EXPLOITING SHORT-LIVED VARIABLES IN SUPERSCALAR PROCESSORS

by
Luis Alfonso Lozano C.

School of Computer Science
McGill University, Montreal

January 1995

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

ISBN 0-612-05585-X

Abstract

Modern superscalar processors use advanced features like dynamic scheduling and speculative execution to exploit fine-grain parallelism. In order to support these features, they use complex hardware mechanisms like reorder buffers, instructions windows and renaming buffers. In this thesis, we have made an observation about the use of these mechanisms: a significant number of program variables are short-lived in the sense that their whole live ranges occur entirely within the reorder buffer. Therefore, the values produced by these short-lived variables do not need to be written back (committed) to the register file. Based on this observation, we have proposed a compiler analysis, which we call *short-live-range analysis*, and a simple architecture feature to avoid the useless commits of the values generated by these short-lived variables. Moreover, we have proposed a new register allocation scheme to assign these variables to the locations provided for register renaming (rather than to the register file), thus decreasing the register pressure.

We have implemented this hardware/software codesign scheme using the McCAT testbed and the simulation results show: (1) the short-live-range analysis and the proposed architecture feature can be successfully used to avoid the useless commit of instructions to the register files; (2) the above mechanism can reduce the number of write ports to the register files without affecting performance; (3) the allocation of short-lived variables to the locations provided by the register renaming mechanism can significantly reduce the introduction of spill code and improve the overall performance.

Résumé

Les machines superscalaires modernes requièrent des mécanismes élaborés tels que l'ordonnancement dynamique et l'exécution spéculative afin d'exploiter au mieux le parallélisme à grain fin. Pour cela, elles s'appuient sur des techniques matérielles complexes telles que les tampons de réordonnancement, les fenêtres d'instructions, le renommage des registres. Dans le cadre de ce mémoire, nous nous sommes attachés à l'étude de ces mécanismes : il s'est avéré qu'un nombre significatif des variables d'un programme a une durée de vie courte, dans ce sens que leur vie entière se déroule dans le tampon de réordonnancement. Les valeurs produites alors par ces variables de courte activité n'ont pas besoin d'être inscrites dans les registres. A partir de cette observation, nous avons proposé une technique de compilation que nous avons appelée "analyse des variables de courte activité" ainsi qu'un mécanisme matériel destiné à éviter les écritures inutiles des valeurs produites par ce type de variables. De plus, nous avons proposé un nouveau schéma d'allocation de registres, afin d'attacher ces variables aux emplacements de renommage des registres (plutôt que dans les registres eux mêmes), afin de réduire la pression sur ces registres.

Nous avons implémenté ces mécanismes, à la fois logiciels et matériels, à l'aide du banc de test McCAT. Les simulations ont donné les résultats suivants : (1) l'analyse des variables de courte activité et les mécanismes matériels proposés ont démontré qu'ils pouvaient réduire de manière significative le nombre d'écritures dans les registres; (2) ces mêmes mécanismes permettent encore de diminuer le nombre de ports d'écriture vers les registres sans dégrader les performances; (3) l'allocation des variables de courte activité dans les emplacements de renommage permet de supprimer beaucoup de code de vidage et d'ainsi améliorer la performance globale.

To my parents: Joaquin and Virginia

Acknowledgments

There are occasions in your life when you want to express something and you just cannot find the right words to use. I think that writing these acknowledgments is one of them. There are no words to express my gratitude to all the people who helped me to pursue my Master's degree.

First, I would like to thank my thesis supervisor, Dr. Guang Gao. Since I came to Mcgill, I saw in Dr. Gao an inspiring lecturer and an infatigable researcher. He always trusted me and listened to my ideas. He encouraged me to go for new goals and to accept difficult challenges. He also supported me for a long period of my studies.

I also thank Dr. Laurie Hendren. Laurie (as everyone calls her) introduced me to the world of compilers. She is one of the best lecturers I have ever seen. She assigned very interesting projects to me and was always willing to help me solve research problems.

There are no words to thank Claudia Pateras. She gave me all her love and support. She spent long hours helping me write this thesis and making sure I was not going to go insane. I hope I can give her back all the time that she gave me. Claudia, wherever we are we will always be *together*. I am also very thankful to the Pateras family. They opened their home to me and let me share with them their happiest moments. Mrs. Pateras, thanks for all those times you sent me your delicious dishes while I was working late at school.

There is no doubt I could not have achieved this goal without the help of my family. My parents saw my eagerness to look for new goals and were willing to do whatever they could to help me reach them. During my stay in Montreal my whole family constantly encouraged me and sent me their love. They always made me feel close to my beloved Colombia. During the difficult times, they gave me strength to continue and, although I was away, we became closer than ever. I will never forget the beautiful letters I received from them, the long conversations I had with my father through computer "talks" and the small drawings and letters from my nephews and nieces. My mother always helped me and took good care of me. She was always present, she will always keep our family together.

I am a man of friends, and I am proud to say that I have many of them. Some of them helped me a lot during my Master's studies. Thanks to my new friends in Montreal: Lucie, Sandro, Mary, William, Alain. They gave me plenty of good moments to enjoy and took me away from my long periods of study. Thanks to my friends from the ACAPS group: Ana, Rakesh, Chris, Nasser, Cecile, Maryam, Sreedhar. They gave me crazy times and helped me stand the moments of pressure. Each of us with different accents, all of us with similar dreams. Thanks to my other friends in North America: Claudia, Mark & Lisa, Robert, Adriana. Claudia, my true friend. Mark, always doing crazy things for me. Robert, my "adoptive" brother. Thanks to my friends in Colombia: John, Jose German and Jaime. They were always willing to help me and my family. Many mountains are waiting for us!

Finally, I should thank two very special people in the School of Computer Science: Lorraine and Lise. They always took good care of all my problems and had a huge smile for me every day.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Vitae summa brevis spem nos vetat incohare longam.
Life's short span forbids us to enter on far-reaching hopes.
Horace

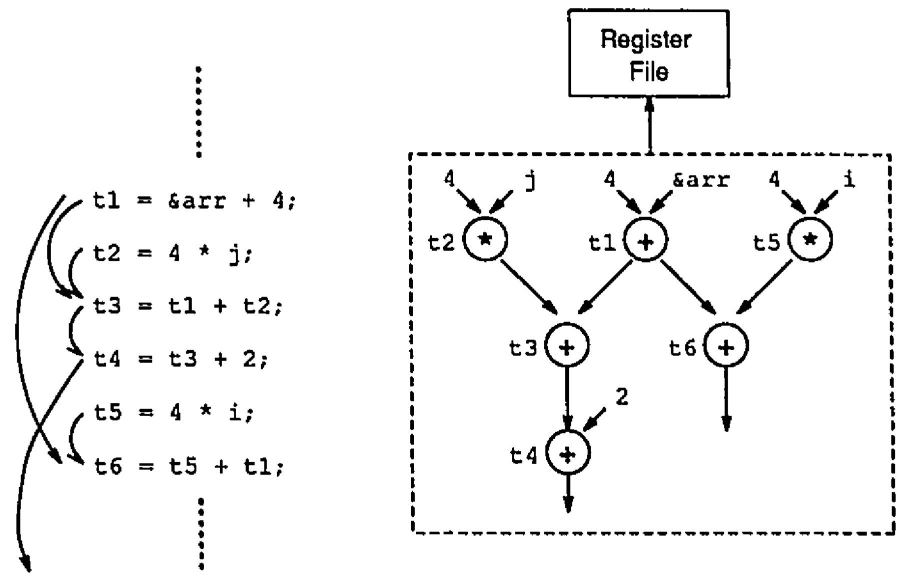
In today's pursuit of high performance computing, processors base their performance not only on high clock rates but also on the number of instructions that can be executed in parallel each cycle. Currently, the most used architecture paradigm in the design of commercial processors is the superscalar paradigm. Examples of modern superscalar machines include the IBM RISC-6000 architecture [BW90], the Motorola 88110 [DA92], the Intel Pentium [AA93], the Hewlett-Packard PA7100 [AAD+93], the TFP [Hsu94] and the PowerPC [DOH94, SD94].

In the superscalar paradigm, parallelism is extracted from code written for a sequential machine using aggressive techniques like dynamic scheduling [Tho64, Smi89], speculative execution [LS84] and register renaming [Kel75]. There are two important aspects to note about the use of these features. First, although these features have been used successfully to uncover the parallelism found in sequential code, the hardware complexity of these mechanisms is very high and, therefore, they are very difficult to implement. Some of these mechanisms require, for example, very wide datapaths and complicated forwarding capabilities. Second, in most cases, the structure of these mechanisms has remained hidden from the compiler. That is, although these features provide powerful schemes to extract parallelism, the compiler is not able to interact with them. Thus, for example, the compiler is not aware of mechanisms like the reorder buffer [SV87] that are used to support these features.

We believe that, in order to continue the current rate of performance growth of RISC processors, it is necessary to expose the structure of these complex mechanisms to the compiler. The compiler can help reduce the complexity of implementation of some of these features and use them more efficiently in order to increase the performance of the machine. We believe that the trend in computer architecture should be towards an increasingly closer relationship between the compiler and the hardware in which the compiler collects information about the program to be executed and passes this information to the hardware. By using the information provided by the compiler, the hardware can be designed more efficiently so as to reduce its complexity. Moreover, this information can also be used by the processor to take better decisions at run time and increase the performance of the model. Besides counting on the information provided by the compiler, the processor should also be powerful enough to be able to make decisions based on events that cannot be anticipated by the compiler such as cache misses.

This relationship between the compiler and the hardware has many facets. In this thesis, we examine one of these facets based on one observation about the behavior of dynamically scheduled superscalar processors. In order to provide dynamic scheduling, processors build, at run time, the data flow graph for a small window (set) of instructions. For each value produced by an instruction, two actions are taken by the current mechanism: first, the value is forwarded to the other instructions that use this value and that are found in the same window of instructions being analyzed; second, the value produced is retired at commit time to the register file from where succeeding instructions that make use of it will be able to fetch it. Our observation is that a large percentage (often more than 90%) of the values produced by the instructions in the program are "short-lived" in the sense that the producer and the last consumer of a value are found very close together in the instruction stream. This implies that there is a big chance that the producer and last consumer of a value are found in the same window of instructions while the superscalar processor is building the dataflow graph for them. In that case, when the value is produced it will be forwarded to *all* the possible consumers of the value. Therefore, when the value is retired to the register file, there will not be any subsequent use of this value. We maintain that, since the value has been forwarded to all the uses before being retired to the register file, there is no need to commit this value to the register file.

This observation can be illustrated with the small example given in Figure 1.1. As it



```
t1 = &arr + 4;
t2 = 4 * j;
t3 = t1 + t2;
t4 = t3 + 2;
t5 = 4 * i;
t6 = t5 + t1;
```

(a) Program fragment

(b) Dataflow graph and retirement of values to the register file

Figure 1.1: Building the dataflow graph for a set of short-lived variables

can be seen in the figure, the values produced by the instructions in the small program fragment are consumed by instructions that follow closely in the instruction stream (with the exception of the value produced for `t4`). Therefore, it is possible that the instructions in this program fragment fit in the same window of instructions while the processor is dynamically building the dataflow graph for the them. In this case, the values produced for `t1`, `t2`, `t3`, and `t5` are forwarded to all their consumers in the window thus making unnecessary the commit of these values to the register file. On the other hand, the value produced for `t4` has to be retired to the register file so that its last consumer can find it there.

As a consequence of this observation, there is another point we make that is related to the use of the register renaming mechanism provided by the type of superscalar processors we are considering. Since we propose that many of the values produced should not be retired to the register file, we note that it is not necessary to assign to these values names from locations in the register file. That is, since these values are never going to reside in the register file, why should we allocate them to the register file? In order to increase instruction level parallelism, superscalar processors implement register renaming to avoid output and anti dependencies. To support this, the register renaming mechanism provides a new location and a different name for each value produced by the instructions in the window being analyzed. We have noticed that, for those values which do not need to be retired to the register file, the processor only refers to them by the name dynamically assigned by the renaming mechanism and not by the name originally assigned to them by the register allocator (the name of a physical register).

The problem with the current superscalar implementations is that the values produced by the short-lived variables are, in fact, committed to the register file. This constitutes an unnecessary runtime overhead and increases the number of required hardware resources, such as the number of register file ports. Also, the conventional compiling schemes for register allocation do not give special treatment to these short-lived values. This yields to inefficiencies and increases the number of physical registers required for the allocation of the variables in the program.

1.1 Thesis Contributions

Based on these observations, we have proposed in this thesis the codesign of a hardware/software scheme to exploit the occurrence of short-lived variables. With this scheme, we aim to reduce the complexity and increase the performance of dynamically scheduled superscalar processors.

We have designed a compile-time analysis method (called short-live-range analysis) and a simple architecture feature to avoid the useless commit of the short-lived variables. Moreover, we propose a novel register allocation scheme by which the compiler can directly assign the short-lived variables to the spaces provided by the register renaming mechanism rather than the physical registers.

We have implemented our scheme on McCAT – The McGill Compiler-Architecture Testbed – which includes an optimizing C compiler [HDE+92] and a cycle-by-cycle superscalar simulator [Mou93]. Our simulation results have demonstrated that:

- The proposed short-live-range analysis can successfully capture most of the short-lived variables: on the average between 80% and 91% of the variables are detected to be short-lived. Furthermore, the combined architecture and compiler method can effectively make use of such analysis and eliminate a great majority of useless writes to the register files: on the average between, approximately, 76% and 89%.

- The mechanism devised to avoid the useless commits of instructions can be used to reduce the number of write ports to the register files without affecting performance. Only a 1% performance loss is detected after reducing the number of write ports to the register file from 4 to 1 for the target superscalar architecture we are studying.

- The proposed method of allocation of short-lived variables to locations provided for register renaming in the reorder buffer can reduce the number of registers needed and, consequently, the amount of spill code introduced by the register allocator thus substantially improving execution time. For instance, the average improvement when the register pressure is high is significant: between, approximately, 15% and 26%.

1.2 Thesis Outline

This thesis is organized as follows:

- In Chapter 2 we provide the necessary background for the understanding of the problems addressed in this thesis and of the proposed solutions. We briefly describe the concepts of Instruction Level Parallelism (ILP) and the architectures to exploit ILP. We focus our attention on superscalar processors and describe the different features used by these processors to expose and exploit instruction level parallelism. Based on the presented concepts, we introduce the execution model in which the discussions in this thesis are going to be based. In the second part of this chapter, we give the background information for the problem of register allocation. This information is necessary to understand our scheme for allocation of short-lived variables.

- Chapter 3 describes in more detail our observation about the useless commit of short-lived values to the register file. We present a motivating example and give experimental evidence of the occurrence of this phenomenon. The problems to be studied and the solution strategy are also stated in this chapter.

- Chapter 4 describes the combined hardware/software mechanism used to reduce the number of useless commits. We describe the possible hardware mechanisms, the compiler analysis and the collaboration between the hardware and the compiler in order to solve this problem.

- In Chapter 5 we describe our scheme for the allocation of the short-lived variables. We describe the advantages of our scheme over the traditional register allocation scheme and we discuss the effects of the introduction of spill code on our method.
- Chapter 6 presents the results of the different experiments performed to show the advantages of the proposed solutions. The benchmarks used and the base superscalar model used for the simulations are introduced.
- In Chapter 7 we briefly summarize the works from other authors that are related to our research. We describe some studies in which the authors have exposed the structure of different hardware mechanisms to the compiler in order to reduce the hardware complexity and increase the performance of their models.
- Finally, in Chapter 8, we summarize our achievements and give directions for future research.

Chapter 2

Background

In this chapter we present the background information required for the understanding of the problems addressed in this thesis. The chapter is split into two sections. The first of them briefly describes the concepts related to Instruction Level Parallelism and the architectures developed to exploit this kind of parallelism. We focus our attention on superscalar processors, present the tasks to be accomplished by this type of processors and introduce some of the hardware mechanisms designed to accomplish these tasks. We also discuss some issues involved in the implementation of this kind of processors. At the end of this section, we present the execution model to be used throughout the rest of the thesis and we briefly review how this execution model works. The reader interested in Instruction Level Parallelism can find a more general overview of the field in [FR91, RF93]. An excellent and exhaustive study on superscalar processors can be found in [Joh91].

