, 1993]. Algorithms that can only schedule a single acyclic basic block are called *local scheduling* algorithms. Algorithms that schedule operations across several basic blocks are known as *global scheduling* algorithms. These algorithms utilise information about the direction of conditional branches, which come from previous execution profiles and heuristics.

Local scheduling causes the execution time of each basic block to be nearly optimum. However, this does not necessarily cause the execution time of the entire program to be nearly optimum, because the processor waits at each branch until all operations before that are in execution. Global scheduling is able to rearrange operations from different basic blocks towards a nearly optimum scheduling for the program.

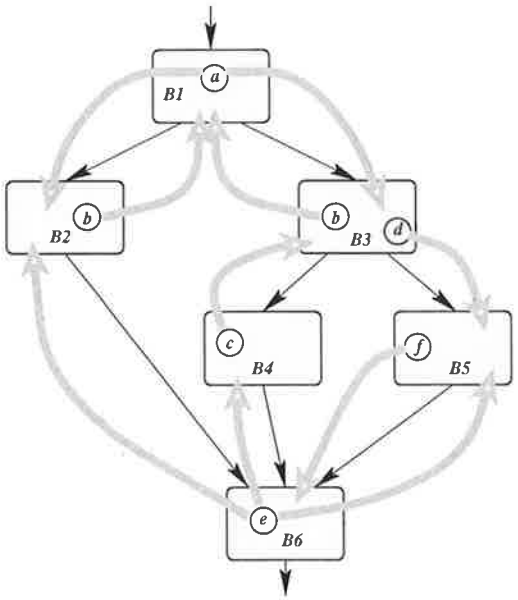
Two types of global scheduling algorithms referred to in the literature [Rau and Fisher, 1993], are reviewed in the following sections.

2.4.1 Acyclic Global Scheduling

Acyclic global scheduling algorithms employ control flow graphs that contain no cycles. Techniques such as loop unrolling can transform a cyclic graph for acyclic scheduling. Loop unrolling is a technique to combine multiple iterations of a loop into a single iteration by duplicating instructions. It is used to increase the number of operations exposed to the scheduler.

Figure 2.8 shows possible opportunities for code motion in acyclic global scheduling. Each box represents a basic block. Extra compensation code may be required in some cases of code motion. For example, in case (d) in Figure 2.8, if the destination of the operation moving from B3 to B5 is live in B4, a copy of this operation will be placed between B3 and B4.

A brief description of the proposed scheduling techniques in the literature, which are based on the general rules shown in Figure 2.8, are presented as follows.



a) Copy operation from B1 to both B2 and B3.

b) Merge identical operations in B2 and B3 into B1.

c) Move operation up from B4 to B3 *speculatively*.

d) Move operation from B3 to B5 if destination is not live in B4.

e) Copy operation from B6 into B2, B4 and B5.

f) Move *predicated* operation down from B5 to B6.

Figure 2.8: Possible code motion opportunities in acyclic global scheduling.

Trace Scheduling

A *trace* is the most likely sequence of operations from an acyclic part of the program control flow graph. Trace scheduling originally was used to pack operations into horizontal microinstructions [Fisher, 1981]. Later, it was employed as the main technique to schedule operations for a Very Long Instruction Word (VLIW) architecture for scientific applications [Fisher, 1983, Ellis, 1986, Colwell et al., 1988].

The key step in trace scheduling is to identify frequently executed sequences and the critical path. Then, this trace is scheduled and optimised regardless of constraints associated with the alternative execution paths, so the result is an efficient schedule for these frequently executed paths at the expense of slower infrequent sequences.

To select a trace, the compiler uses heuristics or profile-based prediction of conditional branches. After scheduling a trace, the next most likely path is selected as another trace for scheduling. This process continues until the entire program is scheduled. To keep the program semantics, special compensation codes are inserted on the off-trace to undo the state changes resulting from mispredictions. Code motion is performed based on the speculative model of the compiler. Figure 2.9 indicates an example of trace scheduling.

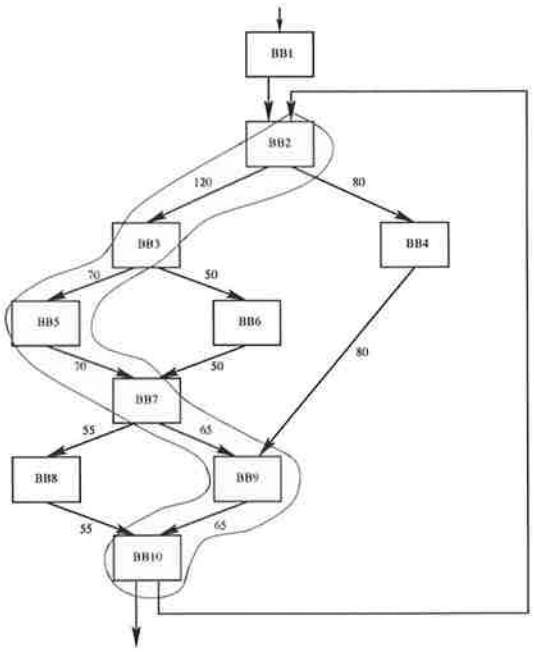


Figure 2.9: An example of trace scheduling. The shaded path is the selected trace, which is optimised and scheduled. (Edges are labeled with the execution frequency.)

Due to the high frequency of conditional branches in general-purpose applications, trace scheduling may be too time consuming and can result in code expansion because of the required compensation codes. Therefore, it is not generally used for general-purpose applications.

Trace scheduling-2 [Fisher, 1993] is an attempt to improve trace scheduling for non-numeric programs. This technique, unlike the original trace scheduling, allows code motion above a conditional branch from both taken and fall-through paths at the same time. Also, calculating the priority of operations for scheduling is based on a criterion called *speculative yield* to reduce the amount of less useful speculative code motion. Figure 2.10 illustrates an example of this technique.

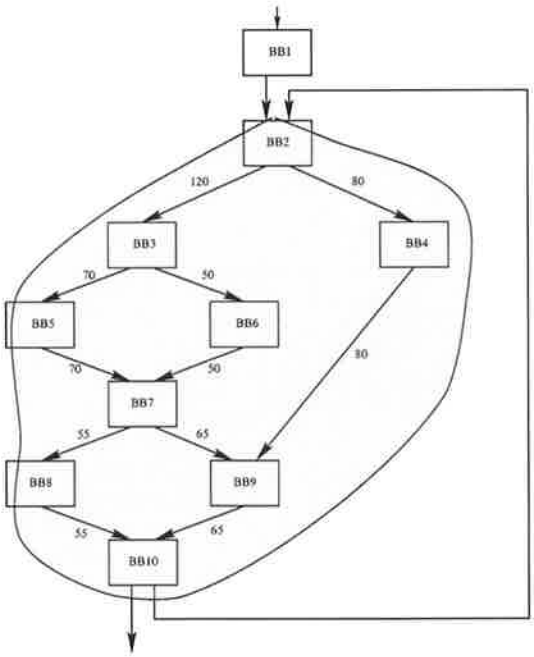


Figure 2.10: An example of trace scheduling-2. The shaded region is optimised and scheduled.

Trace scheduling-2 has the advantage of performing scheduling and code generation at the same phase, like trace scheduling. However, the dynamic nature of data flow analysis and priority calculation for scheduling, which also takes into account the generated compensation code, makes it more complex and difficult for implementation. To the best of our knowledge, no experimental result has been reported about its performance in comparison with the other similar scheduling techniques.

Percolation Scheduling

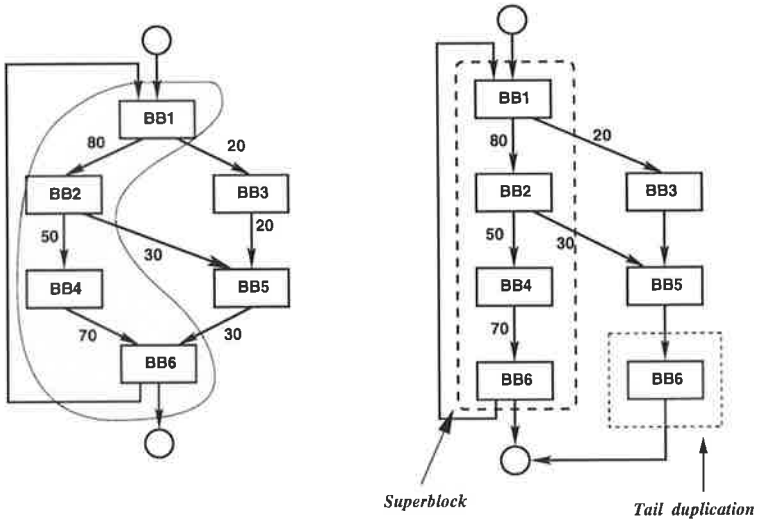
Foster and Riseman proposed a technique called *percolation scheduling* [Foster and Riseman, 1972] to convert a sequence of basic blocks into a larger block through duplication of basic blocks. This results in more code expansion. Nicolau proposed a modified form of percolation scheduling [Nicolau, 1985]. This approach employs a set of primitive code transformations to move an operation up in adjacent nodes towards the starting point in the program. It uses a *parallel program graph* in which each node includes one or more operations that can be executed in parallel, and nodes are connected according to dependencies. As code transformation is done incrementally, moving an operation from a source node X to a target node Y involves checking each intermediate node on the path. This increases the compilation time. *Trailblazing* is an extension to this technique to reduce the amount of compensation code and compilation time through introducing some new code transformations [Nicolau and Novack, 1993]. Percolation scheduling is used in some VLIW compilers like the UCI VLIW compilers [Nicolau and Novack, 1993].

Superblock Scheduling

Superblock scheduling is an extension of trace scheduling which was proposed by the IMPACT compiler group [Hwu et al., 1993]. Similar to trace scheduling, the most frequently executed path with no cycle is selected for optimisation and scheduling. A superblock is an extended block with single entry, and possible multiple exits. Due to the requirement for more compensation code and complexities in optimisation for side entrances, *tail duplication* is performed to remove the side entrances. Published works on implementing trace scheduling [Ellis, 1986, Colwell et al., 1988], described scheduling, code generation and even register allocation to be done in a single phase. In superblock scheduling each of these are considered as a separate phase. Speculative code motion is the main source of more ILP in superblocks. Most local classical optimisations can be simply applied on superblocks in addition to specific ILP optimisations [Hwu et al., 1993]. Figure 2.11 shows an example of superblock formation. Weighted edges represent execution frequency.

Hyperblock Scheduling

Superblocks include only one possible execution path in a region. When the machine parallelism capabilities allow execution of more operations, multiple execution paths should



(a) Original flow graph (b) Superblock formation

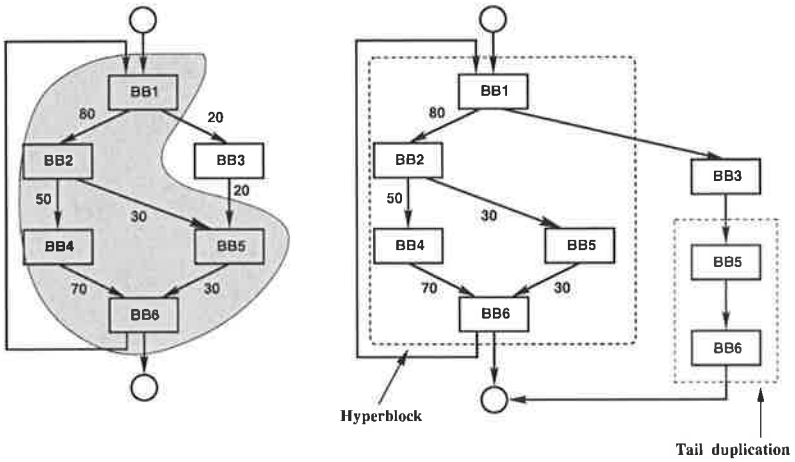
Figure 2.11: An example of superblock formation.

be considered to form the block of code for optimisation and scheduling. For this purpose, conditional branches should be removed through if-conversion.

Conventional if-conversion techniques suffer from two problems [Mahlke et al., 1992b]. They do not consider the execution path frequency in the region of interest. This may limit the performance of the resulting predicated code due to the use of limited machine resources for infrequent execution paths. Also, speculative execution as an important source of ILP is not supported effectively. A predicated operation is speculatively executed when its predicate is not calculated. To overcome these problems, a structure called a *hyperblock* was proposed by the IMPACT compiler group [Mahlke et al., 1992b, Mahlke, 1996].

A hyperblock is a group of basic blocks, which are selected from the most frequent execution paths and then if-converted to form a single unit for optimisation and scheduling. It only has one entry point but one or more possible exit points. To eliminate other entry points to the basic blocks of the hyperblock, *tail duplication* is performed. Figure 2.12 shows an example of a hyperblock.

To select the basic blocks for hyperblock formation, the execution frequency, the size, and the number of hazardous operations in the basic blocks are considered. Hazardous operations like unresolvable memory accesses and function calls are avoided to improve performance [Mahlke et al., 1992b]. Smaller basic blocks with no (or fewer) hazardous operations which are executed more frequently are given higher priority for inclusion in the hyperblock.



(a) Original flow graph (b) Hyperblock formation

Figure 2.12: An example of hyperblock formation.

Speculative execution is supported in hyperblocks through predicate promotion [Mahlke et al., 1992b]. Predicate promotion changes the predicate of an operation to another predicate (called the ancestor), which was used to compute the current predicate. Three types of algorithm have been presented for predicate promotion [Mahlke, 1996].

2.4.2 Cyclic Global Scheduling

Loops are considered as a good source for ILP. *Cyclic* global scheduling is used directly to schedule and optimise a cyclic graph due to loops. It is generally referred to as *software pipelining* in the literature.

Software pipelining is a technique to overlap or pipeline different iterations of a loop [Charlesworth, 1981, Rau and Glaeser, 1981, Lam, 1988]. The result of software pipelining is a new loop whose body contains operations from different iterations of the original loop. This means that, after initiating an iteration of the loop, the next iteration is initiated as soon as possible. The number of cycles between the initiation of two successive iterations is called the *initiation interval (II)*. A smaller initiation interval means higher execution throughput. The II is usually less than the time that it takes to execute a single iteration. The minimum initiation interval (MII), is the maximum of the lower bounds due to the cyclic data dependent constraints caused by recurrences (Rec MII) and the lower bound due to the resource usage constraints (Res MII) [Rau and Fisher, 1993]. The resource which is the most heavily used by the loop body determines the lower bound of the Res MII. Hence, the operation latencies and

the number of registers in the processor affect the MII. The II is dependent on the software pipelining algorithm and can be a single fixed value, a periodic sequence of values, or a sequence of fixed values which depend on the control flow within the loop body [Lavery, 1997]. Figure 2.13 shows a simple example of software pipelining.

Software pipelining can be applied to loops with inter-iteration or loop carried dependencies and with arbitrary control flow including loops with an unknown number of iterations, like *while* loops. Early works on software pipelining addressed only loops with a single block. Rau and his colleagues proposed a general formulation of the software pipelining process for a single basic block loop. Their approach called, *modulo scheduling*, [Rau and Glaeser, 1981] was also used as a basis for other software pipelining algorithms [Jones, 1991, Rau et al., 1992, Huff, 1993]. The Cydrome's Cydra VLIW compiler is based on modulo scheduling with hardware support [Rau et al., 1989, Dehnert and Towle, 1993].

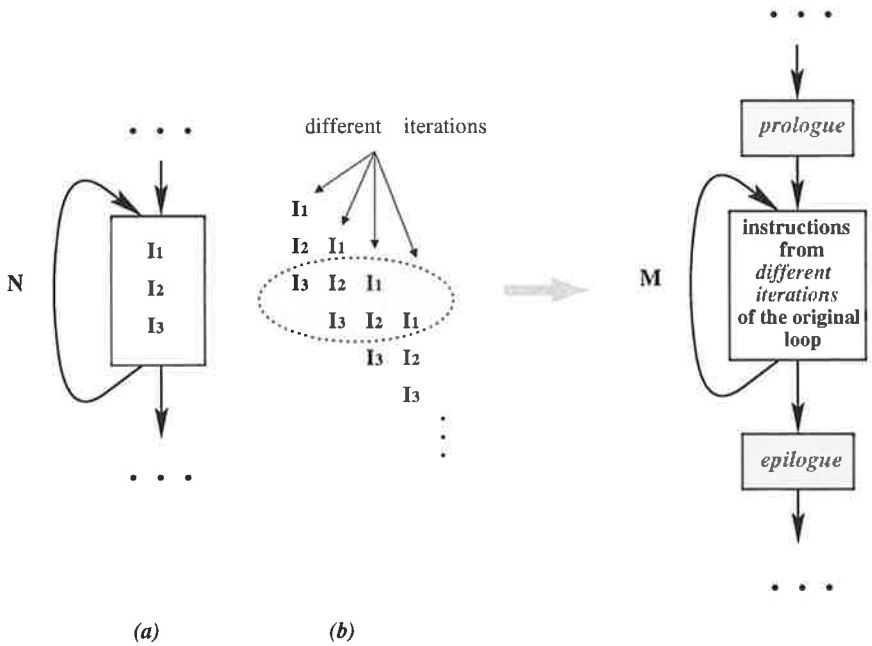


Figure 2.13: Software pipelining: (a) The original loop with N iterations. (b) A new loop with M iterations.

The modulo scheduling algorithm is based on the local scheduling concept and uses list scheduling. It requires that all iterations have a common schedule. It was shown to achieve the MII, so it can be an asymptotically optimal schedule. This algorithm was restricted to

for loops which contain a single basic block and the resource usage of the operations can be viewed as a single resource for a single cycle [Rau and Fisher, 1993]. Generating an optimal resource constrained schedule for loops with arbitrary recurrences is NP-complete [Hsu, 1986, Lam, 1987], but a near optimal scheduling can be performed by using heuristics [Rau and Fisher, 1993].

Later works extended the original modulo scheduling to include conditional structures in the loop body [Lam, 1987, Dehnert et al., 1989, Warter et al., 1992, Lavery and Hwu, 1996], and more accurate Res MII calculation for a complicated resource usage [Rau, 1994].

In Lam's algorithm, called *hierarchical reduction* [Lam, 1987], two paths of a conditional branch are first scheduled independently. NOP operations may be added to one path to make both have the same length. Both paths are considered as a single node for scheduling. This causes an overestimation of resource usage. Also, adding NOPs may decrease the execution speed as well as expanding the code.

In the Cydra 5 compiler, predicated execution was employed to reduce multi paths in the loop to a single one [Dehnert et al., 1989]. Then, the scheduling method for a single block was used.

Another approach for including conditional branches in modulo scheduling employs *if-conversion* before modulo scheduling and *reverse if-conversion* after that [Warter et al., 1992, Warter et al., 1993]. This approach, referred to as *enhanced modulo scheduling* (EMS), was shown to have better performance than the hierarchical reduction approach, due to complicated resource usage patterns in the latter [Warter et al., 1992]. Since the control flow graph is regenerated after scheduling through reverse if-conversion, EMS can be used for processors that do not have hardware support for predicated execution.

Lavery and Hwu proposed a technique for control intensive loops with multiple exits [Lavery and Hwu, 1996]. It only considers the most frequently executed path in the loop body for software pipelining. Superblocks [Hwu et al., 1993] are used to exclude unimportant paths and speculative execution increases parallelism both in each iteration and between successive iterations.

Some approaches were also proposed based on a global scheduling techniques such as percolation scheduling [Nicolau, 1985] or region scheduling [Allan et al., 1992] to make the software pipelined loop.

In *Perfect Pipelining* [Aiken and Nicolau, 1987], the first step is to apply global code

motion in the loop body to schedule operations as early as possible without considering resource limitations. The second step is unrolling the scheduled loop. This is performed as an iterative process until a repeating pattern is found.

Identifying the repeating pattern is a complex task. To avoid it, *Enhanced Pipeline Scheduling* was proposed [Ebcioglu and Nakatani, 1989]. In this technique, which was developed for a VLIW architecture with a tree instruction for multi-way branching, a VLIW instruction is created and placed at the beginning of the loop as a barrier, while the original loop remains intact. Operations in the loop body are moved up to fill this instruction, considering resource conflicts. Once this instruction is filled, it is moved across the loop back-edge and copied into the prologue. In this way, those operations enter the loop body as operations from the next iteration. This process continues until all operations from the original iteration are scheduled.

An approach called *GURPR** [Su and Wang, 1991] avoids the requirement for pattern matching as in perfect pipelining. In this approach, first the loop body is compacted using a global compaction algorithm. Then, it is unrolled and scheduled. Loop unrolling is performed until the last instruction of the first iteration is scheduled. After finding the II, the software pipeline schedule is generated by overlapping iterations, considering loop carried dependencies and resource constraints. Redundant instructions are removed and a new loop with prologue and epilogue is constructed. Theoretically, perfect pipelining can expose more parallelism but, GURPR* may result in less code expansion than perfect pipelining as the pattern is often too long in the latter [Su and Wang, 1991].

2.5 Hardware Techniques

Popular ILP processors referred to as *superscalar* [Johnson, 1991] perform ILP extraction dynamically at run-time. To extract a large amount of ILP at run-time, a relatively large number of operations for parallel execution must be investigated. This is done by fetching multiple operations per cycle, dynamic register renaming and dynamic speculative execution. Dynamic branch prediction techniques [McFarling, 1993, Yeh, 1993], play a key role for this purpose. Depending upon the branch predictor, the execution of speculatively fetched operations can be overlapped if the required resources are available.

For dynamic register renaming, there are more physical registers available than the regis-

ter names (or architectural registers). Register names are mapped to physical registers using a map table. In the decode stage, the source registers are renamed, and the output register is mapped to a new free physical register, then the map table is updated. The usage and efficiency of each method is related to the way dynamic speculation and precise interrupts are handled.

Out of order execution is performed through dynamic scheduling. When an operation is to be issued, its register operands can be either the actual value, or just a symbolic (tag) value. The tag is used for register renaming. The source of a tag might be a centralized or distributed pool depending on the implementation. In Tomasulo's algorithm [Tomasulo, 1967], a number of reservation stations are dedicated to each functional unit. A tag is assigned to the destination register of each operation during the decode cycle or fetch cycle [Moudgill et al., 1993]. After issuing, the operation with its operands is buffered in the reservation station of the specified functional unit. These buffered operations are scanned in each cycle to find an operation ready to execute, which is the operation that has all of its source operands available. Upon completion, the result and destination registers' tags are broadcast to all reservation stations and the register file by a common bus.

The future file mechanism is an extension of Tomasulo's algorithm to implement precise interrupts [Smith and Pleszkun, 1985, Johnson, 1991]. The register file keeps the in-order state, the future file has architectural state. A reorder buffer, which is a FIFO queue, is used to keep the lookahead state. A slot at the top of this queue is allocated to each operation during decoding. Meanwhile, register identifiers are applied to both the future file and the register file to get the source operand values (or tag). When the most recent value of a register is not found in the future file, the value in the register file will be used. After completion, the result is written into the allocated slot in the reorder buffer and the future file. This method is faster and has less hardware complexity in comparison to the associative lookup in the original Tomasulo's algorithm.

Centralised mechanisms like the scoreboard technique, were used earlier than distributed mechanisms [Thorton, 1964]. In this method, the instruction window is used to hold operations that are waiting for operands. The scoreboard as a centralised control structure keeps track of the source and destination registers of each operation in the window, and their dependencies. The Metaflow architecture [Popescu et al., 1991] uses this technique.

The Metaflow architecture employs an approach referred to as DRIS (Deferred-scheduling,

Register renaming Instruction Shelf) for out of order execution and dynamic speculation. In this method, after register renaming, operations are placed in a central structure. To dispatch the ready operations, this central structure is searched every cycle. The DRIS, like the reorder buffer, is a queue. A slot at the bottom of the DRIS is allocated to each issued operation. After completion, the result is written to that slot. When the completed operation reaches the top of the DRIS, the register file is updated and the allocated slot is removed. During the decode cycle (i.e. when the slot is allocated), the DRIS is scanned to find the source operands of the operation which match the destination register of the present operations. If some match is found, the newest one is used. Otherwise, the register file is accessed for source operands. The actual value is read from the DRIS or the register file, when the dependent operation is completed.

Run-time techniques are more successful at resolving memory access dependencies as the address of memory references are known at run time. As discussed in section 2.3.2, a compiler, with the help of a specialised hardware, can speculate on memory access aliases and value prediction leading to a higher performance.

Providing a larger scope for a run-time technique requires a large amount of hardware. This may reduce the execution speed. Regardless of using extra hardware support for ILP extraction, compiler techniques can still be very useful for ILP processors to place independent operations closer together to be picked up at execution time.

2.6 Summary

Code transformations are applied at compile time to extract more ILP in programs. To avoid ILP performance limitations due to dependencies in critical paths, some transformations are applied to reduce the length of these paths. Critical paths are identified with regard to dependency constraints which are considered based on the scheduling model. Data dependencies can be reduced through register renaming, memory address disambiguation and data value prediction. Control dependencies are eliminated or reduced through speculative and predicated execution.

Speculative execution relies on the accuracy of branch prediction techniques. Compilers utilise static branch prediction methods and previous execution profiles (if available) to schedule operations speculatively. On the other hand, dynamic techniques employ dynamic

branch predictors which predict the control flow based on a recent history of branch directions.

Predicated execution is an architecturally supported technique, which is used to reduce the number of conditional branches using a technique referred to as if-conversion. Hyperblock scheduling demonstrates that higher performance can be achieved through selective if-conversion. Hardware mechanisms to nullify operations with false predicates are needed to support predicated execution.

Hardware techniques in general may have an adverse effect on performance due to a potentially longer cycle time. In some cases, such as memory access disambiguation, appealing to a run-time technique seems to be inevitable. However, compiler techniques can still be very effective in improving the performance regardless of whether simple hardware (such as VLIW machine) or a more complex hardware (an out of order superscalar processor) is employed as the ILP processor.

Chapter 3

EVA: An Experimental VLIW Architecture

A Very Long Instruction Word (VLIW) processor exploits ILP that has been extracted through compiler transformations. The advantage of these processors is their ability to exploit large amounts of ILP with relatively simple and fast control hardware. This is in comparison with the popular superscalar processors, in which ILP extraction is performed dynamically at run-time. Although compilers can enhance the performance of superscalar processors by placing the independent operations closer together to be picked up by the dynamic scheduler, they are not generally relied on for this purpose.

In this chapter, the main features and shortcomings of VLIW processors are briefly discussed. Works to remove or reduce the impact of these shortcomings are reviewed. Our experimental VLIW architecture (EVA) is also presented.

3.1 VLIW Architectural Features

In VLIW processors, architecturally visible parallelism capability is exposed to the compiler. This includes an accurate semantic model of the processor, which defines the operation latencies, and a description of the allowed sets of independent operations in a VLIW instruction. The two main features of VLIW architectures are multiple-operation instruction (MultiOp) [Rau et al., 1989] and non-unit assumed operation latency (NUAL) [Rau, 1993].

3.1.1 Multiple-Operation Instruction (MultiOp)

To extract independent operations for parallel issue and execution, dependency checking must be performed on the code. Dependency checking considers the constraints resulting from the current operations in execution (if it is done at run-time) in addition to, destination and source operand references of all candidate operations for issue. If this is to be performed by hardware, a complex circuitry is needed, with a complexity proportional to N^2 for an issue-N processor [Johnson, 1991]. Also, the size of the set of candidate operations is a factor that affects hardware complexity. Therefore, transferring the dependency checking task to the compiler significantly reduces the amount of hardware, which is often on the critical path determining the cycle period.

Operations to be packed in a MultiOp instruction, are determined by the compiler. They are aligned based on the architectural model of the target processor. In this way, the hardware required for instruction alignment is removed. Also the operations are transferred from the instruction buffer to the corresponding decode slot and functional unit in a straightforward manner with no extra hardware.

3.1.2 Non-Unit Assumed Latency (NUAL)

Operation latencies are architecturally visible in VLIW processors. This execution semantic is called non-unit assumed latencies (NUAL) [Rau, 1993].

In conventional RISC and superscalar processors, which are usually based on unit-assumed latencies (UAL) execution semantics, it is assumed that previous operations in the sequence are committed at the time of source operand access for the current operation. If this is not the case (such as an out-of-order superscalar processor), or when the actual latency is greater than specified, additional hardware mechanisms such as register interlocks are provided to preserve the program correctness. Figure 3.1 shows a piece of code to illustrate differences in interpretation of program semantics between NUAL and UAL. In Figure 3.1 (a), operations 3 and 4 use the results of operations 1 and 2 respectively. In Figure 3.1 (b), due to UAL semantics, both operations 3 and 4 use the result of operation 2 and the destination register of operation 1 is overwritten by operation 2. In this manner, the semantics of the same code are different in Figures 3.1 (a) and (b).

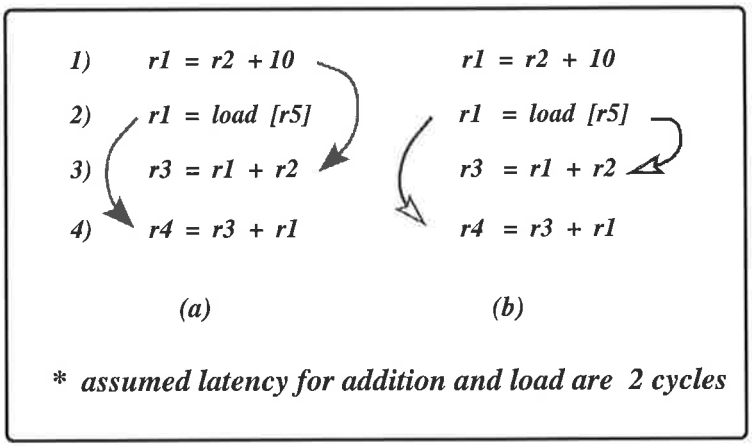


Figure 3.1: Comparison of the execution semantics for (a) NUAL and (b) UAL operations.

NUAL provides more efficiency in code optimisation and generation for the compiler. As the details of the target processor are exposed to the compiler, the resource usage of the target processor can be more efficiently scheduled by the compiler. However, this is at the expense of introducing some problems when the control flow at run-time is not the same as assumed at compile time. Unexpected events like exceptions dynamically change the assumed program behaviour due to control transfer to the exception handling routine, which causes the assumed latency of the scheduled operations to be non-deterministic. Hence, exact NUAL semantics are violated.

To handle this problem, two scheduling models of NUAL referred to as EQ (equals) model and LEQ (less than or equals) model have been defined [Rau, 1993]. An EQ NUAL operation accesses its operands at the specified time and writes back the result exactly at its latency time. In the LEQ model, if the operation latency is L, it is assumed the result is available between one and L cycles after launching the operation. Therefore, unexpected run-time events are handled more easily with LEQ NUAL semantics, while with EQ NUAL, hardware support is required to save the status of the processor before execution of exception handling code.

EQ NUAL is more capable of extracting parallelism due to its deterministic nature. However, its benefits may not justify using more hardware for exception handling.

3.2 Traditional VLIW Shortcomings

3.2.1 Inefficient Memory Usage

Due to statically scheduled code, unused slots appear in a VLIW instruction. A VLIW instruction can be encoded in uncompressed or compressed form, of which the former includes explicit NOP operations for empty slots. Uncompressed encoding, while it involves simpler hardware, results in inefficient use of memory. A study by Conte and et al. [Conte et al., 1996a] indicates a significant loss of performance for an uncompressed I-cache with respect to an I-cache that never misses (perfect cache) due to low space utilisation.

Compressed encoding schemes were proposed for some VLIW machines and CISC architectures. In the Multiflow TRACE VLIW machine, a VLIW instruction is stored in compressed form in memory, and extra mask words are used to identify which field is present in the instruction [Colwell et al., 1988]. During instruction cache refill, each operation is placed in its special field in the cache and NOPs are inserted for non-present fields. The Cydrome Cydra 5 VLIW machine used two different formats for VLIW instructions stored in memory (or cache) which are referred to as MultiOp and UniOp respectively [Rau et al., 1989]. When enough parallelism is not available to form a MultiOp instruction, six UniOps are packed in an instruction word. MultiOps are stored in uncompressed form and NOPs are also used for memory alignment in the case of UniOps. TINKER, a research VLIW machine [Conte and et al., 1995, Conte et al., 1996a] uses compressed encoding of MultiOps. Each operation has four extra bits which are used to identify each field of a MultiOp. These are one header bit, one tail bit to show the first and the last operation respectively, and two bits to indicate the functional unit type. In the IBM VLIW architecture, the program is arranged as a sequence of tree instructions which consist of a set of operations and multi-way branches [Moreno, 1996]. Tree instructions are pruned at run time based on the implementation of the architecture and there are no explicit NOP operations.

Compressed encoding requires hardware support for instruction fetch and expansion. Different instruction fetch mechanisms and cache structures were investigated by Conte and et al [Conte et al., 1996a]. Generally, the basic pipelined fetch model consists of block fetch, next PC computation, and an expander to uncompress a MultiOp. The expander can be placed on the cache refill path or between the cache and the execution pipeline depending on the cache organisation. The branch misprediction penalty may be increased in the latter

case.