The second section of this chapter briefly presents the concepts related to the problem of global register allocation. We review how this problem is mapped to the problem of graph coloring and we describe the heuristic developed by Gregory Chaitin [CAC+81] to solve this problem. We also discuss the advantages and disadvantages of this heuristic. At the end of the section, we describe an improvement to Chaitin's heuristic developed by Preston Briggs [Bri92]. This heuristic is the one used by the register allocator developed for the McCAT compiler.

2.1 Instruction Level Parallelism and Architectures to Exploit ILP

Instruction Level Parallelism (ILP) is a set of processor features and compiler optimizations aimed at increasing the performance of the processor by executing multiple instructions in parallel. The concept of ILP has been exploited by supercomputers and mainframes since the early 60's. It has gained large popularity since its implementation at the microprocessor level and its use in the workstation and the personal computing market. Part of this popularity is due to the fact that ILP can be implemented without affecting the programmer's model. That is, ILP may be used to exploit the parallelism found in programs written to be executed in a sequential machine. This is a direct advantage over other forms of parallelism where the programmer has to specifically describe how parallelism can be exploited.

The amount of parallelism that can be extracted from sequential code is limited by the control and data dependencies found in it. Different compiler and hardware techniques can be used to expose parallelism and make a more efficient use of the resources provided by the processor. At the compiler level, techniques such as loop unrolling, software pipelining and dependence analysis can be used to reduce the dependencies and produce code with more possibilities to exploit parallelism. Techniques like register renaming, dynamic scheduling, and branch prediction can be used at the hardware level to avoid false dependencies and take decisions at run time to increase the number of instructions that can executed in parallel.

Diverse architectures have been proposed to exploit ILP. They can be classified according to whether the decision of which instructions are executed in parallel is taken by the compiler or the hardware, and according to, in the case the decisions are taken by the compiler, how these decisions are communicated to the hardware via the program. The corresponding classification of ILP architectures is as follows [RF93]:

- *Sequential architectures:* In this type of architectures, the hardware decides which instructions are executed in parallel and the program does not give any information to the processor in this respect. Superscalar machines are an example of sequential architectures.
- *Dependence architectures:* In these architectures, the compiler determines the dependencies between instructions and communicates them to the hardware. Typically this is done by specifying, for each operation, the list of other operations that are

dependent on it. The hardware makes the scheduling decisions according to the information provided. Dataflow architectures are representative of this category.

- *Independence architectures:* In these architectures, the compiler determines which instructions can be executed in parallel, and informs the architecture via the program. *Very Long Instruction Word* (VLIW) processors are examples of independence architectures. In the typical implementation of this type of architectures, the hardware does not take any scheduling decisions.

Several processors have been implemented for each kind of architecture. Of special interest for this thesis are the machines that evolved from the pipelined RISC processors since these are prevalent in commercial machines nowadays. As dataflow processors are beyond the scope of this thesis, we will not make any further references to them.

Starting from a conventional pipelined machine there are several ways a processor can be modified to be able to execute several instructions simultaneously. Figure 2.1 depicts their pipelined execution and compares them against the typical single pipelined machine (Figure 2.1(a)).

- *Superpipelined processors:* In a superpipelined machine, the number of stages in the pipeline is increased thus allowing the clock length to be reduced. The clock length is set to a fraction of the latency of a single functional unit. For example, a superpipelined machine of degree 3 (Figure 2.1(b)) will have a clock length that is 1/3 the latency of the simplest functional unit, and since it can issue one instruction per cycle will have several instructions in the execute stage simultaneously.
- *Superscalar processors:* On a superscalar machine, multiple instructions are decoded, issued and executed each cycle. For instance, in a superscalar processor of degree 3 (Figure 2.1(c)), up to 3 instructions can be issued each cycle.
- *VLIW processors:* In a *Very Long Instruction Word* machine, each instruction consists of a set of *operations* that can be executed simultaneously. Therefore, in a VLIW processor of degree 3, up to 3 operations can be specified in each instruction (Figure 2.1(d)).

2.1.1 Superscalar Processors

Superscalar machines have become the standard type of processor implementation for the modern general purpose microprocessors. This kind of machines are considered

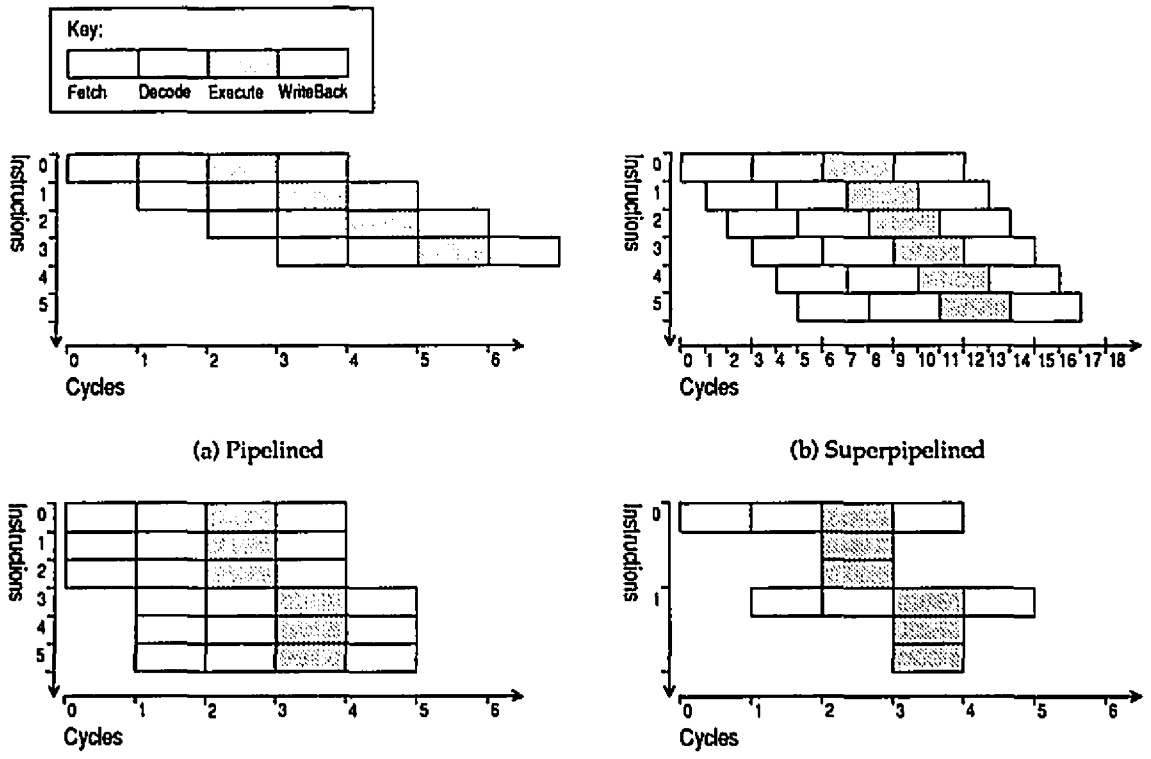


Figure 2.1: Different processors to exploit ILP

the natural next step in the evolution of RISC architectures [Joh91]. The main reason for their success resides in their ability to extract parallelism from code produced for sequential machines. They maintain the binary code compatibility thus preserving the existing software base.

In order to extract parallelism from sequential code, a superscalar processor must be able to perform the following tasks in hardware:

- Fetch and decode several instructions per cycle.
- Detect the dependencies between the instructions in the instruction stream.
- Find several independent instructions to be executed in parallel.
- Schedule the independent instructions and assign them to the available resources.

Various hardware mechanisms have been designed to accomplish these tasks. The compromise is always to try to maximally exploit parallelism while keeping the hardware complexity low. As we will briefly discuss in Section 2.1.2, this objective is not always reached.

The first of the aforementioned tasks, fetching and decoding several instructions per cycle, is critical to the performance of the processor. The number of instructions that can be executed simultaneously is obviously limited by the number of instructions fetched and decoded each cycle. One important limitation of the processor to be able to fetch several instructions per cycle is the occurrence of branches in the instruction stream. Branches limit the number of instructions that can be fetched per cycle because the fetcher has to stall while waiting for the branch to be executed in order to be able to continue bringing target instructions from memory. A technique called *branch prediction* can be used to solve this problem. Branch prediction allows the processor to continue fetching instructions by predicting the outcome of the branch. A number of hardware mechanisms have been designed to provide prediction that go from simple static prediction mechanisms [HP90], to more complex and accurate dynamic branch predictors that achieve up to 97% accuracy [YP93, LS84]. Since the prediction is not always correct, the processor has to provide a mechanism to undo the effect of instructions executed in a wrongly predicted path. Several techniques used to this effect are described later in this section.

Once the instructions have been fetched, the processor has to establish the dependencies between them. As mentioned before, data dependencies are one of the fundamental limits to instruction parallelism. There are two ways the hardware can ensure that data dependencies are respected: by stopping the instruction from being decoded, and by stopping the instruction from being issued. Different mechanisms have been designed to enforce the dependencies in these two ways. The simplest mechanisms enforce the in-order issue of the instructions by using a *scoreboard* to stop the decoder whenever a dependency is found [DA92, McG90]. The scoreboard detects the dependencies by keeping track of registers that will be updated by the instructions currently in execution. However, this mechanism is too restrictive and provides few chances to exploit the parallelism found in the code. A more aggressive use of the *scoreboard* [Tho64] stalls the decoder only when output dependencies occur. This mechanism avoids anti-dependencies by making copies of the operands for each instruction, and enforces the flow dependencies by putting the dependent instructions in an *instruction window* thus avoiding stalling the decoder. From this window, instructions will be issued once their operands have been calculated. *Register renaming* [Kel75] can be used to avoid stalling the decoder in the presence of output dependencies. In order to implement register renaming the hardware has to provide additional internal registers so that each value generated gets allocated to a different location. Providing the space for these new locations and the logic to associate these locations to the instructions makes the hardware even more complex.

By using scoreboarding or register renaming in conjunction with an instruction window, the processor is capable of issuing the instructions to the functional units out of the original program order. The out-of-order issue capability is very important to increase parallelism. The processor selects from the window the instructions that are independent, and dynamically schedules them according to the availability of their operands and the availability of the functional units. The amount of independent instructions that can be selected depends on the size of the instruction window. Basically, the processor is building the dataflow graph [DFL72] for the instructions in the window. Therefore, more parallelism is achieved by using large instruction windows. However, in order to keep the window full, the hardware requires excellent branch prediction techniques. Different mechanisms have been designed to provide the processor with this dynamic scheduling capability. Decentralized schemes buffer the instructions separately for each functional unit. In Thornton's scoreboard mechanism, each functional unit has its own instruction buffers. In the *reservations stations* mechanism [Tom67], the instructions and their operands are kept in stations associated to each functional unit. On the other hand, centralized schemes have one instruction window for all the functional units. The *Dispatch Stack* [AKC86], the *Register Update Unit – RUU* [SV87] and the *Deferred-Scheduling Register-Renaming Instruction Shelf – DRIS* [PSS+91] are examples of centralized mechanisms. The use of a central window can provide better performance because better scheduling decisions can be taken when the instructions are in the same window. However, centralized windows are more difficult to implement.

As mentioned before, the use of branch prediction requires mechanisms that allow the speculative execution of instructions while ensuring the correct execution of the program. In case the processor executes instructions from a wrongly predicted branch, the mechanism should be able to undo the effect of these instructions. Techniques like the *history buffer*, the *reorder buffer*, the *future file* (all described in [SP88]), and *checkpointing* [HP87] have been proposed to this effect. A comparison of the performance of these mechanisms can be found in [BP93]. For this thesis, we are particularly interested in the reorder buffer since this is the mechanism assumed in our execution model and whose characteristics we are going to exploit. The reorder buffer is a FIFO structure that provides additional storage for the results of the instructions executed out-of-order and ensures the in-order update of the register file. Instructions enter the reorder buffer in program order, are executed out-of-order and, when completed, their results are put in the FIFO instead of the register file. The values are copied, or *committed,* to the register file only when the instruction reaches the head of the reorder buffer. If a branch prediction happens to be incorrect, the instructions that are found after the branch are flagged as invalid so when they reach the head of the FIFO they are discarded. In this way, the correct state of the register file is preserved.

The reorder buffer and all the other state maintenance mechanisms are also used to provide precise interrupts [SP88, HP87]. Interrupts are events that disturb the normal execution of the program like a page fault or a hardware error. An interrupt is precise if the state of the processor when the interrupt is handled corresponds to that of the sequential model of the same processor. The out-of-order execution of instructions makes the problem of providing precise interrupts more difficult. The difficulty lies in restoring the processor to the state where all the instructions preceding the faulting instruction have updated the machine state, and none of the succeeding instructions have modified it. This can be accomplished with the help of the reorder buffer by delaying the processing of the interrupt until the point when the faulting instruction reaches the head of the reorder buffer. At this point, all the instructions preceding the interrupt have modified the processor state, the instructions remaining in the reorder buffer can be discarded and the interrupt can be handled.

Another important issue for superscalar processors is to be able to deal with dependencies between instructions that access memory. A memory instruction is dependent on another when the location accessed by both is the same. The simplest way to deal with this kind of dependency is by simply serializing the accesses to memory. A more aggressive solution relaxes the ordering of accesses to memory by allowing the loads to bypass preceding stores whenever possible. Since stores modify the state outside the processor, they have to be issued in strict order with respect to other instructions. However, loads do not need to be issued in order and can be issued before other instructions if there are no dependencies. To accomplish this, stores can be buffered and only sent to memory after *all* the preceding instructions have been committed. Loads can be issued as soon as they are decoded and can bypass the stores in the buffer if there is no address conflict. A further improvement consists in forwarding the values from buffered stores to succeeding loads thus avoiding the loads' accesses to memory. A complete description of *load bypassing* and *load forwarding* can be found in [Joh91].

2.1.2 Complexity Issues

In the previous section we briefly described the problems faced by superscalar processors to extract parallelism from sequential code, and we mentioned some of the techniques used to overcome these problems. Although these techniques are effective to increase the number of instructions that can be executed in parallel, they are far from being simple to implement in a single chip processor. The complexity of the hardware for a superscalar processor depends mainly on the number of uncompleted instructions permitted, the width of the decoder, the mechanism used to restart after a misspredicted branch and the necessity of forwarding results to waiting instructions. Besides the difficulties of implementing each of the major features mentioned, the design of the processor also has to deal with the difficulties caused by the interdependencies between them. The result of all of this is a very complex hardware with long logic delays.

Other than the complexity of implementing a feature, it is also very important to consider the effect that introducing this feature has on the clock cycle length. The overall performance of the processor depends both in the number of instructions executed per cycle and on the clock cycle length. The success of a hardware technique in increasing parallelism can be undermined if the implementation of the mechanism badly affects the clock length. The importance of this tradeoff can be better exemplified by the two current tendencies in superscalar processor implementations. One tendency is towards simpler hardware lacking some of the mechanisms described in the previous section, but attaining high clock speed. The other tendency is towards complex hardware implementing several of the mechanisms described to exploit ILP, but attaining a much slower clock speed [SW94, WHKM93a, WHKM93b]. Although the two tendencies are different in nature, their performance is still comparable and it is unclear at the moment which tendency will predominate.

One of the proposals of this thesis is that the complexity of implementation of some of the superscalar features previously described can be reduced with the help of the compiler. In particular, we are going to address the problem of how to reduce the complexity of the register file and the data path by modifying the compiler. The complexity of the register file is an important issue in architectures that exploit instruction level parallelism. In fact, the inability to build register files with a large number of ports is considered one of the major bottlenecks in realizing an ideal VLIW machine [CDN92]. A processor that is able to issue N instructions per cycle requires a register file with at least $2 \times N$ read ports and N write ports in order to attain peak performance. In the near future, we expect to have superscalar machines able to issue six or eight instructions per cycle. This implies that we will require register files with at least 24 ports. To the best of our knowledge, it is very difficult to build register files with that number of ports [Jol91], and even if they are realizable, they may seriously affect the length of the clock cycle. The chip real-state of a multi-ported register file is proportional to the product of the number of read ports and the number of write ports. Moreover, the access time can be modeled as a logarithmic function of the number of read ports. Hence, it is important to keep the number of ports small in order not to affect the clock cycle. The problem gets even more complicated when we consider that several of the reads/writes can be accessing/modifying the same register each cycle. For these reasons, processors that are able to issue a large number of instructions per cycle partition the register file and the functional units in clusters in which a functional unit can only access the values stored in the register file found in its own cluster [Fis83, FERN84]. In this way, the number of ports for each register file is smaller. However, when necessary, values must be explicitly moved between clusters using a slower communication channel. As a consequence, the compiler has to deal with the problem of intelligently distributing the computation in such a way that the communication between clusters is minimized [CDN92].

To deal with the problem of the complexity of the register file, modern superscalar processors limit the number of register ports and put restrictions on the number of instructions that can be committed each cycle. One interesting approach is adopted in [SD94], where the designers allow the commitment of the instructions from two stages of the pipeline to compensate for the effect of the reduced number of ports.

2.1.3 The Superscalar Processor Execution Model

Figure 2.2 shows the structure of a generic superscalar processor model [Joh91, Mou93] that will be adopted as our execution model. It is very important, for the sake of the comprehension of the different mechanisms discussed in this thesis, to briefly describe how this model works. The superscalar processor model has two important components: the instruction (dispatch) window and the reorder buffer. The instruction window serves as a pool of instructions from where the instructions that are ready to execute are issued to the functional units. The reorder buffer is a FIFO structure which ensures that instructions, upon leaving the buffer, modify the register file in program order and provides the mechanism to support speculative execution and precise interrupts. Although in Figure 2.2 the reorder buffer and the instruction window are drawn as two separate elements, they could

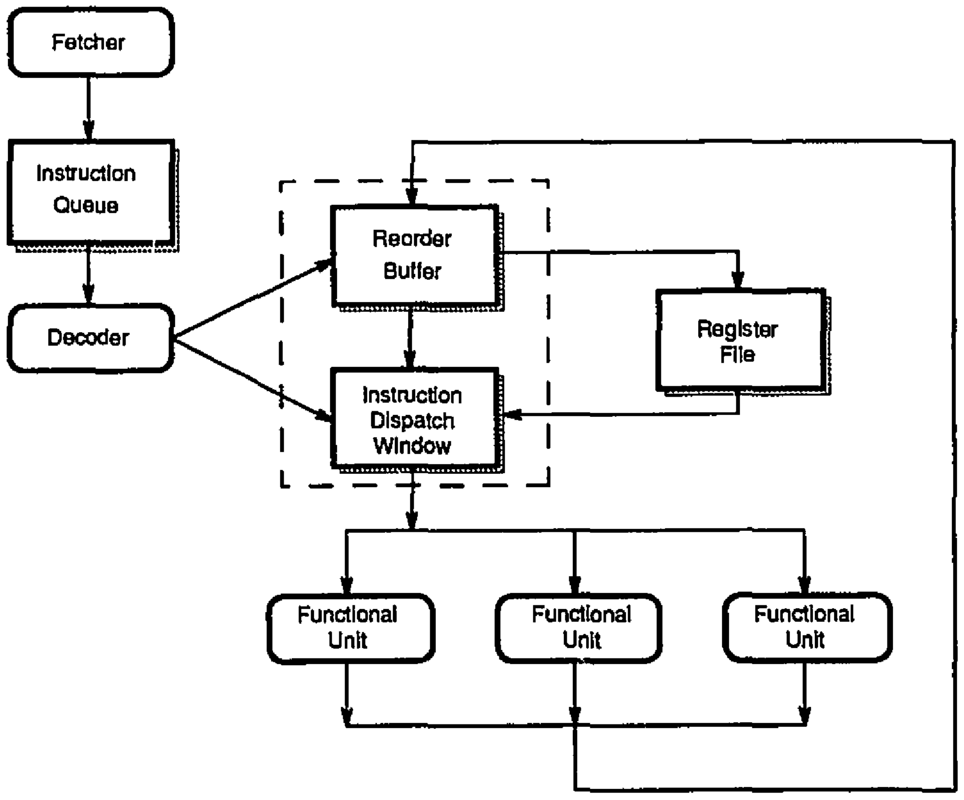


Figure 2.2: Superscalar processor execution model

also be assumed to be joined, as is the case for the Register Update Unit mentioned in Section 2.1.1, without affecting the functionality of our model.

Let us briefly overview how the elements in this model interact to execute instructions concurrently. Instructions are brought from memory by the fetcher which aligns and merges them before putting them in the instruction queue. The decoder reads several instructions each cycle from the instruction queue and puts them in the reorder buffer and in the instruction window. Instructions enter the reorder buffer in program order, and are assigned a tag which identifies them and allows automatic renaming of the destination register. At the same, time when an instruction, say s, enters the instruction window, its operands are searched for in the reorder buffer. If an operand is found in the reorder buffer and its value has already been calculated, the value is directly used as the operand for s in the instruction window. If the operand is found in the reorder buffer but its value has not been calculated, then the tag (corresponding to the instruction which calculates the value) is used as the pending operand for s in the instruction window. Finally, if the operand is not encountered in the reorder buffer, then its value is obtained from the register file instead of the reorder buffer.

The issue stage selects from the instruction window the instructions whose operands have already been calculated, checks the availability of the functional units and sends the instructions that are ready for execution. The instructions are, therefore, *dynamically scheduled* according to the availability of the operands and the functional units. This mechanism thus allows the *out-of-order issue* of the instructions to the functional units. Once an instruction has been executed (completed by a functional unit), its resulting value is written back to the reorder buffer and forwarded to the instructions whose pending operand tags correspond to the tag of the completed instruction. It is important to note that values computed by the functional units are not written back to the register file directly; they are written back to the reorder buffer which allows the *out-of-order completion* of instructions. Instructions can complete and have their values written to the reorder buffer in any order. It is at the commit stage that instructions at the head of the reorder buffer are retired to the register file. This FIFO structure of the reorder buffer also provides the undo capability necessary to support speculative execution. When a branch is executed and its prediction turns out to be incorrect, all the instructions that come after the branch are marked so they can be discarded later at commit time.

2.2 Register Allocation

In this section we review the register allocation problem. This section will provide the necessary background information required to understand the scheme proposed for the allocation of Short-Lived variables.

During recent years a sustained increase in processor performance has been observed; the speed of CPUs has increased in the range of 50% to 100% each year. In contrast, the situation has not been the same for main memories (DRAMs) for which increase in capacity and reduction in cost have been significant, but for which reduction in access time has not progressed at the same rate (i.e: roughly 7% each year) [HP90]. This growing gap between memory cycle time and processor cycle time has been a key motivation for the introduction of additional levels in the memory hierarchy. Smaller, but faster, memories between the main memory and the CPU are essential to allow the processor to receive the data at the rates demanded. Hence, modern microprocessors have adopted the use of larger register files and one or several levels of instruction and data caches.

Furthermore, the introduction of Load/Store architectures, where all the computations are performed in registers and accesses to memory are performed only through explicit Load/Store instructions, increased the importance of the use of register files since the goal in this type of architectures is to keep the maximum amount of data in registers to avoid the expensive accesses to memory. Consequently, the problem of efficiently mapping program variables to machine registers has become one of the predominant compiler optimization techniques [HP90].

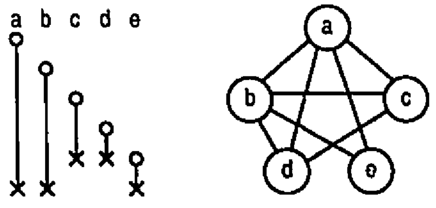
2.2.1 The Problem of Register Allocation

User programs make use of variables to store and manipulate data. Various phases of the compiler introduce temporary variables to calculate intermediate values and perform different transformations. Optimizations performed before register allocation assume that all these variables reside in an unlimited number of *virtual registers*. The goal of the register allocator is to maximize the number of virtual registers that are assigned to physical registers in order to minimize the number of accesses to memory. To accomplish this task, the register allocator tries to map the virtual registers whose live ranges do not interfere to the same physical register. However, sometimes the number of variables that interfere at the same time exceeds the number of registers available and, therefore, the register allocator must carefully decide which variables must be spilled (i.e. temporarily stored in memory) in order to minimize the negative effect on the program's execution time.

Register allocation can be performed at four different levels: 1) at the level of expressions, where the purpose is to reduce the number of allocated registers by finding the best order for the evaluation of the expression; 2) at the level of basic blocks, where the purpose is to reduce the number of registers used by the variables in the basic block [ASU88]; 3) at the level of procedures (intraprocedural), usually called *global register allocation*, where the optimization is performed for the whole body of a routine; and 4) at the level of the entire program (interprocedural), in which case the optimization performs the allocation for several routines simultaneously [Wal86]. In this thesis, we will only consider the problem of register allocation at the global (intraprocedural) and basic block levels.

Traditionally, the problem of global register allocation is solved by converting it into the equivalent problem of graph coloring. The problem is represented by a graph $G(V, E)$, called the *interference graph*, in which the vertices V represent the variables of the program and the edges E represent the interferences between them. The number of colors k that can be used to color the graph represent the number of physical registers available in the target processor. An example, showing a small program fragment, the live ranges for the variables in the program and the corresponding interference graph, is presented in Figure 2.3.

```
a = Read();
b = Read();
c = a * b;
d = a + b;
e = c / d;
print(a,b,e);
```



(a) Program segment (b) Live ranges (c) interference graph

Figure 2.3: Live ranges and interference graph for a small program

Since the problem of coloring an arbitrary graph when the number of colors is greater than two is NP-complete, researchers have developed different heuristics to solve the problem. The first heuristic applied to register allocation was developed by Chaitin [CAC+81, Cha82]. This heuristic was later improved by Briggs [BCKT89, Bri92] as part of his PhD dissertation. In the following subsections, we describe in more detail both heuristics since they are the basis for the register allocator developed for McCAT and constitute the foundation of the register allocation mechanism developed for the allocation of the Short-Lived variables. Other heuristics are briefly described at the end of the chapter.

2.2.2 The Yorktown Allocator

The Yorktown allocator, so named for being developed at IBM Yorktown Heights, was the first implementation of a global register allocator. It was developed by Gregory Chaitin [CAC+81] and his colleagues as part of the PL.8 compiler [AH82].

Chaitin devised a very simple heuristic to color the interference graph that has been proven to work very well in many cases. His reasoning states that, in order to color a graph with k colors, one can simplify the graph by removing a node n with degree less than k because, no matter how the neighbor nodes of n are colored, there will always be a remaining color to assign to n. This process of simplifying the graph by removing nodes with degree less than k can continue until no more nodes can be removed from the graph or the graph is empty. In the case where no more nodes can be removed from the graph, the heuristic has to select one node to be spilled (i.e. kept in memory instead of registers), and hope that by removing that node other nodes with degree less than k will emerge thus allowing the simplification process to continue. The choice of which node to spill must be done carefully. Chaitin's heuristic selects the node with minimum spill cost to degree ratio. The intuition for this criteria is to try to find a variable that is comparatively cheap to spill and, at the same time, interferes with a large number of variables. In this way, when the selected node is removed from the graph a large number of interferences will also be removed thus increasing the possibilities of continuing the simplifying process. The coloring process starts once all the nodes have been removed. The nodes are put back in the graph in the inverse order of their removal. Each time a node is added to the graph a color is assigned to it according to the colors of its current neighbors.

The different steps performed by the Yorktown allocator are depicted in Figure 2.4. A brief description of each step follows:

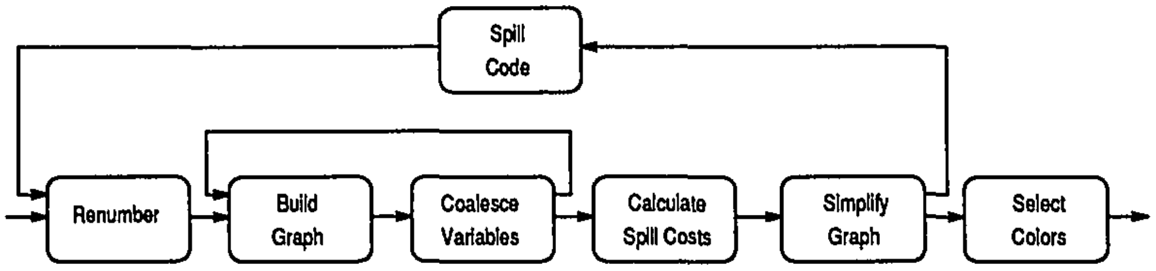


Figure 2.4: Steps of the Yorktown allocator

- *Renumber:* During this step, the compiler finds the right number of live ranges to allocate by calculating the set of *connected du-chains* [ASU88] for all the variables in the routine. Each du-chain is assigned a unique identifier.
- *Build Graph:* Given the set of live ranges, this step goes through the routine finding which ones interfere and constructing the interference graph. Two life ranges are said to interfere if one is alive at any of the definition points of the other.
- *Coalesce Variables:* Also called subsumption, this stage removes copy statements of variables whose live ranges do not interfere by coalescing the respective nodes in the graph into one. This helps to reduce the copy statements produced in the code and improves the targeting of instructions which is useful in optimizing procedure

calls and handling special purpose registers. After this step, the interference graph is built again and coalescing is repeated until no more variables can be subsumed.

- *Calculate Spill Costs:* During this step, the register allocator calculates the cost of spilling each live range. This cost is calculated by predicting the number of load and store instructions that would be executed if the live range were spilled. These costs are only an estimate since they are found at compile time.

- *Simplify Graph:* During this stage the graph is simplified by removing the nodes with degree less than k from graph. Removed nodes are pushed onto a stack. When no more nodes can be removed from the graph, the algorithm selects the node n with minimum $spill\_cost(n)/degree(n)$ to be spilled. This process continues until the graph is empty.

- *Spill code:* In this stage, the allocator introduces the spill code for the variables that were spilled in the *simplify* stage. For each variable spilled, a load instruction is introduced before each use of the variable, and a store instruction is inserted after each definition. The spill code transforms the live range of the variable into a set of small live ranges. Since these small live ranges still interfere with other variables, the algorithm has to rebuild the interference graph and start the coloring process again. The whole process is repeated until no more spill code is required.

- *Select Colors:* Once the simplify stage is completed without spilling any variable, the coloring of the nodes of the graph is started. The nodes are added to the graph in the reverse order in which they were removed by popping them from the stack, and a color is selected for each node added according to the interferences with the nodes already added.

The complexity of the algorithm is $O(V+E)$. Therefore, depending on the number of interferences between variables, the complexity of Chaitin's method can go from $O(V)$ to $O(V^2)$.

The Yorktown allocator has been proven to work well in many cases. However, it has some problems. The first problem is that once a variable has been chosen to be spilled, it is spilled everywhere instead of being spilled only in the places where the register pressure is high. The second problem is that, due to the fact that the live ranges of spilled variables still interfere with other variables, the whole process of register allocation has to be repeated several times until no more spill code is required. Usually this process does not require more than 3 or 4 iterations but it still can be very costly. The third problem is that, since the algorithm is based on a heuristic, it does not guarantee to find a solution for all the tractable cases, even if the problem to solve is very simple. To illustrate this problem, consider the little graph presented in Figure 2.5, and assume we want to find a 2-coloring of this graph using Chaitin's heuristic. During the simplifying process, the algorithm will try to find any node with degree less than 2. However, since no node has degree less than 2, the heuristic will wrongly decide to spill one of the variables. Clearly this graph is very easily colorable with 2 colors by assigning red to *a* and *b*, and white to *c* and *d*, for example.

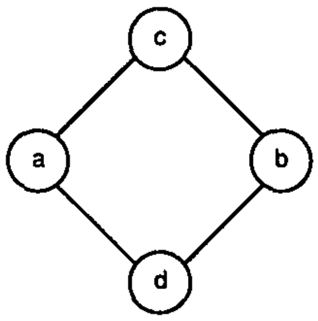


Figure 2.5: A small graph where the Yorktown allocator fails

2.2.3 The Optimistic Allocator

The problem with Chaitin's heuristic for the simplification of the graph is that it looks for nodes with degree less than *k* assuming that all the neighbors of the node will be colored with different colors. This assumption is too restrictive as it can be seen in the subgraph presented in Figure 2.6. If we assume that all the nodes in the graph have a degree of at least 4 and that all of them have the same spill cost, and we attempt a 3-coloring of the graph, Chaitin's heuristic will spill the variable *a*. This spilling is unnecessary because, as can be seen in the figure, the 4 neighbors of *a* can be colored using only 2 colors (red, white), and *a* can be colored with the remaining color (blue).