Among different cache organisations studied for the compressed encoded VLIW instructions, *silo caches* achieved higher performance [Conte et al., 1996a]. The silo cache is made up of several separate caches, which are accessed at the same time to get operations of a MultiOp. Each cache stores one specific operation type. The expander is placed on the cache refill path. To reduce the amount of the required storage, a group of operations which are rarely executed at the same time can be mixed into one cache. This is called a *flexible silo cache* [Conte et al., 1996a].

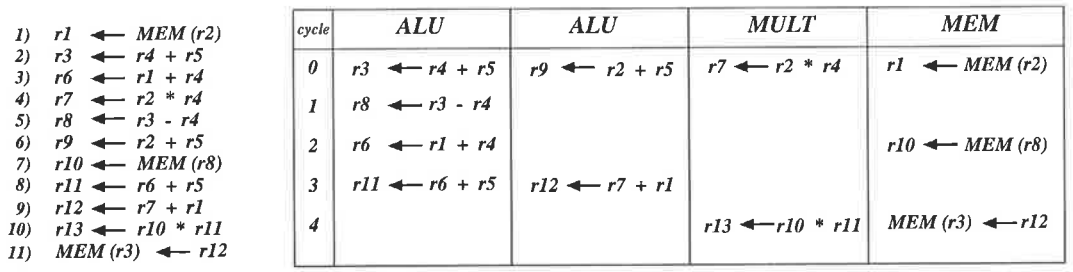
A silo cache needs more storage, as NOPs are explicitly present in the cache and each separate cache has its own tag and length fields. A cache organisation called *banked cache* requires less storage than a silo cache at the expense of lower performance. The banked cache consists of two data and tag arrays (for example, in the Intel Pentium processor [Alpert and Avnon, 1993]) with the block size N for an issue-N processor. Since a MultiOp is compressed, some part of it may be stored in a different bank. Thus, it is necessary for both banks to be accessed at the same time. The header bit in the operation identifies the first operation of the fetched MultiOp. The fetched blocks may be swapped if required. This is done by finding the last operation in the MultiOp through its tail bit. The expander is located on the cache hit path resulting in a higher branch misprediction penalty. Results of Conte's work show less than 10% performance loss with respect to a direct-mapped silo cache of the same size [Conte et al., 1996a].

Valid select lines are used to direct the required Ops to the expander. The expander is located on the hit path, so the misprediction penalty is higher. Addressing in the banked cache is similar to a traditional cache. Each Op in the cache is augmented by 3 offset bits, 1 valid and 1 bank bit which are set at the time of cache refill and are used to compute the next PC. Offset bits indicate where the next sequential MultiOp is with respect to the current one. The next PC is computed at the same time of the cache access [Banerjia et al., 1996].

3.2.2 Object-Code Incompatibility

In VLIW machines operations are scheduled statically by considering architectural features. This may produce object code incompatibility with other implementations of the same architecture when the assumed architectural features such as operation latencies, the number of functional units and register file specifications are changed. Figure 3.2 shows an example of

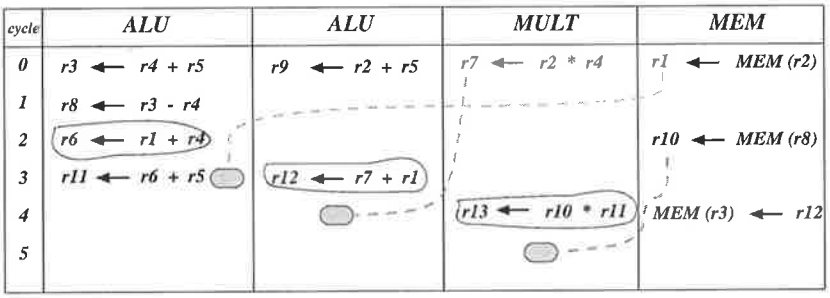
a sequence of operations scheduled for a hypothetical issue-4 VLIW machine. In Figure 3.3, execution of the same code, when the latencies for MULT and MEM functional units have increased, leads to incorrect results. A decrease in latencies does not violate the program semantics for the LEQ model but, this is not the case for the EQ model. In the case of changes in the number of functional units, a narrower issue processor may execute a section of the VLIW instruction from the original VLIW code but, keeping correctness and respecting dependencies is not guaranteed. In a wider issue processor, the original code may be executed correctly but the additional new resources are not utilised.



(a) Original sequence of operations. (b) Scheduled code for VLIW machine (Generation 1).

Latencies assumed for ALU, MULT and MEM functional units are 1, 3 and 2 cycles respectively.

Figure 3.2: Scheduled code example for an issue-4 hypothetical VLIW machine (generation-1).



VLIW machine (assumed Generation 2) - Shaded operations are not executed correctly due to differences between actual and assumed operation latencies.

(New latencies are 1, 4 and 3 for ALU, MULT and MEM functional units respectively).

indicates when actual result is ready.

Figure 3.3: Incorrect interpretation of the code scheduled for VLIW (generation-1) in a new VLIW (generation-2) with different operation latencies.

In chapter 7, research performed to overcome this problem is discussed. Also, a new approach to address the problem is presented.

3.3 Architecture of EVA

To support research in compiler techniques and architectural features for VLIW machines, we use an experimental VLIW architecture (EVA). Some parameters to define architectural characteristics can be set depending on the experiment.

At the time of this research, no suitable experimental or commercial compiler was available to us for our research purposes. Therefore, we designed the EVA architecture, and an accompanying compiler, as the basis for our research. This architecture is based on MIPS-I, which is a classic RISC architecture. Using the MIPS-I architecture provides the opportunity to generate the program execution traces on a MIPS based platform (as discussed in chapter 5). In addition, this allows us to use different C standard libraries for our benchmark programs directly from the MIPS based platform. As the MIPS-I architecture does not have many required features employed in this research, some appropriate features from HPL Play-Doh [Kathail et al., 1994] and the TINKER [Conte and et al., 1995] architectures have been adopted. These features are all discussed in Chapter 4. In all other respects the architecture is identical to MIPS-I.

3.3.1 Instruction Set Architecture

Each operation in the VLIW can be any RISC type instruction that is executed by a functional unit. The number of operations in a VLIW is specified as an architectural parameter. The instruction set architecture (ISA) of EVA is based on the MIPS-I ISA. Each operation is augmented with two extra bits to prevent the need for explicit NOPs in the VLIW instruction. Similarly to the TINKER VLIW architecture [Conte and et al., 1995], one header bit and one tail bit are used, which are set for the first and the last operation in the VLIW instruction respectively. To specify a VLIW instruction with all NOPs, both header and tail bits are set.

3.3.2 Architectural Support for Speculative Execution

For control speculative execution we use the general code motion scheme [Chang et al., 1995]. For this purpose, non-trapping (or silent) versions of excepting operations are added

to the ISA. These include floating-point arithmetic, memory loads and integer divide operations. Operations which may potentially cause an exception are converted into their silent form if they are scheduled for speculative execution. Having silent versions of trapping operations makes it possible to ignore unwanted exceptions.

3.3.3 Architectural Support for Predicated Execution

To support predicated execution each operation has an extra source operand as the predicate. If this input predicate is 'true' the operation is committed, otherwise it is nullified. Also, new compare operations, which are adapted from HPL PlayDoh [Kathail et al., 1994] and the IMPACT architecture [Mahlke, 1996], are used to define predicates in the if-conversion process. Each eliminated branch is replaced by a corresponding predicate define operation. The predicate define operation computes predicate values using semantics similar to those for conventional comparison operations. Its format is as follows:

p<*cmp*> Dest1, Dest2, Src1, Src2, Src3

Src1 and Src2 are the original source operands of the eliminated branch. Src3 is the input predicate, which is normally set to 'true' if this predicate define operation is not itself predicated. Dest1 and Dest2 are the two destination predicate registers. Having two destination registers in one operation makes it possible to combine two predicate definitions into one operation when the source operands for the two predicate definitions are the same. This may happen frequently as defining a predicate and its complement are usually needed. Destination registers are set based on the comparison result and two mode bits which are encoded in Dest1 and Dest2. The first bit indicates if the predicate definition is unconditional or OR-type. The second bit defines normal or complement action.

When the destination type is 'unconditional normal', if the input predicate (Src3) is 'true' and the comparison result is 'true', the predicate register is set to 'true'. Otherwise, a 0 is written to the predicate register. Multiple conditions in programs (like OR, AND) require OR-type predicate definition which can handle all situations for this purpose. If both the comparison result and the input predicate are 'true', a 1 is written to the predicate register (for OR-type normal) otherwise, it is left unchanged. When destination predicates are seen at the first time in an OR-type predicate define operation, they must be initialised before p<*cmp*>

input predicate (Src3)	compare result type	unconditional normal	unconditional complemented	OR-type normal	OR-type complemented
0	0	0	0	-	-
0	1	0	0	-	-
1	0	0	1	-	1
1	1	1	0	1	-

Table 3.1: Predicate comparison behaviour for predicate definition. '-' indicates the previous value is not changed.

operation. This is performed through a special operation which resets the predicate register unconditionally. Table 3.1 briefly summarises the behaviour of predicate define operations.

3.3.4 Architectural Support for Memory Access Disambiguation

Load and store operations may have special modifiers to indicate the latency and control the way the memory hierarchy is accessed for data. [Abraham and et al, 1993]. For example, in the PlayDoh architecture modifiers can specify the first and second level of cache or main memory for data transfer [Kathail et al., 1994]. In our VLIW architecture, the assumed memory latency by the compiler refers to the first level cache. However, a data speculation technique similar to that of the PlayDoh architecture is provided to reduce the latency of the critical paths in the program.

As described in section 2.3.2, compile-time memory disambiguation is a difficult problem and often it cannot produce the complete result. This results in a conservative scheduling of the ambiguous memory access operation. Consequently, the critical path length is increased as load operations are usually located on the critical paths. As an architectural support for run-time memory access disambiguation, two different forms of loads are included in our VLIW architecture. The semantics are the same as for the PlayDoh architecture. Table 3.2 describes the general description of these operations.

3.4 Implementation of EVA Architecture: An Example

Although the aim of this thesis is not to investigate details of implementation issues of ILP processors, an example of the implementation of EVA is provided to illustrate the relation-

Operation Description	Opcode example
pre_load or data speculative load performs normal load operation even if it is not clear whether there is no alias with previous stores.	*s_lw*
check_load or data verify load checks the memory conflict hardware and performs the load if it finds a conflict with previous stores.	*v_lw*
check_br checks the memory conflict hardware and branches to the recovery code if it finds a conflict with previous stores.	*check_br*

Table 3.2: General description of operations to support run-time memory access disambiguation.

ship between different parts of the processor.

The format of the instruction (MultiOp) is shown in Figure 3.4. Operations are prioritised from left to right for the same functional units. In addition to BRU, all IALU functional units can also execute branches. However, the *check_br* operation can only be scheduled in the BRU slot. Only one taken branch can be present in the instruction. It is the responsibility of the compiler to ensure that multiple scheduled branches have mutually independent conditions. If it is not possible for the compiler to guarantee that, only one branch is scheduled in a MultiOp (in the last slot). As the architecture supports predicated execution, few conditional branches are expected after code generation.

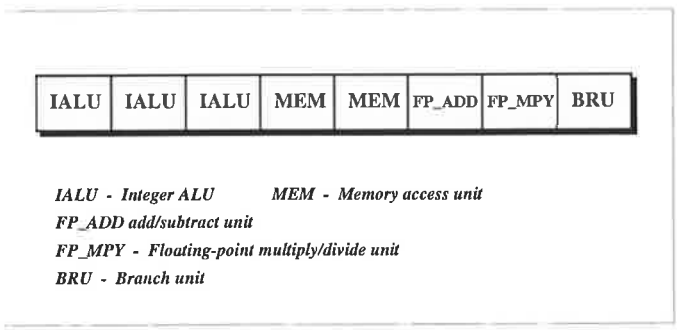


Figure 3.4: Instruction (MultiOp) format for the example implementation.

Figure 3.5 shows the general organisation of an implementation of EVA. The banked cache scheme with two banks is assumed for the instruction cache and the required logic for decompression and alignment are located after the instruction cache. Each operation is forwarded to its specified decode slot.

3.5 Summary

VLIW machines as ILP processors rely on a compiler to extract ILP. This results in simpler hardware and a shorter cycle time. However, the traditional forms of these machines suffer from some shortcomings. These include inefficient memory usage and lack of binary compatibility among different generations of the same architecture. Some techniques to improve memory usage presented in the literature have been reviewed. Chapter 7 addresses the previous work and our new approach to improve object code compatibility.

An experimental VLIW architecture (called EVA) was presented. It has several extensions to common RISC architectures. These include ISA extensions to support speculative and predicated execution. Also, special hardware for dynamic memory access disambiguation is provided. This basic VLIW architecture is used as a base in this research.

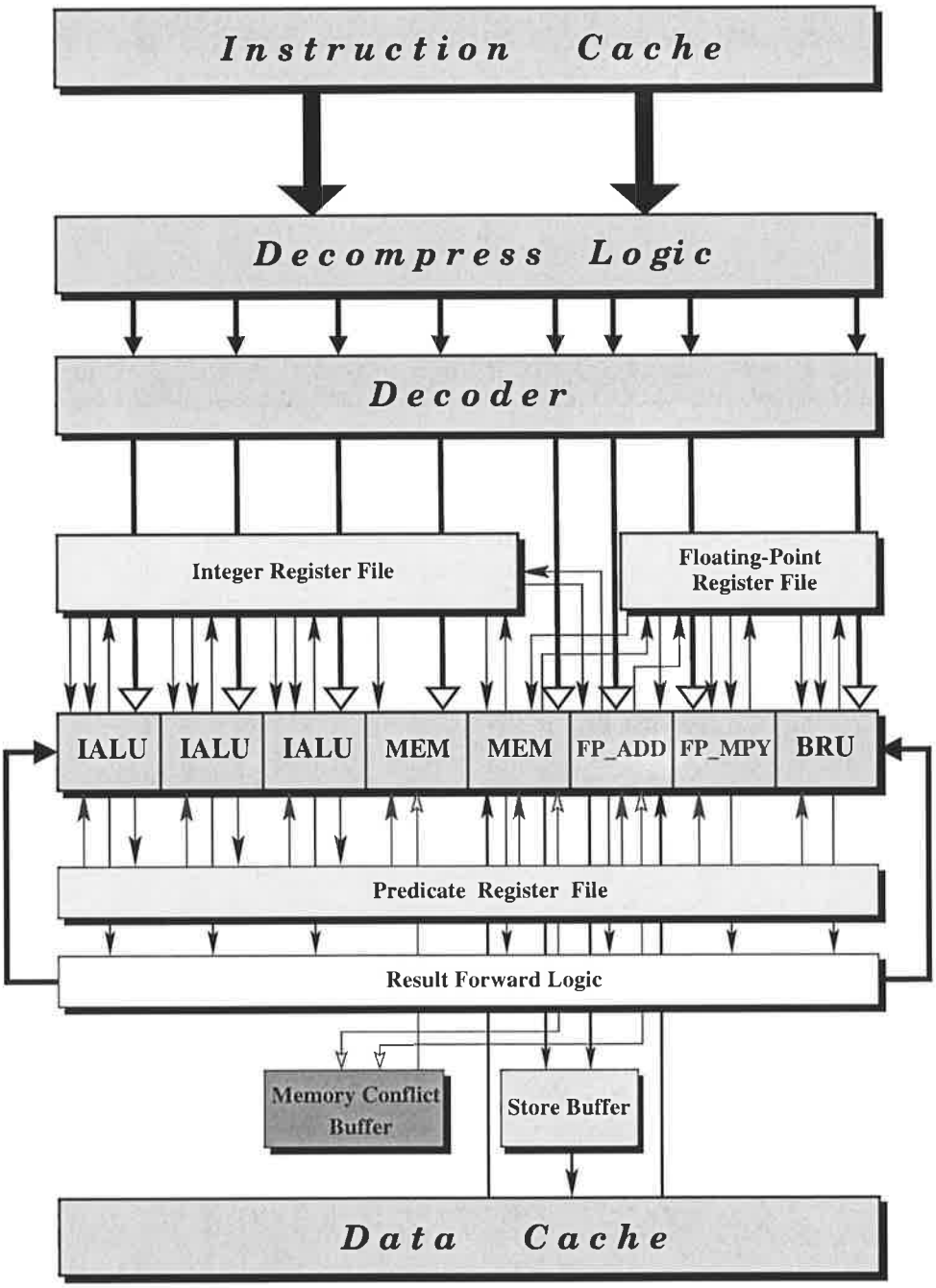


Figure 3.5: General organisation of the example implementation.

Chapter 4

The VLIW Compiler for the EVA

Efficient employment of processor resources requires that enough ILP to be extracted to keep the resources busy. For a VLIW processor, where the complexity is transfered from hardware to the compiler, a powerful and robust compiler is necessary. All of the instruction scheduling in a traditional VLIW machine is performed by the compiler. Due to a larger instruction window available for inspection at compile time, more opportunities for extracting independent operations exist than in a superscalar processor. To investigate performance improvement trade-offs in ILP architectures in general, and VLIW architectures in particular, a powerful experimental compiler is required.

In this chapter we describe our implementation of such a compiler. This compiler employs state-of-the-art algorithms that have been published in recent literature, as well as new back-end algorithms that we have developed. The structure of the compiler is presented, including descriptions of all of the algorithms employed. The new algorithms are described in detail.

4.1 Compiler Structure

4.1.1 Motivation

Implementing an experimental optimising compiler requires a large amount of effort and time. We designed and implemented our compiler for the EVA based on the **SUIF** infrastructure, which includes *SUIF* [SUIF, 1994] and machine SUIF (*machsuif*) [Smith, 1997] from Stanford and Harvard universities respectively. The **SUIF** infrastructure has been used

in other research in ILP [Young, 1998, Gloy, 1998]. Our work is based on **SUIF-1**, the first version of the **SUIF**.

As described in chapter 2, there is generally not enough parallelism within individual basic blocks to achieve significant performance improvements through ILP, and higher levels of ILP may only be obtained through investigation of successive basic blocks. The method of combining several basic blocks into a larger structure depends upon the instruction scheduling technique employed and the capabilities of the target machine architecture . To select the instruction scheduling technique, we realised that employing an acyclic scheduling method is reasonable within the framework of our research, as cyclic scheduling techniques target only innermost loops, while scheduling of code outside the loops needs an acyclic scheduler.

For larger block structures, if the target architecture cannot support predicated execution, the use of superblock scheduling [Hwu et al., 1993] is more efficient than the other techniques for acyclic scheduling mentioned in chapter 2. Employing superblocks uncovers ILP through speculative execution, involving less effort for bookkeeping during upward speculative code motion. However, frequent and unpredictable branches are still present in non-numerical programs. Predicated execution is used to eliminate the conditional branches, resulting in a larger block of linear code. This can be achieved through using the hyperblock structure described in chapter 2, which implies selective conditional branch removal.

Results of work done by the IMPACT compiler group [Mahlke, 1996] indicate that on average, the size of the block presented for optimisation and scheduling increased from 34.5 operations in a superblock to 53.0 in a hyperblock for predicated execution with the same benchmarks. Also, the performance is less changed with predicated execution regardless of the capability to execute more branches per cycle, which can be a performance bottleneck when the branch resources are scarce and handling multiple branches per cycle is difficult. Therefore, for an optimising and aggressive VLIW compiler, we use predicated execution through hyperblock scheduling at the expense of higher implementation cost.

4.1.2 General Structure

The basic structure of a compiler consists of two main parts, the front-end and the back-end. The purpose of the front-end is to convert the input high-level language into an appropriate intermediate code for further processing in the next stages. Optimisations and code generation for the target architecture are performed by the back-end.

The compiler for the EVA is implemented as a multi-pass compiler. The results of each pass are saved in files and used as input for the next pass. This results in a longer compile time, however, as an experimental compiler, it enables us to investigate different optimisation methods and furthermore, to modify each pass irrespective of the implementation details of other passes when the interface between passes is unchanged. We used C++ (as do *SUIF* and *machsuif*) to implement different passes in the compiler. Preliminary work of the structure and implementation of our compiler was presented in [Biglari-Abhari et al., 1997, Biglari-Abhari et al., 1998]. Figure 4.1 indicates the main blocks of the compiler.

The front-end is from the *SUIF* system, and produces the *SUIF* intermediate code (IC). The purpose of the *intermediate code restriction* is to convert the first level intermediate code to a second level to provide a one to one correspondence between each IC and machine operations. Classical optimisations need a single form of each operation. As our VLIW operations are based on the MIPS instructions, this is performed by the *mgen* pass, which is a slightly modified version of the original one from the *machsuif* and transforms the suif intermediate code to MIPS-I instructions.

Profile information is used mainly for hyperblock formation and optimisation (such as loop peeling). Information derived from the execution profile such as the execution frequency of the subroutine calls may be used during code generation too.

Figure 4.2 indicates the steps of profile information generation. We modified **HALT** [Young and Smith, 1996], which is an ATOM [Eustace and Srivastava, 1995] like profiler to gather profile information. The file produced by *mgen* is augmented by annotations which for example, indicate the unique numbers associated with branches and basic blocks and other required parts for instrumentation. This is done by the *label* pass. Then, *halt* inserts calls to the analysis routines, which employ the unique numbers introduced before, and other relevant information for the instrumentation points. The output of the *halt* pass is converted to an executable code and linked with the analysis routines to produce an executable code to be run on a MIPS-based platform. The profile results are saved in a file and are further processed to place the profile annotations into the input file to produce information in a suitable form to be used in the next passes. This can be performed several times if necessary to collect the information of different profiling processes.

Predicated execution introduces new challenges in compiler optimisation and scheduling phases for highly-optimised code generation. Conventional program analysis tools must be

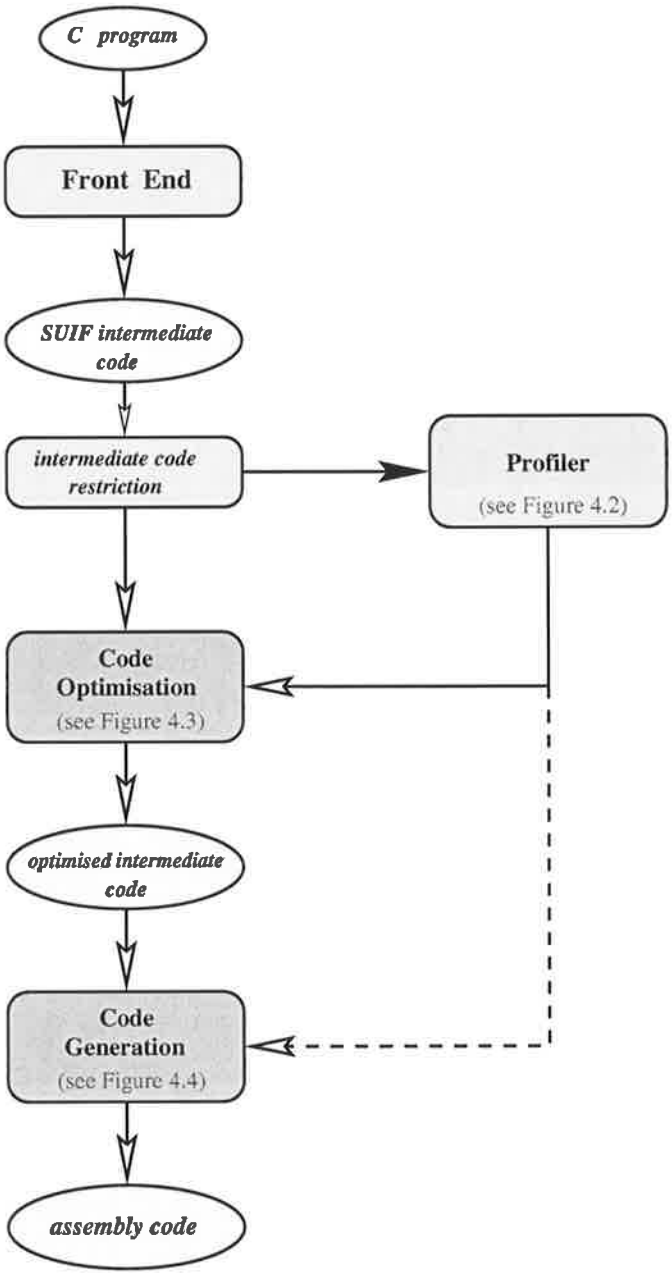


Figure 4.1: General structure of the main phases of the EVA VLIW compiler.

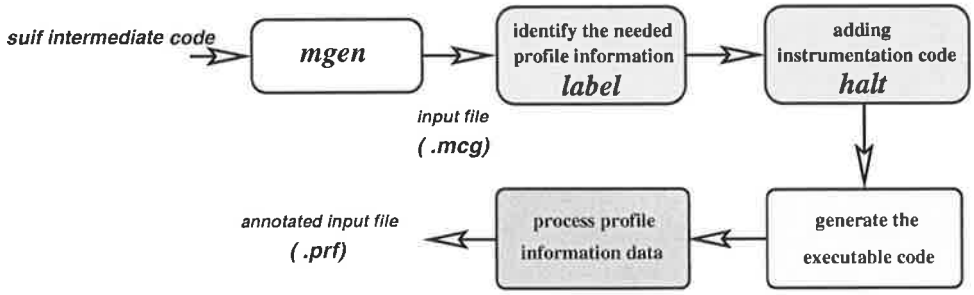


Figure 4.2: Main steps to generate the execution profile information for the benchmark programs.

augmented by information about the relationships between predicates. For example, conventional data flow analysis is not efficient in scheduling and register allocation, resulting in conservative code generation. Thus, a library was developed to provide the required information.

Figures 4.3 and 4.4 illustrate the structure of the two main parts of the back-end in our compiler, which are the optimiser and the code generator. More details of the important passes are described in the following sections. The libraries developed to facilitate implementation of our compiler are described in the appendix A.

4.2 Predicate-Sensitive Analysis

Conventional program analysis tools assume that operations are executed unconditionally in each basic block. To analyse programs containing predicated operations the conventional tools must be modified to consider the existence of predicates and their relationships.

Two structures have been introduced to communicate information regarding predicates to the compiler. The predicate hierarchy graph (PHG) [Lin, 1992] was used to find certain relationships among predicates in a hyperblock by the IMPACT compiler group. A PHG is a directed acyclic graph which represents the boolean equations under which each predicate is defined. The PHG consists of predicate nodes and condition nodes. Each predicate has a single predicate node in the PHG. Condition nodes correspond to predicate define operations. Edges show how predicates and conditions are determined. The PHG is queried to find if two predicates are disjoint.

Another structure referred to as a *partition graph* was introduced by the HP lab compiler

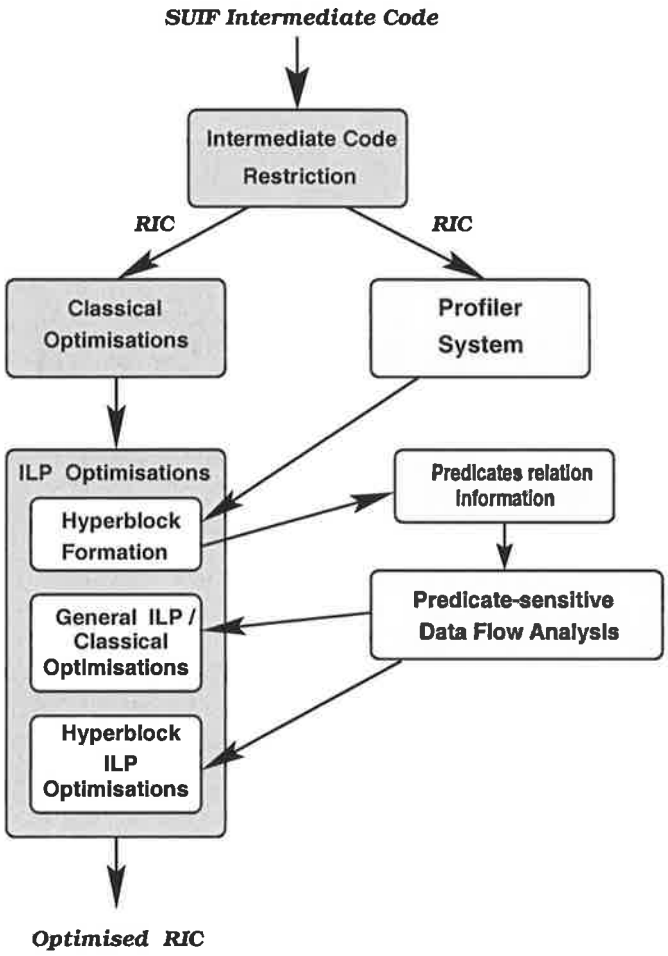


Figure 4.3: General structure of the code optimisation phase of the VLIW compiler.

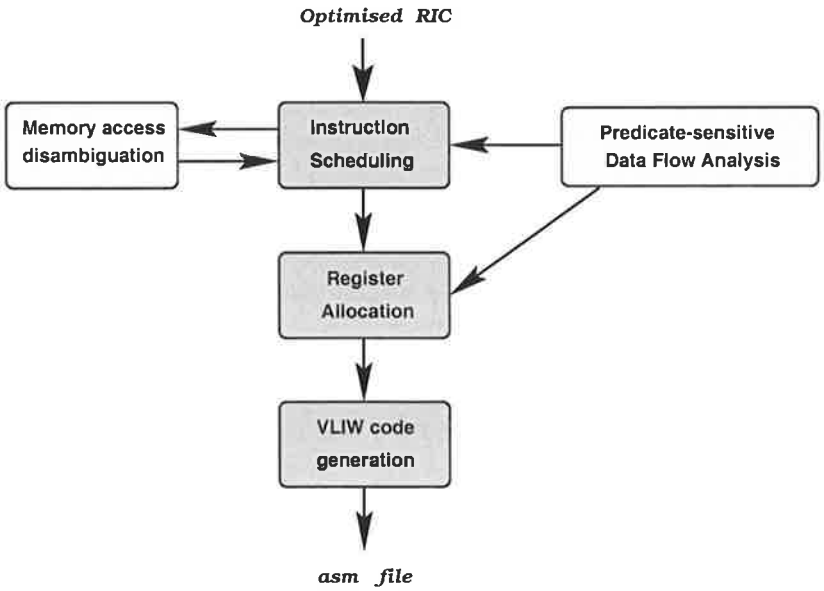


Figure 4.4: General structure of the code generation phase of the VLIW compiler.

group [Johnson and Schlansker, 1996]. A partition graph is a directed acyclic graph in which nodes represent execution sets and the labeled edges represent partition relations among the nodes. An execution set for a predicate is considered as the set of execution traces in which the predicate is assigned 'true'. A partition of a predicate is a division of the domain of the predicate into multiple disjoint subsets, where the union of these subsets is equal to the domain. Figure 4.5 illustrates an example of the set relation between predicate domains. In operation *peq p2(UN),p3(UC),r1,r2 (p1)* if $r1 = r2$ and the guard predicate *p1* is 'true', predicates *p2* and *p3* are set as 'true' and 'false' respectively. So, *p1* is partitioned into *p2* and *p3*. Here UN represents unconditional normal predicate behaviour and UC represents unconditional complement (see Table 3.1).

To use traditional optimisation techniques, which are based on control flow graphs, the predicated code can be transformed back to produce control flow graphs. Using the PHG, reverse if-conversion [Warter et al., 1993] is applied to the predicated code to produce the control flow graphs for the hyperblocks. Since this new control flow graph may in general contain some paths that cannot be traversed during execution, it produces conservative optimisation results.

It is more efficient to be able to perform data flow analysis directly on the predicated code. The partition graph and its related query algorithms for predicate relationships can

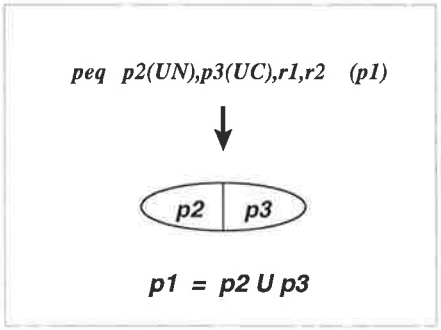


Figure 4.5: Set relation between predicate domains.

facilitate this.

We perform the required data flow analysis and predicate relation queries for instruction scheduling and register allocation based on the partition graph algorithms. Currently, *SUIF* and *machsuif* libraries do not have specific tools to support predicated execution. We have developed the required tools for our compiler.

4.2.1 Overview of Partition Graph Construction

The construction process for the partition graph involves two steps. First, an initial graph is built and then the graph is complemented, so that all nodes can be reached from the root.

The initial graph can be generated either from the original control flow graph when the predicates are assigned to basic blocks just before if-conversion, or directly from the predicated code. The former method seems easier. However, to keep the predicate relations after each code transformation the partition graph must be kept along with each code transformation process. Also, this method may not include the effect of other predicated code generated through optimisations or control critical path reduction [Schlansker and Kathail, 1995]. Therefore, we employ the second method which generates the partition graph directly from the predicated code.

To extract predicate relations, the predicate define operations are translated to a sequential form [Johnson and Schlansker, 1996]. In this form, the destination predicate value is a boolean function of the compare condition and the guard predicate. Table 4.1 indicates how the translation is performed.