Briggs' heuristic [BCKT89, Bri92] solves this problem by optimistically assuming that, even if *a* has 4 neighbors, *a* and its neighbors can be colored using 3 colors or less. In order to do this, Chaitin's algorithm is modified so that the spill decisions are moved from the simplify step to the later step of selection of colors as shown in Figure 2.7. During *simplify*, the optimistic allocator looks for a node that has degree less than *k*, removes it from the graph and pushes it into the auxiliary stack. It repeats this process until no more nodes

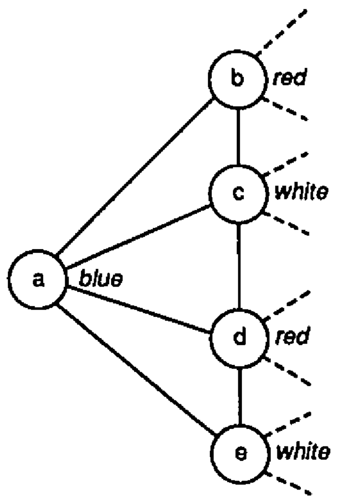


Figure 2.6: Generalization of the problem of the Yorktown allocator

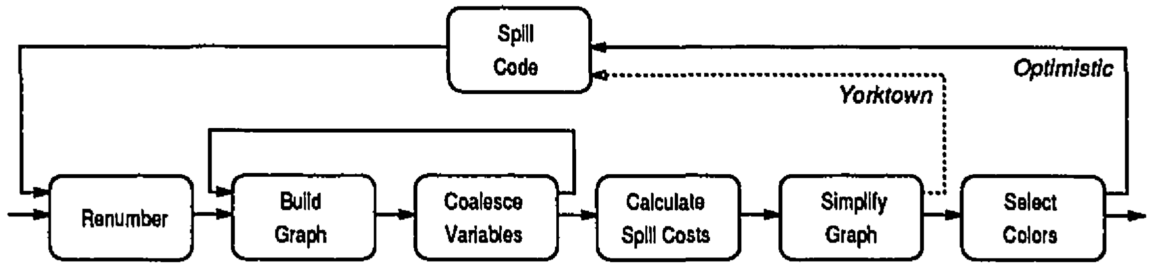


Figure 2.7: The Optimistic allocator versus the Yorktown allocator

with degree less than k can be found. At this point, the allocator chooses the node with lowest spill cost to degree ratio, optimistically removes it from the graph and pushes it into the stack assuming that the *select colors* phase will be able to assign a color to it. During the *select colors* step the nodes are put back in the graph in inverse order and colored according to the colors of the neighbor nodes. If at some point a node cannot be colored because there are no free colors, the node is marked to be spilled and it is not added to the graph. Briggs shows in his PhD thesis that if his heuristic decides to spill a node that node would have also been spilled by Chaitin's heuristic and, therefore, his heuristic always works at least as well as Chaitin's heuristic.

2.2.4 Other Allocators

There have been other heuristics developed to solve the problem of global register allocation. Register allocation is a hot topic within the compiler research community. The current focus is towards integrated schemes of register allocation and instruction scheduling. Due to space limitations, we will just mention a few of them and point out their most relevant characteristic.

Chow and Hennessy's mechanism [CH90] colors the interference graph by prioritizing the variables according to the their benefits of residing in registers. Callahan and Koblens [CK91] present a method to color the interference graph guided by the hierarchical structure of the program. Gupta and his colleagues [GSS89] presented a method to color the interference graph by partitioning it in its clique separators. Proebsting and Fischer [PF92] use a stochastic approach to allocate registers based on the probability that a value will be held in a register. Kolte and Harrold [KH93] propose a method that partitions the live ranges in Load/Store ranges providing more flexibility for the allocation. Norris and Pollock [NP94] propose to guide allocation with the use of the Program Dependence Graph – PDG. Finally, an allocation method based on cyclic interval graphs instead of the interference graph is proposed by Hendren et al. [HGAM93].

Chapter 3

Short-Lived Variables and Useless Commits

In this chapter we describe the problem of the useless commit of instructions to the register file. We start by presenting a simple example which points out the inefficiencies of the execution model as a result of this phenomenon. We then provide experimental evidence of the useless commits of instructions for a set of benchmarks. After defining some concepts, we formulate the problems to be studied in this thesis, we show the importance of addressing these problems, and we give an intuitive description of how these problems should be solved. At the end of the chapter, we describe the main obstacles we face to apply the solutions proposed, we illustrate the circumstances under which these obstacles are present, and for each one we give a small description of the method used to overcome them.

3.1 Useless Commits: A Motivating Example

One important observation of this thesis is that a significant portion of the values generated by the functional units are used only while they reside inside the reorder buffer. The cause of this is that a large percentage of the values generated have a very short live range. In general, the last use of a value can be found in the instruction stream soon after its definition point.

As an example, consider the small loop extracted from the Tomcat benchmark and reproduced in Figure 3.1(a). Figure 3.1(b) shows the DLX [HP90] assembly code for the body of the loop produced with *dlxcc*, the GCC Compiler targeted to produce code for the DLX architecture[1]. Figure 3.1(c) shows the live ranges for the different values generated and the respective register assignments. In this figure, small circles (o) denote definition points while cross signs (×) denote the points where the values die. Overlapped cross and circle signs (⊗) denote redefinition points. It can be seen that, for all live ranges, once a new value is defined, it is last used a few instructions later. If we count the length of a live range by the number of instructions between the producer and the last consumer, we can see that the length of the longest live range in this example is 14 instructions. If this code were to be run on a machine with a reorder buffer of 16 entries, all values produced could be consumed while they reside in the reorder buffer. If this were the case, the commit of the instructions to the register file would be useless because none of the values would ever be obtained from the register file.

Another important point to note from this example is that nine[2] physical registers have been used for the allocation of the temporal variables used in the body of the loop. This is a waste of resources because, as we pointed out before, the values stored in these registers are never being acquired from the register file. In other words, the register allocator is assigning register names to values that are never going to be obtained from the register file.

From the analysis of this example, two questions emerge: can we avoid the useless commits of instructions to the register file? Also, can we improve the register allocation process so that register names are not assigned to values that do not require them?

3.2 Experimental Observations

After considering this example, we want to determine how often these useless commits occur during the execution of real programs. For this purpose, we modified our superscalar simulation testbed to establish the percentage of useless commits found during the

[1]Whenever appropriate, we will use in our examples assembly code produced for the DLX architecture. The mnemonics used for the instructions are in general easy to understand, and most of the instructions have this format:`<Op. code> <dest. register>,<src. register 1>,[<src. register 2>]`. A complete description of the instruction set can be found in [HP90].

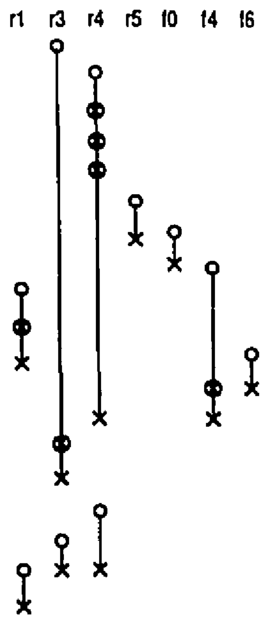
[2]`f4` and `f6` are being used as double registers.

```
for (I = 1; I <= N; I++) {
  X[I][1] = ((double) (I-1))
          /(double) (N-1);
}
```

(a) C Program

```
L5:
  lw r3,0(r6)
  slli r4,r3,#2
  add r4,r4,r3
  slli r4,r4,#3
  add r4,r7,r4
  add r5,r3,#-1
  movi2fp f0,r5
  cvti2d f4,f0
  lhi r1,(LC0>>16)&0xffff
  addui r1,r1,(LC0&0xffff)
  ld f6,0(r1)
  divd f4,f4,f6
  sd 8(r4),f4
  add r3,r3,#1
  sw 0(r6),r3
  lw r4,0(r6)
  addi r3,r0,#4
  sle r1,r4,r3
  bnez r1,L5
```

(b) Assembly code



(c) Live Ranges

Figure 3.1: Examples of short-live-range variables

execution of a set of benchmarks. The simulator was modified to count the number of values written to the register file, and to detect which of these values are never read from the register file. The results of these experiments are summarized in Table 3.1. In this table, we give the percentage of useless commits detected when running each benchmark with different sizes of the reorder buffer. A description of the benchmarks used will be given in Section 6.1, and a description of the testbed will be given in Appendix A.

From the results presented in this table, it can be seen that a large percentage of the instructions committed to the register file are useless. The percentage of useless commits increases with the size of the reorder buffer because there is a bigger chance that the last use of a value and its definition point reside in the reorder buffer at the same time. Even with a reorder buffer of 8 entries the percentage of useless commits is very high (on average 89.56%). With a reorder buffer of 32 entries the percentages are even higher, on average 95.11%. These results suggest that all these useless commits constitute a significant waste of resources and allow for architecture and compiler optimizations.

	Buffer Size		
Benchmark	8	16	32
Alvinn	85.80	88.90	89.89
Bubble	93.82	95.93	97.14
L8	89.66	98.81	98.81
L14	91.65	95.93	97.08
L8unroll	89.54	98.70	98.70
L14unroll	91.77	95.87	97.12
Linpack	92.32	92.61	93.33
Quickrand	81.81	87.52	88.19
Tomcat	95.95	97.61	97.78
Whetstone	83.29	90.21	93.04
Average	89.56	94.21	95.11

Table 3.1: Percentage of useless commits

3.3 Problem Formulation

In this section we formulate the problems to be studied in this thesis. We start by defining some of the related concepts. Some of these concepts are also exemplified in Figure 3.2.

Let $d_0, d_1, d_2 \ldots d_n$ be the definition points for a set of values $v_0, v_1, v_2 \ldots v_n$. Let $u_{i_0}, u_{i_1}, u_{i_2} \ldots u_{i_m}$ be the set of last-use points for a value v_i. We define the live range r_{i_j} of a value v_i as the sequence of instructions in the interval $[d_i, u_{i_j}]$, and the length of r_{i_j}, $\mathcal{L}(r_{i_j})$, as the number of machine instructions in the longest path between d_i and u_{i_j} including d_i and u_{i_j}. Assume we have a superscalar machine $\mathcal{M}$, like the one described in our execution model (Section 2.1.3), with a reorder buffer of length $\mathcal{LR}$ entries. We say that the commit of the value v_i produced by the definition point d_i to the register file is a *useless commit* if all the references to v_i are obtained from the reorder buffer rather than from the register file. We say a live range r_{i_j} is short if it has a length that is smaller than or equal to the size of the reorder buffer, i.e., $\mathcal{L}(r_{i_j}) \leq \mathcal{LR}$. Consequently, we call a variable x a *short-lived variable* if all the live ranges associated to x are short. Note that for an instruction at d_k to be a useless commit, it is necessary that *all* the associated live ranges, r_{k_x} for any x, be short.

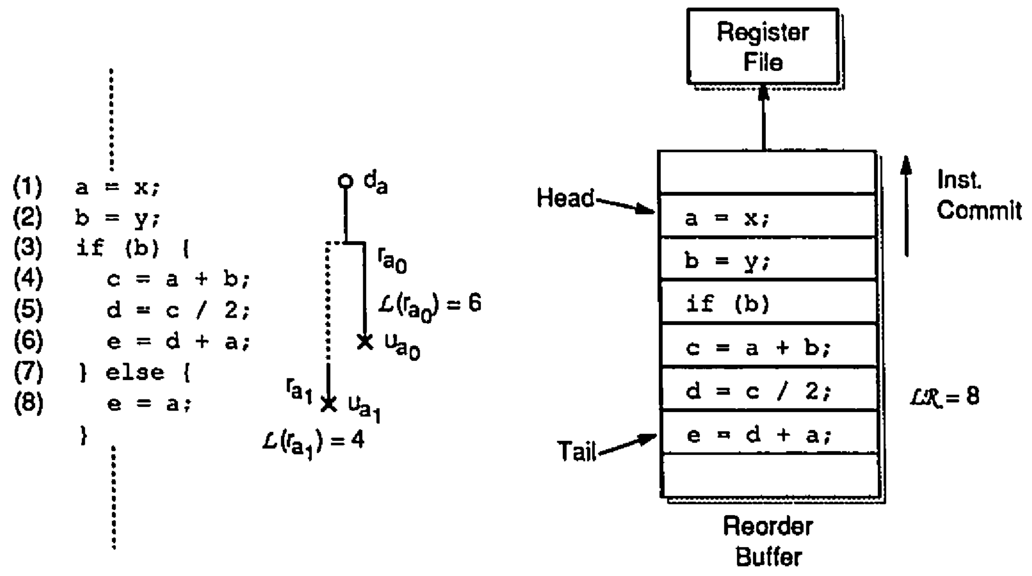


Figure 3.2: Some concepts related to short-lived variables

Given these definitions, the problems to be studied in this thesis can be stated as follows:

- **Problem 1 - Reduction of Useless Commits:** Given a machine $\mathcal{M}$ with a reorder buffer of $\mathcal{CR}$ entries and a program $\mathcal{P}$, we want to develop architecture mechanisms and compiler techniques to reduce the number of useless commits in $\mathcal{M}$, while executing $\mathcal{P}$.

- **Problem 2 - Allocation of short-lived variables:** In conjunction with the solution for Problem 1, we want to propose a register allocation scheme which ensures that the short-lived variables in $\mathcal{P}$ do not occupy physical registers of the machine whenever possible.

Solving Problem 1 will allow the architecture to reduce the number of write ports to the register file without affecting the execution time of the program. The reduction of write ports is an important issue because it simplifies the structure of the data path of the processor [UJ92, Joh91]. The complexity of implementation of the reorder buffer, the register file and the associated busses depends on the number of ports to the register file. As we mentioned in Section 2.1.2, the area complexity of a multiported register file is roughly proportional to the square of the number of ports [CDN92]. In fact, due to the complexity of the register file, the cycle length for many architectures is determined by the access time to the register file. Recent superscalar designs, like the PowerPC 604 [SD94], provide register files with more than 10 ports. However, even 10 register ports may not be enough to satisfy the demands posed by issuing several instructions each cycle, and the designers are forced to put restrictions on the number of instructions that can be committed per cycle or to add stages to the pipeline to handle the possible contention [SD94].

Solving Problem 2 will allow the compiler to utilize more efficiently the physical registers by using them exclusively for the allocation of variables with long live ranges. The effective utilization of the registers is also an important issue. As we saw in Section 2.2, the register allocator tries to map a usually large number of virtual registers to a small number of physical registers which, therefore, constitute a precious resource. Furthermore, compiler optimizations for ILP try to increase parallelism which in turn increases register pressure [CF87]. We need to find mechanisms to reduce the register pressure created by these optimizations and utilize as efficiently as possible the register names provided. If we avoid the use of physical registers by the short-lived variables we will increase the number of registers available for long-lived variables thus reducing register pressure and the amount of spill code introduced.

3.4 Solution Strategy

As observed in Section 3.2, a large percentage of the instruction commits are not necessary. Therefore, we propose to reduce this percentage (Problem 1) by providing the architecture with specific information, so that, at commit time, it can decide whether a value should be committed to the register file or just discarded. This information should be collected by the compiler by analyzing the characteristics of the live ranges of the variables, and provided to the architecture through the instruction set. Note that only the compiler can detect whether the use of a value is the last one or not. Thus, it is the responsibility of the compiler to flag the instructions that are going to be discarded at commit time.

To solve Problem 2, we propose to modify the register allocation process, so that variables whose live ranges are not committed to the register file are not assigned to physical registers since these live ranges are not going to make use of them. We maintain that these variables should be assigned to and stored in the reorder buffer instead of the register file. Then, the reorder buffer will be considered an extension of the register file in which the values of short live ranges will be temporarily stored before they are discarded at commit time. The actual storage for the value will be dynamically allocated by the renaming mechanism of the architecture. In short, we propose to use the space provided by the renaming capabilities of the reorder buffer as additional registers to allocate the short-lived variables.