Then, the predicated code is translated to the sequential single assignment (SSA) form. The SSA code is processed to extract partition relations. Each partition relation is recorded

Predicate define operation	Sequential form
P_{out} = **cmp**(*un*) (r1 <cond> r2) if P_{in}	P_{out} = (r1 <cond> r2) . P_{in}
P_{out} = **cmp**(*uc*) (r1 <cond> r2) if P_{in}	P_{out} = (!(r1 <cond> r2)) . P_{in}
P_{out} = **cmp**(*on*) (r1 <cond> r2) if P_{in}	P_{out} = P_{out} + (r1 <cond> r2) . P_{in}
P_{out} = **cmp**(*oc*) (r1 <cond> r2) if P_{in}	P_{out} = P_{out} + (!(r1 <cond> r2)) . P_{in}

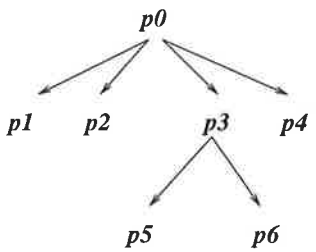
Table 4.1: Sequential form for predicate define operations

```
LB1:  load    r4, 0(r3)
      plt     p2(UN), p1(UC), r4, r11
      add     r1, r1, 1     (p1)
      add     r2, r2, 1     (p2)
      ple     p4(UN), p3(UC), r4, r7
      move    r7, r4        (p3)
      load    r3, 4(r3)     (p4)
      bne     LB1, r3, 0
      peq     p6(UN), p5(UC), r3, r7   (p3)
```

(a) assembly code after if-conversion

```
LB1:  load    r4, 0(r3)
      p2 =    (r4 < r11) . TRUE
      p1 =    ! (r4 < r11) . TRUE
      add     r1, r1, 1     (p1)
      add     r2, r2, 1     (p2)
      p4 =    (r4 <= r7) . TRUE
      p3 =    ! (r4 <= r7) . TRUE
      load    r3, 4(r3)     (p4)
      bne     LB1, r3, 0
      p6 =    (r3 = r7) . p3
      p5 =    ! (r3 = r7) . p3
```

(b) Sequential SSA form



(c) Partition graph

Figure 4.6: An example illustrating partition graph generation.

as a set of labeled edges in the partition graph. To make every node reachable from the root, additional partitions consistent with the existing partitions are added. In this way, the graph is completed and can be exposed to the query algorithms. The process is illustrated through an example in Figure 4.6.

4.3 Supporting Predicated Execution

Compilation for predicated execution involves converting sequences of operations representing the program control flow into predicated operations through if-conversion. If-conversion transforms control dependencies into data dependencies. Code should be if-converted early in the compilation process (early in the back-end) to be effective in both ILP optimisations and scheduling. The amount of code to be if-converted is dependent upon the number of resources in the target processor. Full if-conversion may reduce the performance for general applications due to lack of resources [Mahlke et al., 1992b]. Hyperblocks are formed through selective if-conversion of the most frequently executed paths. A study by August and his colleagues [August et al., 1997] indicates that performance can be improved if a balance is made between the amount of existent control flow and predicated code after hyperblock formation. This can be achieved through partial reverse if-conversion.

4.3.1 Hyperblock Formation

After classical optimisations, the hyperblock formation pass is performed. Hyperblock formation involves three main steps: Block selection, tail duplication and if-conversion [Lin, 1992, Mahlke, 1996].

Block Selection

Basic block X *dominates* basic block Y if X is visited on every path from the start point in the CFG to Y. A group of basic blocks with a single entry block for control flow, which dominates all other blocks in the group is called a region [Aho et al., 1986]. First, the largest regions are identified in each procedure. A block is only considered as a member of a single region with no internal cycles. Cycles are due to loop back edges. A loop back edge is an edge from basic block Y to block X if X dominates Y. Since only branches that are not loop back edges or region exit edges are considered for if-conversion, all loop back edges

are coalesced into a single back edge. For this purpose, a new basic block is added to the region, which contains the control transfer operation required for the back edge. Then, all other previous back edges are modified to target this new basic block. Figure 4.7 shows an example.

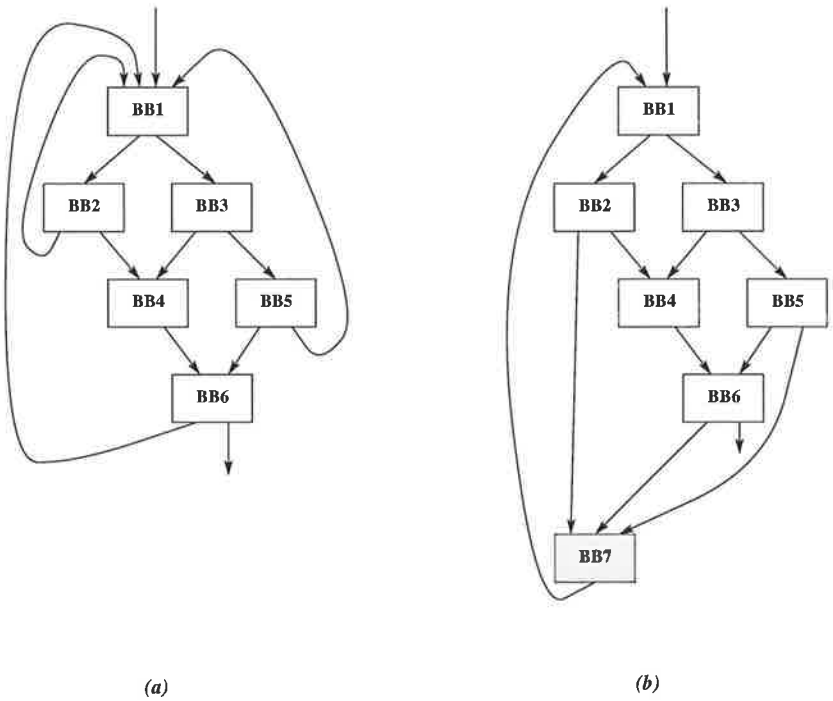


Figure 4.7: An example illustrating the loop back edge coalescing. (a) The original loop with multiple back edges. (b) The modified loop with a new back edge.

The block selection algorithm heuristically considers several criteria in determining whether to include different execution paths in the region for the hyperblock. Profile information provides the execution frequency of each basic block. To employ the processor resources more efficiently, less frequent execution paths are avoided. In this manner, the processor resources are not occupied for these paths. In addition, paths with a smaller number of operations, lower dependency height, and less hazardous operations are given higher priority. Hazardous operations like unresolvable memory accesses and subroutine calls limit the effectiveness of optimisation and scheduling of the hyperblock [Mahlke, 1996]. Higher priority paths are included in the hyperblock having regard for the estimated available execution resources and the characteristics of the path. The set of blocks in the selected paths form the hyperblock.

Tail Duplication

To remove side entry points to the hyperblocks, tail duplication is performed. In this process, basic blocks in the hyperblock, which are the target of control flow from outside of the hyperblock are cloned and the original control transfer of side entries are adjusted to the corresponding duplicated blocks. Basic blocks are duplicated at most once, even for other entry points.

If-conversion

If-conversion, as the final step of hyperblock formation, eliminates the control flow among the basic blocks of the hyperblock through predicated operations. The RK if-conversion algorithm [Park and Schlansker, 1991] has been employed in our compiler for this purpose.

For if-conversion, first the predicates are assigned to the basic blocks. Then, the operations which define the predicates are inserted into the basic block to keep the program semantics. In the RK algorithm, the R function determines how to assign the predicates to basic blocks, and the K function indicates how a predicate is defined and where it is placed.

Control dependencies inside the hyperblock are calculated from the control flow graph [Ferrante et al., 1987]. Then, it is decomposed into the R and K functions. Based on the R function, one predicate register is assigned to each set of basic blocks with the same control dependencies. Using the K function, predicate defining operations are inserted in all basic blocks, which are the source of the control dependency. The predicate compare condition is determined by the corresponding branch condition, and then the branch is removed. All operations in the basic block are conditioned under the predicate assigned to it. Using a topological sort of the hyperblock control flow graph, the predicated code is placed into the final hyperblock. Figure 4.8 shows a simple program before and after if-conversion. The shaded area indicates the code selected for if-conversion.

4.4 Code Optimisation

To produce more efficient and parallel code both general and machine-dependent code optimising transformations are required. Code optimisation for our purpose can be classified as either classical or ILP optimisations, which are described in the following sections.

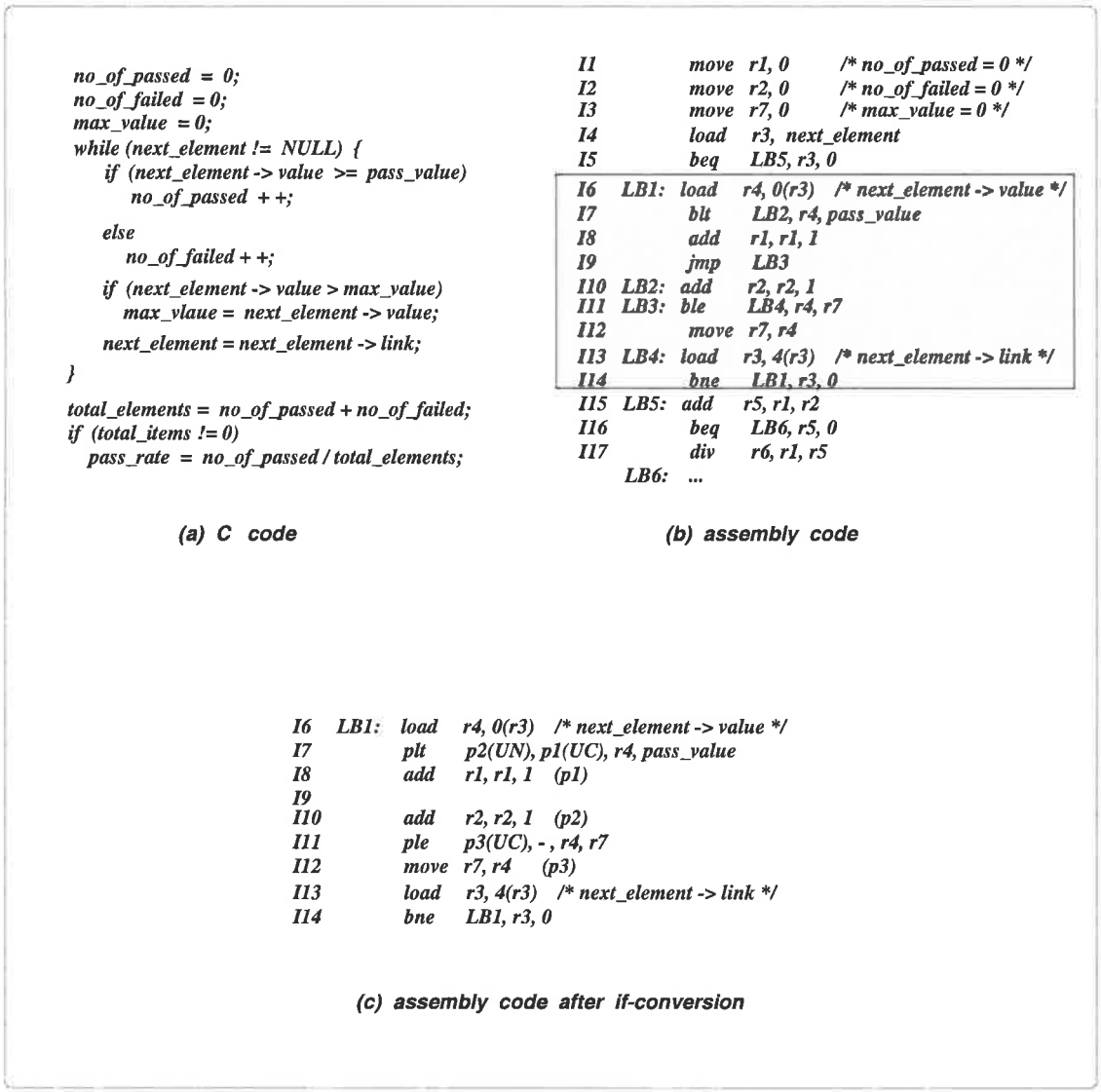


Figure 4.8: A simple program and its assembly code before and after if-conversion.

We have implemented most of the optimisation techniques discussed here based on algorithms from [Aho et al., 1986, Mahlke, 1992, August, 1996].

4.4.1 Classical Optimisations

The aim of classical optimisation is to reduce the size of a program and improve its execution time. Some conflicts may exist between these aims but our main concern in this work is reducing the execution time even at the expense of increasing the code size. Depending on the program area considered for code transformation, three types of optimisations are performed. These include local, global and loop optimisations.

Local Optimisations

Optimisations performed on the individual basic blocks are called *local optimisations* [Aho et al., 1986]. No information regarding other basic blocks is required for this type of optimisations. The following local optimisation techniques have been implemented in our VLIW compiler. Some of them are also applied for the global optimisations.

Copy Propagation - Copy propagation replaces the future uses of a variable x, which is assigned by the value of another variable y, with the variable y. This helps eliminate useless assignments.

Common Subexpression Elimination - If a computed expression appears again in the code sequence, while its source operands have not been changed, it is not necessary to recompute the expression. The previously computed value is stored in a register and is reused for the next occurrence of the expression.

Constant Folding - When the operands of an operation are constant values, the operation can be evaluated at compile time. Also, operations such as add with zero and multiply by one or zero can be evaluated at compile time. This optimisation removes the need to execute these operations.

Dead-code Elimination - The code that computes a value, which is never used, is a dead code. Dead code may be introduced after performing some other optimisations. *Dead-code elimination* prevents executing unnecessary operations in the final code.

Strength Reduction - Operations like multiply, divide and remainder have longer latency then other ALU operations. *Strength reduction* is a technique to replace these operations with an operation with a lower latency like add, subtract, or shift, where possible to reduce the

dependency height. This optimisation is performed when the compiler can make sure that the new code is executed faster.

Redundant Memory-access Elimination - When the compiler can make sure that the value in a memory location has not been changed then additional load or store operations can be removed.

Global Optimisations

The scope for global optimisation covers the entire procedure. Therefore, data flow information among the basic blocks is needed. To gather these information, systems of equations relating the information at different points in a procedure are solved iteratively. Available definition information, live variable information, and available expression information are required for global optimisations. Due to the existence of predicated code, a predicate-sensitive data flow analysis is required to avoid conservative optimisation decisions.

Copy propagation, common subexpression elimination, dead-code removal and redundant memory-access elimination are applied as global optimisation.

Loop Optimisations

Many programs spend most of their execution time within loops. So, reducing the number of operations in a loop will improve the execution speed even at the expense of increasing the amount of code outside the loop in some cases. Loop optimisations are applied to *natural* loops, which have a unique entry node called the *header* and at least one path to the header referred to as a *back edge*.

Natural loops are identified through finding loop headers and back edges in the control flow graph. For this purpose, dominator information for basic blocks is first calculated.

Some transformations include moving operations before the loop header to a preheader block. A loop preheader is the only predecessor block of the loop header which is not in the loop and the header is its only successor. If the loop does not have a preheader block, a new block is created for this purpose and the control transfers to the loop are adjusted accordingly.

The following transformations are considered in our VLIW compiler as the classical loop optimisations.

Invariant Code Removal - Operations in the loop whose source operands do not change

in each iteration are moved to the loop preheader block. This is referred to as *invariant code removal*. In this way, these operations are executed only once.

Induction Variable Strength Reduction - Basic induction variables are used as index in each loop iteration. Also, address calculations for data structure elements like an array are based on these variables. Changes on the induction variables are typically increment or decrement. Induction variable strength reduction replaces variables whose value is a linear function of the basic induction variable with a new basic induction variable and a simpler increment or decrement operation. This may decrease the dependency height in the loop.

Induction Variable Elimination - Induction variable elimination considers as useless the induction variables which are not live after the loop exit or whose contents are not used in the loop, and removes them.

Global Variable Migration - Global frequently accessed variables which live in memory are moved into a register for the duration of the loop. So, memory accesses are replaced by faster register accesses within the loop.

4.4.2 ILP Optimisations

In order to extract more ILP to form the VLIW instruction, a number of ILP optimisations are performed after classical optimisations. The ILP optimisations considered in this compiler are as follows:

Loop Unrolling

Loop unrolling is a technique to combine multiple iterations of a loop into a single iteration. It is used to increase the number of operations exposed to the scheduler for further optimisations.

Register Renaming

Output and anti-dependencies limit the amount of ILP extraction. These dependencies do not arise from the program characteristics and can be removed or reduced through *register renaming* [Cytron and Ferrante, 1987]. Register renaming is very useful especially after loop unrolling or loop peeling to reduce the dependency height.

Accumulator Variable Expansion

In some loops the value of a variable is recomputed and accumulated in each iteration. This variable is called an *accumulator variable*. Redefinitions of these variables impose more dependencies after loop unrolling. To remove these dependencies, for a loop unrolled N times, N temporary accumulator variables are placed instead of the original one. At exit points of the loop all of these temporary accumulator variables are combined again to produce the correct result. Using this technique removes additional dependencies and provides more ILP.

Induction Variable Expansion

Induction variables are used as loop indices or to access array elements inside the loop. These variables may be referenced many times in an unrolled loop. To remove the dependencies between induction variable definition and their use, N temporary induction variables are created. N is the number of operations which define the induction variable. Each new induction variable replaces one definition of the original induction variable and is initialised in the loop preheader. To avoid dependency between the definition and use of the temporary induction variable, its update operations are placed near the end of the unrolled loop body.

4.4.3 Hyperblock-specific Optimisations

In addition to the above optimisations, there are some optimisation techniques which are only applied on hyperblocks. The following hyperblock-specific optimisations have been implemented in our VLIW compiler.

Predicate Promotion

Research by the IMPACT compiler group indicates that the performance of hyperblock scheduling is highly dependent on speculative execution [Mahlke, 1996]. In addition to the conventional speculative motion of operations above the branches which they depend on, *predicate promotion* provides another form of speculative execution for predicated code [Mahlke et al., 1992b]. Predicate promotion changes the predicate of an operation to another predicate (called the ancestor), which was used to compute the current predicate. The ancestor predicate is less constrained than the original predicate. Therefore, the promoted

operation is executed under fewer conditions than the original operation. Predicate promotion reduces the dependency height of the code.

Three types of algorithms have been presented for predicate promotion [Mahlke, 1996]. Currently, the *simple predicate promotion* algorithm is implemented in our compiler. Other algorithms are *multi-definition predicate promotion* and *renaming predicate promotion.*

Simple predicate promotion is applied on operations whose predicates are computed by a single predicate definition operation (which we call the promotion target operation). The candidate predicated operation for simple predicate promotion has some conditions. Its destination register must not be live where its ancestor predicate was defined. Also, the destination register must not be written by other operations in the control paths between the candidate and the promotion target operation. Then, depending on the speculative execution model adopted, the proper form of the candidate operation is inserted and its predicate is promoted to predicate of the promotion target operation. Simple predicate promotion is iteratively applied where possible. Multi-definition predicate promotion is similar to the first one and is used for predicates which are defined by more than one operation. The same conditions exist for this type of predicate promotion, but the predicate is promoted to 'true' predicate in this case. In renaming predicate promotion, first an evaluation of the profitability of this transformation is made and then it is applied if profitable.

Loop Peeling

Loop peeling is a technique to separate the first several iterations of the loop. Each peeled iteration is a copy of the loop body and is placed before the rest of the modified original loop. Determining which loop is suitable for peeling and the number of peeled copies is a complicated task. Loop peeling is an effective optimisation for a hyperblock. To perform loop peeling, loop behaviour, dependency height, and resource utilisation patterns must be considered. Execution profiles can provide information regarding loop behaviour. Dependency height and resource utilisation patterns can be estimated by the compiler.

In a hyperblock, too many peels cause much code expansion and excessive resource utilisation. This leads to a possible loss in performance. Also, too few peels cause early exit from the hyperblock and therefore, the advantage of a hyperblock optimisation cannot be employed effectively.

Figure 4.9 illustrates an example of hyperblock loop peeling and its benefits. As men-

tioned before, a hyperblock may not contain inner loops. When a frequently invoked loop has only a few iterations per invocation, conventional techniques cannot expose ILP because other ILP loop optimisations like loop unrolling and software pipelining [Charlesworth, 1981] rely on many iterations on each loop invocation. In addition, loops which do not have enough parallelism by themselves cannot be optimised by these techniques. On the other hand, loop peeling makes it possible to overlap the execution of loop operations with their surrounding code.

In situations where the regions considered for hyperblock formation are not large due to inner loop structures, loop peeling can help form a larger hyperblock. As shown in Figure 4.9 (a), the original control flow graph is divided into three small regions for hyperblock formation. After loop peeling in Figure 4.9 (b), a larger hyperblock can be formed. Figure 4.9 (c) indicates the flow graph after hyperblock formation.

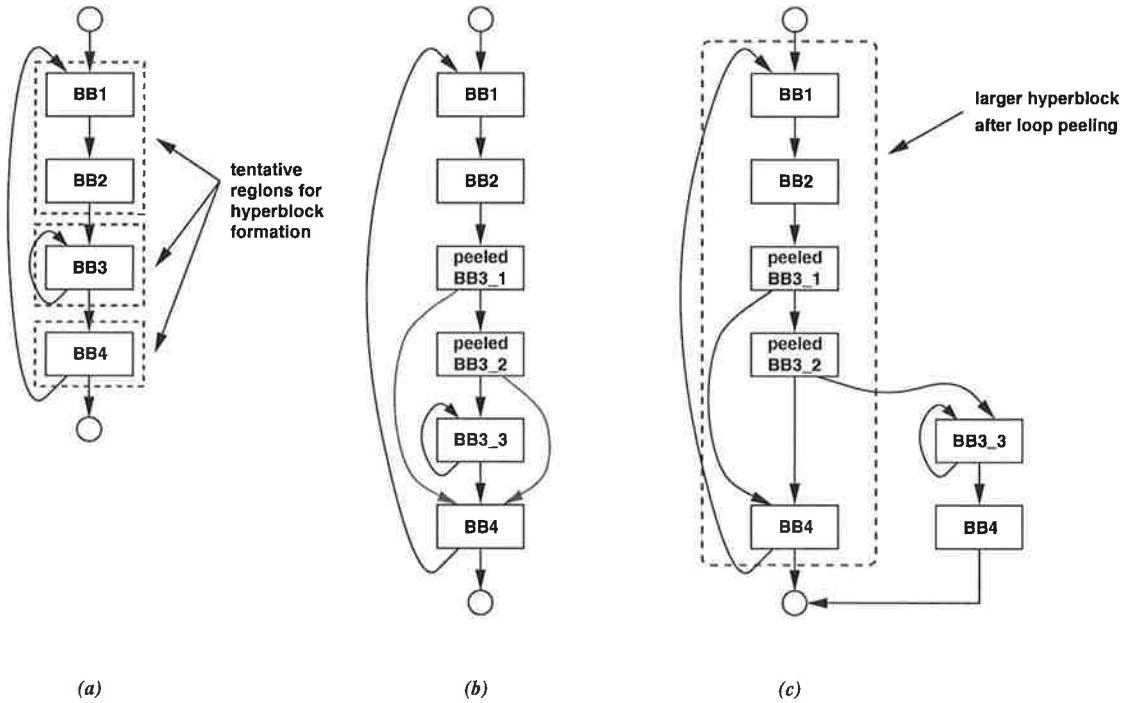


Figure 4.9: An example of hyperblock loop peeling. (a) Original control flow graph with tentative regions for hyperblock formation. (b) Loop BB3 is peeled three times. (c) A large hyperblock is formed after loop peeling.

To select suitable loops for peeling, some general guidelines can be followed. Generally,

low iteration count loops are considered as good candidates for loop peeling. When the number of peeling is more than the number of iterations, it is less likely that the program exits the hyperblock before completion. Some other loops may have different iteration counts for each invocation. When the loop has a low iteration count it benefits from the above mentioned advantages of peeling. When it has a high iteration count, the rest of the loop after peeling, which is referred to as the *recovery loop* [August, 1996] will be invoked several more times and therefore, it is useful to optimise it by conventional loop optimisation techniques.

Optimisations and scheduling after loop peeling may change the characteristics of the loop, so it is not possible to know exactly how much to peel a loop. The loop peeling selection heuristics should be such as to not make a hyperblock useless by too many peels, causing more operations to saturate the processor resources, or too few peels causing the recovery loop to be invoked more often.

The following heuristic, which is based on work by the IMPACT compiler group [August, 1996], provides a reasonable number by which to peel the candidate loop. In this heuristic the number of operations in a peeled loop must be less than a specified value. This will help to estimate the resource usage of the peeled loop. Also, there is an upper bound on the number of peels. Profile information is also used to estimate the number of invocations for the recovery loop and the efficiency of loop peeling.

4.5 Instruction Scheduling

Instruction scheduling is used to rearrange the sequence of operations so that the execution of the longest sequence of operations can be started as soon as possible and to increase the number of concurrent operations for execution. Scheduling is an NP-complete problem [Dewitt, 1976], and obtaining an optimal solution for all cases is not possible. Therefore, it is attempted to find a nearly optimal solution having regard to dependencies among operations and the available resources in the target architecture.

Scheduling algorithms are based on the nature of the control flow graph that they can schedule [Rau and Fisher, 1993]. Algorithms that can only schedule a single acyclic basic block are called *local scheduling* algorithms. Algorithms that schedule operations across several basic blocks are known as *global scheduling* algorithms. These algorithms utilise information about the direction of conditional branches, which come from previous execution

profiles and heuristics.

Local scheduling causes the execution time of each basic block to be nearly optimum. However, this does not necessarily cause the execution time of the entire program to be nearly optimum, because the processor waits at each branch until all operations before it are in execution. Global scheduling is able to rearrange operations from different basic blocks towards a nearly optimum scheduling for the program.

In this section, issues concerning the scheduler in the EVA VLIW compiler are discussed. Figure 4.10 shows the basic parts of the scheduler.

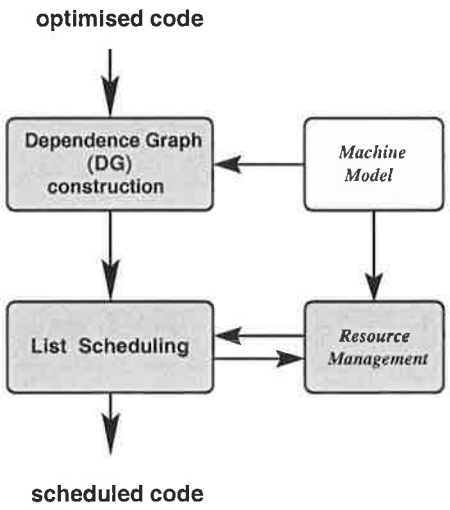


Figure 4.10: General structure of the scheduler.

4.5.1 Dependency Representation

Dependencies among the operations in the program are often represented by an extended directed acyclic graph. Each operation may only be dependent upon the operations preceding it, resulting in the acyclic nature of the graph. Nodes in the dependency graph (DG) represent the operations that are to be executed in order to perform the semantics specified in the source program. Usually, additional information is included in each node to identify both the fields needed and the resources used and defined by the operation. This provides all information needed by the scheduler at each node.

Flow dependencies (RAW), anti-dependencies (WAR) and output dependencies (WAW)

are marked on DG edges. Of these three types of edges, only flow dependency edges are necessary to keep the program semantics. Output and anti-dependencies are the result of reusing processor resources. The removal of these dependencies has a large impact on the resulting schedule. Resource binding should occur as late as possible to reduce this impact. Control dependency edges indicate the ordering constraint between a branch operation and the operations before and after it. Other types of edges such as constraint edges based on the adopted speculation model may be added to enforce some restriction of operation movement with respect to each other. Generally, all the required constraints regarding the operation scheduling are summarised in the dependency graph.

4.5.2 Resource Management

Resource modelling is employed to check for resource conflicts during instruction scheduling. Two approaches have been used in compilers for this purpose. One method looks backward over already scheduled code when an operation is considered for scheduling in the current cycle. To perform this task, a list of scheduled operations is maintained to find the cycle in which the operations were issued. To schedule a new operation, this list is checked to find any resource conflict between the new operation and every previously scheduled operation. Therefore, it is required to do several checks prior to scheduling each operation, resulting in a slow scheduling process. Also, the implementation of this type of resource modelling is dependent on the target processor implementation.

The second method looks forward over the resources which are already dedicated to the scheduled operations. In this method, the pipeline of the processor is simulated by the compiler. A bit matrix referred to as *reservation table* is maintained in each future cycle and is checked against the resources already committed in the current and future cycles when scheduling a new operation. The size of the reservation table is proportional to the number of resources and the length of the longest pipeline. Every check for a resource conflict needs an AND operation on bit matrices. When the operation can be scheduled another OR operation is needed to update the bit matrices. Also, the required storage space for the reservation table is large. Some techniques have been introduced to overcome the large space problem [Eichenberger and Davidson, 1996, Gyllenhall et al., 1996], but checking a resource conflict is still a bit-matrix operation.

Other techniques employ finite state automata [Müller, 1993, Bala and Rubin, 1997] in

order to increase the speed of resource conflict detection through a fast table lookup. These methods have the advantage that the finite state automaton is built only once for a given processor implementation. A lookup into the automaton transition table indicates if there is a legal state transition and thus the candidate operation can be scheduled.

As the EVA has a relatively simple hardware complexity in comparison to out-of-order superscalar processors, we employed a technique similar to the reservation table. Two resource description files are defined for the processor. The first one describes the resource group used by each operation. The second file defines the processor resources and indicates the relative cycles in which these resources are used.

4.5.3 Scheduling Process

Figure 4.11 shows the main steps of the scheduling process. These are described in more detail in the following sections.

```
// Scheduling main steps

read machine model specification files and construct the required data structures
for each procedure
    identify scheduling regions (e.g. hyperblock, superblock or a single basic block)
    for each scheduling region
        construct the dependency graph
        perform list scheduling
        convert the speculative operations into their non-trapping version
    write the new procedure
```

Figure 4.11: Main steps of scheduling.

Dependency Graph Generation

Figure 4.12 indicates the basic algorithm to construct the dependency graph (DG). In addition to the data dependency edges between nodes in the DG, constraint edges are added to maintain program correctness and impose the required restrictions on operation scheduling. In our scheduler, control transfer operations are not allowed to be reordered, so constraint

edges are added between them. Also, control dependency edges are included from operations above a branch (or subroutine call) to the branch (or subroutine call), and from the branch (or subroutine call) to the operations below it if their destination registers are in the live-out set of the control transfer operation. To prevent illegal speculative code motion, other constraint edges may be added depending on the underlying speculation model. In all these cases, the predicate-sensitive data flow analyser is queried to determine the required information (such as liveness).

At first, memory dependency edges from loads to stores, stores to loads, and stores to stores are included if there is a real or ambiguous dependency between them. Dependent store operations are not allowed to be reordered. However, other memory dependency edges may be modified if a run-time memory access disambiguation mechanism (such as the memory conflict buffer described in section 2.3.2) is provided.

In order to perform scheduling for the memory conflict buffer (MCB) [Chen, 1993, Gallagher, 1995], a check operation (*check_br*) is added immediately after each load operation. Since the source operand of the check operation is the destination operand of the corresponding load, a dependency edge is added from the load to the inserted check operation. Then, ambiguous dependencies between the load and previous stores are removed. This is performed for only a limited number of stores to prevent more register pressure and reduce the chance of false conflicts [Gallagher, 1995]. Control dependencies which exist for the load are added to the check operation too. Figure 4.13 shows an algorithm to modify the dependency graph to prepare for MCB scheduling.

At scheduling time, if the load is not scheduled above any stores upon which it was dependent, the corresponding check operation is removed. Otherwise, the load is converted to a *pre_load* and the required correction code is generated. The correction routine includes the load and all operations dependent upon it. Meanwhile, the compiler makes sure the operands which are used in the correction routine are not destroyed. The correction routine branches to tail duplication code as it is not allowed to jump into the hyperblock [Gallagher, 1995]. Later, after scheduling, some optimisation can rearrange those jumps to a more appropriate position in the code.

```
for each operation <op1> in the scheduling region
    create a node in the dependency graph
    add dependency edges (RAW, WAR, WAW) to the previous operations
    if <op1> is a memory access operation then
        // add memory constraint edge
        if <op1> is a store operation then
            for each previous load or store <op2>
                if there is a real or ambiguous dependency between <op1> and <op2> then
                    add (MEM) edge from <op2> to <op1>
        else   // for loads
            for each previous store  <op3>
                if there is a real dependency between <op3> and <op1> then
                    add (MEM) edge from <op3> to <op1>
                else if there is an ambiguous dependency between <op3> and <op1> then
                    add (MEM-A) edge from <op3> to <op1>

    if <op1> is a control transfer operation then
        for each non control transfer operation <op4> above <op1>
            if dst of <op4> is in the live-out set of <op1> then
                add constraint edge (CTS) from <op4> to <op1>

    for each previous control transfer operation <op5>
        if <op1> is a control transfer operation then
            // control transfer operations are not allowed to be reordered
            add constraint edge (CTR) from <op5> to the <op1>
        else
            if dst of <op1> is in the live-out set of <op5> then
                add constraint edge (CTS) from <op5> to the <op1>
```

RAW, WAR, WAW : Data dependencies.
CTR : Control dependency between control transfer operations.
CTS : Control dependency between a control transfer and a non-control transfer operation.
MEM : Real memory dependency between memory access operations.
MEM-A : Ambiguous memory dependency between memory access operations.

Figure 4.12: Basic algorithm for dependency graph construction.

```
// Input:  dependency graph of the scheduling region
           K : the parameter to limit data over-speculation.

for each operation <op> in the scheduling region
  if <op> is load and has ambiguous memory dependency edge (MEM-A) then
     insert an appropriate check_br after <op> in the operation list
     add a new node to the dependency graph for check_br
     add control and memory dependencies of <op> for check_br
     add RAW dependency from <op> to check_br
     remove dependencies of <op> from K preceding stores above it with (MEM-A) edge
```

Figure 4.13: Algorithm to modify the dependency graph to prepare for MCB scheduling.

List Scheduling

In most instruction schedulers a technique called list scheduling is employed to pack independent operations to be executed concurrently [Beaty, 1991, Bringmann, 1995]. A list scheduler constructs the schedule based on the priority of each operation. List scheduling is relatively easy to implement and has been shown to produce good results in the presence of good heuristics [Davidson et al., 1981].

After building the dependency graph, the earliest issue cycle for all operations is calculated. In our list scheduling algorithm, two lists are used. These are *candidate list* and *scheduled list*. After prioritising all operations in the dependency graph, operations whose earliest issue cycle is less than or equal to the current cycle are placed in the candidate list. The candidate list is sorted from highest to lowest priority.

In each cycle, the candidate list is examined. If the operation is not dependent on scheduled operations whose latencies are not fulfilled, or on other unscheduled operations, it is considered as ready for scheduling. Then the resource manager is queried. If the resource is available the operation is scheduled. Otherwise, the earliest issue cycle of the operation and all other operations dependent upon it are incremented. When all operations in the candidate list are processed, the current cycle is advanced by one. The scheduling terminates when all operations in the DG are scheduled. Figure 4.14 shows the basic scheduling algorithm.

Due to resource limitations, to select the best ready operation to schedule in each cycle, these operations should be prioritised. Dependency height is often used for this purpose. In

```
// Input:  Dependency Graph   (DG)
           Priority calculation algorithm

// Output:  scheduled_list


Calculate the earliest issue cycle for each operation in the DG
Prioritise operations in the DG based on the input algorithm
current_cycle = 0
candidate_list and scheduled_list are empty
do
  // extract candidate operations
     for each unprocessed operation <op> in the DG
        if earliest_issue_cycle of <op> <= current_cycle  then
              append <op> to candidate_list
              set <op> as processed

  // prioritise candidate operations
     sort candidate_list based on priority

  // process candidate operations
     for each operation <op> in candidate_list
        if <op> does not have dependency & resources are available then
             append <op> to scheduled_list  (<op>, cycle, issue_slot, speculative_flag)
             update the state of the resource manager
             remove <op> from candidate_list
        else
             increment earliest_issue_cycle of <op> by one cycle
             update earliest_issue_cycle of operations dependent upon <op>

     advance current_cycle by one cycle
  while non-scheduled <op> left
```

Figure 4.14: Basic scheduling algorithm.

path-oriented scheduling schemes such as superblock and hyperblock scheduling, which rely on profile information to select the best paths to be included in the scheduling region, it is important to reduce the impact of changes, due to differences in assumptions made for profiling and what happens at the execution time, on performance. When the run-time behaviour of the scheduled code is different from what was assumed during scheduling based on profile information, the overall performance may degrade. In addition, the usefulness of speculation is dependent upon the accuracy of profile information. Speculating an operation above a frequently taken branch may delay the execution of the branch resulting in performance loss.

Fisher [Fisher, 1993] proposed a heuristic called *speculative yield* in order to measure the effectiveness of speculating an operation by considering branch probabilities. It has been used with the dependency height to calculate the priority of each operation for superblock scheduling [Bringmann, 1995]. In Bringmann's work priorities are calculated statically once for each operation x before list scheduling using the following equation.

$$Priority_x = \sum_{i=1}^{n} (Prob_i \times (MaxLT + 1 - LT_x)) \tag{4.1}$$

Prob_i is the probability of exit_i (branch_i). n is the number of exits in the superblock or hyperblock region. LT_x is the late time of operation x with respect to exit_i. A late time for an operation is the latest time that an operation can be scheduled without delaying an exit [Deitrich and Hwu, 1996]. MaxLT is the maximum late time in the dependency graph.

In Bringmann's algorithm, late time is calculated based on the dependency height. However, not considering resource usage may reduce the effectiveness of priority calculation. For this purpose, we modified this algorithm so that the resource restrictions can be taken into account. We calculate two late times for each operation for an exit. These are based on dependency height and resource usage. The maximum of two late times is used as the late time in 4.1. MaxLT is the maximum late time considering the dependency graph and resource usage. Figure 4.15 illustrates our algorithm, which we refer to as the modified static priority calculation algorithm. In this algorithm, late times with respect to resource limitations are calculated through considering issue-width, type and number of functional units in the target processor.

Deitrich and Hwu proposed a heuristic, which is referred to as *speculative hedge* to calculate priority for unscheduled operations dynamically [Deitrich and Hwu, 1996]. In their algorithm, which we refer to as the dynamic priority calculation algorithm, first a critical

```
// Input: dependency graph (DG)
determine dependecy height for all operations in the DG
for each exit_j (branch) in the scheduling region
      calculate late time (LT1) regarding dependency height in the DG
      calculate late time (LT2) regarding resource restrictions
      LT_j = Max (LT1, LT2)
MaxLT = Max (LT_1, LT_2, ..., LT_j)
for each operation <OP_i> in the DG
      for each exit_j
          calculate LT_i, j
          priority_i = prob_j * LT_i, j
```

Figure 4.15: Modified static priority calculation algorithm.

need for each exit is identified. A critical need is related to the dependency height or any processor resource limitation. In this way, this algorithm takes resource limitations into account to prevent delaying of exits unnecessarily. Operations which result in earlier retirement of more branches (exits) are assigned a higher priority. This is calculated by considering processor issue width, availability of branch functional units and late time of operations ready to schedule. This calculation is performed in each scheduling cycle. It was shown that for SPEC CINT92 benchmarks, the dynamic priority algorithm improved the performance however, the compile time is 26% to 49% longer than the static Bringsmann's algorithm [Deitrich and Hwu, 1996].

In hyperblocks, the number of branches is much lower than in superblocks. Therefore, it seems the performance advantage of using a dynamic algorithm cannot justify large compilation time. We use the modified static algorithm to calculate operation priorities. Experimental results are presented in chapter 5.

4.6 Register Allocation

Register allocation is the process of assigning the best possible physical registers among those provided by the target architecture, to the virtual registers (and possibly variable symbols) in the intermediate code. Optimal register allocation is an NP-complete problem [Sethi,

1975]. Early techniques developed for this purpose attempt to approach the optimal solution for a single procedure [Chaitin, 1982, Chow and Hennessy, 1984, Briggs, 1992]. Recently, inter-procedural allocation techniques have also been studied [Kurlander, 1996].

Our register allocator uses one scheduling region (hyperblock, superblock or simply a basic block) each time for register assignment while considering software conventions for register usage on MIPS-based systems. The register set of the processor is divided by software convention into two different classes referred to as caller-saved and callee-saved. This may reduce the procedure call penalty at the expense of more complexity in the register allocation. In the EVA, the general-purpose, floating-point and predicate registers are divided into these classes.

To perform register allocation, the target machine registers are classified into different register banks. Each register bank is identified by the type of data and usage convention (such as integer callee-saved). Similarly to Chaitin's work [Chaitin, 1982] and the IMPACT compiler register allocator [Hank, 1993], we consider virtual registers for the allocation process. The allocation process assigns a location from an array of n registers to each virtual register. The index of the array can be considered as an abstract name for the register. Later, in the final phase, a proper physical register is assigned to each element of this array.

Basic steps of the allocation process are live range construction, interference graph construction and graph colouring, which are described in the following sections.

4.6.1 Live Range Construction

Live ranges indicate where a virtual register is live. For predicated code, it is necessary to perform liveness analysis by considering predicate relations in order to avoid conservative register allocation and more spill code generation. A study by Gillies and et al [Gillies et al., 1996] shows the importance of a predicate-aware register allocator. Predicate-sensitive analysis reduces register pressure and allows more efficient register allocation.

Live ranges are constructed for the selected region (basic block or a hyperblock). First a partition graph (as described in section 4.2.1) is constructed. Then, liveness information is determined. The live range of a virtual register is constructed by applying definition-use analysis on each operation.

4.6.2 Interference Graph Construction

The interference graph is used to take into account the relations among the virtual registers for the purpose of register assignment. The interference graph consists of one node for each live range. An arc joins two live ranges if they interfere. The importance of using predicate-sensitive analysis in interference graph construction is indicated in Figure 4.16. As this Figure shows, it seems at first that there is interference between the live ranges of virtual registers *v2* and *v3*. However, predicates *p1* and *p2* are disjoint. Therefore, the live ranges of *v2* and *v3* do not interfere. So, both of them can be assigned to the same physical register.

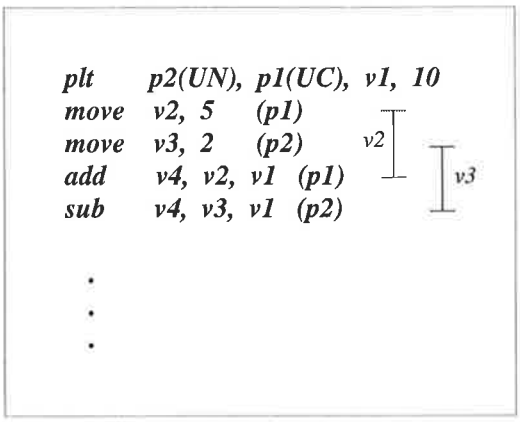


Figure 4.16: An example of non-interfering live ranges due to predicate-aware analysis.

The number of physical registers required is dependent upon the number of nodes in the interference graph. Some transformations may be used to reduce the size of the interference graph. A process called as *coalescing* [Briggs, 1992] is performed to merge two non-interfering live ranges, which are the source and destination of copy operations, into one live range.

As was mentioned before, we perform register allocation for a hyperblock or a basic block. Figure 4.17 illustrates the algorithm for interference graph construction. Predicate-sensitive liveness analysis using the partition graph for each scheduling region is performed. In this process, liveness information is combined conservatively at the boundaries of regions to reduce the complexity (as shown in Figure 4.18). This means that considering two regions *Reg1* and *Reg3*, liveness under predicate p_x in *Reg3* is promoted to the liveness under the 'true' predicate at the boundary.

```
// Input: scheduled regions (in the procedure)

for each scheduled region (SR)
    construct the predicate partition graph
perform liveness analysis for the procedure
for each SR
    determine the live-out information (liveness-set)
    for each operation in the SR in backward order
        q = qualifying (or guard) predicate
        for each dst operand (def) of the operation
            for each virtual register (vr) in liveness-set
                p_set = set of predicates under which vr is live
                for each predicate p in p_set
                    if (p and q are not disjoint )
                        add edge from def to vr
                update liveness-set
    for each src operand of the operation
        update liveness-set
```

Figure 4.17: The algorithm to construct the interference graph.

Similar approximations are performed in the predicate-aware register allocation method proposed by Gillies and his colleagues [Gillies et al., 1996]. In their method, the partition graph is constructed for the entire procedure by considering all predicates in the procedure. To employ the conventional bit vector data flow analysis, an array of bit vectors is created. The size of the array indicates the number of virtual registers in the procedure. The size of the bit vectors are based on the number of predicates in the procedure. The bit vectors are referred to as the *basis*. Then, the basis of basic blocks in the CFG are combined based on the required scheduling region. However, only some of the predicates (depending on the size limit of the basis) are considered in the analysis and the effects of the rest are approximated. The size of the bit vectors in our implementation is the maximum number of predicates in the scheduling regions. This is less than the total number of predicates in the procedure. The approximation in our case is the promotion of predicates in the liveness information at the boundaries of the regions if necessary.

After performing liveness analysis for the procedure, each scheduling region is traversed backwardly to make nodes in the interference graph and apply the effect of each operation on the liveness information. The updating of liveness information is illustrated in Figure 4.19,

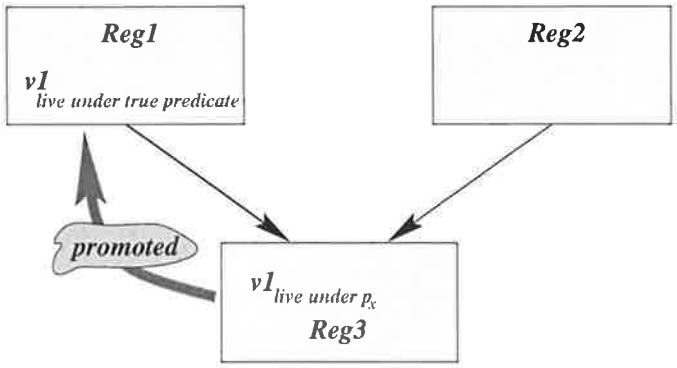


Figure 4.18: Predicate promotion to approximate the liveness of a virtual register at the region boundaries.

which gen/kill scheme represents generation and destroy of live values. This is based on Johnson and Schlansker's work [Johnson and Schlansker, 1996] and the EVA architecture.

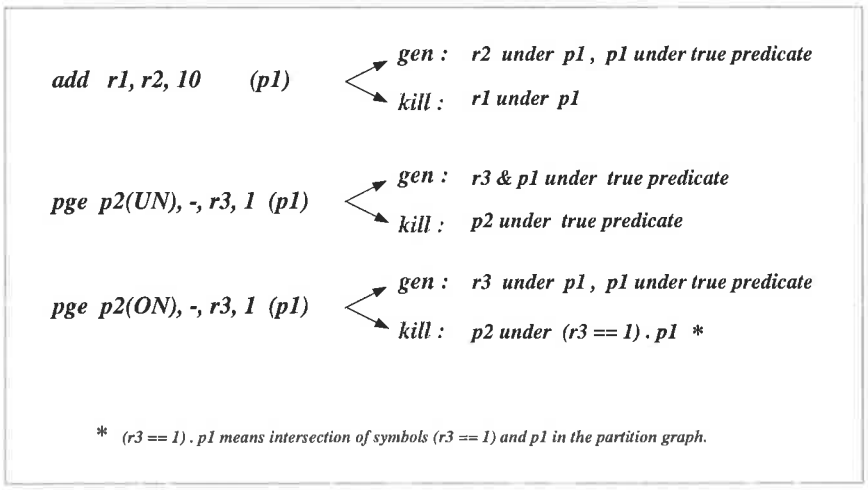


Figure 4.19: Liveness update in the presence of predicates. Each type of operation has a different gen/kill scheme for the liveness.

4.6.3 Graph Colouring

The purpose of graph colouring is to assign colours to graph nodes so that adjacent nodes have different colours. This approach is employed in register allocation to assign *n* physical

registers of the target architecture to m nodes in the interference graph [Chaitin, 1982, Chow and Hennessy, 1984]. When $m > n$, there are not enough physical registers available for all virtual registers. In this case, some registers are spilled. Spill operations are inserted at each definition and use of the virtual register in the live range. The spill cost is the time required to execute the spill code.

To reduce the spill cost, nodes in the interference graph are prioritised. The priority is used to determine the virtual register that is to be assigned next. Depending on the number of subroutine call operations in the live range, a register is selected from the caller-saved or callee-saved convention. To control the amount of spill code, live ranges with a few or no subroutine calls are assigned to the caller-saved register. For live ranges with more subroutine calls, the callee-saved registers are used. In this case, spill code is needed when the callee routine uses those registers. Two values are calculated, which are referred to as *caller_factor* and *callee_factor*, based on the spill cost in order to make a decision to use a caller-saved or a callee-saved register. The spill cost and the priority are calculated as follows [Hank, 1993].

$$spill\_cost_v = \sum_{i=1}^{k}(def_i + use_i) \times w_i - PF \tag{4.2}$$

This equation determines the spill cost of virtual register v. k is the number of operations in the live range. def_i is 1 if operation i defines v; otherwise, it is 0. If v is a source operand of the operation i, use_i is 1; otherwise, it is 0. w_i represents an estimation of the execution frequency of operation i based on the execution profile. We use PF (predicate effect factor), which represents the number of disjoint predicates of defs and uses in the live range. For unpredicated code this would be 0.

The caller_factor and callee_factor for a live range are defined as follows.

$$caller\_factor_v = spill\_cost_v - call\_weight \times leaf \times K_{\mathbf{r}} \tag{4.3}$$

$$callee\_factor_v = spill\_cost_v - (fn\_weight + 1) \times K_{\mathbf{e}} \tag{4.4}$$

In equations 4.3 and 4.4, $\mathrm{K_r}$ and $\mathrm{K_e}$ are the cost factors for caller-saved and callee-saved respectively. $\mathrm{K_r}$ and $\mathrm{K_e}$ are processor dependent. The *call_weight* is the sum of the execution frequency of any subroutine call in the live range. The parameter *leaf* is 0 when the current subroutine calls another subroutine, otherwise it is 1. The *fn_weight* is the execution frequency of the current subroutine.

Equation 4.5 shows how priority for a live range is determined and was adopted from [Hank, 1993].

$$priority_v = \frac{spill\_cost^3}{N} \tag{4.5}$$

N is the number of operations in the live range. So, smaller live ranges are given a higher priority. The significance of spill cost in calculation of the priority is reflected as power of 3 in equation 4.5.

After each virtual register is allocated to a location in the array of registers (or possibly spilled) in the register allocation process, the elements of this array are mapped to the physical registers corresponding to a particular bank. Finally, the VLIW assembly code is generated.

The spill cost and the priority calculation schemes for a live range that we use are similar to the IMPACT register allocator [Hank, 1993]. However, we modified it to consider the impact of the predicated code, which was not available in the IMPACT register allocator. We added the *predicate effect factor* (indicated as PF in equation 4.2), to consider the number of disjoint predicates of defs and uses in the live range. We do not expect our predicate-aware register allocator to be as efficient as the method presented in [Gillies et al., 1996] in considering the effects of predicates in register allocation. This is because we consider predicates only in our scheduling region, which is the scope of register allocation, and promote predicates outside the regions, while in [Gillies et al., 1996], the aim is to consider all predicates in the procedure where practical. As the scope of most optimisations (especially, ILP optimisations) in our compiler is a hyperblock, it is expected that register allocation to be effectively performed for the EVA by our method.

4.7 Summary

In this chapter, we described the design and implementation of a VLIW compiler for the EVA. No experimental or commercial compiler was available to capture the requirements of our research based on the characteristics of the EVA architecture. We employed state-of-the-art techniques to implement a suitable compiler for the EVA.

This compiler is based on the **SUIF** infrastructure, which includes *SUIF* and machine SUIF (*machsuif*) library and passes. The front-end, which converts a C program into a

RISC type *SUIF* intermediate code, is from the *SUIF* system. We implemented the back-end including the code optimiser and generator. We used some algorithms mostly from the research of the IMPACT and HP lab compiler groups to implement the back-end.

The EVA provides architectural features for predicated execution. Therefore, its compiler should generate predicated code. This is achieved through employing hyperblocks. A hyperblock is built through selective if-conversion of the conditional branches in the selected region. The control transfer information is kept by the generated predicates. The presence of predicates increases the complexity of the compiler. It is also necessary to perform predicate-sensitive data flow analysis in order to optimise and schedule operations efficiently. This also affects the final step of code generation, which is register allocation. Neglecting the relationship among the predicates reduces the quality of the code and increases register pressure. These effects were considered in the implementation of the compiler for the EVA and we developed the required tools for this purpose. All phases in the back-end (after forming hyperblocks) work on the predicated code.

After performing a group of classical and ILP optimisations including predicate-specific optimisations, code generation is performed. This step generates the VLIW assembly code after performing operation scheduling and register allocation.

The EVA compiler is the first VLIW compiler based on **SUIF** infrastructure. Our major contributions in design and implementation of this compiler are as follows. Currently, *SUIF* and *machuif* libraries do not have specific tools to support predicated execution. We have developed the required tools for our compiler. The libraries and new passes are described in appendix A. We proposed an algorithm to calculate the priority of operations in the scheduling process. It considers both the dependency height and resource usage to calculate the priority. Also, an algorithm for predicate-sensitive register allocation for hyperblocks was proposed. It considers a hyperblock as the scope of register allocation and takes the effects of disjoint predicates into account for register assignment.

Chapter 5

Experimental Evaluation of the EVA Compiler

Performance evaluation of an ILP processor and its compiler is a complex process and requires a systematic approach. Depending on the stage of design, the performance model may use varying levels of abstraction. In this thesis, architectural performance is considered, so we use the total number of execution cycles as the performance measure. This depends on both architectural features and the target processor workload or benchmark programs.

In this chapter, first different approaches to performance evaluation of ILP processing are reviewed. Some experimental results to indicate the capability of the EVA VLIW compiler in extracting ILP are presented. Also, experimental results for the modified static priority calculation algorithm, which was described in section 4.5.3, are presented.

5.1 Simulation Techniques for ILP Processing

As the complexity of processors and their compilers have increased, design and analysis of them require increasingly complex models and robust simulators. The simulator software takes the program compiled for an experimental target machine and executes it on an existing host machine. The complexity of the simulator is based on the performance evaluation criteria. Information about the behaviour of a program at run-time is captured by tracing tools. The accuracy and level of detail of this information depends on the analysis technique employed. For example, to study memory systems, address traces are collected. The dynamic number of operations is used for optimisation studies, and branch outcome frequencies drive

branch prediction analysis tools.

Direct simulation of detailed functional and pipeline timing of the target machine requires several thousands of instructions of the host machine (which runs the simulator) to interpret the behaviour of each target machine instruction [Bose and Conte, 1998]. To increase the simulation speed, a decoded execution trace of the program is fed into the simulator. Figure 5.1 shows a diagram of the traditional *trace-driven* simulator, which was used by Smith and his colleagues [Smith et al., 1989]. First, the benchmark program is executed on the host machine to generate a *trace* file, which captures information such as target address of branches and their direction, memory access addresses, operand values for selected operations, and the contents of the PC. The *trace* decoder uses the trace and object files to generate the appropriate input for the simulator. Details of the information provided by the trace file depends on the type of experiment. This simulator models only the *cycle-by-cycle* pipeline flow of the target machine. In some cases, *event-driven* simulation can be used [Bose and Conte, 1998].

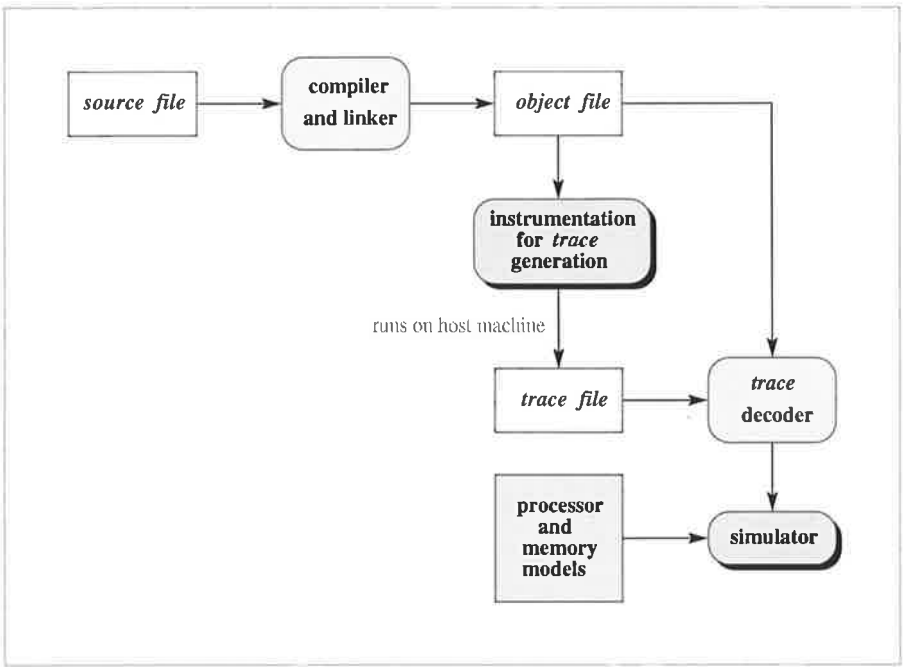


Figure 5.1: Basic diagram of a traditional trace-driven simulator.

A trace-driven simulator is faster than the direct interpretation method, because it does not require the processing of data values to maintain the machine state and for this purpose,

it relies on the execution trace information fed into it.

Nevertheless, trace-driven simulation requires a long time, which often limits the number of design alternatives that the researcher can consider. It may also need a large disk area to save execution traces. To reduce these problems, statistical trace sampling has been employed [Conte and Gimarc, 1995]. In this method, simulation results are produced using a sample representative of the entire workload. The method employed to collect the sample is critical. Statistical methods are used to predict the accuracy of the results. The use of statistical trace sampling methods on SPEC92 benchmarks indicated the maximum relative error in the predicted performance to be less than 3% [Conte and Gimarc, 1995].

A potential source of error in this method is the state-loss problem [Bose and Conte, 1998]. This occurs because, at each sample point of the trace, the state of the simulated machine is unknown. Therefore, the state should be built before applying the next trace sample. For this purpose, the beginning part of a trace sample can be used only to build the state. The next part of the trace sample is used to gather statistical performance information. Conte and et al [Conte et al., 1996b] presented techniques for calculating confidence intervals to reduce this problem.

Another approach which is referred to as *execution-driven* simulation [Covington and et al., 1988, Bose and Conte, 1998] or *compiled simulation* [Young, 1998] is faster than trace-driven simulation. It is more efficient when the instruction set of the simulation host machine is the same as, or very similar to, that of the machine being simulated. This method is based on *shade* [Cmelik and Keppel, 1993], which is an instruction-set simulator to generate execution traces to be used by analyser code. It dynamically cross-compiles executable code for the experimental machine into an executable code that runs directly on the host machine. Analyser code can be called by shade, which can determine the type and size of the required tracing information. Shade lacks timing-level simulation [Cmelik and Keppel, 1993]. An extension of this approach has been used for simulation of the IBM VLIW architecture [Moreno et al., 1996, Altman et al., 1996].

Figure 5.2 indicates the basic diagram of an execution-driven simulator. It consists of two main parts, the *translator* and the *cycle timer*. The translator converts the assembly code from the target experimental architecture into the host processor executable code. The translator also generates the instrumentation code, which includes counters at selected points in the code and calls to the cycle timer, and generates descriptors to indicate when and how

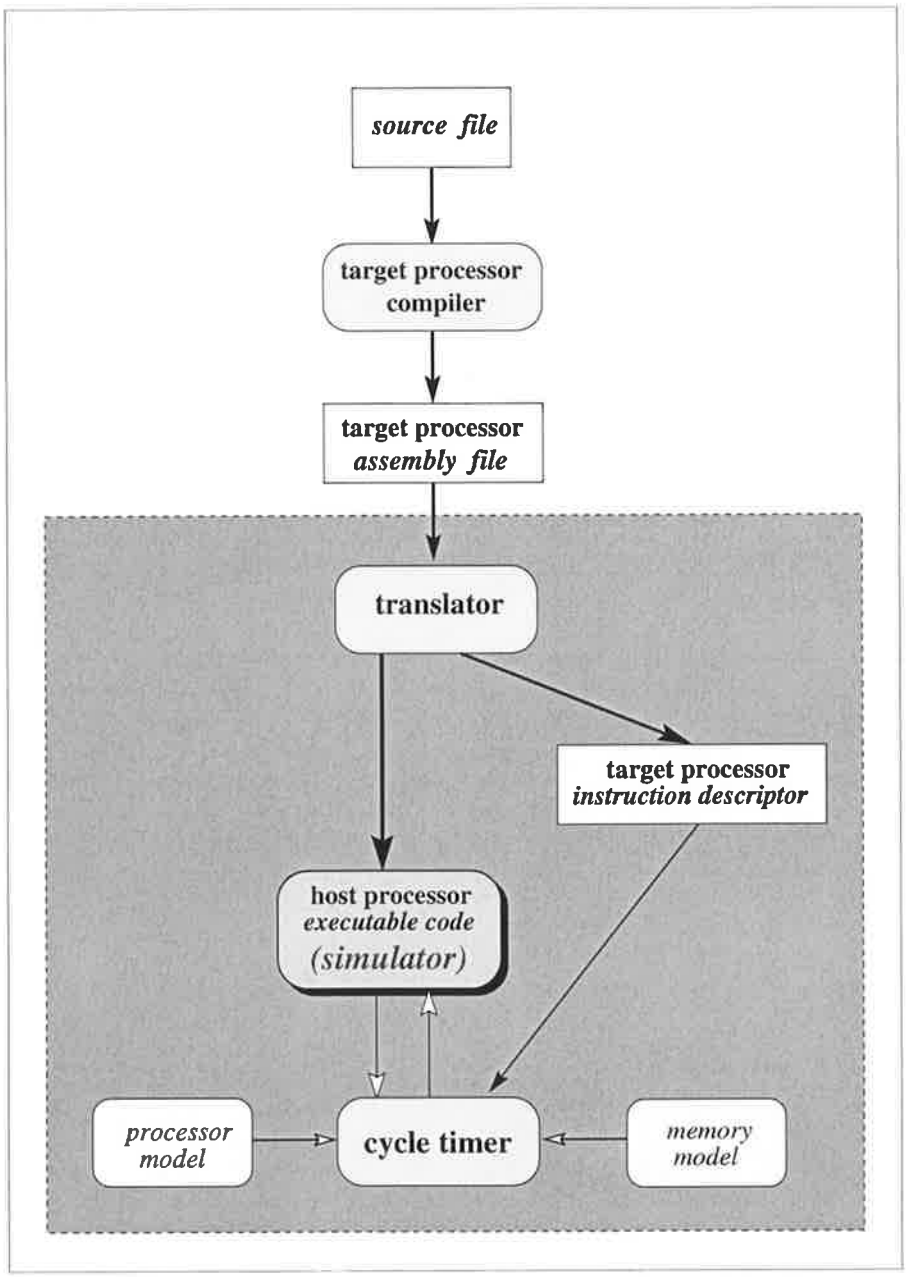


Figure 5.2: Basic diagram of an execution-driven simulator.

resources are used by the program. Then, cycle by cycle simulation is performed using the output of the translator with alternative memory and processor models.

A third approach which is simpler and less accurate than the two previous methods can be used to estimate the execution time of an application based on the static code schedules weighted by dynamic execution frequencies obtained from profiling. It was reported that the accuracy of results is better than 95% in comparison with simulation results (generated by a trace-driven simulator) assuming perfect caches [Bringmann, 1995, August et al., 1997].

5.2 Experimental Results

In this section, some experiments performed to evaluate the capability of the EVA compiler at extracting ILP are described and results are presented. These include measurement of speedup and code size changes for the basic structure of the compiler.

5.2.1 Methodology

To evaluate the performance improvements quickly, we use the third method described in section 5.1, which is based on execution profile information and static code schedules. Perfect cache models are assumed for both data and instruction caches. The possible effects of non-prefect cache on performance are discussed in section 5.2.4. More accurate results with a detailed target processor model and non-perfect cache model can be generated by a complex simulator (such as an execution-driven simulator) but, this requires a long time to implement. As mentioned before, the accuracy of results in our case is more than 95% in comparison with results generated with a simulator, assuming perfect caches. This is because, a VLIW processor can be considered similar to the in-order issue processor used in [Bringmann, 1995] and [August et al., 1997]. However, we use different sets of profile information for profile-based compilation passes and performance estimation to increase the accuracy. We used train inputs in the former and reference inputs in the latter for the SPEC95 benchmarks.

Our work mainly considers performance improvement for general-purpose applications. Thus, as is common practice in architecture research, SPEC95 integer benchmarks and some common Unix utility programs are used as the workload in our experiments. Tables 5.1 and 5.2 indicate the benchmark set and the inputs used for profiling. The results are generated

Benchmark	Group	Description
099.go	SPEC95 - INT	The game of GO
129.compress	SPEC95 - INT	File compressor
130.li	SPEC95 - INT	XLISP Interpreter
132.ijpeg	SPEC95 - INT	JPEG encoder
134.perl	SPEC95 - INT	Interpreted Programming Language
147.vortex	SPEC95 - INT	Object-oriented database
cmp	Unix utility	File comparison
grep	Unix utility	Pattern search
wc	Unix utility	Word count
yacc	Unix utility	Parser generator

Table 5.1: Benchmark programs used in our experiments.

with three different machine models of the EVA VLIW architecture (chapter 3), which are shown in Figure 5.3.