3.5 Obstacles

In the preceding sections, we have proposed exploiting the fact that a large percentage of the live ranges occur while they reside in the reorder buffer. However, in order to be able to exploit this fact, we need to overcome some problems. There are 3 main obstacles to surpass: the compiler's lack of knowledge about which instructions are present in the reorder buffer at any given point, the detection of useless commits in the presence of the speculative execution of instructions, and the need to provide support for precise interrupts. In the following subsections, we will illustrate with examples these obstacles and give an intuition on how to overcome them. Some of the presented ideas will be elaborated in subsequent chapters.

3.5.1 Determining Useless Commits at Compile Time

The first obstacle preventing us from applying our ideas is that in current superscalar designs, like the one described in our execution model in Section 2.1.3, the compiler cannot safely determine whether an instruction commit will be useless based only on the fact that the associated live range has a length smaller than the size of the reorder buffer. The compiler can establish that a live range is short, but that does not imply that the associated definition point will be a useless commit. The problem lies in that, with the current implementation of the reorder buffer, an instruction can be committed to the register file as soon as it reaches the head of the reorder buffer without waiting for the last use of the value to enter the buffer.

This problem is illustrated in Figure 3.3. Consider a machine with a reorder buffer having 12 entries ($\mathcal{LR} = 12$), which is able to fetch and decode 2 instructions per cycle ($\mathcal{DB} = 2$). As it can be seen in the figure, the length of the live range of a is 11. According to our definition, this live range is short. However, from the moment the definition point of a, d_a, enters the reorder buffer, to the moment the last use, u_a, enters the reorder buffer, five clock cycles will occur. In these five cycles, instruction (1) will be able to execute and write back its result to the reorder buffer. If this instruction reaches the head of the reorder buffer before the last use u_a enters the buffer, then the value of a will be retired to the register file. Then, when u_a enters the reorder buffer the value of a will have to be obtained from the register file and therefore, the commit of d_a will not be useless. However, if we had a different policy for the commit process of the instructions, the commit of d_a could have been avoided.

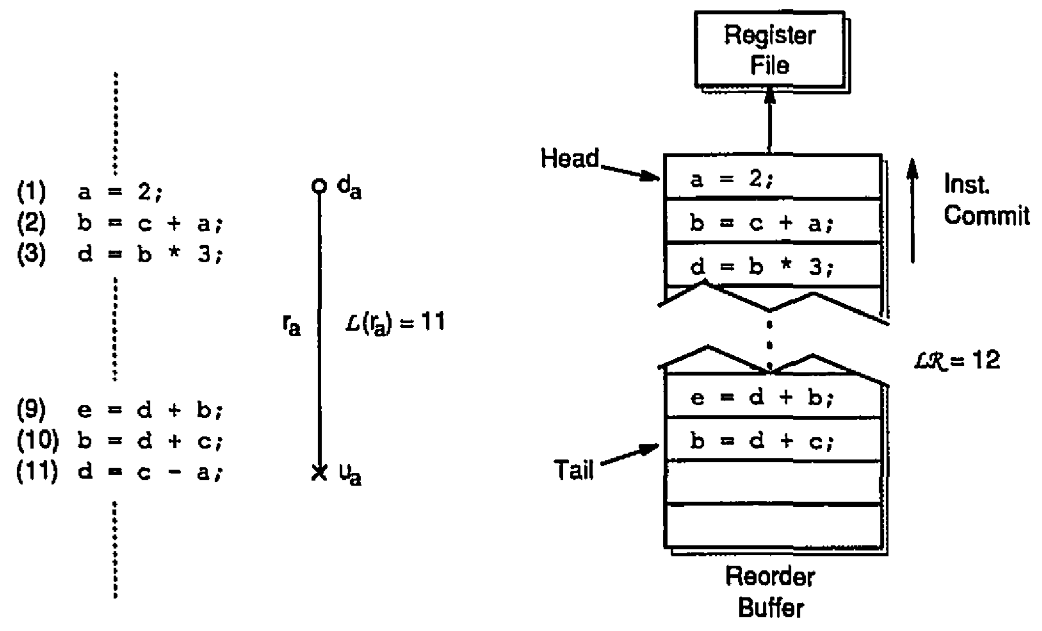


Figure 3.3: Impossibility of determining useless commits at compile time

To address this issue, we propose that the compiler should mark in the code the definition and last use points of the short live range, while the architecture should be augmented to ensure that the instruction defining the value will not leave the reorder buffer before the last use enters the reorder buffer. We will elaborate on this mechanism and its possible variations in Chapter 4 where we discuss the compiler mechanism and architectural modifications required.

3.5.2 Detecting Useless Commits in the Presence of Speculative Execution

The second obstacle to overcome is related to the speculative execution of instructions. The problem of knowing at compile time when an instruction is going to be a useless commit is even more difficult in the presence of speculative execution. After a definition point has entered the reorder buffer, one or several branches can be predicted before the last use of the value enters the reorder buffer. Yet, even after the last use has entered, it is not safe to say that the definition point can be discarded once it arrives to the head of the reorder buffer because some of the branches in the middle could have been wrongly predicted. If the definition point is discarded and the prediction for one of the branches happens to be incorrect, then another use of the value can be fetched from the new path, and this use will not be able to find the value in the reorder buffer nor in the register file.

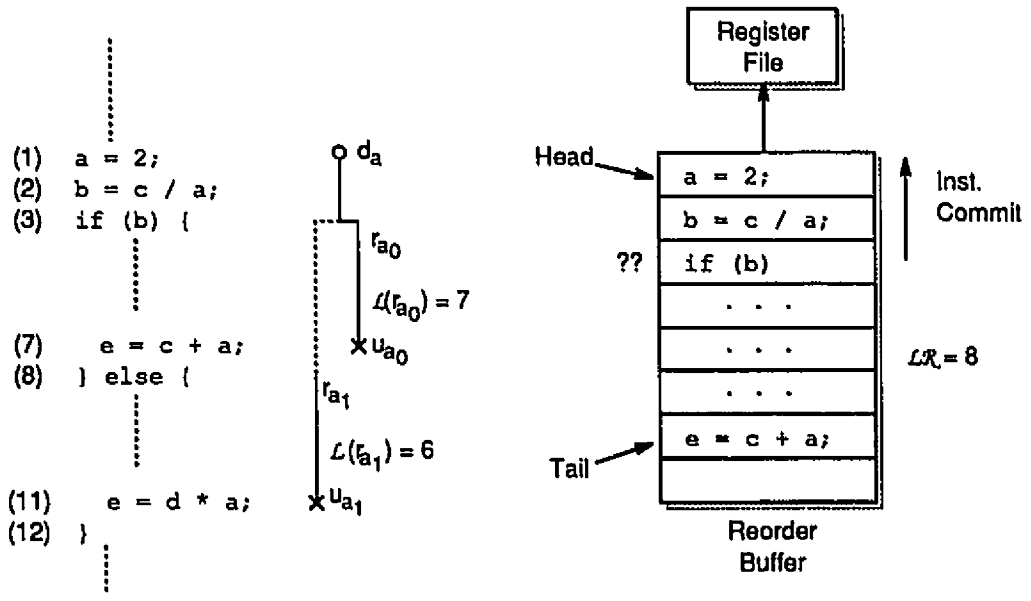


Figure 3.4: Dealing with speculative execution

To illustrate this problem, consider the example given in Figure 3.4. Assume the processor has a reorder buffer with 8 entries ($\mathcal{LR} = 8$). The definition point of a, d_a, is found in instruction (1). In this case there are two last uses for a, one at the end of the if-body and the other at the end of the else-body. The respective lengths of both live ranges are six and seven. Therefore, based on our definition, both live ranges are short and a is a short-lived variable. Assume that the branch prediction mechanism foretells that the if-body is going to be taken. In that case, instruction (7) would be put in the reorder buffer and we could assume that since both the definition point d_a and the last use u_a are present in the reorder buffer, the commitment of d_a would be a useless commit. Then, when retiring the definition point d_a we could discard the value of a instead of committing it to the register file. If the prediction of the branch is correct the value of a will not be required anymore and the mechanism would work properly. However, if the prediction happens to be incorrect, the instructions in the else-body will be fetched and when instruction (11) is decoded the value of a will not be found either in the register file nor in the reorder buffer.

There are two possible ways to get around this problem. One way is to design a hardware mechanism to ensure that the definition point of a short live range will not leave the reorder buffer until the last use is also in the reorder buffer and the branches between them have been resolved. The other way is to simplify our problem and decide that we are only going to avoid the useless commits of short live ranges that do not cross the basic clock boundaries. Both mechanisms will be discussed in Chapter 4.

3.5.3 Dealing with Interrupts

The last obstacle we have to overcome in order to allow the compiler to exploit the occurrence of short-lived variables is related to the need to provide precise interrupts. Interrupts are events that disturb the normal execution of the program like a page fault or an arithmetic overflow. An interrupt is said to be precise, if at the moment when the interrupt is handled all the instructions preceding the faulting one in the sequential code have updated the state of the processor, none of the succeeding instructions have updated the processor state, and the saved program counter is pointing to the instruction that caused the interruption [SP88, HP87]. Precise interrupts are easily achievable in sequential processors where all the instructions are executed in order. However, in ILP processors precise interrupts are difficult to achieve due to the fact that in this kind of machines instructions can be executed out of order and thus update the state of the processor in a non sequential order. As we described in Section 2.1.1, the reorder buffer provides a mechanism to support precise interrupts. The interrupt is not treated until the faulting instruction reaches the head of the reorder buffer. In this way, we can ensure that all the instructions preceding the faulting one have updated the state of the processor and that the saved program counter is pointing to the correct instruction.

Even if our execution model uses a reorder buffer, we face a new problem when we are trying to avoid the useless commits produced by short-lived variables and at the same time still providing precise interrupts. The problem resides in the recoverability of the processor state after handling the interrupt. If we discard the value produced by the definition point, and an instruction lying between the definition point and the last use point of a live range causes an interrupt, then the last use of the value will not be able to find this value after the interrupt has been handled.

To illustrate this problem, consider the example given in Figure 3.5. The range of a in this example is a short live range. Assume the situation in the reorder buffer is as depicted in the figure. Both the definition point (inst. 1) and the last use (inst. 5) of a are in the reorder buffer. Assume further that when retiring instruction (1) we discard the value produced instead of committing it to the register file. A problem arises if an instruction, say instruction (3), causes an interrupt. In this case, the processor will wait until instruction (3) reaches the head of the reorder buffer and, at this point, it will call the routine to handle the interrupt. The routine will save the state of the registers, process the interrupt and restore the register file to the state it had before the interrupt. After the handling routine has returned the processor will start executing the program from the point where the fault occurred. However, when the processor tries to decode instruction (5) again, it will look for the value of a and, since we discarded it before the interrupt occurred, it will not be able to find it. In this case, the interrupt was handled at the point where it should have been, that is, the interrupt was precise. Nevertheless, the interrupt was not recoverable.

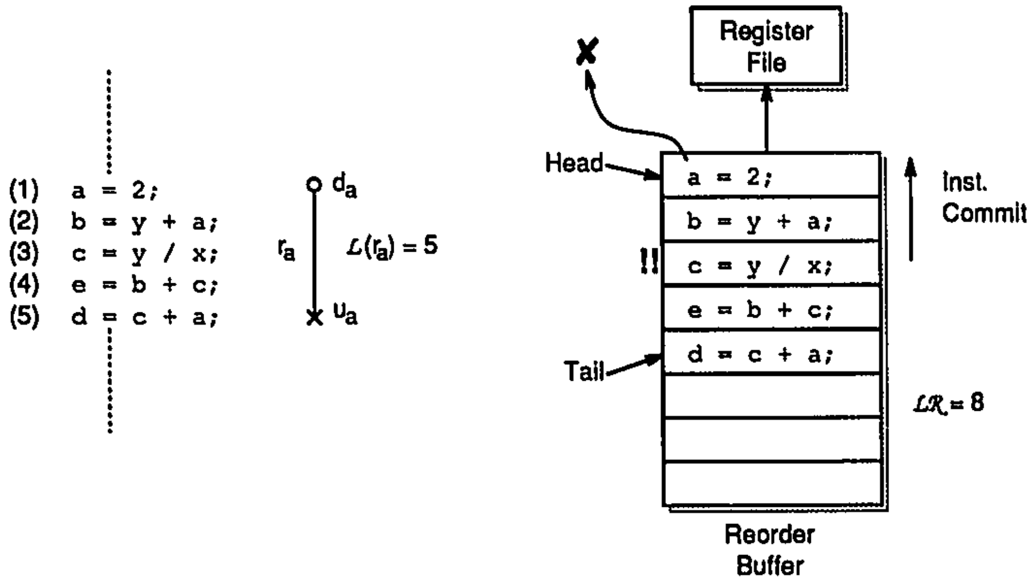


Figure 3.5: The problem of handling interrupts

In order to provide the recoverability of the interrupts, we propose that at the moment of handling the interrupt, not only the state of the register file and the program counter should be saved, but also the state of the reorder buffer should be saved. The process of saving the state of the reorder buffer can be expensive given that the size of the reorder buffer can be large. However, since interrupts do not occur frequently (around 1 instruction every 5000 instructions [HP87, BP93]), the performance penalty of saving the state of reorder buffer should be negligible. The advantage of saving the state of the reorder buffer is that the processor can restart fetching the instructions from the last instruction that entered the reorder buffer before the interruption was treated instead of restarting fetching from the instruction that caused the interrupt.

As we will see in Section 4.2, an improvement to this mechanism can be implemented if we consider discarding only the values of short live ranges that do not cross the basic block boundaries.

Chapter 4

Reduction of Useless Commits

In the previous chapter, we illustrated with examples the fact that, with our current execution model, the detection of a short live range by the compiler does not imply the detection of a useless commit to the register file. We pointed out that the problem with the current model is that an instruction can be retired to the register file as soon as it reaches the head of the reorder buffer without waiting for the last use to be present. We also noted that, even if the architecture ensures that the definition point of a value is not retired until the last use is also in the reorder buffer, it is not safe to decide that a value should be discarded if the last use of this value is being speculatively executed. In this chapter, we describe the hardware and software mechanisms used in this thesis to overcome the problem of detection of useless commits both with and without the presence of speculative execution. With the help of this mechanisms we are able to solve the problem of reduction of useless commits, which is the first problem this thesis aims at resolving.

4.1 Detecting Useless Commits

The simplest mechanism, though not the most efficient, for the detection of the useless commit of instructions could be implemented at the hardware level without requiring the intervention of the compiler. At commit time, the processor could decide to discard the value to be stored in a register if it detects that a succeeding instruction in the reorder buffer is also writing to the same register. Although this mechanism does not require any information provided by the compiler, it is not as precise as the mechanism we propose in this thesis. That is, it would not detect the useless commits of instructions whose corresponding registers are not redefined by any of the succeeding instructions in the reorder buffer. Moreover, this mechanism would also have problems dealing with the speculative execution of instructions since the instruction redefining the register could be in the reorder buffer after an unresolved branch.

Since we want to maximally reduce the number of useless commits, we decided to implement the mechanism suggested in our solution strategy in Section 3.4. That is, to use the compiler to collect information about the live ranges and communicate this information to the processor through the instruction set. Hence, in our mechanism the compiler decides which values should be discarded at commit time and tags these values. In addition, the architecture provides a mechanism to ensure that the whole live range of a tagged value will occur inside the reorder buffer. To achieve this, the architecture has to impose more constraints on the commit process of an instruction to guarantee that the definition point of a value chosen by the compiler will not be discarded until the last use enters the reorder buffer. This can be implemented using two different schemes:

- **Scheme 1: Marking the last use of a value**

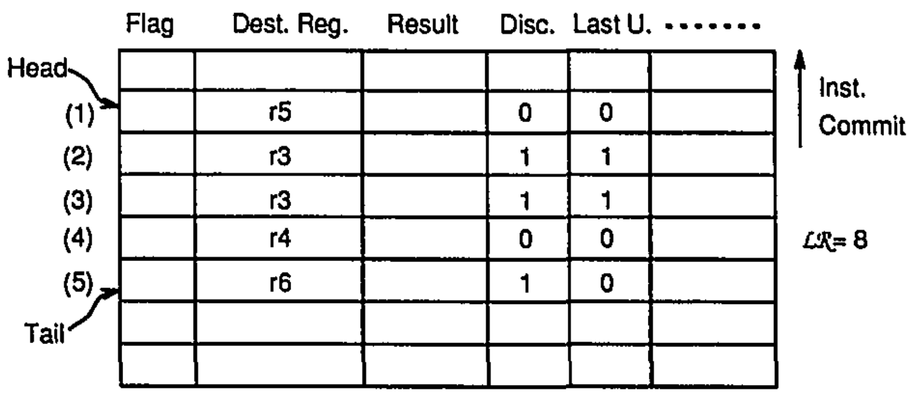
  In this scheme, the compiler is responsible for informing the architecture which of all the uses of the value is the last one. When the last use enters the reorder buffer an associative search is performed in the reorder buffer and the definition point is marked so that when it arrives at the head of the reorder buffer it can be discarded. This should not increase the time required to enter an instruction into the reorder buffer since the mechanism defined in the execution model is already performing the associative search to find the instruction that is defining the value.