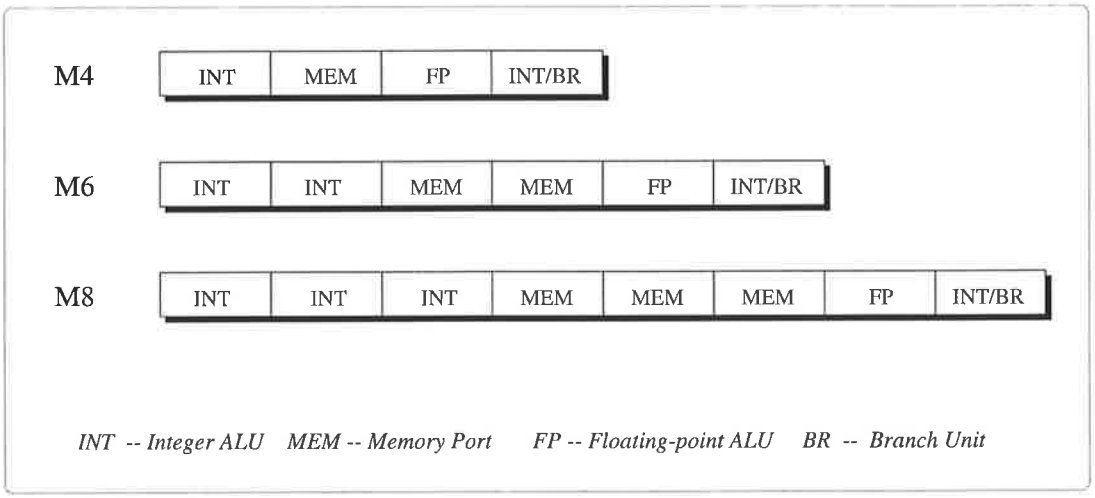


Figure 5.3: Issue slot configuration of three machine models.

The number of functional units and the arrangement of the issue slots in the VLIW instruction word for each model are based on the reports of average dynamic frequency of each operation type in typical workloads [Jourdan et al., 1995]. Table 5.3 indicates the assumed latency of operations, which are similar to MIPS R10000 processor [Yeager, 1996].

The platform used in our experiment is a multiprocessor supercomputer composed of 20

Benchmark	Profiling Data Set
099.go	SPEC train input
129.compress	SPEC train input
130.li	SPEC train input
132.ijpeg	SPEC train input
134.perl	SPEC train input (prime)
147.vortex	SPEC train input
cmp	g23.c with g25.c (from 099.go)
grep	string "if" from g23.c (099.go)
wc	Postscript file of chapter 3
yacc	jv-exp.y from GNU gdb-4.18

Table 5.2: Profiling data set for benchmark programs.

Operation Group	Latency
Integer ALU	1
Load	2
Store	1
Int-Multiply	3
Int-Divide	9
Branch	1
FP ALU	3
FP Multiply	3
FP Divide (SP)	9
FP Divide (DP)	14

Table 5.3: Latency of operations.

64-bit 200MHz MIPS R10000 processors connected by a 1.2 Gbyte/sec system bus with Irix 6.2 operating system.

5.2.2 Results

To evaluate the effectiveness of predicated code in the performance of the EVA compiler, relative dynamic frequencies of different groups of operations and code size increase were measured for the M8 machine model. Tables 5.4 and 5.5 indicate the percentage of each group of operations for basic block and hyperblock scheduling respectively. The hyperblock scheduling is based on the heuristics presented in [Mahlke, 1996] which we call HB1. A new algorithm called HB2 is presented in chapter 6 but in this section, results for HB1 are presented as this scheme is used as the basic hyperblock scheduling algorithm in the EVA compiler.

The results in Table 5.5 are generated with speculative code motion (both conventional code motion above a conditional branch and predicate promotion.) The impact of hyperblock scheduling on the percentage of different groups of operations can be explained as follows. It is clear that the dynamic number of branches should be decreased as some branches would be converted to predicate define operations (which are considered as IALU operations). There is an increase in the percentage of IALU operations as expected. The scheduler tries to schedule those operations which are on the critical paths and other operations that depend on them as early as possible. For this reason, load operations are often candidates for speculation when a mechanism is provided to avoid unwanted exceptions. In the EVA, non-trapping version of loads are used for speculative loads.

Table 5.6 shows the relative code size after hyperblock scheduling. Combining multiple execution paths into a larger block, tail duplication and employing speculative code motion potentially increase the code size. If most operations in a hyperblock are executed when control is transfered to the beginning of the hyperblock, the impact of tail duplication on the dynamic number of operations is less as they are not executed often. Speculation causes some redundant operations to be executed when their effects are nullified later. The number of operations can be decreased as more optimisation opportunities (such as global variable migration and loop invariant code elimination) are enabled for hyperblocks.

Figure 5.4 illustrates the speedup achieved for hyperblock scheduling (HB1) with respect to basic block scheduling for the three machine models. Speedup is calculated as the ratio

Benchmark	load	store	IALU	branch	FP
099.go	20.87	6.89	58.68	13.56	0
129.compress	21.36	14.59	46.71	16.74	0.6
130.li	31.23	16.81	28.08	23.88	0
132.ijpeg	27.35	3.63	58.74	10.28	0
134.perl	25.44	16.35	47.68	10.24	0.29
147.vortex	21.36	14.27	54.55	9.8	0.02
cmp	48.28	21.4	13.63	16.69	0
grep	27.13	9.48	12.51	50.88	0
wc	30.46	19.22	26.01	24.31	0
yacc	29.81	3.64	36.67	29.88	0
Average	28.32	12.62	38.32	20.62	0.91

Table 5.4: Dynamic percentage of each group of operations in the benchmarks with basic block scheduling for the M8 machine model.

Benchmark	load	store	IALU	branch	FP
099.go	17.32	8.01	64.86	9.81	0
129.compress	15.62	12.32	61.42	10.1	0.54
130.li	23.13	14.64	44.02	18.21	0
132.ijpeg	20.59	4.12	68.04	7.25	0
134.perl	20.78	17.02	53.75	8.35	0.1
147.vortex	25.42	9.61	58.64	6.33	0
cmp	27.86	1.02	61.79	9.33	0
grep	22.43	9.11	47.29	21.17	0
wc	10.2	1.31	75.77	12.72	0
yacc	22.33	3.75	58.29	17.63	0
Average	20.56	8.09	59.38	12.09	0.64

Table 5.5: Dynamic percentage of each group of operations in the benchmarks with hyper-block scheduling (HB1) for the M8 machine model.

Benchmark	Code size increase
099.go	1.53
129.compress	1.68
130.li	1.18
132.ijpeg	1.41
134.perl	1.29
147.vortex	1.13
cmp	0.41
grep	0.83
wc	1.85
yacc	1.35
Average	1.266

Table 5.6: Code size for hyperblock scheduling relative to basic block scheduling for the M8 machine model.

of the number of cycles for basic block scheduling to the number of cycles for hyperblock scheduling.

As expected, the performance is increased for wider-issue processors. For some benchmarks such as *cmp*, *grep*, and *wc* larger speedup achieved. This is due to high reduction in the dynamic number of conditional branches in these benchmarks. In this way, hyperblock scheduling is more effective in performance increase for these benchmarks.

5.2.3 Results for modified static priority calculation algorithm

Calculation of the priority of operations for scheduling affects the achieved performance. The results presented in the previous section are based on Bringmann's algorithm to calculate the priority of operations described in section 4.5.3. We proposed a modified version of that algorithm (the modified static priority calculation algorithm) to take resource constraints into account to prevent delaying of branches (exits) unnecessarily.

Figure 5.5 indicates the relative improvement in speedup by applying this algorithm. As the number of branches (exits) are reduced in hyperblocks, the number of operations between branches increases. Also, more operations contend for processor resources. Therefore, it is expected that including some aspects of resource usage awareness in calculating the priority

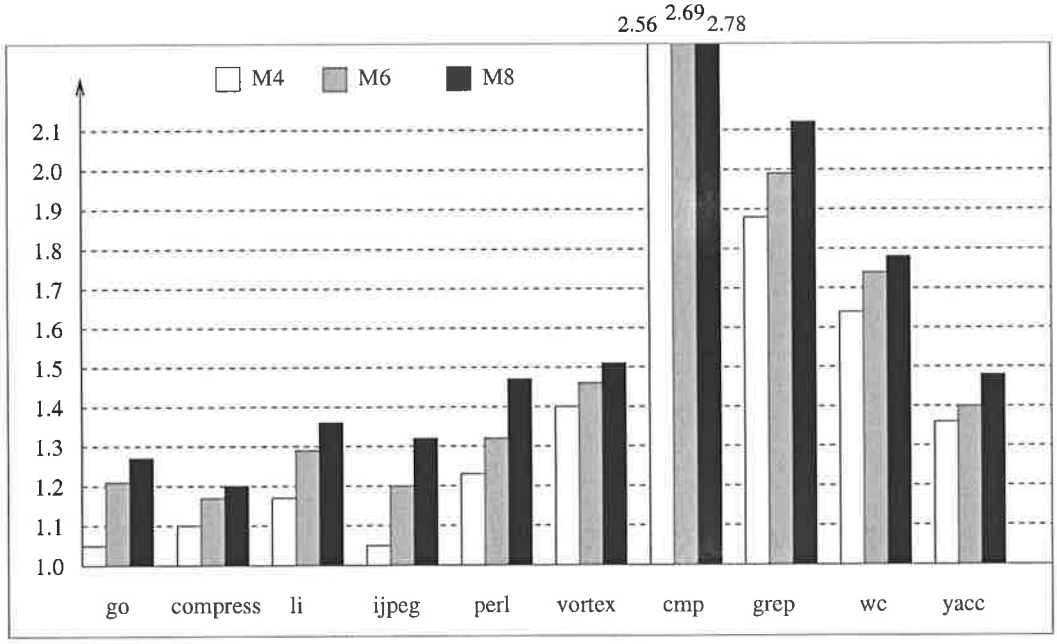


Figure 5.4: Speedup of hyperblock scheduling with respect to basic block scheduling in the EVA compiler for different machine models.

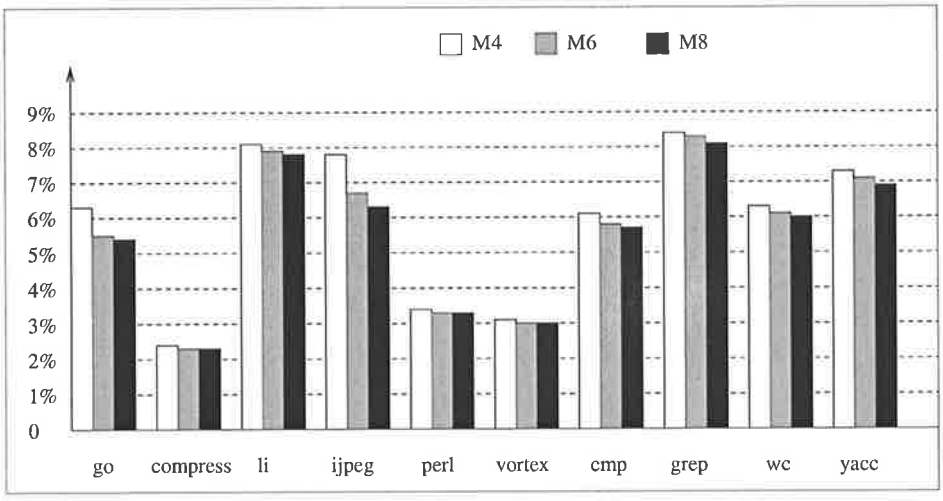


Figure 5.5: Speedup of the modified static priority calculation algorithm with respect to the original algorithm.

of operations would result in a better schedule. The results indicate that this method is more effective when ILP is limited because of fewer processor resources.

5.2.4 Effects of Non-perfect Cache

The impact of non-perfect caches is greater for wider-issue processors. Cache miss penalty is increased as the issue rate of the processor increases, resulting in performance loss. This is because, for the same finite caches, a larger part of the total execution cycles are wasted due to cache misses.

In order to evaluate the impact of non-perfect caches, we use the following relation to adjust the cycle count obtained for the perfect cache.

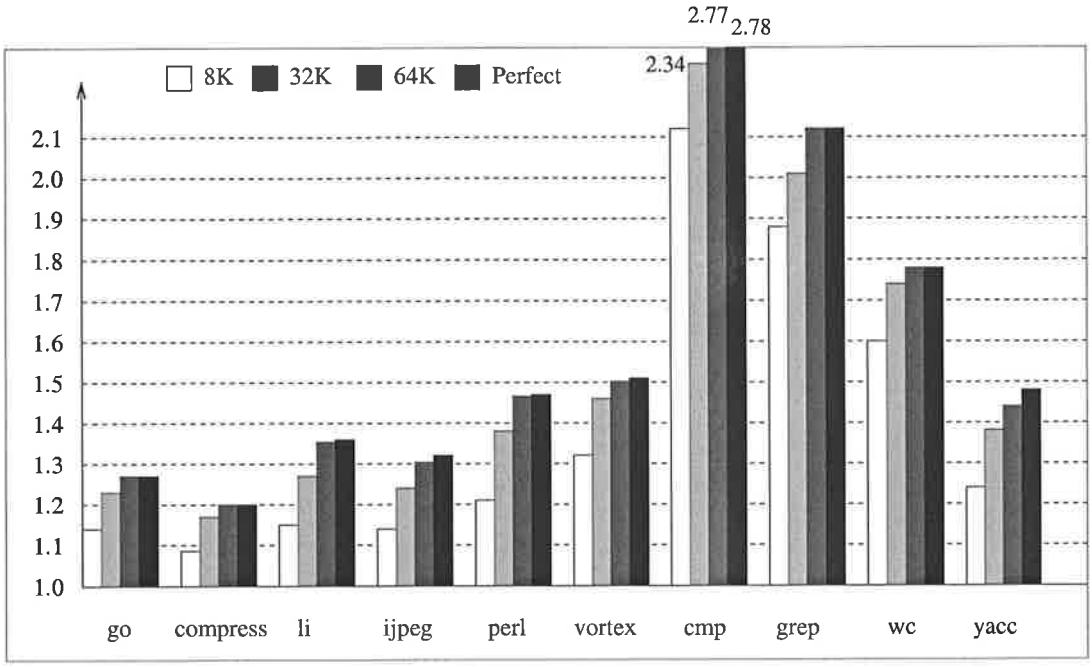
$$cycle\_count_{NP} = (c_R * m_R + c_W * m_W) * num\_instrs + cycle\_count_P \quad (5.1)$$

$cycle\_count_P$ represents the cycle count obtained for the perfect cache. m_R and m_W indicate miss rates for cache read and write. c_R and c_W are miss penalties for read and write operations respectively. *num_instrs* is the dynamic number of operations.

Equation 5.1 is an approximation used to obtain cycle count in the case of using non-perfect caches. For example, in an out-of-order superscalar processor, some part of miss penalty cycles are covered by executing other operations which are not dependent on the operation resulted in the cache miss. In our case, which is a VLIW machine, equation 5.1 is a reasonable approximation.

To generate the results, we use *mlcache* [Tam et al., 1997], which is a multi-lateral cache simulator to get miss rates. Instruction traces are generated in the appropriate format and fed into the *mlcache*. Figure 5.6 shows the impact of finite instruction cache on the performance of the EVA compiler. A perfect data cache is assumed to generate these results.

Speedup is decreased for all benchmarks with a small cache. The non-perfect instruction cache effects for hyperblock scheduling can be mainly considered due to code size increase in a block and changes in the code layout. The number of operations between two branches (which makes a code block for execution) increases after hyperblock formation through inclusion of multiple execution paths and removal of their branches with if-conversion. Also, speculative operations increase the dynamic number of operations. When the number of operations in a block are higher than can be handled by the underlying finite instruction cache,



All caches are direct-mapped.

Figure 5.6: Speedup of hyperblock scheduling with respect to basic block scheduling in the EVA compiler for different instruction caches for the M8 machine model. A perfect data cache is used.

capacity misses [Hennessy and Patterson, 1996] increase. If the cache is direct mapped, additional collision misses occur. So, it is expected that finite instruction caches have negative impact on the performance of hyperblock scheduling. For most benchmarks, the performance for a 64K direct mapped instruction cache is very close to the case with a perfect cache.

Table 5.7 indicates the relative normalised speedup with finite data caches. The speedup for a perfect data cache is assumed 1. A perfect instruction cache is used to generate these results. Figure 5.7 shows the speedup for this case.

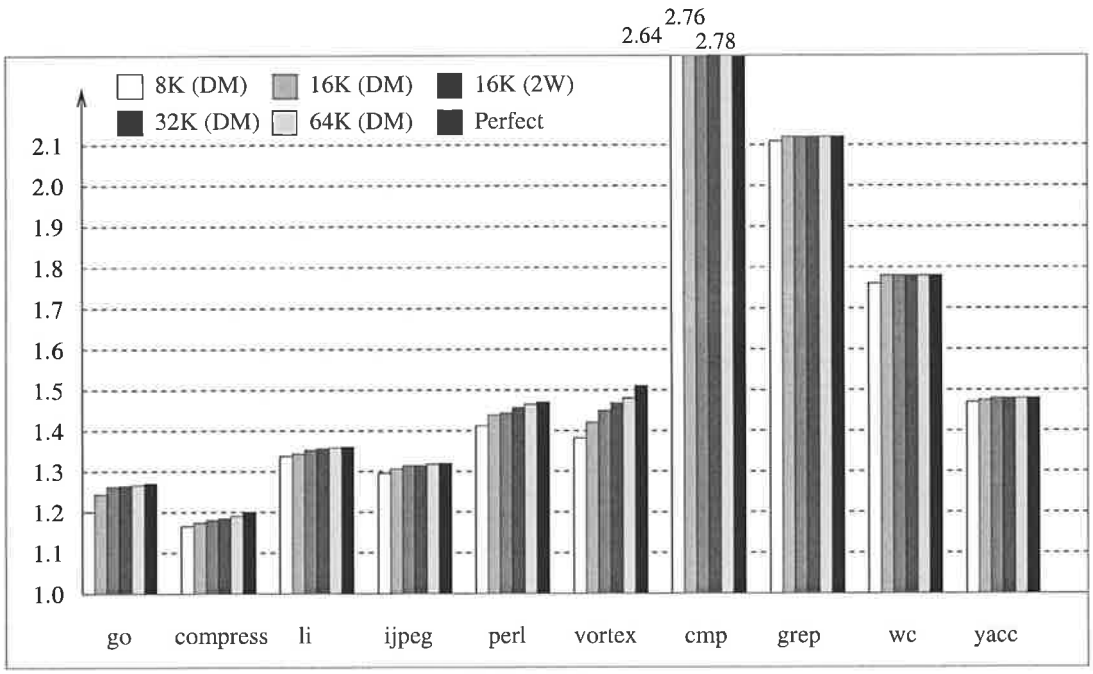
For finite data caches, speculative loads can be a cause of cache misses. Due to the higher scheduling priority of loads (as a sequence of operations are often dependent on them), these operations from different execution paths after hyperblock scheduling tend to be clustered, and in this way data from different parts of memory are requested and more *conflict misses* occur. Results indicate that the working data set of benchmarks fit into a 64K data cache, resulting in little performance loss with respect to a perfect data cache.

5.3 Summary

In this chapter, first different approaches in performance evaluation in ILP processing were discussed. In order to generate the experimental results efficiently within an acceptable accuracy, we used execution profile information and static code schedules to calculate the number of execution cycles. In addition, the effects of non-perfect caches were evaluated. Experimental results show effectiveness of hyperblock scheduling in the EVA compiler over basic block scheduling. Also, significant improvement due to our modified Bringmann's algorithm was achieved. In addition, experimental results indicate that 64K data and instruction caches produce performance almost equivalent to a perfect memory system, justifying our use of a perfect memory system in earlier results.

Benchmark	8K(DM)	16K(DM)	16K(2W)	32K(DM)	64K(DM)
099.go	0.741	0.9	0.97	0.981	0.989
129.compress	0.826	0.868	0.91	0.921	0.952
130.li	0.939	0.953	0.979	0.989	0.996
132.ijpeg	0.922	0.958	0.981	0.981	0.994
134.perl	0.877	0.934	0.943	0.971	0.99
147.vortex	0.749	0.824	0.881	0.918	0.942
cmp	0.952	0.99	1	1	1
grep	0.989	0.998	1	1	1
wc	0.978	1	1	1	1
yacc	0.979	0.991	1	1	1

Table 5.7: Relative normalised speedup with finite data caches with respect to a perfect data cache for the M8 machine model. A perfect instruction cache is assumed.



DM and 2W represent direct-mapped and 2-way set associative data caches.

Figure 5.7: Speedup of hyperblock scheduling with respect to basic block scheduling in the EVA compiler for the M8 machine model and different data caches. A perfect instruction cache is assumed.

Chapter 6

Partial Path Selection for Hyperblock Formation

It has been shown that hyperblock scheduling in some cases suffers from performance loss in comparison with superblock scheduling [Mahlke, 1996, August et al., 1997]. The most common cause of poor hyperblock formation is excessive resource consumption. Paths which are combined together to construct the hyperblock share processor resources and when the amount of resource usage is not well estimated, execution time for overlapped paths may increase.

In this chapter, an algorithm is presented for hyperblock formation which considers an estimation of the resource usage to include partial paths when beneficial.

6.1 Issues in Hyperblock Formation

Two issues should be considered in order to generate predicated code at compile-time efficiently. These are related to when the if-conversion should be applied and what is to be if-converted [August et al., 1997]. Making proper decision with regard to these issues is a complex process. Figure 6.1 shows two possible strategies for a compiler which generates predicated code.

In Figure 6.1 (a), if-conversion is applied early in the compilation process. This provides more opportunities for code improvement, especially employing predicate-sensitive optimisations. In this way, the cost of some optimisations related to conditional branches (such as branch combining [Mahlke et al., 1995] and control height reduction [Schlansker et al.,

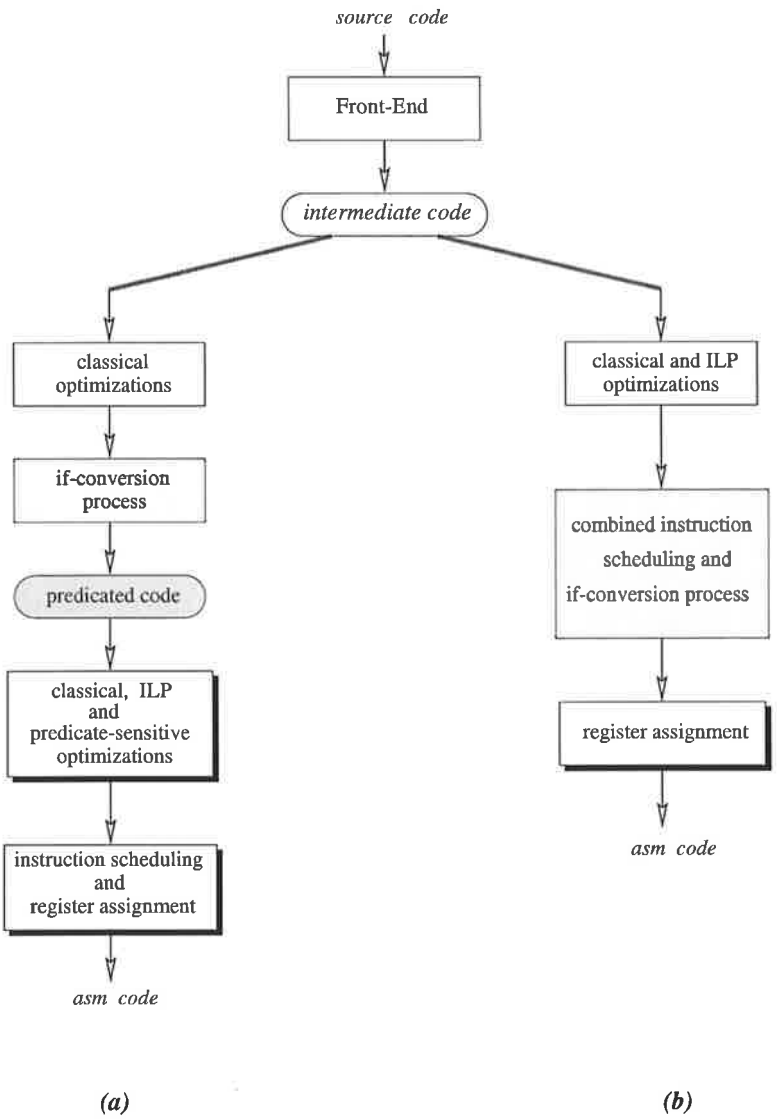


Figure 6.1: Two possible strategies in a compiler to generate predicated code. (a) If-conversion before ILP optimisation and instruction scheduling. (b) Combined if-conversion and instruction scheduling.

1994]) are also avoided [August et al., 1997]. Figure 6.1 (b) indicates another approach, in which if-conversion and instruction scheduling are combined into one phase. At this stage, more information about the target processor characteristics (such as resource usage) is available to the if-converter and this may result in higher performance. However, applying if-conversion at the time of scheduling increases the complexity of scheduler. Also, it seems not to be practical to apply predicate-sensitive optimisation techniques at this time due to additional complexity, so the opportunity for these optimisations is lost.

To decide on what to be if-converted, previous studies especially for general-purpose applications indicated that selective if-conversion should be employed in order to achieve higher performance [Mahlke et al., 1992b].

As discussed in [August et al., 1997], optimisation performed after hyperblock formation may change the code characteristics so that decisions made at the time of hyperblock formation may no longer result in an efficient hyperblock at execution time. Some effects of these optimisations may be estimated in advance. However, it is not possible to estimate the impact of all code transformations applied on the hyperblock. For example, optimisations related to dependency height reduction have different opportunities before and after if-conversion.

6.2 Related Work

Basic blocks in a region are selected based on a heuristic which indicates their usefulness when included in the hyperblock. In the original work on hyperblock scheduling from the IMPACT compiler group [Mahlke et al., 1992b], three features are considered in order to calculate the usefulness of a basic block for this purpose. These are the size of the block, its execution frequency and the number of hazardous operations (which limit the applicability of optimisation and scheduling). Smaller and more frequently executed blocks with fewer hazardous operations have higher priority for inclusion in the hyperblock. Dependency height is added to the heuristic in [Mahlke, 1996] to give higher priority to paths with lower dependency height. The overall dependency height of a hyperblock is the maximum of dependency heights of all included paths, so the execution of a hyperblock is not completed until the path with maximum dependency height completes. This has a negative impact on the other included paths when their execution frequency is higher.

Figure 6.2 indicates the basic steps of the heuristic. The priority of each path is propor-

tional to its execution probability, number of hazardous operations, and normalised factors indicating the relative number of operations in the path and the relative dependency height with regard to the highest priority path. The resource estimation is simple and is proportional to the target processor issue-width and the dependency height of the path. Including the basic blocks of a path is considered beneficial if the path does not have a large relative dependence height with respect to other included paths.

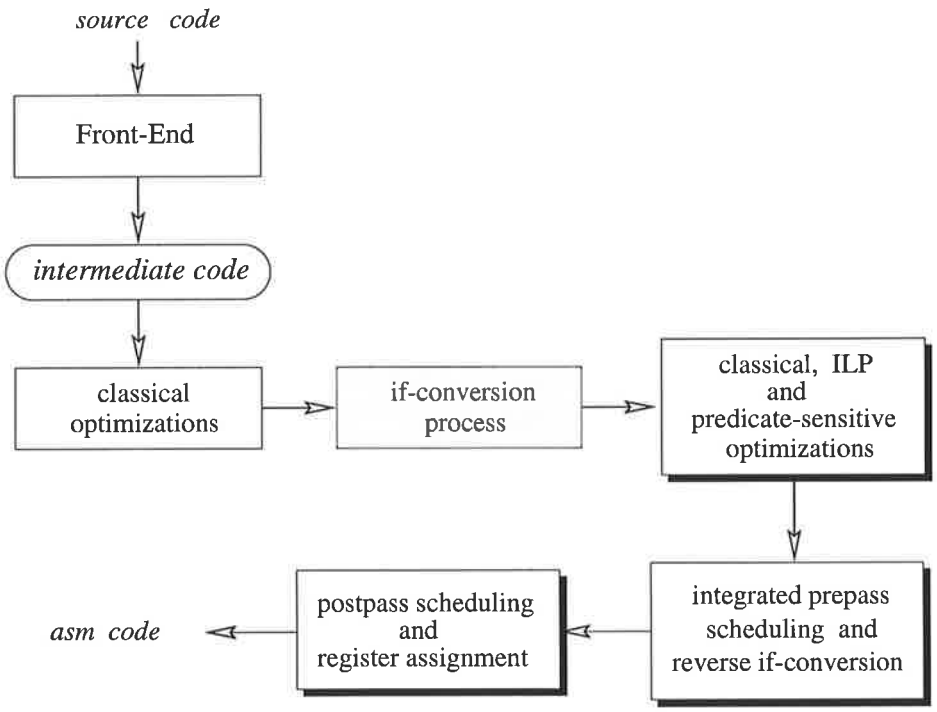
```
// Input:  candidate region
// Output:  selected_blocks

selected_blocks is empty
enumerate all paths in region
calculate priority of each path
sort paths based on priority
add all basic blocks of the highest priority path to the selected_blocks
 for each execution path <path>
        estimate resource usage for <path>
        if resource available then
                if including <path> is beneficial then
                        add all basic blocks of <path> to the selected_blocks
```

Figure 6.2: Original heuristic for hyperblock formation.

Results presented in [Mahlke, 1996] indicated effectiveness of the above heuristic for a target processor with regular resources. When this is not the case, performance loss occurs in comparison with superblock scheduling [August et al., 1997]. To solve this problem, August and his colleagues proposed a technique called *partial reverse if-conversion* [August et al., 1997]. Figure 6.3 indicates the compilation phases proposed in this approach. To generate predicated code, aggressive if-conversion is performed in the early phases of the compilation process. Some of the predication is converted back to control flow at scheduling time through reverse if-conversion [Warter et al., 1993]. Partial reverse if-conversion is integrated with the prepass scheduling.

To apply reverse if-conversion, execution paths in the predicated code should be identified. A structure called a *predicate flow graph* (PFG) is used for this purpose. A PFG is a



Blocks with shadow represent compile passes which work on predicated code.

Figure 6.3: Compiler structure to adjust the amount of generated predicated code based on resource usage through reverse if-conversion.

control flow graph (CFG) in which predicate execution paths are also represented. Execution paths are determined having regard to the relations among the predicates in the region. These can be provided through the predicated code analysis techniques discussed in [Johnson and Schlansker, 1996, Gillies et al., 1996].

To generate the PFG, a path is created at a predicate definition for both 'true' and 'false' cases. The number of paths in the PFG is proportional to the number of independent predicates with overlapped live ranges. Therefore, a complete PFG has a large number of paths which might not be practical to implement in some cases. For this reason, it is useful to create only those paths of the PFG that are required for the specific analysis. For example, in the approach discussed in [August et al., 1997], the PFG is generated only for the predicate considered for reverse if-conversion.

A process called *predicate partial dead code removal* is applied to eliminate dead code, like other types of partial dead code elimination [Knoop et al., 1994].

Partial reverse if-conversion results in separate code segments: the code before the reverse if-converting branch, the code removed from the hyperblock through this process, and the remaining code below the reverse if-converting branch. Using the generated PFG, operations are inserted in the proper code segment. An operation which exists in both paths is placed in both segments after the reverse if-converting branch. Because of the duplicated code generated in this process, the code size may increase significantly. A simple technique was discussed in [August et al., 1997] to reduce this problem in some cases.

Predicates to be if-converted are considered from top to bottom in the scheduled order of their predicate define operations. Three schedules are considered for each candidate predicate to evaluate possible benefits of its reverse if-conversion. These are the schedule without reverse if-conversion, the schedule of the new hyperblock after reverse if-conversion, and the schedule of the removed code which is put in a separate block. If the number of cycles of the scheduled code without reverse if-conversion is greater than the combined number of cycles for removed code, the new hyperblock, and the effect of miss prediction of the reverse if-converting branch (that is, miss prediction rate * miss penalty) then, the reverse if-conversion is beneficial. Generating three schedules to make a decision for reverse if-conversion increases the compile time. Some methods to re-use previous schedules are discussed in [August et al., 1997] in order to minimise the increase in the compile time.

6.3 A New Approach

In order to construct a more efficient hyperblock, we improve the heuristic of Figure 6.2 through a better resource estimation for non-regular processors. The complete resource usage pattern in a processor can be known at the time of scheduling, so if the if-conversion process is performed before scheduling, resource usage may only be estimated.

Figure 6.4 indicates our algorithm for block selection to construct a hyperblock. First local scheduling (or basic block scheduling) is performed on the selected region in order to collect the resource usage information of a basic block. Later, this information is used to assess each basic block (BB) in an execution path to find out if it would be profitably included in the hyperblock. This estimation cannot reveal accurate resource usage, as speculation and predication applied later would change the sequence of operations. However, since the branches cannot be reordered in our scheduler, and inclusion of each basic block is based on its control flow predecessor which was selected previously, this method may give a reasonable estimation.

A parameter which we call *hb_factor* is calculated for each BB to indicate its value for inclusion in the hyperblock. It is proportional to the execution probability of the BB and the number of hazardous conditions. In the current implementation, we use hazard factor of 1, when no such conditions exist. Otherwise, its value is 0.3. This indicates the relative importance of hazard conditions, which should be considered in the BB selection.

The algorithm uses three lists. *BB_queue* holds basic blocks that are previously selected for the hyperblock and their control flow successors, which are candidates for selection. These successor blocks are added to *succ_list*. Selected basic blocks are inserted in the *selected_BB_list*.

The algorithm works as follows. First the head BB of the region is selected. Each selected BB is placed into *BB_queue* and *selected_BB_list*. Unprocessed successors of each BB in the *BB_queue* are placed into *succ_list* and the BB is popped from *BB_queue*. *succ_list* is sorted based on *hb_factor*. Each BB in *succ_list* is examined to check if it is beneficial to include it in the hyperblock. The resource usage pattern of this BB is compared with the record of used resources of previously selected basic blocks. This information is kept in a structure called *res_usage_record*. Adding a new BB to the partially selected path may add a delay on the completion of the previously included paths. This can be due to the changing of the scheduling priorities at the time of scheduling. Therefore, a parameter called *delay_cycle_count* is

```
// Input:  Candidate region of basic blocks
// Output:  list of selected basic blocks for hyperblock formation

Perform preliminary local scheduling for each basic block (BB) in the region
Calculate hb_factor for each BB in the region using the following relation:
                hb_factor = BB_exec_prob  *  hazard_factor
selected_BB_list, BB_queue, and succ_list are empty
add the first BB (head BB) of the region to BB_queue
add head BB to  selected_BB_list
while BB_queue not empty
      BB<i> =  top node of BB_queue
      add unprocessed successor nodes of  BB<i> to succ_list
      sort succ_list based on hb_factor
      for each BB<x> in succ_list
            mark BB<x> as processed
            selected_BB = NULL
            compare BB resource usage with res_usage_record
            if no conflict with regard to the specified delay_cycle_count then
                  selected_BB = BB<x>
            else if partial of BB<x> is beneficial then
                  move extra operations of BB<x> to a new BB (BB<y>)
                  adjust appropriate control edges of BB<x> to BB<y>
                  selected_BB = BB<x>
            if selected_BB then
                  add selected_BB to selected_BB_list
                  add selected_BB to BB_queue
                  update res_usage_record
```

Figure 6.4: The block selection algorithm for hyperblock formation.

used to consider this delay. *delay_cycle_count* indicates the maximum acceptable delay imposed on the completion of other included paths. For example, if *delay_cycle_count* is set to 1, this means that the candidate BB can be included into the hyperblock if the completion of other included paths is estimated to be delayed at most one cycle.

If inclusion of the candidate BB does not conflict within the specified acceptable delay then this BB is selected and added to the selected_list. In some cases, inclusion of only a part of the BB can prevent additional delay. If *hb_factor* of the BB is greater than a threshold value (0.4 in our implementation), it is considered for partial inclusion . This is performed by creating a new basic block and moving the additional operations from the selected BB to this new BB. Control edges are adjusted accordingly. The selected BB is inserted in *selected_list* and *BB_queue*. This process ends when there is no basic block in *BB_queue*.

Figures 6.5 and 6.6 show an example on how the inclusion of a part of basic block can be useful.

In Figure 6.5, after selection of BB1, both BB2 and BB3 are candidates for selection. However, as indicated in their schedule, inclusion of both basic blocks results in a resource conflict (because of operations I_5 and I_{12}). Therefore, after selection of BB2 (as its execution probability is higher), a part of BB3 is selected and those operations that cause resource conflict are transferred out of the hyperblock as shown in Figure 6.6.

6.4 Experimental Results

The experimental results are generated based on the method described in section 5.2.1. Our approach attempts to include partial paths, when beneficial, with relatively simple resource usage estimation. To evaluate the effectiveness of this approach, results are presented for three different machine models. Speedups are relative to basic block scheduling scheme. To measure the performance improvement, speedup for superblocks (SB), original hyperblock scheme (HB1) and our approach (HB2) are calculated.

Figures 6.7, 6.8, and 6.9 show the speedup results for issue-4, issue-6 and issue-8 machine models respectively.

As the results indicate, in some cases the performance is less for hyperblock scheduling (HB1) compared to superblock scheduling technique. This is due to over-saturation of the processor resources The performance loss is greater for lower issue processors.

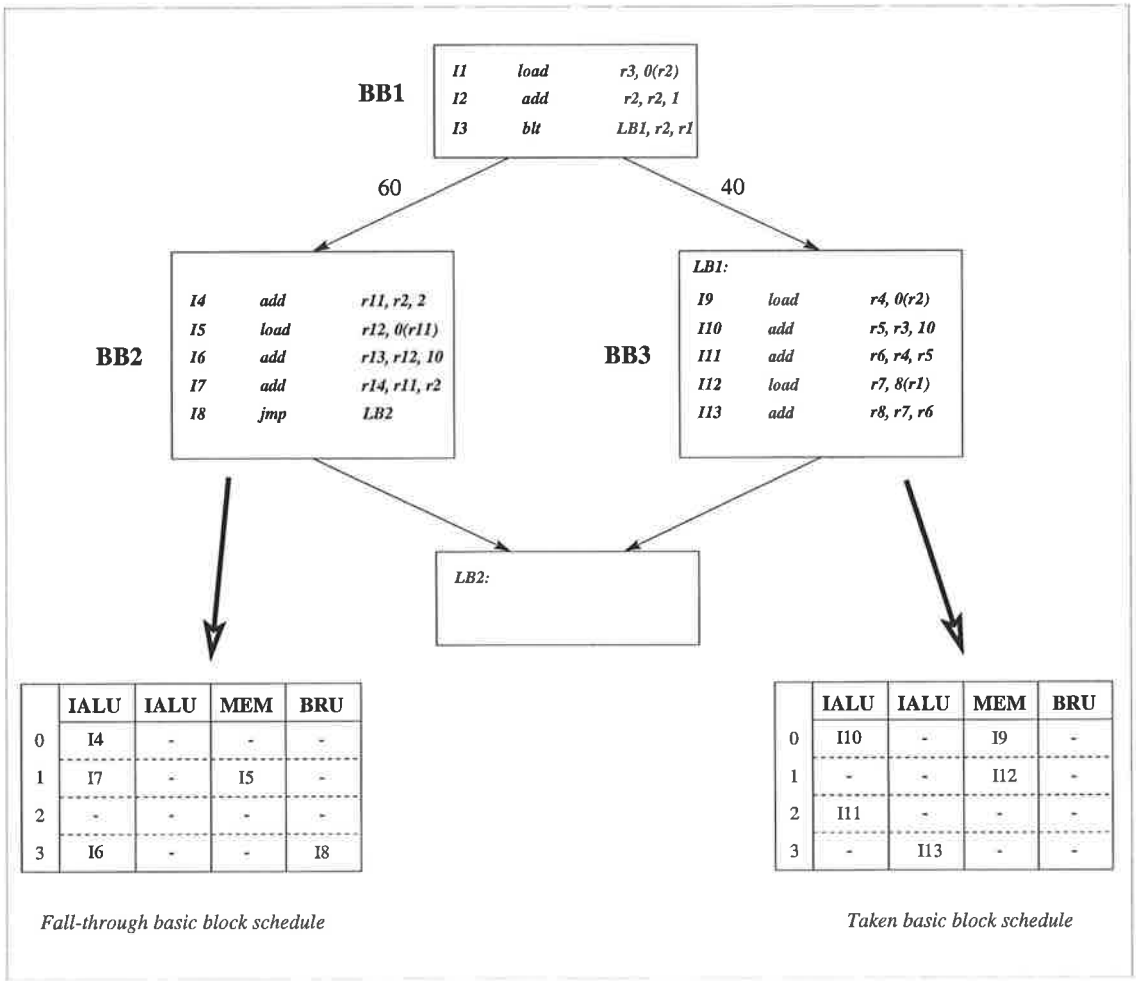


Figure 6.5: Example of a candidate region for hyperblock formation with local schedules for the candidate basic blocks.

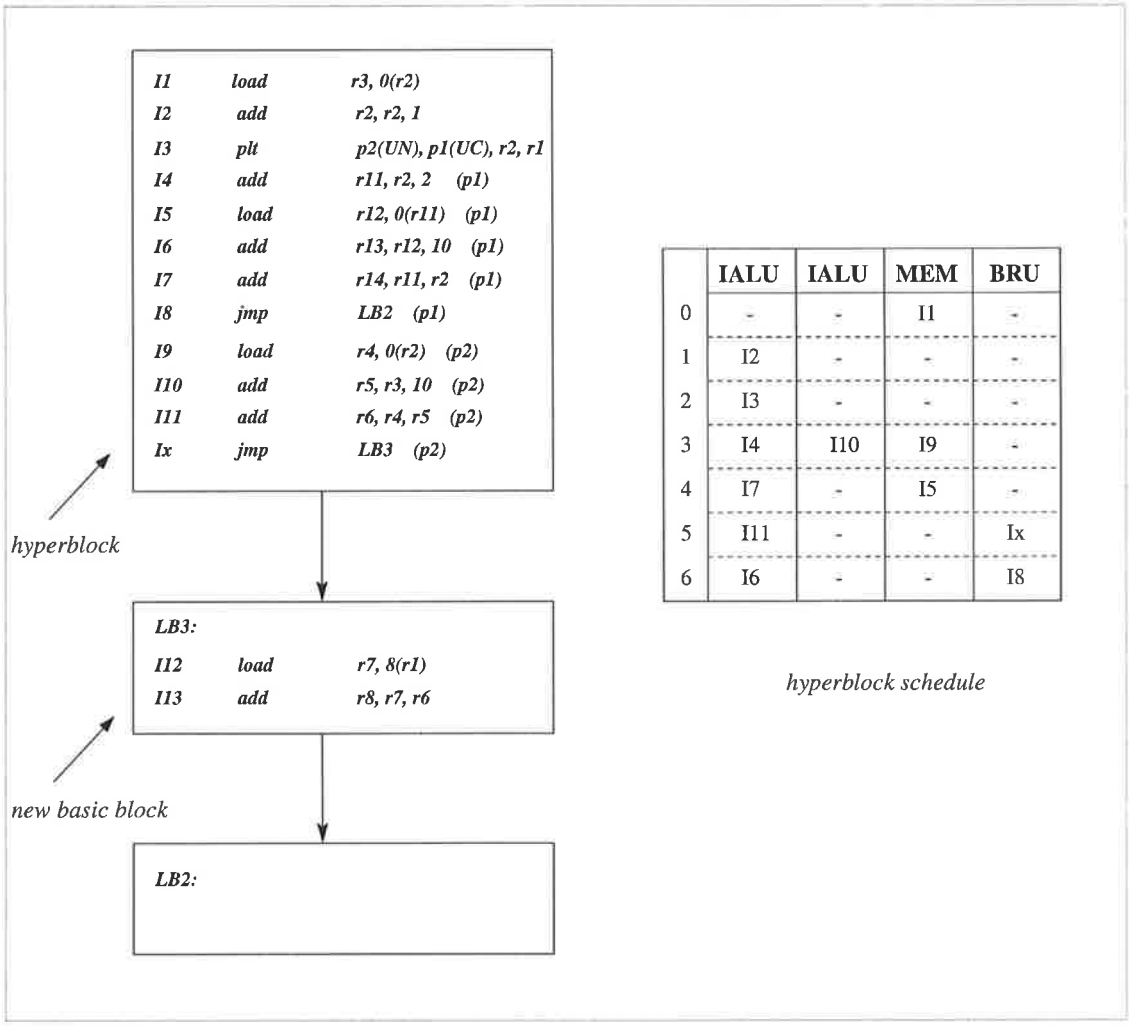


Figure 6.6: Constructed hyperblock based on our heuristic.

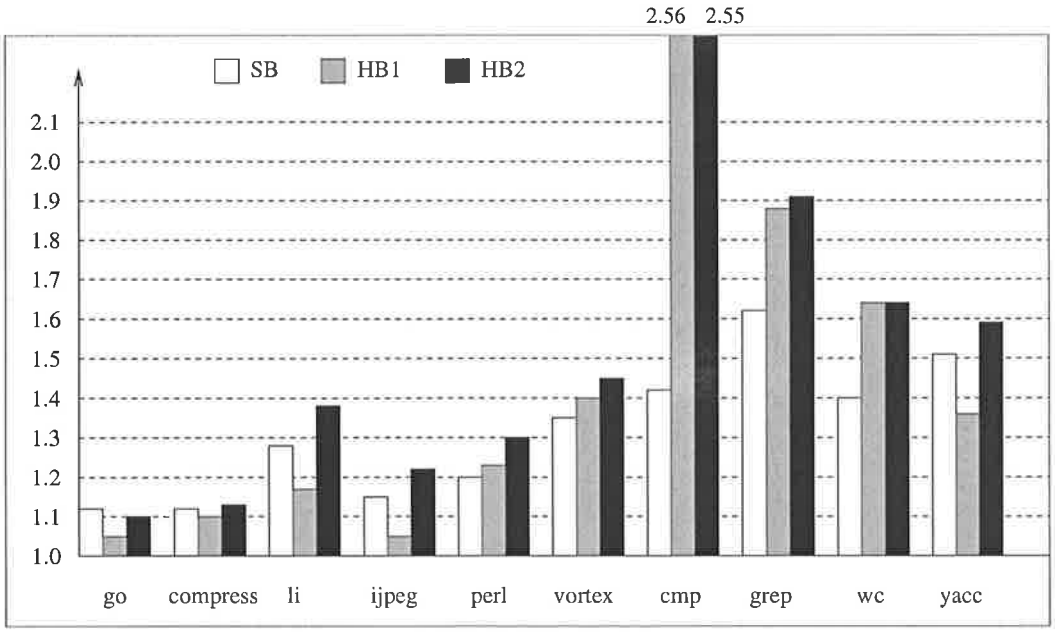


Figure 6.7: Speedup of superblock (SB), traditional hyperblock (HB1), and new hyperblock (HB2) scheduling methods with respect to basic block scheduling for the M4 machine model.

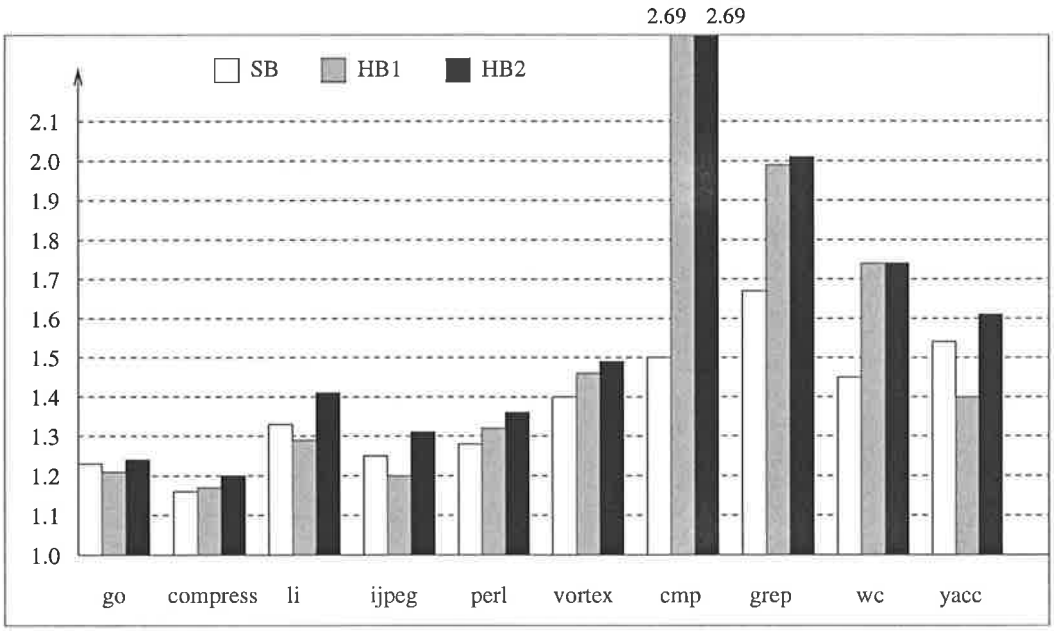


Figure 6.8: Speedup of superblock (SB), traditional hyperblock (HB1), and new hyperblock (HB2) scheduling methods with respect to basic block scheduling for the M6 machine model.

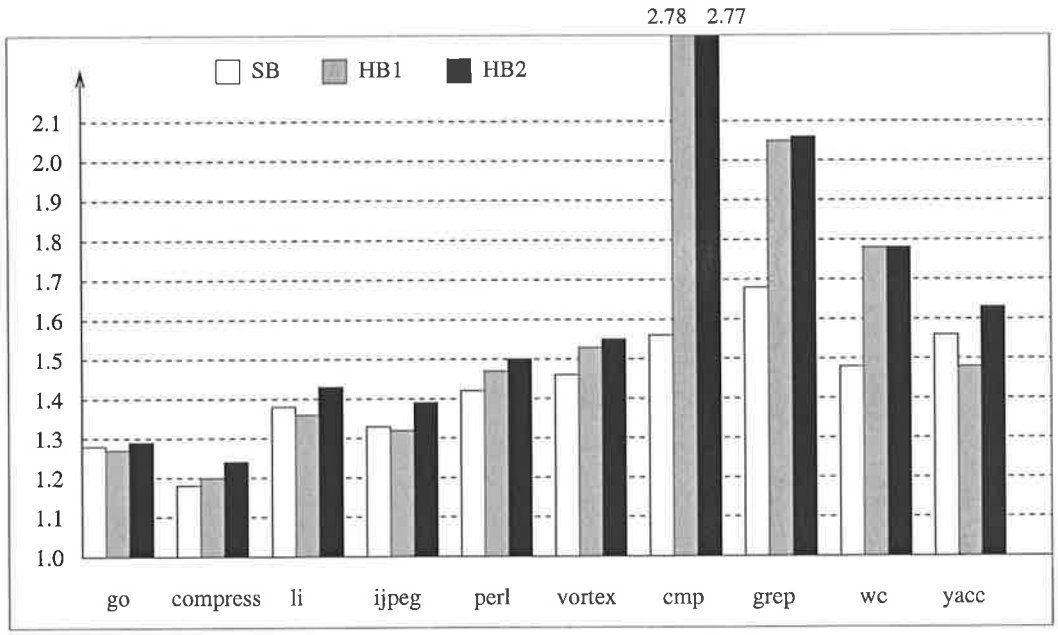


Figure 6.9: Speedup of superblock (SB), traditional hyperblock (HB1), and new hyperblock (HB2) scheduling methods with respect to basic block scheduling for the M8 machine model.

Applying our heuristic for block selection in hyperblock scheduling improves the performance. For those benchmarks like *cmp*, and *wc*, there is no improvement in performance. In these benchmarks, traditional hyperblock formation method successfully selects blocks on the main execution path which dominates the total execution time. So, it seems less potential improvement exists for these benchmarks.

6.5 Summary

In this chapter, the issues in hyperblock formation were discussed. Applying if-conversion to generate predicated code can be done early in the compilation phase (after classical optimisation in the back-end) or late at the scheduling time when more information about the target processor characteristics is available. The performance of hyperblock scheduling depends on the accuracy of resource estimation when the basic blocks are selected to construct the hyperblock.

We proposed a new algorithm to overcome the performance loss of the original hyperblock formation heuristic for non-regular processors. We estimate resource usage through performing local scheduling before block selection. Also, to avoid delays on execution of

more frequent execution paths, partial paths are selected in cases where it appears to be beneficial. Experimental results indicate the effectiveness of our approach, particularly for low issue processors.

Chapter 7

Supporting Binary Compatibility in VLIW Machines

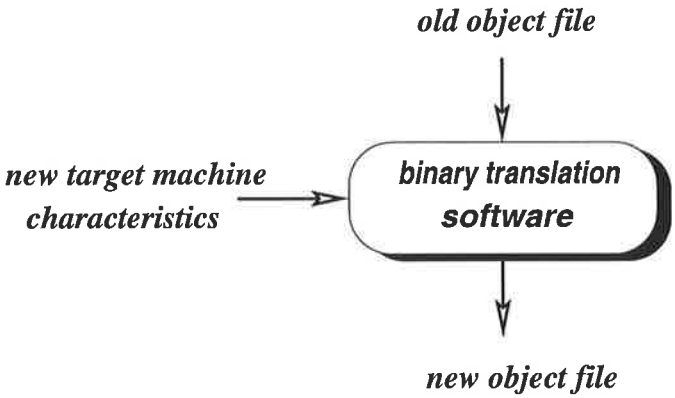
One of the main problems that prevented extensive use of VLIW architectures for non-numeric programs is lack of object code (or binary) compatibility. As explained in section 3.2.2, this is due to exposing all architectural features to generate code at compile time. In this manner, new features of a VLIW machine may lead to incorrect results by executing the code compiled for the old machine.

In this chapter, a new approach to overcome this problem is presented. It is performed with the help of code annotation provided by the compiler, to reduce the complexity of the required hardware.

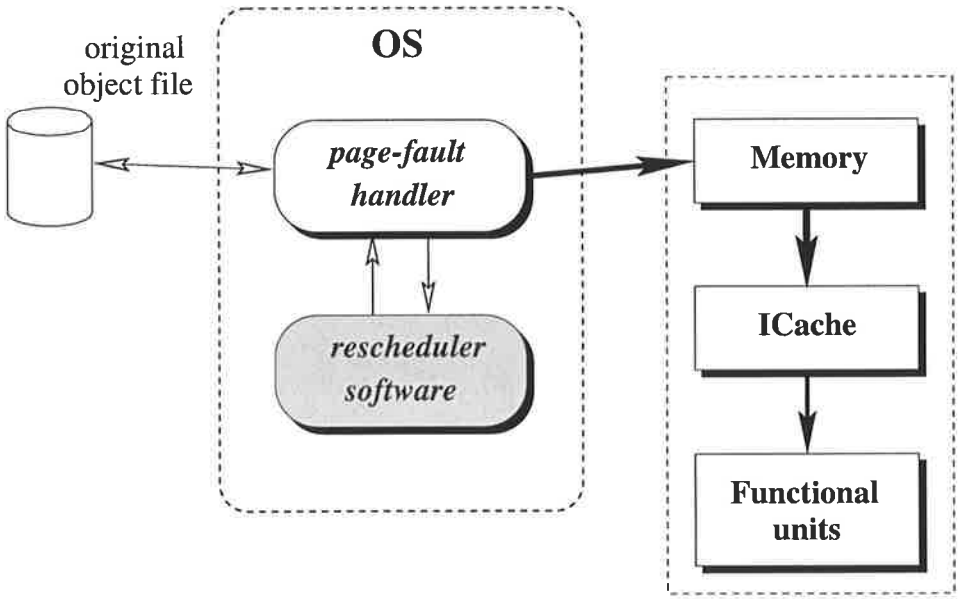
7.1 Related Works

Several software and hardware approaches to overcome the binary incompatibility problem have been proposed by researchers. Figures 7.1 and 7.2 outline the general methods which have been used. An overview of research based on these approaches follows.

The first technique is binary translation. An application program which was compiled for one architecture can be converted to a new code for another architecture through binary translation. Binary translation does not require the original source code of the application program. As indicated in Figure 7.1 (a), the old object file and the characteristics of the target machine are required for new code generation. This is an off-line approach which does not affect the run-time behaviour of the processor. This technique has been used for



(a)



(b)

Figure 7.1: Outline of software-based techniques to achieve object code compatibility for different generations of the VLIW architecture. (a) Binary translation. (b) OS-based dynamic rescheduling.

large scale migration of programs from one generation of machine to the next [Sites and et. al., 1993, Silberman and Ebcioglu, 1993].

Binary translation has some advantages over re-compilation of the source code for the new machine. When the source code is not available or an understanding of the structure of the source code is necessary for re-compilation, binary translation is preferred. Also, it is possible to apply some code optimisation techniques at translation time to improve the quality of the executable code.

Issues such as self-modifying code and the operating system interface impose problems for binary translation. The old machine's OS traps may require that their arguments be translated to equivalent calls on the new machine's OS. These issues should be addressed in the binary translation software.

Run-time rescheduling can be performed by a software method, or a hardware technique, or a combination of both. Figure 7.1 (b) shows a run-time rescheduling technique through the operating system of the underlying machine. Two major researches have been reported which use this approach. Conte and Sathaye described a dynamic rescheduling approach which works as follows [Conte and Sathaye, 1995].

On a page-fault, the OS performs context switching and loads the page from the next level of the memory hierarchy. At the first-time page fault, the OS detects the difference between the current machine's characteristics and those assumed to generate the executable code. This may be possible through saving the machine characteristics in the header of the binary file. This page of code is rescheduled by special scheduling software that is invoked by the OS to generate a correct executable code for the current machine. When the same pages are accessed again, they are retrieved from the text swap space of the OS, where they were saved when displaced by new pages. As the page size is constant, a special operation encoding has been designed to keep the size of the rescheduled code the same as the original code [Conte and et al., 1995].

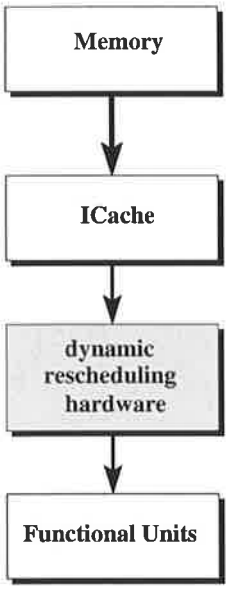
Rescheduling of each page at each first-time page fault introduces additional overhead on the page fault handling process by the OS. When a program is to be executed several times, the scheduled pages can be saved for later use. A technique called the persistent rescheduled-page cache (PRC) was proposed to reduce the overhead in these cases [Conte et al., 1996d]. The PRC, as a part of the OS file system, keeps the rescheduled pages that were written into it at the last termination of the program. For subsequent program execution, when a page fault

occurs with generation mismatch, the PRC is searched for the existence of the rescheduled version of the page. If it is present, it will be used and in this way, the time for rescheduling is saved.

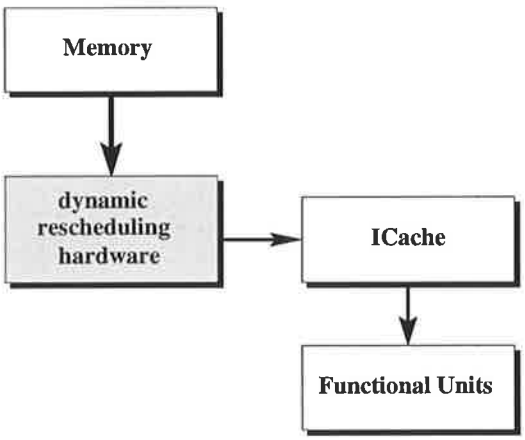
Another work based on the approach shown in Figure 7.1 (b) was presented by Ebcioglu and Altman [Ebcioglu and Altman, 1996]. The aim of this work is to emulate an old architecture on a new architecture and follows similar steps to those mentioned above, to generate a new schedule for a tree-based VLIW processor.

Figure 7.2 illustrates the general outline of hardware-based techniques to achieve object code compatibility among different generations of a VLIW architecture. In Figure 7.2 (a), dynamic rescheduling is performed in the execution pipeline path. Rau [Rau, 1993] presented dynamic scheduling for non-unit assumed operation latencies (NUAL) execution semantics in VLIW machines. In his method, which is called *Split-Issue*, each operation is partitioned into two phases. Phase1 includes access of source operands and computation. The result of the phase1 is written into a renamed register. Phase2 is copying the content of the renamed register into the actual architectural register. The latencies of phase1 and phase2 are **L-1** and **1** respectively, if the assumed latency of the operation is **L**. Phase2 is scheduled to issue at **L-1** cycles after phase1 issue to maintain the program semantics. In this way, the computation part and writing of the result into the architectural registers are performed at different phases and by different functional units (the main computation functional unit and the copy-back unit, respectively). This approach in general, requires complex hardware which potentially has an impact on cycle time. To achieve simpler hardware, a special mechanism referred to as *latency stalling* was proposed [Rau, 1993]. In this case, the instruction issue stalls when the assumed latency of an operation elapsed, but the actual latency does not.

Arita and his colleagues presented an extended VLIW architecture called V++ to overcome the problem of increase in code size and changes of the assumed operation latencies at run time [Arita et al., 1994]. Their rescheduling technique is based on predetermined and adaptive restructuring. Predetermined restructuring is performed by a set of delay registers for each functional unit. The amount of delay is assigned to each operation by the compiler. Adaptive restructuring employs the ultimate-barrier method [Takagi et al., 1991], which is a high-speed synchroniser. It is used to apply the precedence relations among operations dynamically through blocking those operations whose precedence relations have not been satisfied. In this manner, it provides a producer-consumer synchronisation process. In sum-



(a)



(b)

Figure 7.2: Outline of hardware-based techniques to achieve object code compatibility for different generations of the VLIW architecture. (a) Dynamic rescheduling at the execution pipeline path. (b) Dynamic rescheduling before moving operations to the instruction cache.

mary, functional unit assignment and detection of precedence relations in V++ is performed by the compiler. However, resolution of precedence relations is done by the hardware.

In Figure 7.2 (b), dynamic rescheduling is performed out of the execution pipeline path. One proposed approach in this category is based on a mechanism called the fill unit. The fill unit was originally proposed to form larger executable units from microoperations [Melvin et al., 1988]. In this approach, operations are prefetched from memory or instruction cache and are converted into their corresponding microoperations. Then, these microoperations are placed into a buffer. When the operation is a branch or the buffer is full, the group of microoperations (which is called *multinode* in [Melvin et al., 1988]) are placed into the decoded cache. Later, instructions are fetched from the decoded instruction cache. Hardware mechanisms are provided to handle data dependencies and exception recovery.

The original fill unit approach was extended by Franklin and Smotherman [Franklin and Smotherman, 1994] to form VLIW instructions dynamically from operations that can be issued together. This works as follows. The fill unit receives operations in the program order from the conventional instruction cache and decoder. A group of decoded operations which can be issued at the same time are formed in the fill unit buffer. When the buffer is full, it is written into an extra cache called the *shadow cache*. At each cycle, both the conventional instruction cache and the shadow cache are accessed and if the shadow cache is hit, the VLIW type instruction from the shadow cache is sent to the functional units. Otherwise, one operation fetched from the conventional I-cache is executed. Similarly to the original fill unit technique, the fill unit buffer is finalised when a branch is encountered. However, in this technique both paths of a branch can be fetched to form a tree-like instruction.

The DIF (Dynamic Instruction Formating) machine [Nair and Hopkins, 1997] is another approach which can be considered to be based on Figure 7.2 (b). In the DIF machine, operations are executed for the first time by a separate processing engine referred to as *primary engine* in [Nair and Hopkins, 1997]. At the same time, the dependency information is provided by the primary engine to translator hardware, which reschedules these executed operations as a DIF group. A DIF group is the unit of execution and consists of a sequence of VLIW instructions. DIF groups are stored in the DIF cache which is connected to another execution engine called the *parallel engine* and is similar to a VLIW execution pipeline. This approach is able to schedule operations speculatively above a few conditional branches. This is performed through a path prediction mechanism, rather than a branch prediction method.

The direction of a path is determined based on the results from the primary engine at translation time. When a misprediction or exception occurs, all operations in the DIF group are re-executed in the primary engine and a new DIF group is generated.

Miss Path Scheduling (MPS) introduced by Banerjia and et al. [Banerjia et al., 1998] is also based on the approach of Figure 7.2 (b). In this technique, operations are rescheduled at cache miss time when the operations are received from a higher level of memory. The rescheduling hardware is able to schedule operations speculatively above conditional branches. Then, scheduled blocks are provided to an in-order execution engine. A scheduled block contains a set of VLIW-type instructions. The MPS technique uses one cache to hold operations fetched by the processor pipeline, in comparison with two different caches (shadow cache and instruction cache) required for the fill unit approach and the DIF method.

7.2 A New Approach

This section describes our proposed approach to overcoming the binary incompatibility problem between different generations of VLIW machines. We call this *dynamic VLIW generation* (DVG). It requires ISA support and additional hardware, but this hardware is not located on the critical path of the execution pipeline. Operations are rescheduled at the cache miss repair. Thus, DVG is based on the approach in Figure 7.2 (b).

To simplify dependency checking hardware, some form of dependency information is encoded for each operation at compile time. Each operation has a *dependency_word*. This information can be combined into the final binary code or may be provided in a separate file, which can be loaded into memory by the OS loader at execution time.

To schedule operations, the compiler generates a dependency graph to capture all dependency information in a scheduling region (such as a hyperblock). Transferring all of this information through the object file is not practical as it requires a large amount of disk space. Also, this information should be available in a suitable form to be used by a dynamic rescheduler with less overhead. Therefore, for DVG, the compiler provides a limited form of dependency information which will be processed at the time of instruction cache miss repair.