  This mechanism is exemplified in Figure 4.1. Figure 4.1(a) shows a fragment of assembly code that has been annotated by the compiler with ampersand signs to indicate which values can be discarded at compile time and to flag the corresponding last uses of these values. Thus, for example, the value of `r3` defined in instruction (2) can be discarded at compile time and the last use of this value is found in instruction (3). The value of `r3` in instruction (3) can be discarded at commit time and its last use is found in instruction (5). On the other hand, the values of `r5` in instruction (1) and of `r4` in instruction (4) must be committed to the register file.

  Figure 4.1(b) shows a partial representation of these instructions in the reorder buffer. We have added two bits to each entry in the reorder buffer. The "Discard" bit (Disc)

```
(1)   add r5,r0,r3
(2)   slli &r3,r4,#2
(3)   add &r3,&r3,r4
(4)   slli r4,r3,#3
(5)   add &r6,r4,&r3
```

(a) Assembly code



	Flag	Dest. Reg.	Result	Disc.	Last U.	·······
(1)		r5		0	0	
(2)		r3		1	1	
(3)		r3		1	1	
(4)		r4		0	0	
(5)		r6		1	0	

(b) Reorder buffer representation

Figure 4.1: Marking the last use of a value

indicates that the value being defined in this entry can be discarded at commit time. The value of this bit is determined when the instruction is decoded and entered in the reorder buffer. The "Last Use" bit indicates that the last use of the value being defined in this entry is present in the reorder buffer. This bit is set when the instruction with the last use of the value enters the reorder buffer. When the instruction reaches the head of the reorder buffer, the value is retired to the register file if the "Discard" bit is zero. If both the "Discard" bit and the "Last Use" bit are one, the value can be discarded. Otherwise, if the value of the "Discard" bit is one and the "Last Use" is zero, then the value cannot be discarded until the corresponding last use enters the reorder buffer. Thus, in the example, the value produced for r5 must be retired to the register file since the value of the "Discard" bit is reset. On the other hand, the value produced for r3 can be discarded since both the "Discard" and the "Last Use" bit are set. Once instruction (5) arrives to the head of the reorder buffer, the value produced for r6 can be discarded only if the "Last Use" bit has been set, indicating that the last use of r6 has entered the reorder buffer.

Special care has to be taken by the compiler so that the use of this mechanism does not produce stalls in the decode stage. If the length of the reorder buffer is $\mathcal{LR}$ and the decode bandwidth of the processor is $\mathcal{DB}$, then the length of the live ranges chosen by the compiler must not exceed $\mathcal{LR} - \mathcal{DB}$. In this way, we avoid the case where instructions cannot enter the reorder buffer because another instruction is at the head waiting for the last use to enter the reorder buffer and stopping the commit of other instructions. So, as an example, if $\mathcal{LR}$ is 16 and $\mathcal{DB}$ is 4 then the length of the live ranges chosen to be discarded should not exceed 12 instructions.

- **Scheme 2: Keeping a minimum number of instructions in the reorder buffer**

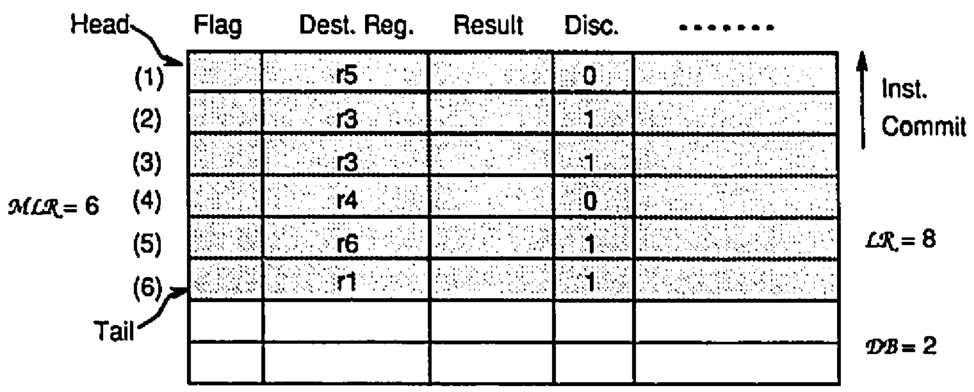
  In this scheme, an instruction cannot be committed unless there is a minimum number of instructions in the reorder buffer. In this way, the compiler can be sure that if two instructions are separated by a number of instructions that is less than the minimum number of instructions kept in the reorder buffer, the two instructions are going to be simultaneously present in the reorder buffer. Here again, special care has to be taken when choosing the minimum number of instructions to be kept in the reorder buffer in order to avoid decode stalls. The minimum number of instructions $\mathcal{MLR}$ must be set to $(\mathcal{LR} - \mathcal{DB})$. That is, $\mathcal{MLR}$ will be an architectural parameter which depends on the reorder buffer size $\mathcal{LR}$ and the decode bandwidth $\mathcal{DB}$ (See Figure 4.2). Knowing $\mathcal{MLR}$, the compiler will be able to decide which live ranges will completely occur inside the reorder buffer and mark them, so the architecture can safely discard the associated value at commit time.

  We should emphasize here that the use of this scheme does not delay the execution of the instructions in the program. The fact that an instruction i cannot be retired from the reorder buffer until a minimum number of succeeding instructions are present in the buffer, does not delay the execution of the instructions that depend on i. The reason is that the reorder buffer forwards the value produced by the instruction i to all dependent instructions at the time i finishes execution rather than when it is retired. Moreover, this scheme does not delay the decoding of instructions because the minimum number of instructions $\mathcal{MLR}$ was set in such a way that it ensures there is space in the reorder buffer to put $\mathcal{DB}$ new instructions each cycle. The only situation where the reorder buffer could be full is when the instruction at the head of the buffer has not finished execution, and this situation is not caused by the fact that we are keeping a minimum number of instructions in the reorder buffer.

  An advantage of this mechanism is that the compiler does not need to flag the last use of a value to be discarded. Only the definition point has to be flagged. This is

```
(1)    add r5,r0,r3
(2)    slli &r3,r4,#2
(3)    add &r3,r3,r4
(4)    slli r4,r3,#3
(5)    add &r6,r4,r3
(6)    addi &r1,r5,#8
```

(a) Assembly code



(b) Reorder buffer representation

Figure 4.2: Keeping a minimum number of instructions in the reorder buffer

illustrated in Figure 4.2. In Figure 4.2(a) a fragment of assembly code annotated with the information of which values should be discarded at commit time is shown. As it can be seen, the compiler only needs to mark the values that can be discarded at commit time. In this example, the values produced by instructions (2), (3), (5) and (6) can be discarded instead of committed to the register file. Figure 4.2(b) shows a partial representation of these instructions in the reorder buffer. In this case we only need to add one bit to each entry of the reorder buffer. As in the previous schema, we call this bit the "Discard" bit. It is determined when the instruction is decoded, and it is used at commit time to decide if the value should be discarded or committed to the register file. The reorder buffer has to be modified to ensure that an instruction at the head of the buffer can only be removed if the number of instructions in the reorder buffer is at least $\mathcal{MLR}$.

These two schemes are equivalent in terms of effectiveness. The selection of the mechanism will depend, then, on the complexity of the implementation. As mentioned before, Scheme 2 has the advantage that only the definition points need to be marked by the compiler. This implies a smaller change in the instruction format and, depending on the instruction set, could imply that object code compatibility could be preserved. On the other hand, the implementation of Scheme 2 could be more difficult for some architectures particularly in the presence of branch prediction. When a branch is wrongly predicted, some architectures scrap the instructions following the branch while others just mark them so that at commit time they can be discarded. For the architectures that actually scrap the instructions, it could be more difficult to keep an account of how many instructions are in the reorder buffer. In this case, Scheme 1 could be easier to use.

4.2 Dealing with Instruction Speculation

As we showed in Section 3.5.2, the problem of detecting useless commits at compile time is more difficult when we consider the speculative execution of instructions. The problem is that it is not safe to discard a definition point even if the last use is also present in the reorder buffer because the last use could be speculatively executing. There are 2 schemes that can be use to solve this problem:

- **Scheme 1: Using a hardware mechanism to detect useless commits in the presence of speculative execution**

  With this scheme, the reorder buffer has to be modified to ensure that the definition point of a short live range is not discarded until we are certain that the last use is not executing speculatively. That is, the value produced by a tagged instruction at the head of the reorder buffer cannot be discarded until the last use has entered the reorder buffer, and there are no unresolved branches between the definition point and the last use of the value. This can be accomplished in hardware by keeping track of the difference in speculation level between the definition point and the last use of the value. For this, each entry in the reorder buffer can be augmented with two fields to indicate the level (depth) of speculation at which the entry currently is ("CurrLevel"), and the level of speculation at which the last use of the value being produced is ("LastUseLevel"). Each can be implemented with no more than four bits, since no more than four levels of speculation are required to achieve nearly maximum parallelism in a superscalar processor, as shown in [Joh91].

This mechanism is described in Figure 4.3. Figure 4.3(a) shows a small fragment of assembly code which includes a branch instruction (instruction (3)). We will focus our attention on the value of r6 since it has been tagged by the compiler and its last use occurs after the branch. Figure 4.3(b) shows the state of the reorder buffer after the first three instructions have entered. Bit 0 of the "CurrLevel" of instructions (1) and (2) is set to one whereas bit 1 of "CurrLevel" of instruction (3) is set to one to indicate that instructions after the branch are in the next level of speculation. Assume that the branch predictor foretells that the branch is not going to be taken. When instruction (4) enters the reorder buffer (Figure 4.3(c)), the bit 1 of its "CurrLevel" is set to one and, since this instruction has the last use of r6, bit 1 of the "LastUseLevel" for instruction (1) is also set to one. When the branch is resolved, two actions can be taken. If the branch was correctly predicted (Figure 4.3(d)), bit 1 of "CurrLevel" for all the instructions preceding the branch is set to one to indicate that the branch corresponding to that level was correctly predicted. If the branch was mispredicted, bit 1 of "LastUseLevel" for all the instructions preceding the branch is set to zero to indicate that all the instructions with last uses in that level were wrongly entered in the reorder buffer.

The value defined by a tagged instruction (Discard bit set to one) can be discarded only if the same bit is set in both the "LastUseLevel" and the "CurrLevel" fields. In Figure 4.3(d), the value defined in instruction (1) can be discarded because bit 1 for both the "CurrLevel" and the "LastUseLevel" is set, indicating that there are no unresolved branches between this instruction and the last use of the value being produced.

Although this mechanism is able to map short live ranges to useless commits in the presence of instruction speculation, there are several arguments against its use. The first argument is that it seems difficult to implement. One of the goals of this thesis is to use information provided by the compiler to reduce the hardware complexity. The use of this mechanism would certainly not help to achieve this goal. The second argument is that this mechanism can introduce stalls in the decoder. In effect, the definition point of a short live range cannot be discarded until there are no unresolved branches between it and the last use of the value. Therefore, a definition point could be at the head of the reorder buffer waiting for some succeeding branches to be resolved and stopping the decoding process because of lack of space in the reorder buffer to introduce more instructions. Although we did not implement this mechanism and, therefore, we are not certain of how often this situation can occur,

```
(1)   add &r6,r0,#3      ; r6 Definition Point
(2)   sgt &r3,r3,r5      ; r3 = (r3 > r5)
(3)   beqz &r3,Lelse1    ; If (r3 == 0) goto Lelse1
(4)   add r5,r0,&r6      ; r6 Last Use
```

(a) Assembly code

	Flag	Dest. Reg	Result	Discard	....	CurrLevel 3	2	1	0	LastUseLevel 3	2	1	0
Head													
(1)		r6		1					1				
(2)		r3		1					1				1
?? (3)		-		-				1					
Tail													

(b) Insertion of instructions up to the branch

	Flag	Dest. Reg	Result	Discard	....	CurrLevel 3	2	1	0	LastUseLevel 3	2	1	0
Head													
(1)		r6		1					1			1	
(2)		r3		1					1				1
?? (3)		-		-				1					
(4)		r5		0				1					
Tail													

(c) Insertion of the last use of r6

	Flag	Dest. Reg	Result	Discard	....	CurrLevel 3	2	1	0	LastUseLevel 3	2	1	0
Head													
(1)		r6		1				1	1			1	
(2)		r3		1				1	1				1
(3)		-		-				1					
(4)		r5		0				1					
Tail													

(d) Resolution of the branch

Figure 4.3: Detecting useless commits in the presence of instruction speculation

this scenario is, of course, undesirable.

- **Scheme 2: Considering only live ranges that do not cross basic block boundaries**

  The other way to get around this problem is to simply avoid it by considering as short-lived variables only those variables whose live ranges are short and whose definition and last use points are found in the same basic block. The intuition behind this idea is that, since the live ranges we are considering are short, there is a high probability that their definition and last use points are in the same basic block. In general, many of these short live ranges are produced as a consequence of the use of temporary variables by the compiler. Most of the time, these variables introduced by the compiler are used only inside the basic block. In order to verify this observation, we modified our simulator to count the number of useless commits that are caused by live ranges that do not cross the basic block boundaries. The results of our measurements for different reorder buffer sizes are presented in Table 4.1.

	Buffer Size		
Benchmark	8	16	32
Alvinn	85.8	89.35	89.35
Bubble	93.83	95.90	95.92
L8	89.66	98.81	98.81
L14	91.26	95.16	95.16
L8unroll	89.54	98.70	98.70
L14unroll	91.39	94.85	94.85
Linpack	92.31	92.50	92.64
Quickrand	81.37	84.11	84.50
Tomcat	96.31	97.45	97.57
Whetstone	82.82	89.26	89.26
Average	89.43	93.61	93.68

Table 4.1: Percentage of useless commits produced by live ranges that do not cross basic block boundaries

Comparing these results with the ones presented in Table 3.1, it can be seen that most of the short live ranges do not cross the basic block boundaries. Even for a reorder buffer of 32 entries, the difference of the measurements presented in Tables 3.1 and

4.1 for a particular benchmark is at most 4%. The differences are further reduced when we consider the measurements for reorder buffers of 8 and 16 entries. This small difference confirms that this scheme can be used, without a significant loss of precision, to detect the useless commits, thus avoiding in a simple way the problem posed by the speculative execution of instructions.

As can be seen, there are good reasons to choose Scheme 2 over Scheme 1. The complexity of implementation and the potential stalling of instruction decoding, disfavoring Scheme 1, outweighs the slight loss of precision of Scheme 2. By choosing scheme 2, we are applying the principle of "making the common case fast" [HP90] since we are avoiding the use of a complicated hardware mechanism that would only be necessary in a small percentage of the cases. As a consequence of this selection, we will modify our definition of a short-lived variable given in Section 3.3. We redefine a short- lived variable as a variable whose live ranges are short and whose definition and last-use points are found in the same basic block.

Another advantage of the use of Scheme 2 is related to the recoverability of the interrupts. As we explained in Section 3.5.3, in order to make an interrupt recoverable under our scheme, it is necessary to save the state of the whole reorder buffer before handling the interrupt. An improvement to this mechanism can be obtained if we consider that the short live ranges to be discarded do not cross the basic block boundaries. In this case, the values discarded will only be required by the instructions between the faulting instruction and the next instruction in the reorder buffer that changes the control flow, i.e., by the instructions that are in the same block of the instruction that produced the fault. According to this observation, it is not necessary to save the status of the whole reorder buffer, but only the status of the entries from the head of the reorder buffer to the next jump or branch instruction. Since jump and branch instructions constitute around 13% of the instruction mix [HP90], we will be, in general, saving less than eight entries of the reorder buffer each time an interrupt occurs.