The simplest way of encoding dependency information is by using a bit vector for each group of operations. Each bit can represent a dependency to a previous operation based on its position in the bit vector. For example, if an operation has a dependency on the 14th op-

eration before it, then bit 13 in the bit vector is set. This scheme however, increases the code size. If each operation is 32 or 64 bits, then having a 64-bit dependency_word is unacceptable. To reduce the size of the dependency_word, each bit may represent dependency to a packet of previous operations. For example, if each bit represents dependency to a packet of eight operations, keeping dependency information for 64 previous operations requires only 8 bits. This is at the expense of imposing more limitations at the time of rescheduling. As an example, if operation x is only dependent on one of the operations in the corresponding packet, it cannot be scheduled before all eight operations in that packet unless it is known which operation in that packet should precede it. This can be done by using special hardware to resolve the issue. Details are described in the next sections.

7.2.1 Speculative Scheduling

It was shown that speculative scheduling is an essential technique to achieve higher amounts of ILP in general-purpose applications [Lam and Wilson, 1992]. In order to perform speculative code motion in DVG, several issues should be considered.

The first issue is a method to predict conditional branches. In our scheme, two possibilities are available. One is an ISA extension, so that each conditional branch operation has an additional bit to indicate the prediction of its direction. This is similar to architectures presented in [HP, 1994, Weiss and Smith, 1994]. This bit is set if the branch is predicted taken. The prediction can be based on profile information or the heuristics used by the compiler. The other one is modifying the code layout, so that the most likely direction of a forward conditional branch is its fall-through path.

The second issue is providing a mechanism to keep and restore the processor state when the prediction direction is not valid. Speculative motion of operations which write into a register is not allowed when the destination register is live on the other path. For this purpose, it is necessary to rename the architectural registers.

Register renaming involves mapping an architectural register into a physical register where the result of the operation is written. Following operations refer to this physical register to retrieve the value for their input operands. In a typical hardware register renaming scheme in superscalar processors, a mapping table is looked up to determine the current mapping of the register when the contents of the register is read. The mapping table is updated when the register is written. A large number of ports is required to perform this process when

multiple operations are executed concurrently.

To avoid this complexity, we extend a form of the *rotational remap* renaming scheme which was presented in [Nair and Hopkins, 1997]. In our scheme, the register file has several physical locations and status fields for each architectural register. The status fields indicate which physical location holds the most recent value and how the original physical location can be identified when a branch is mispredicted. Figure 7.3 shows an example to indicate how this scheme works.

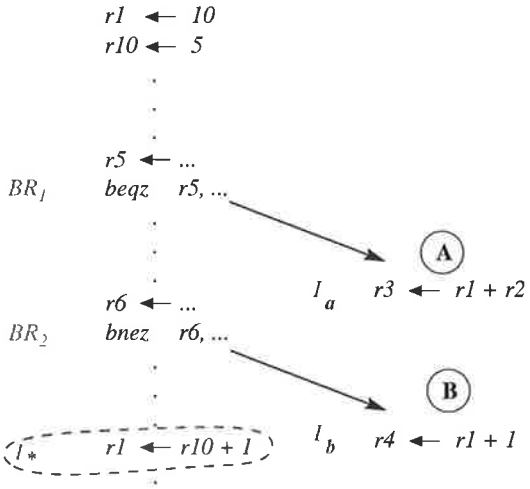
A sequence of operations with two conditional branches is shown in Figure 7.3 (a). The aim is to move operation I_* above branches BR_1 and BR_2. Figure 7.3 (b) shows the state of register r1 if I_* is not scheduled above BR_1. After moving it up above BR_2, the state is changed to that shown in Figure 7.3 (c). The *active_L* field indicates the current physical location mapped to r1. *Speculation adjustment (SA)* determines how many locations should be moved to the left (considering a rotating remapping scheme) to reach the correct value of the architectural register when the branch is mispredicted. In the case of moving I_* above BR_2, the value in *speculation adjustment* field is set to 01, which means one shift to the left to access the original value of r1 if BR_2 is taken.

Figure 7.3 (d) shows the case when I_* is scheduled above BR_1. In this case, I_* is speculated over two conditional branches and if one of them is taken the original state is preserved through moving two locations to the left (based on *speculation adjustment* of 10). It should be noted that, no operation exists in the fall-through path of BR_1 to change r1. Otherwise, it would not be possible to schedule I_* above BR_1 due to that dependency constraint.

7.2.2 DVG Scheduling Algorithm

We propose the DVG approach in the context of predicated code. The beginning and end points of each hyperblock are encoded in the ISA by the compiler. Any code motion is performed in the hyperblock. This means that the original hyperblock is rescheduled for the new machine.

Figure 7.4 illustrates the DVG algorithm. At instruction cache miss time, the value of the PC is used to load at most N operations from the upper level of the memory hierarchy. N is implementation dependent. The issue-width of the processor and the implementation details of the DVG hardware are used to determine N, so that the amount of rescheduling overhead is tolerable.



(a)

architectural index	*L0*	*L1*	*L2*	*L3*	*active_L*	*SA*
r1	-	*10*	-	-	*01*	*00*

(b)

architectural index	*L0*	*L1*	*L2*	*L3*	*active_L*	*SA*
r1	-	*10*	*6*	-	*10*	*01*

(c)

architectural index	*L0*	*L1*	*L2*	*L3*	*active_L*	*SA*
r1	*6*	*10*	-	-	*00*	*10*

(d)

- means don't care.
L0 to L3 indicate physical locations.
active_L shows the index of current physical location for the register.
SA is speculation adjustment.

Figure 7.3: An example of using the rotational remapping scheme in DVG with special status fields to restore registers' original state when a branch is mispredicted. (a) A sequence of operations to be rescheduled. (b) State of r1 before rescheduling of I_*. (c) State of r1 after moving I_* above BR_2. (d) State of r1 after moving I_* above BR_1.

```
// Input:  N  (maximum numbers of operations to be  rescheduled)
           PC  (address of missed operation in the ICache)
// Output:  rescheduled code in the scheduling_buffer

last_scheduled_cycle  =  0
for  N  operations
     use PC to fetch operation <op> and its dependency_word from memory
     if <op> is a branch  then
           start_cycle  =  last_scheduled_cycle
     else
           start_cycle  =  0
     use start_cycle and dependency_word to find the sch_cycle for <op>
     if valid sch_cycle not found  then
           // this means that no resource available.
           terminate the scheduling process
     if scheduled <op> is speculative and has dst_reg  then
           set SP_status of the register dst_reg
     place scheduled <op> in the scheduling_buffer
     if sch_cycle  >  last_scheduled_cycle  then
           last_scheduled_cycle  =  sch_cycle
     if <op> is the last operation in the hyperblock  then
           terminate the scheduling process
     increment PC  (PC  =  PC  +  4  in our case)
```

Figure 7.4: Main steps of the DVG scheduling algorithm.

After receiving each operation from the upper level of memory, it is scheduled and placed in the *scheduling buffer*. The scheduling buffer is similar to the reservation table in [Banerjia et al., 1998]. Its structure is like a matrix with m rows and n columns, where m is the maximum number of cycles (or the maximum number of generated VLIWs), and n is the issue-width of the processor. Also, a structure called the *operation scoreboard* is used to record the status of each processed operation. The cycle in which the result of each scheduled operation will be ready is recorded in the *operation scoreboard.*

Each architectural register has a counter which is loaded with the latency value of the operation writing into it. The counter is decremented each cycle when the execution of an operation is started until it is 0. Then, the *ready* bit is set. When a register is accessed for reading, if the result is not ready, all operations in the current VLIW are stalled until the required result is ready. The main purpose of the counter is to handle cases when the latency of an operation scheduled in the previous scheduling region is not fulfilled. At the same time, this mechanism preserves the processor state at the time of unexpected events which may increase the latency of an operation. The value of these counters may be saved and restored when necessary to keep the processor state.

When the maximum number of generated VLIWs is reached, or the last operation of a hyperblock is scheduled, or N operations are processed, the rescheduling process is terminated and the contents of the *scheduling buffer* are transferred to the instruction cache. The amount of time required for this step depends on the structure of the instruction cache. In [Banerjia et al., 1998], two cache organisations for holding VLIWs were considered. These are the uncompressed I-Cache and the compressed banked I-Cache [Conte et al., 1996a]. As the structure of the uncompressed I-Cache is similar to the *scheduling buffer*, transferring VLIWs from the *scheduling buffer* to the I-Cache does not require a long time in comparison to the compressed banked cache. In the latter, NOP operations are not included in the cache and special encoding is used to handle this. This results in longer time to transfer operations from the *scheduling buffer* to the I-Cache.

To schedule operations speculatively, the rotational remapping scheme is employed as discussed above. To prevent scheduling of a branch before operations which originally preceded it, the latest scheduled cycle is kept into a special register. Branches cannot be reordered. Also, an operation cannot be moved above a call subroutine and an indirect jump. Note that for applying the DVG approach to the EVA, the ISA of EVA includes non-trapping

version of excepting operations, so it does not require additional hardware to keep the precise exception handling. Otherwise, similar hardware schemes to those used in any multiple-issue processor to achieve precise exception handling would be necessary.

We use the example in Figure 7.5 to describe how the DVG approach works. Figure 7.5 (a) shows a code segment scheduled for an issue-2 VLIW processor which is considered as the old-machine. Operations I_1 to I_{15} are located consecutively in memory with their dependency words. In this example, predicate operands are assumed as "TRUE" and are not shown. Also, we assume each bit in the *dependency_word* is used to indicate a dependency to one previous operation.

Figure 7.5 (b) illustrates the scheduled operations for the new VLIW machine, which is an issue-3 processor and the latency for multiplication is reduced to 2 cycles in comparison to the old machine. The scheduling process is as follows.

When the cache miss occurs, the address of the operation resulting in the cache miss is used to get the operation from the higher level of memory hierarchy with its *dependency_word*. The *dependency_word* is applied to the *operation scoreboard* to find the earliest cycle for scheduling. Operations I_1, I_2 and I_3 do not have dependency on previous operations (as the region begins with I_1). Therefore, they can be scheduled in cycle 0 in the new machine. Then, operation I_4 is processed. It is dependent on I_1 so, referring to the *operation scoreboard* determines that I_4 can be scheduled in cycle 2. Other operations are processed and scheduled in this way. Operations I_{10}, I_{11} and I_{12} are speculative and the status fields in the related architectural registers are set appropriately as described before. When an excepting operation is speculated, its non-trapping version is used as the speculative operation.

In the case of backward branches, which are marked by the compiler, a limited form of loop unrolling can be applied which is dependent on N (the maximum number of operations to be rescheduled) and the distance between the branch and its target.

7.3 Comparison with Related Works

The DVG approach, similarly to the fill unit [Franklin and Smotherman, 1994], the DIF machine [Nair and Hopkins, 1997], and the MPS [Banerjia et al., 1998], reschedules operations before placing them in the cache which is accessed by the parallel execution engine in the machine. The fill unit method lacks speculative scheduling and requires two instruction

cycle	slot 0		DW	slot 1		DW
0	I_1	r2 ← MEM (r1)	(0,0,0,0,0,0)	I_2	r4 ← MEM (r3)	(0,0,0,0,0,0)
1	I_3	r6 ← MEM (r5+8)	(0,0,0,0,0,0)			
2	I_4	r2 ← r2 + 1	(0,0,0,1,0,0)	I_5	r7 ← r4 - 1	(0,0,0,1,0,0)
3	I_6	r8 ← r6 - r2	(0,0,0,1,1,0)	I_7	r9 ← r2 * 3	(0,0,0,1,0,0)
4	I_8	r10 ← MEM (r11 + 4)	(0,0,0,0,0,0)	I_9	beqz r8, L1	(0,0,0,1,0,0)
5	I_{10}	r6 ← r2 + r7	(1,1,0,0,0,0)	I_{11}	r12 ← MEM (r8)	(0,1,0,0,0,0)
6	I_{12}	r13 ← r10 + 2	(0,0,1,0,0,0)			
7	I_{13}	r14 ← r10 * r9	(1,1,0,0,0,0)	I_{15}	bnez r13, L2	(0,0,0,0,1,0)

A piece of code scheduled for an issue-2 VLIW processor.
L1 and L2 refer to the code outside of the region.
DW - Dependency_Word
Operation latency: load = 2, multiplication = 3, other = 1.

(a)

cycle	slot 0	slot 1	slot 2
0	I_1	I_2	I_3
1	I_8		
2	I_4	I_5	
3	I_6	I_7	I_{10}
4	I_{11}	I_{12}	I_9
5	I_{13}		I_{15}

Code scheduled for an issue-3 VLIW processor.
Operation latency: load = 2, multiplication = 2, other = 1.
Speculative operations are shown shaded.

(b)

Figure 7.5: An example of DVG rescheduling process. (a) Code scheduled for the old machine. (b) Code rescheduled for the new machine.

caches (the conventional I-Cache and the shadow cache). The DIF machine also employs two caches and has two different execution engines. The DVG method on the other hand requires one instruction cache. It is an extension of the conventional VLIW processor which reschedules operations dynamically at cache miss time. The most similar published work to the DVG method is the MPS method. In the MPS, operations are also rescheduled at the cache miss repair. The major difference between the DVG approach with the previous works (particularly with the MPS method) is the use of code annotation generated by the compiler to direct the rescheduling process. Also, a limited form of loop unrolling can be performed for backward branches in the DVG.

7.4 Experimental Results

To evaluate the effectiveness of the DVG approach, experimental results were generated based on the following methodology.

We modified the *mlcache* [Tam et al., 1997], so that at each cache miss at most N operations are fetched from the next level of the memory hierarchy and scheduled. The scheduled operations are placed into the appropriate locations of the instruction cache. A perfect data cache is assumed for all experiments. The instruction cache is a 64K direct mapped banked cache (as described in section 3.2.1). The assumed bandwidth between the I-Cache and the next level of memory is one word (32 bits) per cycle with six cycles latency. These are contemporary assumed values typical of current memory technology. To approximate the scheduling overhead, five additional cycles are added to the total miss repair latency. This assumption is reasonable especially, if pipelined hardware is employed.

To record the number of cycles, one cycle latency is assumed for each cache hit, and the processor stalls for the period of the cache miss repair. The number of execution cycles for the old machine and the speedup are obtained as described in section 5.2.1.

Six SPEC95 integer programs (from our benchmark set) are used as the benchmarks in this experiment. Each benchmark program is compiled with the EVA compiler for the old machine, and the required code annotations (*dependency_word* and other operation marking such as backward branches and so on) are generated. Program traces (using the train inputs of the benchmarks) are generated in a proper format and are used by the *mlcache*.

Figure 7.6 shows the speedup achieved through moving the older machine code to the

new wider machine and applying the DVG technique. This indicates that the DVG technique uses additional resources in the new machine to improve the performance of the code compiled for the old machine.

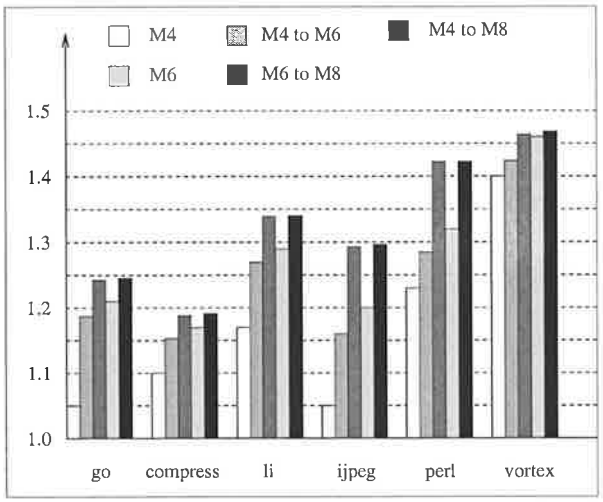


Figure 7.6: Speedup of the rescheduled old machine code on a wider new machine model.

Table 7.1 shows the decrease in speedup when the code complied for the old machine is rescheduled through the DVG method for the new machine in comparison to the code compiled for the new machine. For example, the number in the M4 to M8 column indicates the reduction in speedup for the code originally compiled for M4 and rescheduled for M8 compared to the speedup of the code compiled for the M8 machine model. Speedup Figures are based on basic block scheduling as discussed in section 5.2.1.

Results are shown in Figures 7.7, 7.8, and 7.9 for three cases. Figure 7.7 shows the performance of the code rescheduled for machine model M4. The speedup is compared with the code compiled for the M4 model. Figures 7.8 and 7.9 indicate the performance for the M6 and M8 models respectively. The results show that performance of the DVG technique is higher for wider-issue processors. This is due to the more opportunities for scheduling provided when more resources are available.

Benchmark	M4		M6		M8	
	M6 to M4	M8 to M4	M4 to M6	M8 to M6	M4 to M8	M6 to M8
099.go	12%	12%	11%	10.3%	10.2%	9.1%
129.compress	8.5%	10%	10%	9.1%	9.4%	9%
130.li	7.9%	8.9%	7%	6%	5.8%	5.6%
132.ijpeg	10.8%	10.9%	9.1%	8.9%	8.5%	7.6%
134.perl	11%	11.3%	11%	10.5%	10.2%	10.2%
147.vortex	9.4%	10%	9.5%	9.3%	9.1%	8.9%

Table 7.1: Relative performance of the rescheduled code in comparison to the code generated by the compiler for different machine models.

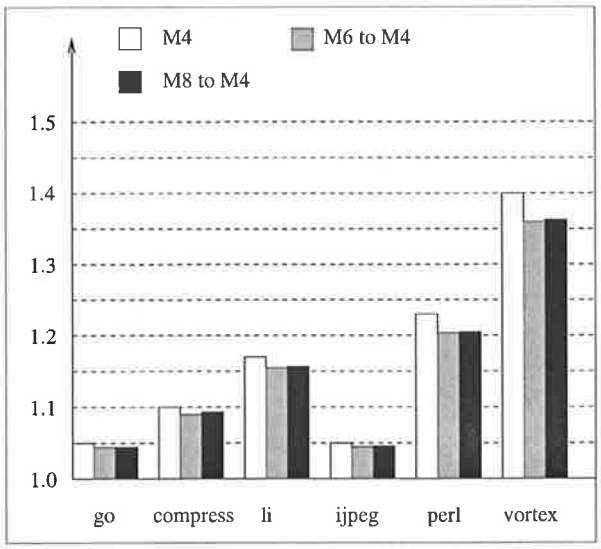


Figure 7.7: Speedup of the rescheduled code for machine model M4.

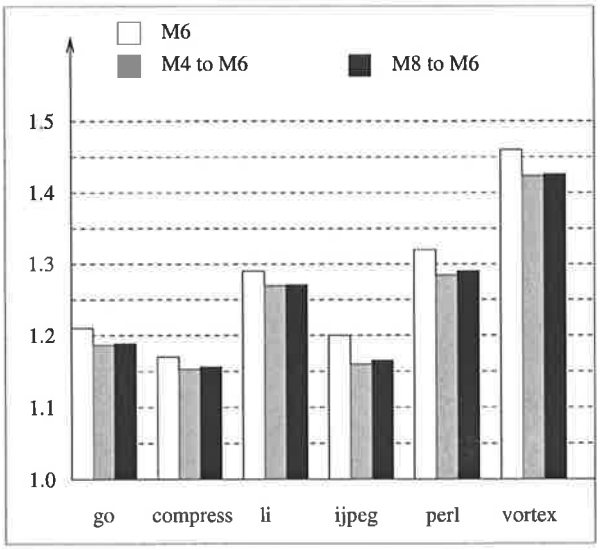


Figure 7.8: Speedup of the rescheduled code for machine model M6.

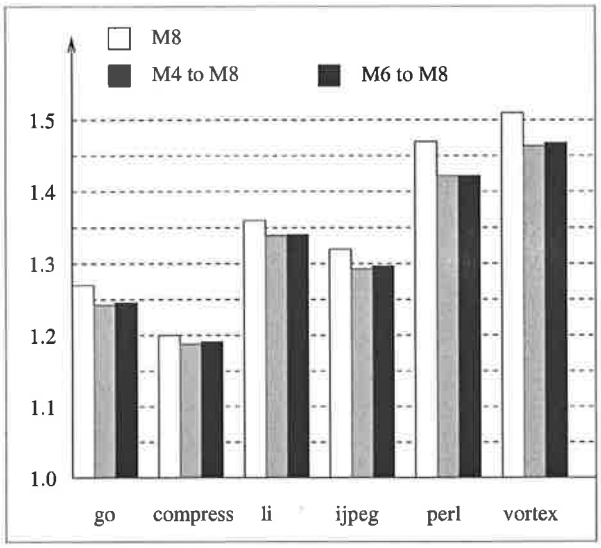


Figure 7.9: Speedup of the rescheduled code for machine model M8.

7.5 Summary

In the process of ILP extraction and code generation for a VLIW processor, the architectural features are exposed to the compiler and operations are statically scheduled based on the target machine characteristics. This leads to binary incompatibility among different implementations of the same VLIW architecture. We presented a new approach called dynamic VLIW generation (DVG) to overcome this problem.

In the DVG technique, operations are rescheduled for the new machine at the time of instruction cache miss repair. In this way, the rescheduler hardware is not located in the execution pipeline engine avoiding potentially longer cycle times. To simplify the dependency checking hardware, dependency information is encoded for each operation at compile time. This information can be combined into the final binary code, or may be provided as a separate file, which can be loaded into memory at the execution time by the OS loader. In this technique operations can be rescheduled speculatively and a mechanism is presented to prevent destroying the contents of live registers.

The rescheduled code can be fetched from the instruction cache in the conventional form. Experimental results show that the performance of rescheduled code using the DVG technique is only a few percent worse, for both wider and narrower issue processors, than code compiled directly for the target processor.

Chapter 8

Summary and Conclusion

8.1 Summary

The effective exploitation of ILP involves extracting as much ILP as possible from the application programs with regard to the available processor resources. Possible side effects which may potentially result in lower processor speed should be avoided and their impact reduced. This research targets these issues through implementing an optimising compiler for VLIW architectures. The task of ILP extraction is achieved by the compiler mainly through exposing the architectural features and processor characteristics to the compiler.

Predicated execution has been employed in our compiler to reduce the serious obstacles to ILP extraction due to frequent conditional branches in general-purpose programs. Predicated execution allows the compiler to eliminate some conditional branches through utilising conditional execution. Predicated execution is exploited in the compiler through a structure called a hyperblock. Hyperblocks include basic blocks from multiple execution paths. The performance of hyperblock scheduling depends on the accuracy of resource estimation when the basic blocks are selected to construct the hyperblock. We proposed a new algorithm to reduce the performance loss of the original hyperblock formation heuristics for non-regular processors. An estimation of the resource usage is made through performing local scheduling before block selection. Partial paths or part of a basic block are selected if it is shown to be profitable. Also, a new technique for predicate-aware register allocation was presented. We implemented a complex VLIW compiler system based on SUIF/machsuif to test these ideas. Experimental results indicate improvement of the order of 10%-20% using our approach.

VLIW machines have not been used extensively as an ILP processor for general-purpose

programs. One of the major reasons is the lack of object code compatibility among different generations of the same architecture. This is due to the assumed architectural features such as operation latencies, the number of functional units and register file specifications at the time of code generation. When the characteristics of the processor is changed, the assumptions made at compile time may not be valid and preservation of program correctness is not guaranteed. We presented a new approach to solve this problem which we call dynamic VLIW generation (DVG).

In the DVG technique, operations are rescheduled for the new machine at the time of instruction cache miss repair. In this manner, the rescheduler hardware is not located in the execution pipeline engine. To simplify dependency checking hardware, dependency information is encoded for each operation at compile time. This information can be combined into the final binary code, or may be provided as a separate file, which can be loaded into memory at the execution time by the OS loader. This technique is able to perform speculation in the rescheduling process. The rescheduled code is placed into the instruction cache and the next cache accesses, which are cache hits, receive the rescheduled code. Experimental results show that the performance of rescheduled code using the DVG technique is about 10% worse than code compiled directly for the target processor.

8.2 Future Directions

Possible future works along this line of research can be classified into two different parts. These are related to new optimisation techniques for the predicated code and improving the compiler assisted object code compatibility among different VLIW processors.

For the first part, the following are considered as possible further research topic.

- Inter-region code motion - In our research, we performed code optimisation and scheduling in the context of hyperblocks. Therefore, the possibility of code improvement through performing special case optimisation and code motion between hyperblocks can be investigated.

- Predicated loop peeling - Loop peeling is an effective technique to enlarge loop regions with inner loops for predicated code generation. As optimisation and scheduling after loop peeling may change the characteristics of the loop, a heuristic is required to estimate the possible changes in the loop characteristics for loop peeling.

- Inter-procedural register assignment in the predicated code - We proposed a technique for predicate-aware register allocation. Further works are required to adjust it to achieve more efficient code, especially through taking account of the inter-procedural characteristics.

For the second part, we notice that our approach to improve binary compatibility was investigated in the context of predicated code. So, further research is required to assess its performance when predicated execution is not supported by the architecture. Also, the overhead of the rescheduling process and its impact on the performance should be evaluated for different cases.

Appendix A

A Brief Description of Libraries and Different Passes in EVA VLIW Compiler

The VLIW compiler for the EVA is developed based on **SUIF-I** and *machine* **SUIF-I** (*mach-suif*) infrastructures [SUIF, 1994, Smith, 1997]. The front-end of the compiler generates SUIF intermediate code. In addition to the SUIF library, a general machine instruction library and a CFG library are used in our work. To implement different compiler optimisation techniques for ILP processing, we also developed several new libraries. Figure A.1 illustrates different libraries and their dependency upon each other. A brief description of each new library is given in Table A.1.

The EVA compiler is designed as multi-pass compiler. The results of each pass are saved in files and used as input for the following passes. This provides an easier way to modify the current passes and to implement new passes without the need to know the details of other passes. Table A.2 indicates a brief description of typical passes in the EVA compiler. The sequence of theses passes is shown in Figure A.2.

The instruction set architecture (ISA) of the EVA is an extension of MIPS-I instructions. In Figure A.2, *mgen* converts the SUIF intermediate code to MIPS-I instructions. Optionally, the control transfer operations can be modified to change the code layout when necessary. This is performed by *cti_optimize*. Other passes including *raga*, *label*, *mfin*, *halt* and *print-machine* are used to generate code to be run on a MIPS-based host machine. In this way, the profile information is generated. *raga* performs register allocation. *label* assigns unique numbers to branches, basic blocks and other required parts for the instrumentation. *mfin* finishes the required steps related to the stack frame. *halt* inserts calls to the analysis routines,

Library name	Description
gen_utils	Provides general utility functions (e.g. finding number of each type operations in a procedure, bitset array, etc.)
pred_sup	Provides functions to generate and access predicate operands in the predicated code.
prof_info	Includes functions and data structures to insert and retrieve profile information through code annotations.
pdfa	Predicate-sensitive data flow analysis routines.
machine_model	Provides functions to define and use information about the target machine characteristics.
region	Provides functions to define and access different code regions for optimisations and scheduling.
dep_graph	To generate dependency graphs including both data and control dependencies.
list_sch	Provides functions and data structures for list scheduling based schedulers.

Table A.1: A brief description of new libraries in the EVA compiler.

Name	Description
cti_optimize	Optimises control transfer instructions and modifies code layout.
reg_dealloc	Register de-allocation. Undo the previous register allocation and converts most hardware registers to virtual registers.
process_prof_trace	Process execution trace file to generate the required instruction annotations.
classic_optimize	Perform selected classical optimisations.
hypergen	Generate hyperblock structures.
ILP_optimize	Perform selected ILP optimisations.
scheduler	Instruction scheduler.
reg_alloc	Perform register allocation.
vliw_gen	Generate final VLIW instructions (based on the specified coding information).

Table A.2: A brief description of typical passes in the EVA compiler.

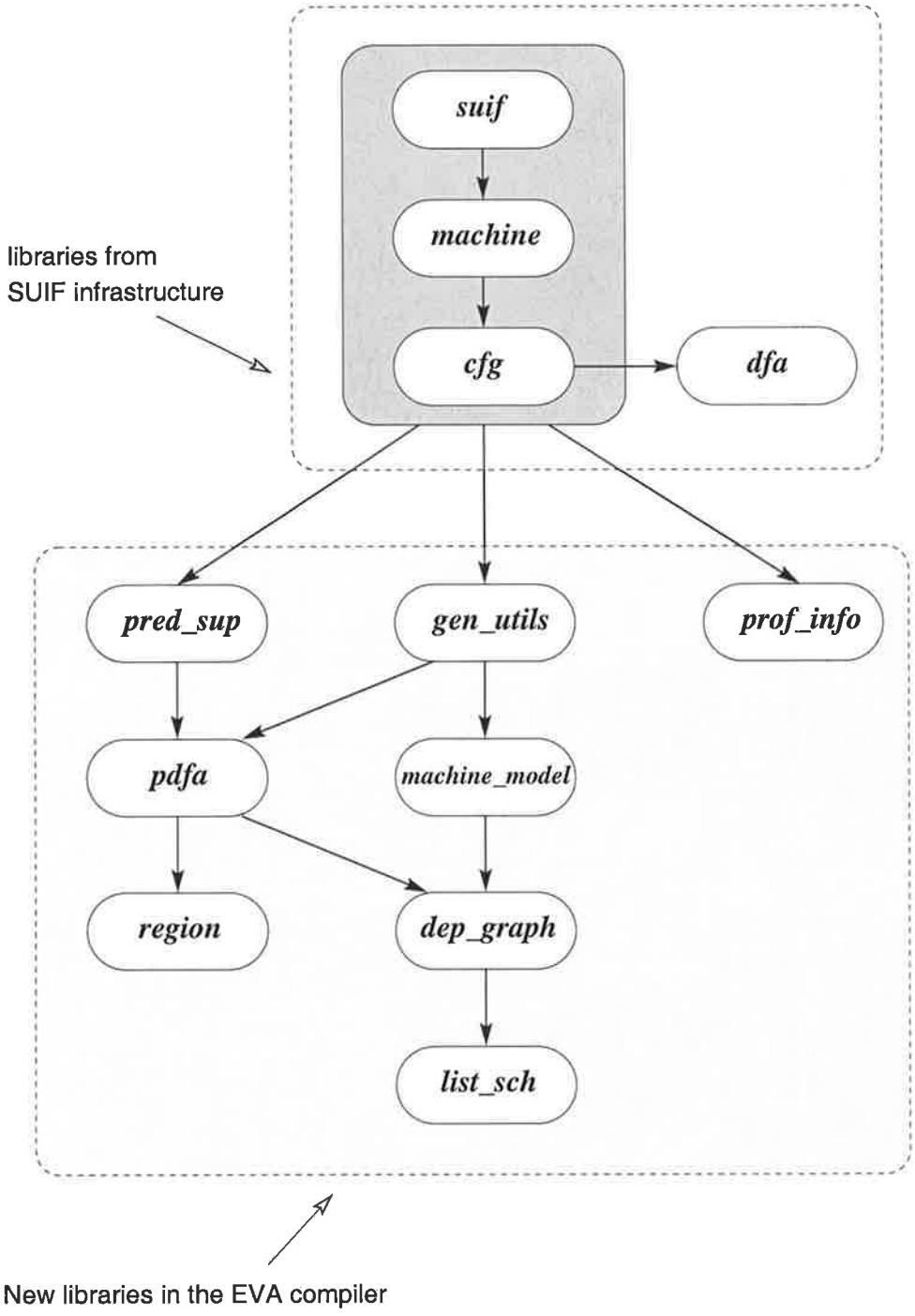


Figure A.1: Different libraries and their dependency on each other in the EVA compiler.

which use the unique numbers introduced before, and other relevant information for the instrumentation points. Passes *label* and *halt* are a part of **HALT** [Young and Smith, 1996], which is an ATOM [Eustace and Srivastava, 1995] like profiler to gather profile information. *printmachine* generates the MIPS assembly file. The output of *printmachine* is linked with the instrumentation code and the executable code is generated by the host compiler. The executable code is run on the host machine to generate profile information traces.

The new passes in the EVA compiler work as follows. *reg_dealloc* converts most registers assigned by *raga* back to virtual registers. This is necessary for the following passes. Then, *process_prof_trace* generates annotations such as the execution frequency of a basic block, branch direction, memory reference address and so on, using the profile trace files. This step may be performed several times if required to handle all trace files. Classical optimisations are performed by *classic_optimize*. *hypergen* generates the code with hyperblocks. This pass can be replaced by another pass to generate other structures such as superblock. After ILP optimisation by *ILP_optimize*, the code is scheduled through *scheduler* pass. Register allocation and final VLIW code assembly files are generated by *reg_alloc* and *vliw_gen* respectively.

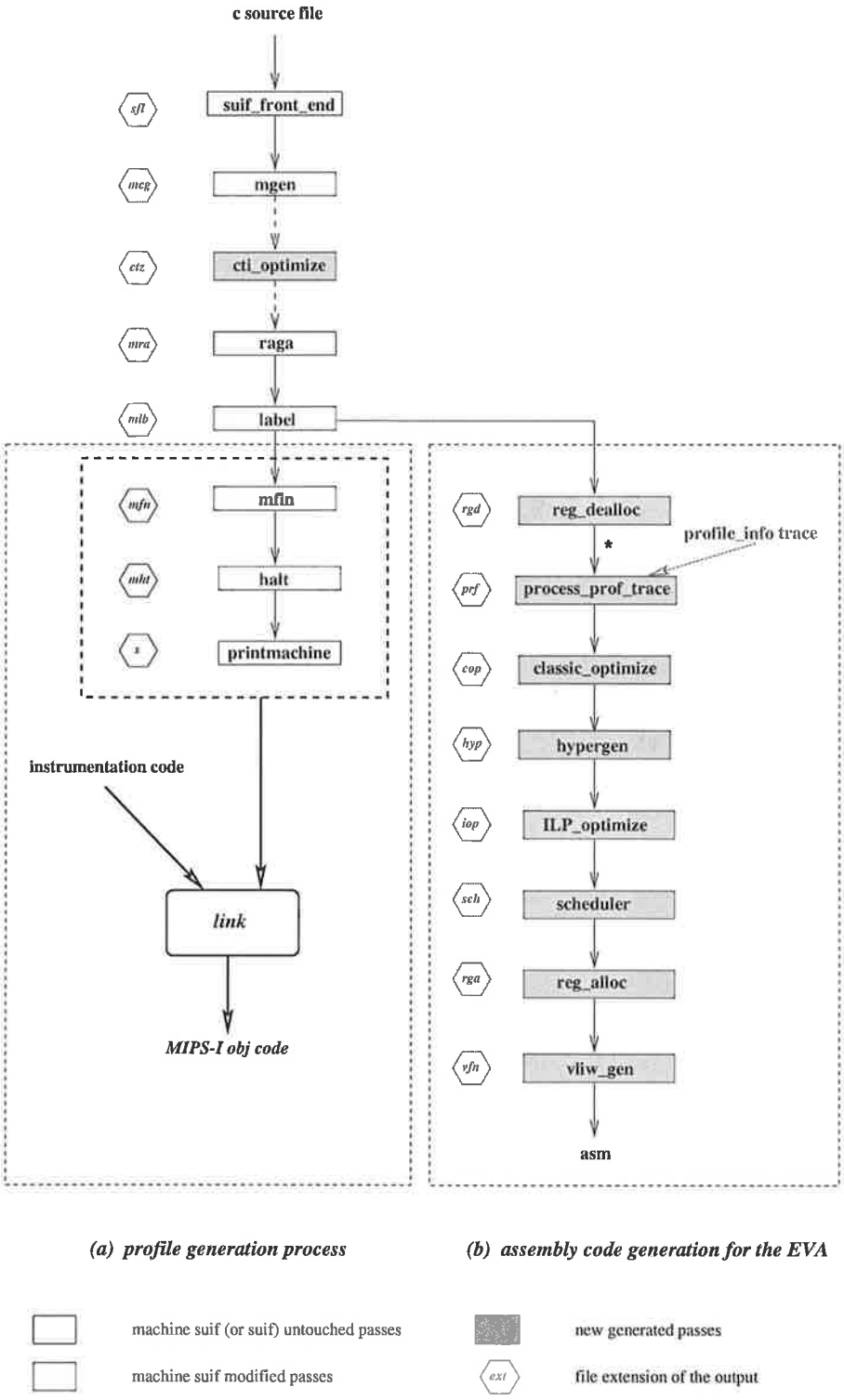


Figure A.2: Typical passes in the EVA compiler.

Bibliography

[Abraham and et al, 1993]

S. G. Abraham and et al, "Predictability of Load/Store Instruction Latencies," In *Proc. of the 26th Annual International Symposium On Microarchitecture*, pages 139–152, December 1993.

[Aho et al., 1986]

A. V. Aho, R. Sethi, and J. D. Ullman *Compilers, principles, techniques, and tools* Addison-Wesley, 1986.

[Aiken and Nicolau, 1987]

Alexander Aiken and Alexsandru Nicolau, "Perfect Pipelining: A New Loop Parallelization Technique," Technical Report Tr 88-873, Dept. of Computer Science, Cornell University, Ithaca, New York 14853-7501, October 1987.

[Allan et al., 1992]

V. Allan, J. Janardhan, R. M. Lee, and M. Srinivas, "Enhanced Region Scheduling on a Program Dependence Graph," In *Proc. of the 25th International Workshop on Microprogramming and Microarchitecture*, pages 72–80, December 1992.

[Allen et al., 1983]

J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of Control Dependence to Data Dependence," In *Proc. of the 10th ACM Symposium on Principles of Programming Languages*, pages 177–189, January 1983.

[Alpert and Avnon, 1993]

Donald Alpert and Dror Avnon, "Architecture of the Pentium Microprocessor," *IEEE Micro*, Vol. 13, No. 3, pages 11–21, June 1993.

[Altman et al., 1996]

E. R. Altman, R. Miranda, J. Moreno, and C. B. Hall, "An Integrated Approach to Architectural Simulation, Timing and Memory Hierarchy Evaluation," In *Proc. of the PAID'96*, 1996.

[Ando et al., 1995]

Hideki Ando, Chikako Nakanishi, Tetsuya Hara, and Masao Nakaya, "Unconstrained Speculative Execution with Predicated State Buffering," In *Proc. of the 22nd Annual International Symposium on Computer Architecture*, pages 126–137, Santa Margherita Ligure, Italy, 1995.

[Arita et al., 1994]

Takaya Arita, Hiromitsu Takagi, and Masahiro Sowa, "V++: An Instruction-Restructurable Processor Architecture," In *Proc. of the 27th Annual Hawaii International Conference on System Sciences*, pages 398–407, 1994.

[August, 1996]

David I. August *Hyperblock Performance Optimization for ILP Processors* Master's thesis, Dept. of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1996.

[August et al., 1995]

D. I. August, B. L. Deitrich, and S. A. Mahlke, "Sentinel Scheduling with Recovery Blocks," Technical Report CRHC-95-05, Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, February 1995.

[August et al., 1997]

David I. August, Wen mei W. Hwu, and Scott A. Mahlke, "A Framework for Balancing Control Flow and Predication," In *Proc. of the 30th International Symposium on Microarchitecture*, December 1997.

[Bala and Rubin, 1997]

Vasanth Bala and Norman Rubin, "Efficient Instruction Scheduling Using Finite State Automata," *International Journal of Parallel Programming*, Vol. 25, No. 2, pages 53–82, 1997.

[Banerjia et al., 1996]

Sanjeev Banerjia, Kishore N. Menezes, and Thomas M. Conte, "NextPC Computation for a Banked Instruction Cache for a VLIW Architecture with a Compressed Encoding," Technical report, Dept. of Electrical and Computer Engineering - North Carolina State University, 1996.

[Banerjia et al., 1998]

S. Banerjia, K. N. Menezes, S. W. Sathaye, and T. M. Conte, "MPS: Miss-path Scheduling for Multiple Issue Processors," *IEEE Transactions on Computers*, Vol. 47, No. 12, December 1998.

[Beaty, 1991]

Steven J. Beaty *Instruction Scheduling using Genetic Algorithms* PhD thesis, Colorado State University, 1991.

[Biglari-Abhari et al., 1997]

Morteza Biglari-Abhari, Michael J. Liebelt, and Kamran Eshraghian, "Implementing a VLIW Compiler using SUIF," In *Proc. of the Second SUIF Compiler Workshop*, Stanford University, CA, August 1997.

[Biglari-Abhari et al., 1998]

Morteza Biglari-Abhari, Michael J. Liebelt, and Kamran Eshraghian, "Implementing a VLIW Compiler: Motivation and Trade-offs," In *Proc. of the Third Australasian Computer Architecture Conference - ACAC'98*, pages 37–46, Perth, Western Australia, Published by Springer-Verlag, February 2-3 1998.

[Bose and Conte, 1998]

Pardip Bose and Thomas M. Conte, "Performance Analysis and Its Impact on Design," *IEEE Micro*, pages 41–49, May 1998.

[Briggs, 1992]

P. Briggs *Register Allocation via Graph Coloring* PhD thesis, Dept. of Computer Science, Rice University, Houston, TX, 1992.

[Bringmann, 1995]

Roger A. Bringmann *Enhancing Instruction Level Parallelism through Compiler-Controled Speculation* PhD thesis, University of Illinois at Urbana-Champaign, 1995.

[Bringmann et al., 1993]

R. A. Bringmann, S. A. Mahlke, R. E. Hank, J. C. Gyllenhaal, and W. W. Hwu, "Speculative Execution Exception Recovery using Write-back Suppression," In *Proc. of the 26th Annual International Symposium on Microarchitecture*, pages 214–223, December 1993.

[Chaitin, 1982]

G. J. Chaitin, "Register Allocation and Spilling via Graph Coloring," In *Proc. of the ACM SIGPLAN 82 symposium on Compiler Construction*, pages 98–105, June 1982.

[Chang et al., 1995]

P. P. Chang, N. J. Warter, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, "Three Architectural Models for Compiler-Controlled Speculative Execution," *IEEE Transactions on Computers*, Vol. 44, No. 4, pages 481–492, April 1995.

[Charlesworth, 1981]

A. E. Charlesworth, "An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS-164 Family," *Computer*, Vol. 14, No. 9, pages 18–27, 1981.

[Chen, 1993]

William Y. Chen *Data Preload for Superscalar and VLIW Processors* PhD thesis, University of Illinois at Urbana-Champaign, 1993.

[Chow and Hennessy, 1984]

F. Chow and J. Hennessy, "Register Allocation by Priority-based Coloring," In *Proc. of the ACM SIGPLAN 84 Symposium on Compiler Construction*, pages 222–232, June 1984.

[Cmelik and Keppel, 1993]

Robert F. Cmelik and David Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling," Technical Report UWCSE 93-06-06, University of Washington and Sun Microsystems, Inc., June 1993 1993.

[Colwell et al., 1988]

R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman, "A VLIW Architecture for a Trace Scheduling Compiler," *IEEE Trans. on Computers*, Vol. 37, No. 8, pages 967–979, August 1988.

[Conte et al., 1996a]

T. M. Conte, S. Banerjia, S. Y. Larin, K. N. Menezes, and S. W. Sathaye, "Instruction Fetch Mechanisms for VLIW Architectures with Compressed Encodings," In *Proc. of the 29th Annual International Symposium on Microarchitecture*, pages 201–211, Paris, France, 1996.

[Conte and et al., 1995]

T. M. Conte and et al., "The TINKER Machine Language Manual," Technical report, North Carolina State University, Raleigh, NC 27695-7911, April 1995.

[Conte and Gimarc, 1995]

Thomas M. Conte and Charles E. Gimarc, editors *Fast Simualtion of Computer Architectures* Kluwer Academic Publishers, Boston, 1995.

[Conte et al., 1996b]

Thomas M. Conte, Mary A. Hirsch, and Kishore N. Menezes, "Reducing State Loss for Effective Trace Sampling of Superscalar Processors," In *Proc. of the 1996 International Conference on Computer Design*, pages 468–477, Austin, TX, 1996.

[Conte et al., 1996c]

Thomas M. Conte, Burzin A. Patel, Kishore N. Menezes, and J. Stan Cox, "Hardware-Based Profiling: An Effective Technique for Profile-Driven Optimization," *International Journal of Parallel Programming*, Vol. 24, No. 2, pages 187–206, February 1996.

[Conte and Sathaye, 1995]

T. M. Conte and S. W. Sathaye, "Dynamic Rescheduling: A Technique for Object Code Compatibility in VLIW Architectures," In *Proc. of the 28th International Symposium on Microarchitectures*, pages 208–218, December 1995.

[Conte et al., 1996d]

Thomas M. Conte, Sumedh W. Sathaye, and Sanjeev Banerjia, "A Persistent Rescheduled-Page Cache for Low Overhead Object Code Compatibilty in VLIW Architectures," In *Proc. of the 29th Annual International Symposium on Microarchitecture*, pages 4–13, Paris, France, December 1996.

[Covington and et al., 1988]

R. C. Covington and et al., "The Rice Parallel Processing Testbed," In *In Proc. of the 1988 ACM SIMGMETRICS*, pages 4–11, 1988.

[Cytron and Ferrante, 1987]

R. Cytron and J. Ferrante, "What's in a name? - or The Value of Renaming for Parallelism Detection and Storage Allocation," In *Proc. of the 1987 International Conference on Parallel Processing*, pages 19–27, August 1987.

[Davidson et al., 1981]

S. Davidson, D. Landskov, B. D. Shriver, and P. W. Mallett, "Some Experiments in Local Microcode Compaction for Horizontal Machines," *IEEE Transactions on Computer*, Vol. C-30, No. 7, July 1981.

[Dehnert et al., 1989]

J. C. Dehnert, P. Y. Hsu, and J. P. Bratt, "Overlapped Loop Support in Cydra 5," In *Proc. of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–38, April 1989.

[Dehnert and Towle, 1993]

James C. Dehnert and Ross A. Towle, "Compiling for the Cydra," *Journal of Supercomputing*, Vol. 7, No. 1, pages 181–227, 1993.

[Deitrich and Hwu, 1996]

Brian L. Deitrich and W. W. Hwu, "Speculative Hedge: Regulating Cimpile-Time Speculation Against Profile Variation," In *Proc. of the 29th Annual International Symposium on Microarchitecture*, Paris, France, December 2-4 1996.

[Dewitt, 1976]

D. J. Dewitt *A Machine-Independent Approach to the Production of Optimal Horizontal Microcode* PhD thesis, Department of Computer and Communication Sciences, University of Michigan, Ann Arbor, MI, 1976.

[Dunn and Hsu, 1996]

D. A. Dunn and W. C. Hsu, "Instruction Scheduling for the HP PA-8000," In *Proc. of the 29th Annual International Symposium on Microarchitecture*, pages 298–307, December 1996.

[Ebcioglu and Altman, 1996]

Kemal Ebcioglu and Erik R. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility," Technical Report RC20538, IBM T. J. Watson Research Center, Yorktown Heights, NY, August 1996.

[Ebcioglu and Nakatani, 1989]

K. Ebcioglu and T. Nakatani, "A New Compilation Technique for Parallelising loops with Unpredictable Branches on a VLIW Architecture," In *Proc. of the Second Workshop on Languages and Compilers for Parallel Computing*, August 1989.

[Eichenberger and Davidson, 1996]

A. E. Eichenberger and E. S. Davidson, "A Reduced Multipipeline Machine Description that Preservs Scheduling Constraints," In *Proc. of the ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, Philadelphia, May 1996.

[Ellis, 1986]

J. R. Ellis *Bulldog: A Compiler for VLIW Architectures* MIT Press, Cambridge, MA, 1986.

[Eustace and Srivastava, 1995]

A. Eustace and A. Srivastava, "ATOM: A Flexible Interface for Building High Performance Program Analysis Tools," In *Proc. of the Winter 1995 USENIX Technical Conference on UNIX and Advanced Computing Systems*, pages 303–314, January 1995.

[Ferrante et al., 1987]

Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren, "The Program Dependence Graph and its Use in Optimization," *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 3, pages 319–349, July 1987.

[Fisher, 1981]

Joseph A. Fisher, "Trace Scheduling : A technique for global microcode compaction," *IEEE Transactions on Computers*, Vol. 30, pages 478–490, July 1981.

[Fisher, 1983]

Joseph A. Fisher, "Very Long Instruction Word Architectures and the ELI-512," In *Proc. of the 10th Annual Symposium on Computer Architecture*, pages 140–150, 1983.

[Fisher, 1993]

Joseph A. Fisher, "Global Code Generation for Instruction-Level Parallelism: Trace Scheduling-2," Technical Report HPL-93-43, HP Laboratories, Compiler and Architecture Research, June 1993.

[Fisher and Freudenberger, 1992]

J. A. Fisher and S. M. Freudenberger, "Predicting Conditional Branch Directions from Previous Runs of a Program," In *Proc. of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–95, October 1992.

[Foster and Riseman, 1972]

C. C. Foster and E. M. Riseman, "Percolation of Code to Enhance Parallel Dispatching and Execution," *IEEE Transaction on Computers*, Vol. C-21, pages 1411–1415, December 1972.

[Franklin and Smotherman, 1994]

Manoj Franklin and Mark Smotherman, "A Fill-Unit Approach to Multiple Instruction Issue," In *Proc. of the 27th International Symposium on Microarchitectures*, pages 162–171, San Jose, CA, December 1994.

[Gallagher, 1995]

David M. Gallagher *Memory Disambiguation to Facilitate Instruction-Level Parallelism Compilation* PhD thesis, Dept. of Electrical Engineering, University of Illinois at Urbana-Champaign, 1995.

[Gallagher et al., 1994]

D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu, "Dynamic Memory Disambiguation Using the Memory Conflict Buffer," In *Proc. of the 6th International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 183–195, San Jose, California, October 1994.

[Gillies et al., 1996]

D. M. Gillies, D. R. Ju, R. Johnson, and M. Schlansker, "Global Predicate Analysis and its Application to Register Allocation," In *Proc. of the 29th Annual International Symposium on Microarchitecture*, pages 114–125, Paris, France, December 2-4 1996.

[Gloy, 1998]

Nikolas C. Gloy *Code Placement using Temporal Profile Information* PhD thesis, Harvard University, Division of Engineering and Applied Sciences, September 1998.

[Gyllenhall et al., 1996]

J. C. Gyllenhall, W. W. Hwu, and B. R. Rau, "Optimization of Machine Description for Efficient Use," In *Proc. of the 29th Annual International Symposium on Microarchitecture*, Paris, France, December 2-4 1996.

[Hank, 1993]

Richard E. Hank *Machine Independent Register Allocation for the IMPACT-1 C Compiler* Master's thesis, University of Illinois at Urbana-Champaign, 1993.

[Hennessy and Patterson, 1996]

John L. Hennessy and David A. Patterson *Computer Architecture: A Quantitative Approach* Morgan Kaufmann Publishers, San Francisco, CA, second edition, 1996.

[Horel and Lauterbach, 1999]

Tim Horel and Gary Lauterbach, "UltraSPARC-III: Designing Third-Generation 64 bit Performance," *IEEE Micro*, pages 73–75, June 1999.

[HP, 1994]

HP *PA-RISC 1.1 Architecture and Instruction Set Reference Manual* Hewlett Packard, Palo Alto, CA, 1994.

[Hsu and Davidson, 1986]

P. Y. Hsu and E. S. Davidson, "Highly Concurrent Scalar Processing," In *Proc. of the 13th International Symposium on Computer Architecture*, pages 386–395, June 1986.

[Hsu, 1986]

P. Y-T. Hsu *Highly Concurrent Scalar Processing* PhD thesis, University of Illinois at Urbana-Champaign, 1986.

[Huff, 1993]

R. A. Huff, "Lifetime-sensitive Modulo Scheduling," In *Proc. of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 258–267, 1993.

[Hwu et al., 1993]

W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The Superblock: An Effective Technique for VLIW and Superscalar Compilation," *The Journal of Supercomputing*, Vol. 7, No. 1, pages 229–248, January 1993.

[Johnson, 1991]

Mike Johnson *Superscalar Microprocessor Design* Prentice Hall, Englewood Cliffs, NJ, 1991.

[Johnson and Schlansker, 1996]

Richard Johnson and Michael Schlansker, "Analysis Techniques for Predicated Code," In *Proc. of the 29th Annual International Symposium on Microarchitecture*, pages 100–113, Paris, France, December 2-4 1996.

[Jones, 1991]

R. B. Jones *Constrained Software Pipelining* Master's thesis, Dept. of Computer Science - Utah State University, Logan, UT, 1991.

[Joupi and Wall, 1991]

N. P. Joupi and D. W. Wall, "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines," In *Proc. of 18th Annual International Symposium on Computer Architecture*, pages 34–42, 1991.

[Jourdan et al., 1995]

Stephan Jourdan, Pascal Sainrat, and Daneil Litaize, "Exploring Configurations of Functional Units in an Out-of-Order Superscalar Processor," In *Proc. of the 22nd Annual International Symposium on Computer Architecture*, pages 117–125, Santa Margherita Ligure, Italy, 1995.

[Kathail et al., 1994]

V. Kathail, M. Schlansker, and B. R. Rau, "HPL PlayDoh Architecture Specification: Version 1.0," Technical Report HPL-93-80, HP Laboratories, Compiler and Architecture Research, February 1994.

[Keller, 1975]

R. M. Keller, "Look-Ahead Processors," *Computing Surveys*, Vol. 7, No. 4, pages 177–195, December 1975.

[Kessler, 1999]

R. E. Kessler, "The Alpha 21264 Microprocessor," *IEEE Micro*, pages 24–36, April 1999.

[Knoop et al., 1994]

J. Knoop, O. Ruthing, and B. Steffen, "Partial Dead Code Elimination," In *Proc. of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 147–158, June 1994.

[Kurlander, 1996]

Steven M. Kurlander, "Approaches to Interprocedural Register Allocation," Technical Report CS-TR-96-1294, Computer Science Department of the University of Wisconsin - Madison, January 1996.

[Lam, 1987]

M. Lam *A Systolic Array Optimizing Compiler* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1987.

[Lam, 1988]

M. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," In *Proc. of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 318–327, Atlanta, 1988.

[Lam and Wilson, 1992]

Monica S. Lam and Robert P. Wilson, "Limits of Control Flow on Parallelism," In *Proc. of the 19th Annual International Symposium on Computer Architecture*, pages 46–57, Gold Coast, Australia, May 19-21 1992.

[Lavery, 1997]

Daniel M. Lavery *Modulo Scheduling for Control-Intensive General-Purpose Programs* PhD thesis, Dept. of Electrical Engineering, University of Illinois at Urbana-Champaign, 1997.

[Lavery and Hwu, 1996]

D. M. Lavery and W. W. Hwu, "Modulo Scheduling of Loops in Control-Intensive Non-Numeric Programs," In *Proc. of the 29th Annual International Symposium on Microarchitecture*, pages 327–337, December 1996.

[Lin, 1992]

David Chu Lin *Compiler Support for Predicated Execution in Superscalar Processors* Master's thesis, Dept. of Electrical Engineering, University of Illinois at Urbana-Champaign, 1992.

[Lipasti and Shen, 1997a]

Mikko H. Lipasti and John Paul Shen, "Approaching 10 IPC via Superspeculation," Technical Report CMU-CSC-97-1, Department of Electrical and Computer Engineering, Carnegie Mellon University, January 1997.

[Lipasti and Shen, 1997b]

Mikko H. Lipasti and John Paul Shen, "Exceeding the Dataflow Limit via Value Prediction," In *Proc. of the 29th Annual International Symposium on Microarchitecture*, December 1997.

[Lipasti and Shen, 1997c]

Mikko H. Lipasti and John Paul Shen, "The Performance Potential of Value and Dependence Prediction," In *Proc. of the EUROPAR-97*, Passau, Germany, August 1997.

[Lipasti et al., 1996]

Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen, "Value Locality and Load Value Prediction," In *Proc. of the Seventh International Conference on Architectural Support for Programming Languages and Operatings Systems (ASPLOS-VII)*, Boston, MA, October 1996.

[Mahlke, 1992]

Scott A. Mahlke *Design and Implementation of a Portable Global Code Optimizer* Master's thesis, Dept. of Electrical Engineering, University of Illinois at Urbana-Champaign, 1992.

[Mahlke, 1996]

Scott A. Mahlke *Exploiting Instruction Level Parallelism in the Presence of Conditional*

Branches PhD thesis, Dept. of Electrical Engineering, University of Illinois at Urbana-Champaign, 1996.

[Mahlke et al., 1993]

S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W. W. Hwu, B. R. Rau, and M. S. Schlansker, "Sentinel Scheduling: A Model for Compiler Controlled Speculative Execution," *ACM Transactions on Computer Systems*, Vol. 11, No. 4, November 1993.

[Mahlke et al., 1992a]

S. A. Mahlke, W. Y. Chen, W. W. Hwu, B. R. Rau, and M. S. Schlansker, "Sentinel Scheduling for VLIW and Superscalar Processors," In *Proc. of the ASPLOS-V Conference*, 1992.

[Mahlke et al., 1995]

Scott A. Mahlke, Richard E. Hank, James E. McCormick, David I. August, and Wen mei W. Hwu, "A Comparison of Full and Partial Predicated Execution Support for ILP Processors," In *Proc. of the 22nd Annual International Symposium on Computer Architecture*, pages 138–149, Santa Margherita, Italy, 1995.

[Mahlke et al., 1992b]

S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective Compiler Support for Predicated Execution using the Hyperblock," In *Proc. of the 25th International Symposium on Microarchitecture*, pages 45–54, 1992.

[McFarling, 1993]

S. McFarling, "Combining Branch Predictors," Technical Report TN-36, Digital Equipment Corp., Maynard, Mass., 1993.

[Müller, 1993]

Thomas Müller, "Emplyoing Finite Automata for Resource Scheduling," In *Proc. of the 26th Annual International Symposium on Microarchitecture*, pages 12–20, December 1993.

[Melvin et al., 1988]

S. Melvin, M. Shebanow, and Y. Patt, "Hardware Support for Large Atomic Units in Dynamically Scheduled Machines," In *Proc. of the 21th Annual Workshop on Microprogramming and Microarchitecture*, pages 60–66, San Diego, CA, December 1988.

[Moreno, 1996]

Jaime H. Moreno, "Dynamic Translation of Tree-Instructions into VLIWs," IBM Research Report RC 20505 (90881), IBM T.J. Watson Research Center, July 18 1996.

[Moreno et al., 1996]

J. H. Moreno, M. Moudgill, K. Ebcioglu, E. Altman, B. Hall, R. Miranda, S. K. Chen, and A. Polyak, "Architecture, Compiler and Simulation of a Tree-based VLIW Processor," IBM Research Report RC 20495, IBM T.J. Watson Research Center, July 9 1996.

[Moudgill et al., 1993]

Mark Moudgill, Keshav Pingali, and Stamatis Vassiliadis, "Register Renaming and Dynamic Speculation: An Alternative Approach," In *Proc. of the 26th International Symposium on Microarchitectures*, pages 202–213, 1993.

[Nair and Hopkins, 1997]

Ravi Nair and Martin Hopkins, "Expoliting Instruction Level Parallelism in Processors by Caching Scheduled Groups," In *Proc. of the 24th Annual International Symposium on Computer Architecture*, pages 13–25, Denver, CO, June 1997.

[Nicolau, 1985]

Alexandru Nicolau, "Percolation Scheduling: A Parallel Compilation Technique," Technical Report TR 85-678, Dept. of Computer Science - Cornell University, Ithaca, NY. 14853, May 1985.

[Nicolau and Novack, 1993]

Alexandru Nicolau and Steven Novack, "Trailblazing: A Hierarchical Approach to Percolation Scheduling," In *Proc. of the 1993 International Conference on Parallel Processing*, 1993.

[Papworth, 1996]

David B. Papworth, "Tuning the Pentium Pro Microarchitecture," *IEEE Micro*, pages 8–15, April 1996.

[Park and Schlansker, 1991]

J. C. H. Park and M. S. Schlansker, "On Predicated Execution," Tech. Rep. HPL-91-58, HP Laboratories, Palo Alto, CA, May 1991.

[Popescu et al., 1991]

V. Popescu, M. Schultz, J. Spracklen, G. Gibson, B. Lightner, and D. Isaman, "The Metaflow Architecture,," *IEEE Micro*, pages 10–13, 63–73, June 1991.

[Rau, 1993]

Bob R. Rau, "Dynamically Scheduled VLIW Processors," In *Proc. of the 26th International Symposium on Microarchitectures*, pages 80–90, 1993.

[Rau, 1994]

Bob R. Rau, "Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops," In *Proc. of the 27th International Symposium on Microarchitectures*, pages 63–74, December 1994.

[Rau and Fisher, 1993]

Bob. R. Rau and Joseph. A. Fisher, "Instruction-Level Parallel Processing: History, Overview, and Perspective," *Journal of Supercomputing*, Vol. 7, No. 1, pages 9–49, January 1993.

[Rau and Glaeser, 1981]

B. R. Rau and C. D. Glaeser, "Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing," In *Proc. of the Fourteenth Annual Workshop on Microprogramming*, pages 183–198, 1981.

[Rau et al., 1992]

B. R. Rau, P. P. Tirumalai, and M. S. Schlansker, "Register Allocation for Software Pipelined Loops," In *Proc. of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 283–299, 1992.

[Rau et al., 1989]

B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The Cydra-5 Departmental Supercomputer : Design Philosophies, Decisions, and Trade-offs," *IEEE Computer*, Vol. 22, pages 12–35, January 1989.

[Schlansker et al., 1994]

M. Schlansker, V. Kathail, and S. Anik, "Height Reduction of Control Recurrences for ILP Processors," In *Proc. of the 27th International Symposium on Microarchitecture*, pages 40–51, December 1994.

[Schlansker et al., 1997]

M. Schlansker, B. R. Rau, S. A. Mahlke, V. Kathail, R. Johnson, S. Anik, and S. G. Abraham, "Achieving High Levels of Instruction-Level Parallelism with Reduced Hardware Complexity," Technical Report HPL-96-120, HP Laboratories, Compiler and Architecture Research, February 1997.

[Schlansker and Kathail, 1995]

M. S. Schlansker and V. Kathail, "Critical Path Reduction for Scalar Programs," In *Proc. of the 28th Annual International Symposium on Microarchitecture*, pages 57–69, November 1995.

[Sethi, 1975]

R. Sethi, "Complete Register Allocation Problems," *SIAM Journal of Computing*, Vol. 4, pages 226–248, March 1975.

[Silberman and Ebcioglu, 1993]

Gabriel M. Silberman and Kemal Ebcioglu, "An Architectural Framework for Supporting Heterogeneous Instruction-Set Architectures," *Computer*, Vol. 26, No. 6, pages 39–56, June 1993.

[Sites and et. al., 1993]

R. Sites and et. al., "Binary Translation," *Communications of the ACM*, Vol. 36, No. 2, pages 69–81, 1993.

[Smith and Pleszkun, 1985]

James E. Smith and Andrew R Pleszkun, "Implementation of Precise Interrupts in Pipelined Processors," In *Proc. of the 12th Annual International Symposium on Computer Architecture*, pages 36–44, June 1985.

[Smith, 1997]

Michael D. Smith *Extending SUIF for Machine-specific Optimizations* Harvard University, May 1997.

[Smith et al., 1992]

Michael D. Smith, Mark A. Horowitz, and Monica S. Lam, "Efficient Superscalar Performance through Boosting," In *Proc. of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 248–259, 1992.

[Smith et al., 1989]

Michael D. Smith, Mike Johnson, and Mark A. Horowitz, "Limits on Multiple Instruction Issue," In *Proc. of the Third International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 290–302, 1989.

[Smith et al., 1990]

Michael D. Smith, Monica S. Lam, and Mark A. Horowitz, "Boosting Beyond Static Scheduling in a Superscalar Processor," In *Proc. of the 17th Annual International Symposium on Computer Architecture*, pages 344–354, 1990.

[Su and Wang, 1991]

B. Su and J. Wang, "GURPR*: A New Global Software Pipelining Algorithm," In *Proc. of the 24th Annual Workshop on Microprogramming and Microarchitecture*, pages 212–216, November 1991.

[SUIF, 1994]

SUIF *The SUIF Library* Stanford Compiler Group - Stanford University, 1994.

[Takagi et al., 1991]

H. Takagi, T. Arita, and M. Sowa, "A Static Scheduling Algorithm for the Ultimate Barrier," In *1991 Summer United Workshops on Parallel, Distributed and Cooperative Processing, CPSY-91-15 (in Japanese)*, pages 91–98, 1991.

[Tam et al., 1997]

E. S. Tam, J. A. Rivers, and E. S. Davidson, "Flexible Timing Simulation of Multiple Cache Configurations," Technical Report CSE-TR-348-97, University of Michigan, November 1997.

[Thorton, 1964]

J. E. Thorton, "Parallel Operation in the Control Data 6600," In *Proc. of the Fall Joint Computer Conference 26*, pages 33–40, 1964.

[Tomasulo, 1967]

R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development*, Vol. 11, No. 1, pages 25–33, 1967.

[Warter et al., 1992]

N. J. Warter, G. E. Haab, and J. W. Bockhus, "Enhanced Modulo Scheduling for Loops with Conditional Branches," In *Proc. of the 25th Annual International Symposium on Microarchitecture*, 1992.

[Warter et al., 1993]

N. J. Warter, S. A. Mahlke, W. W. Hwu, and B. R. Rau, "Reverse if-conversion," In *Proc. of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 290–299, 1993.

[Weiss and Smith, 1994]

Shlomo Weiss and James E. Smith *POWER and PowerPC* Morgan Kaufmann, San Francisco, CA, 1994.

[Yeager, 1996]

Kenneth C. Yeager, "The MIPS R10000 Superscalar Microprocessor," *IEEE Micro*, April 1996.

[Yeh, 1993]

T.-Y. Yeh *Two-level Adaptive Branch Prediction and Instruction Fetch Mechanisms for High Performance Superscalar Processors* PhD thesis, EECS Department, University of Michigan, Ann Arbor, 1993.

[Young and Smith, 1996]

Cliff Young and Michael D. Smith, "Branch Instrumentation in SUIF," In *Proc. of the Second SUIF Compiler Workshop*, Stanford University, CA, January 1996.

[Young, 1998]

R. C. Young *Path-based Compilation* PhD thesis, Harvard University, Division of Engineering and Applied Sciences, January 1998.