4.3 Compiler Analysis

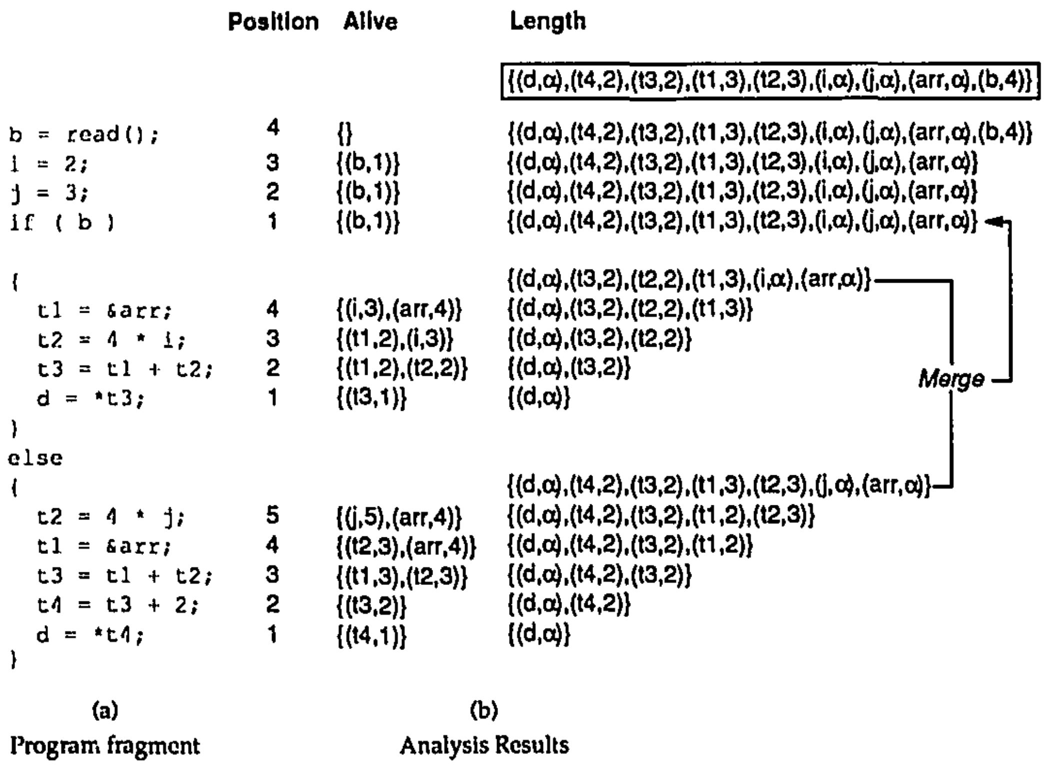
Now that we have selected the hardware mechanism and have simplified the problem in such a way that we can exactly map short live ranges to useless commits, we need to design the compiler analysis to find out which live ranges are going to be tagged so that the values produced are discarded at commit time.

This analysis, which we call the *short-live-range analysis*, must perform the following tasks:

1. Find the length of the longest live range of each variable.
2. Detect variables whose live ranges cross basic block boundaries.
3. Based on the previous information, on the size of the reorder buffer $\mathcal{CR}$, and on the processor's decode bandwidth $\mathcal{DB}$, find the variables that correspond to our definition of a short-lived variable.

The analysis is carried out on the low level abstract syntax tree representation (LAST) [Don94] of the McCAT compiler (See Appendix A). It performs a backwards walk over the tree keeping track of the definition and last-use points of each live range and the distance between these points. In order to do this, the analysis makes use of a counter which we call *Position* and two sets called *Alive* and *Length*. The *Position* counter always keeps the distance of the instruction being analyzed relative to the end of the basic block. This distance is measured in machine instructions. The *Alive* set keeps a list of the variables that are alive at the point where the analysis is being performed. It consists of a set of tuples $< x, u_x >$, where x is a variable and u_x is the position of the instruction containing the last use of x relative to the end of the basic block. The *Length* set keeps the information of the variables whose live ranges have already been analyzed. It consists of a set of tuples $< y, l_y >$, where y is a variable and l_y is the length of the longest live range found for that variable. The symbol α can be used as a value for l_y to indicate that at least one live range belonging to the variable y crosses the limits of a basic block. Since this is a backwards analysis, we are going to describe the actions taken by it when it advances, statement by statement, from the end to the beginning of a basic block. We then describe the rules used to merge the information obtained from different basic blocks. A small example, illustrating the results of the analysis for each statement of a program fragment, is given in Figure 4.4.

Initially, the analysis sets the value of the *Length* set to empty. When the analysis reaches the end of a basic block, it resets the *Position* counter and initializes the *Alive* set to be empty. For each statement in the basic block of the form $x = y \ op \ z$, the following actions are taken:



```
                      Position  Alive              Length

                                                   {(d,α),(t4,2),(t3,2),(t1,3),(t2,3),(i,α),(j,α),(arr,α),(b,4)}

b = read();           4         {}                 {(d,α),(t4,2),(t3,2),(t1,3),(t2,3),(i,α),(j,α),(arr,α),(b,4)}
i = 2;                3         {(b,1)}            {(d,α),(t4,2),(t3,2),(t1,3),(t2,3),(i,α),(j,α),(arr,α)}
j = 3;                2         {(b,1)}            {(d,α),(t4,2),(t3,2),(t1,3),(t2,3),(i,α),(j,α),(arr,α)}
if ( b )              1         {(b,1)}            {(d,α),(t4,2),(t3,2),(t1,3),(t2,3),(i,α),(j,α),(arr,α)}

{                                                  {(d,α),(t3,2),(t2,2),(t1,3),(i,α),(arr,α)}
  t1 = &arr;          4         {(i,3),(arr,4)}    {(d,α),(t3,2),(t2,2),(t1,3)}
  t2 = 4 * i;         3         {(t1,2),(i,3)}     {(d,α),(t3,2),(t2,2)}
  t3 = t1 + t2;       2         {(t1,2),(t2,2)}    {(d,α),(t3,2)}                              Merge
  d = *t3;            1         {(t3,1)}           {(d,α)}
}
else
{                                                  {(d,α),(t4,2),(t3,2),(t1,3),(t2,3),(j,α),(arr,α)}
  t2 = 4 * j;         5         {(j,5),(arr,4)}    {(d,α),(t4,2),(t3,2),(t1,2),(t2,3)}
  t1 = &arr;          4         {(t2,3),(arr,4)}   {(d,α),(t4,2),(t3,2),(t1,2)}
  t3 = t1 + t2;       3         {(t1,3),(t2,3)}    {(d,α),(t4,2),(t3,2)}
  t4 = t3 + 2;        2         {(t3,2)}           {(d,α),(t4,2)}
  d = *t4;            1         {(t4,1)}           {(d,α)}
}
```

(a) Program fragment

(b) Analysis Results

Figure 4.4: An example of the short live range analysis for a program fragment

1. Since x is being defined, then the tuple $< x, u_x >$ is removed from the *Alive* set to reflect the fact that x is dead before this instruction. The tuple $< x, u_x >$ should have been added to the *Alive* set by the analysis when it found the last use of x in the basic block. The length of the live range of x, l_x, is set to the number of instructions between this statement and the last use of x, which can be expressed as $l_x = Position - u_x + 1$. If x is not already in the *Length* set, then the tuple $< x, l_x >$ is added to the set. If a tuple $< x, l_{x_{old}} >$ is already in the *Length* set then the tuple with the maximum length is put in the set; that is, the old tuple is removed and the tuple $< x, max(l_{x_{old}}, l_x) >$ is added to the set (Note that we define $max(\alpha, l_w) = \alpha$ for any value of l_w).

2. Since y is being used, then we have to check if this is the last use of y. That is, if y is not already in the *Alive* set, then we set $u_y = Position$ and we add the tuple $< y, u_y >$ to the *Alive* set, to indicate that y is alive up to the current instruction and to record the point where the last use of y is. The same action is taken for z.

3. The *Position* counter is increased by the number of machine instructions produced

by this statement. In general, a statement in the LAST tree corresponds to one machine instruction.

In order to clarify these actions let's illustrate how they work for an statement in the example program. Let's take the instruction `t4 = t3 + 2` at the end of the else-body. Before analyzing these statement, the value for the *Alive* set is $\{(t4, 1)\}$ indicating that the last use of $t4$ was found in the instruction at position 1. The value for the *Length* set is $\{(d, \alpha)\}$ indicating that d is a variable that was previously analyzed and has at least one live range that crosses the boundaries of a basic block. The current value of *Position* is 2. Since $t4$ is being defined in this instruction, we remove $(t4, 1)$ from *Alive* and we add $(t4, 2)$ to *Length* to indicate that the live range of $t4$ has a length of 2 instructions. Since $t3$ is being used in this instruction, we add $(t3, 2)$ to the Alive set to indicate that the last use of $t3$ in this basic block is at position 2. Finally, we increment the position counter before proceeding to the next instruction. The same actions are taken for all the instructions until we reach the beginning of the basic block.

When the analysis reaches the begin of a basic block, we are certain that the variables that are in the *Alive* set cross the border of this basic block. Therefore, for each variable x in the *Alive* set a tuple $< x, \alpha >$ must be added to the *Length* set overwriting the value of any previous tuple for x. When joining the information of two basic blocks A and B, the corresponding sets $Length_A$ and $Length_B$ have to be merged into the set $Length_{Final}$ so that the length of the longest live range for each variable is preserved. The result obtained in $Length_{Final}$ is given as input for the analysis of the basic block that precedes A and B. The rules to merge the information of the length of the live ranges are the following:

1. If a variable y is in both $Length_A$ and $Length_B$, then the tuple $< y, max(l_{yA}, l_{yB}) >$ must be added to $Length_{Final}$.
2. For any tuple $< w, l_w >$ in $Length_A$ or in $Length_B$, if the variable w is either in $Length_A$ or in $Length_B$, but not in both, then the tuple $< w, l_w >$ must be added to $Length_{Final}$.

Lets consider how this rules apply in our example when we merge the information obtained for the else-body and the if-body. For example, in the else-body, the length of the live range of $t2$ is 3 instructions. For the if-body, the length of the live range of $t2$ is 2 instructions. Therefore, when merging the information we take the longest live range, i.e., $(t2, 3)$. On the other hand, from the information obtained from the else-body it can be seen that j crosses the boundaries of that basic block; therefore, (j, α) must be added to the merged $Length$ set. The same rules are applied for all the variables in both sets.

After the analysis is performed we obtain, for every variable in the routine, the length of its longest live range. With this information, and the knowledge of the length of the reorder buffer $\mathcal{LR}$ and the decode bandwidth $\mathcal{DB}$, the analysis can decide which variables are short-lived and pass this information to the code generator. The code generator can produce the code with the flagged live ranges so that the values generated by these live ranges can be discarded at commit time.

From the final result in our example (the set drawn inside the box), we can see that in this program the variables d, i, j, arr cross the basic block boundaries and, therefore, cannot be considered short-lived variables. The other variables have very short live ranges and, if we assume we have a machine with a reorder buffer of 8 entries ($\mathcal{LR}$=8) and decode bandwidth of 2 instructions per cycle ($\mathcal{DB}$=2), then all of them would be considered short-lived variables.

Chapter 5

Allocation of Short-Lived Variables

In the previous chapter, we presented the hardware and software mechanisms to reduce the number of useless commits to the register file. We also discussed the issues that we had to solve in order to achieve the goals stated in the description of the first problem to be studied in this thesis (Section 3.3). In this chapter, we describe the software scheme used to solve the second problem stated. That is, the need for a different register allocation scheme, so that variables that are short-lived are not allocated to architected registers.

Up to now, we have only discussed how to modify the architecture so that the values produced by short-lived variables are discarded at commit time. However, the register allocator continues to assign these variables to physical registers even though the associated values are never going to be put there. As we explained in Section 3.1, this constitutes a waste of resources. The names provided to address the spaces located in the register file should be used only for the variables that will, in effect, make use of them. That is, these names should be used exclusively to allocate the long-lived variables. Since the values produced by the short-lived variables are only used from the reorder buffer, these values should be allocated into the space provided by the reorder buffer. Hence, our idea is to expose the reorder buffer to the compiler as an extension of the programmable registers for storing short-lived values.

With our proposal, the space in the reorder buffer is treated as additional registers which we call *symbolic registers*. There are as many symbolic registers as entries in the reorder buffer. However, a symbolic register is not tied to any particular location in the reorder buffer. For example, if a value is assigned to the symbolic register two (`sr2`), that does not mean the value will be stored in the second entry of the reorder buffer. The actual association between a symbolic register and a reorder buffer entry is done at runtime. When the instruction defining the value of the symbolic register enters the reorder buffer, the renaming mechanism renames the symbolic register and assigns an entry in the reorder buffer to store the value being produced.

Given the set of symbolic registers, the problem now is to modify the compiler to make an efficient use of them. The register allocation process has to be modified to ensure that short-lived variables are allocated independently and are assigned to symbolic registers. In order to achieve this, we propose a register allocation scheme that is performed in four steps:

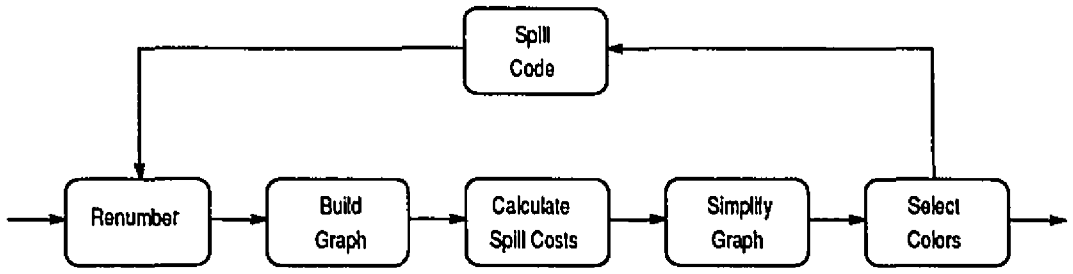
1. Short-live-range analysis.
2. Allocation of short-lived variables.
3. Modified Chaitin-like allocation for remaining variables.
4. Introduction of spill code.

These steps are depicted in Figure 5.1(b) where they can also be compared to the steps in the traditional register allocation process, which are represented in Figure 5.1(a) and explained in detail in Section 2.2. As it can be seen, we are adding two steps to the traditional Chaitin-like allocation process to allocate the short-lived variables first. As shown in the figure, the whole allocation process has to be repeated after the introduction of spill code. Later in this chapter, we will explain why the allocation of short-lived variables must also be repeated after the introduction of spill code.

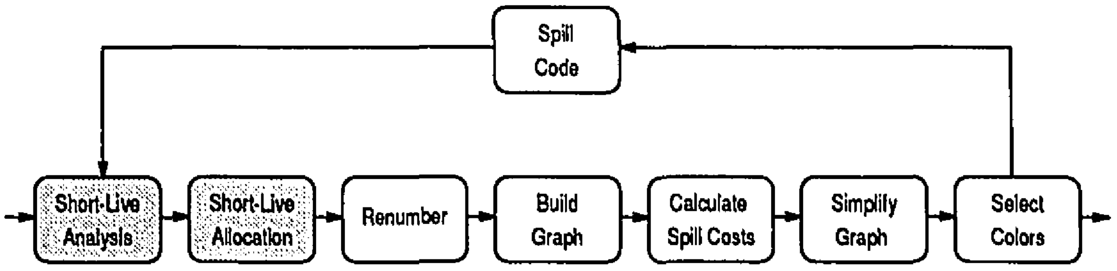
In the following subsections, we will explain, in more detail, the steps involved in the proposed allocation scheme.

5.1 Allocation of Short-Lived Variables

The first step is to perform the short-live-range analysis, as we described in Section 4.3, to find out which variables can be allocated to the symbolic registers. The second step is to assign the symbolic registers to these short-lived variables. Since the live ranges of



(a) Traditional register allocation



(b) Register allocation with short-lived variable analysis

Figure 5.1: Traditional register allocation vs. short-lived variable allocation

these variables do not cross the basic block boundaries, they can be allocated in linear time using a simplified version of the algorithm for register allocation at the basic block level presented in [ASU88].

For each basic block, the algorithm performs a backwards walk over the tree assigning symbolic registers to the short-lived variables it finds in each statement. The information on the registers assigned to the variables is stored in the tree so that it can later be accessed by the code generator. The algorithm makes use of two sets which we call *Position* and *Registers Used*. The *Position* set keeps track of the variables that already have symbolic registers assigned. Each element in the set is a tuple of the form $< x, sr_x >$, where x is a short-lived variable and sr_x is the symbolic register assigned to it. The *Registers Used* set keeps track of the symbolic registers that are currently used. Initially both sets are empty. The analysis is exemplified in Figure 5.2. In this example, we assume that the variables *a,b,c,d,i,j* are long-lived and, therefore, the algorithm does not allocate symbolic registers to them.

For each statement of the form $x \; = \; y \; op \; z$, the algorithm performs the following actions:

	Position	Symbolic registers used
`t1 = &a;`	{}	{}
`t2 = 8 * i;`	{<t1,1>}	{1}
`t3 = t2 + 4;`	{<t1,1>,<t2,2>}	{1,2}
`t4 = t1 + t3;`	{<t1,1>,<t3,2>}	{1,2}
`t1 = &b;`	{<t4,2>}	{2}
`t2 = 8 * j;`	{<t4,2>,<t1,1>}	{1,2}
`t3 = t2 + 4;`	{<t4,2>,<t1,1>,<t2,3>}	{1,2,3}
`t5 = t1 + t3;`	{<t4,2>,<t1,1>,<t3,3>}	{1,2,3}
`c  = *t4;`	{<t5,1>,<t4,2>}	{1,2}
`d  = *t5;`	{<t5,1>}	{1}

(a) Program fragment (b) Allocation of short-lived variables

Figure 5.2: An example of the analysis for the allocation of short-lived variables

1. If x is a short-lived variable, since x is being defined, then the symbolic register associated to x, sr_x, can be freed. To reflect this, the tuple $< x, sr_x >$ is removed from the *Position* set and the register assigned to it, sr_x, is removed from the *Registers Used* set.

2. If y is a short-lived variable, then the algorithm checks if y is not already in the *Position* set. If it is not, then the algorithm assigns to it one symbolic register that is not in the *Registers Used* set, say sr_y, and adds the tuple $< y, sr_y >$ to the *Position* set. Also, the symbolic register sr_y is added to the *Registers Used* set. The same action is taken for z, if z is a short-lived variable.

Using this algorithm, independent live ranges that belong to the same variable can be allocated to different registers, as for the variables *t2* and *t3* in the previous example. In this case, *t3* has 2 live ranges. In one of them, *t3* gets allocated to the symbolic register three (`sr3`). In the other, it gets allocated to the symbolic register two (`sr2`). *t2* live ranges get allocated analogously.

It is important to note here that, since the number of symbolic registers available is equal to the number of entries in the reorder buffer, and since the length of each live range does not exceed the length of the reorder buffer (the short-live-range analysis ensures this is true), the number of symbolic registers is always enough to allocate all the short-lived variables without requiring the introduction of spill code.

The assembly code produced after the symbolic registers have been allocated is shown in Figure 5.3. In this figure, we show how the code looks when the variables are allocated using the traditional Chaitin-like allocation scheme (Figure 5.3(b)), compared to the code produced when we use our proposed allocation scheme. Note that our scheme uses three symbolic registers and two physical registers compared to the five physical registers used by the traditional allocation scheme[1].

```
t1 = &a;          add r6,r30,#-36       add sr1,r30,#-36
t2 = 8 * i;       slli r3,r5,#3         slli sr2,r5,#3
t3 = t2 + 4;      addi r3,r3,#4         addi sr2,sr2,#4
t4 = t1 + t3;     add r5,r6,r3          add sr2,sr1,sr2
t1 = &b;          add r6,r30,#-40       add sr1,r30,#-40
t2 = 8 * j;       slli r3,r4,#3         slli sr3,r4,#3
t3 = t2 + 4;      addi r3,r3,#4         addi sr3,sr3,#4
t5 = t1 + t3;     add r4,r6,r3          add sr1,sr1,sr3
c  = *t4;         lw r8,0(r5)           lw r4,0(sr2)
d  = *t5;         lw r3,0(r4)           lw r5,0(sr1)
```

(a) Program fragment (b) Code produced after Chaitin-like allocation (c) Code produced after separate allocation for short-lived variables

Figure 5.3: An example of the code produced for the two allocation schemes

5.2 Allocation of the Long-Lived Variables

After all the short-lived variables are allocated, the next step is to allocate the remaining variables using Chaitin's allocator with the Briggs improvement [CAC+81, Bri92]. The advantage of our scheme at this point is that, since many of the variables have already been allocated in the previous step, in this step the size of the interference graph is smaller than what it would be for the traditional allocator, thus simplifying the problem. This also implies that fewer variables are competing for the physical registers which means a decrease in the amount of spill code required. Yet, it is still possible that during the selection of the colors for the interference graph, the Chaitin allocator decides to spill some variables to memory thus requiring the introduction of spill code.

[1]We do not count r30 here since in DLX this register is reserved to keep the value of the frame pointer

5.3 Introduction of Spill Code

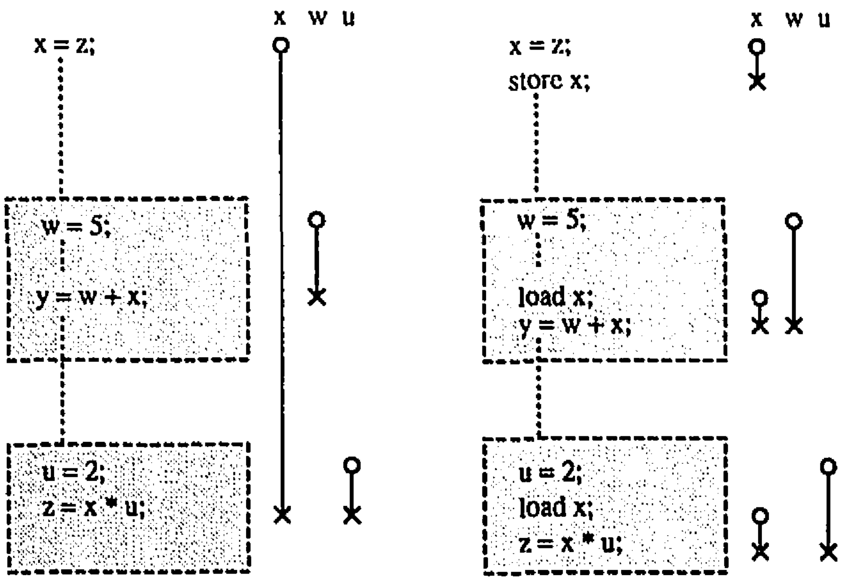
When introducing spill code, the register allocator inserts a load before each use and a store after each definition for each variable being spilled. The introduction of these load/store instructions has two effects on our mechanism:

1. The variables that are spilled now become short-lived variables.
2. The introduction of loads and stores may cause some short-lived variables to become long-lives.

These two effects can be better understood by examining the example presented in Figure 5.4. Figure 5.4(a) presents a fragment of a program together with the representation of the corresponding live ranges before the spill code is introduced. The gray boxes represent zones where the register pressure is assumed to be high enough to require the introduction of spill code[2]. In the example, x is not short-lived and has several uses, while w and u are short-lived variables. For the purposes of our example, we are going to assume that the length of the live range of w is the maximum length for w to be considered short-lived. Let us assume that, since the register pressure is high, the register allocator selects x to be spilled, and let us examine the effect of introducing spill code for this variable, as depicted in Figure 5.4(b). The first effect is that since x was spilled, its live range is now composed of several small live ranges causing x to become a short-lived variable that can be allocated using symbolic registers. This is an advantage of our scheme over Chaitin's allocator which repeats the whole allocation process again because spilled variables continue to interfere with the other variables in the interference graph.

The second effect of the introduction of spill code is that since the length of the live range for w was on the limit of being considered short, and since some instructions were introduced for the spill code of x, then the live range for w is not short anymore, and w has to be allocated using the physical registers. This forces the repetition of the whole process to check if anymore spill code is required for the variables that interfere with w. We have observed, though, that this does not happen too often and that our process converges faster than in the traditional Chaitin allocation. Finally, note that despite the fact that the live range for u is longer now, u is still a short-lived variable and can still be allocated using symbolic registers. This implies that no additional spill code is going to be introduced in this zone with high register pressure.

[2]We assume that some registers are occupied by other program variables not shown in the figure.



(a) Before spill code (b) After spill code

Figure 5.4: Effect of spill code on the allocation of short-lived variables

Chapter 6

Experimental Results

In the two previous chapters, we presented the hardware and software schemes used to solve the problems of reducing the number of useless commits to the register file and the allocation of the short-lived variables to the storage provided by the reorder buffer. In this chapter, we perform experiments using our testbed to show the effectiveness of the proposed solution.

In order to determine the effect of the suggested methods, we performed the following measurements:

1. Effectiveness of the short-live-range variable analysis in terms of the percentage of program variables that are detected to be short-lived variables, and in terms of the percentage of writes to the register files that can be discarded by using the information provided by this analysis.
2. Effect of the reduction of the write ports to the register files on the performance of the current execution model and on the performance of our proposed (optimized) model in which values produced by short-lived variables are not committed to the register file.
3. Effectiveness of the scheme for the allocation of short-lived variables to the symbolic registers in the reorder buffer in terms of the percentage of improvement in the execution time for the different benchmarks.

We start our discussion by presenting the conditions under which the experiments were carried out. First, in Section 6.1, we list the different tools used to conduct the experiments and the set of benchmarks employed. Then, in Section 6.2, we instantiate the execution model by specifying the configuration parameters for what we call the *base model*. This base model specifies the different resource sizes and latencies for the processor used in our simulations.

Before presenting the detailed results, we give a short summary to facilitate the understanding of the analyses (Section 6.3). After this, we present and analyze the detailed data obtained through the experiments (Section 6.4, Section 6.5, and Section 6.6).

6.1 Testbed and Benchmarks

All analyses and experiments were carried out using the McGill Compiler and Architecture Testbed (McCAT) testbed (For a more detailed description see Appendix A). This testbed consists of an optimizing C compiler (McCAT) [HDE+92] and a superscalar processor cycle-by-cycle simulator (SuperDLX). SuperDLX [Mou93] simulates many of the features described by Johnson [Joh91] including dynamic scheduling, branch prediction, register renaming and the use of a reorder buffer.

For our experiments, we have used 10 benchmarks which are briefly described here:

- *Alvinn:* Single precision floating point benchmark from the SPEC 92 suite. It trains a neural network using backpropagation in order to keep an autonomous vehicle on the road.
- *Bubble:* Kernel integer benchmark. Performs recursive bubble sort.
- *L8:* Loop 8 of the Livermoore loops. Double precision floating point benchmark. Chosen because it is one of the loops with the most register pressure.
- *L14:* Loop 14 of the Livermoore loops. Double precision floating point benchmark. Chosen because it is one of the loops with the most register pressure.
- *L8unroll:* Loop 8 of the Livermoore loops unrolled twice to increase register pressure.
- *L14unroll:* Loop 14 of the Livermoore loops unrolled twice to increase register pressure.

- *Linpack:* Kernel benchmark from the "LINPACK" package of linear algebra routines developed by Dongarra.
- *Quickrand:* Kernel integer benchmark. Implements a recursive version of Quicksort.
- *Tomcat:* Floating point benchmark from the SPEC 92 benchmark suite. It performs mesh generation.
- *Whetstone:* Synthetic benchmark intended to be representative of floating point intensive programming.

As it can be seen, our set of benchmarks comprises code from different types of applications. We included some integer kernels taken from non scientific applications, some floating point kernels from scientific code and some complete floating point scientific applications. We believe that although the sample of benchmarks is small, it is representative of a wide set of applications.

6.2 Base Model

All experiments were performed by simulating the behavior of the execution model described in Section 2.1.3. We instantiated this model by specifying the values of the different parameters as shown in Table 6.1. The processor resulting from this specification is our *base model* for the experiments to be carried out. Some of the parameters of the base model, like the decode bandwidth, the number of memory accesses per cycle and the memory latency, were selected according to real values found in some recent superscalar processor designs. The base model uses a reorder buffer size of 16 entries and assumes 4 write ports per register file. The different measurements are obtained by varying some parameters of the base model, e.g., the reorder buffer size between 8, 16 and 32 entries, and the number of register write ports per register file. In such cases, the sizes of the instruction dispatch window and the load and store buffers are always kept equal to the size of the reorder buffer. For simplicity, we assume that the base model and its variations have a perfect cache and we briefly comment on the effect of a non-perfect cache on our results. Also for simplicity, we assume that we have enough functional units (five of each type) to execute the operations that are not related to memory accesses and that the latency of these operations is one cycle.

Parameter	Value
Instruction queue	16 instructions
Decode bandwidth	4 instructions per cycle
Reorder buffer	16 entries
Memory latency	2 cycles
Memory accesses	1 per cycle
Address resolution functional units	1
Other functional units	5 of each type
Latency of other functional units	1 cycle
Write ports per register file	4

Table 6.1: Configuration parameters specified for the base model

6.3 Summary of Results

The major results of our experiments are:

- The short-live-range analysis can be successfully used to avoid the useless commit of instructions to the register files. The proposed analysis successfully captures most of the short-lived variables: on the average, close to 90% (89.34) of all variables are detected to be short-lived when we assume a reorder buffer with 16 entries, and more than 90% (91.92) when the reorder buffer size is increased to 32. The combined architecture and compiler scheme can effectively make use of such an analysis and eliminate a great majority of the useless writes to the register files: on the average 87% (87.98) for the base model and close to 90% (89.71) when reorder buffer size is increased to 32.

- The mechanism devised to avoid the useless commits of instructions can be used to reduce the number of write ports to the register files without affecting performance. In fact, using this mechanism we could reduce the number of write ports to one and obtain a loss on performance of only one percent. We performed these measurements with the model using a reorder buffer of 32 entries. In this way, we obtained the largest performance impact when we reduced the number of write ports.

- The proposed method for allocation of short-lived variables to symbolic registers can reduce the number of physical registers required and decrease the amount of spill code needed thus improving execution time. The average improvement compared to the traditional allocation method is substantial. When the register pressure is high (only 4 registers are effectively available) the improvement exceeds 22% (22.35%) for a reorder buffer of size 16, and 26% (26.43%) when the reorder buffer size is increased to 32.

We will further elaborate on these results in the following three sections.

6.4 The Effect of Short-live-range Analysis and Architecture Support for Useless Commit Elimination

The effectiveness of the short-live-range compiler analysis (explained in Section 4.3) and of the hardware mechanisms that allow the elimination of the useless writes to the register file (explained in Section 4.1) is illustrated in Table 6.2. In this table are the measurements of the effectiveness of our combined hardware/software mechanism for different sizes of the reorder buffer. For each size, we present the percentage of variables that were detected by the compiler to be short-lived variables, and the percentage of writes that were discarded at run time. It can be seen, that even for a small reorder buffer (of size 8), on the average more than 80% of the variables used at the low level representation of the program are being detected as short-lived variables. This is caused, as explained before, by the large number of temporal variables introduced by the compiler. Most of these variables, plus the ones that are spilled to memory by the register allocator, are successfully captured by our compiler analysis: on the average close to 90% (89.34) when the reorder buffer size is 16 (base model), and 80.13% and 91.92% when the reorder buffer sizes are 8 and 32 respectively.

Furthermore, the architecture mechanism proposed can make use of the information provided by the compiler and eliminate a great majority of useless writes to the register files. The reduction on the average is 88% (87.98) for the base model, and is 76.34% and 89.71% when the reorder buffer sizes are 8 and 32 respectively. These results are illustrated in Figure 6.1. In this graph, we show the lowest, the highest and the average percentage of discarded writes for each size of the reorder buffer. It can be seen that in some cases

Benchmark	Reorder Buffer					
	8		16		32	
	Short Variables	Discarded Writes	Short Variables	Discarded Writes	Short Variables	Discarded Writes
Alvinn	75.33	69.38	84.44	88.13	86.44	88.47
Bubble	81.25	79.56	89.58	89.76	91.67	91.81
L8	73.15	72.04	89.49	96.06	94.16	98.80
L14	83.33	83.95	88.33	88.37	91.67	90.78
L8unroll	81.37	70.90	95.24	95.60	96.70	98.34
L14unroll	89.35	83.97	92.73	88.41	94.29	90.80
Linpack	80.20	84.37	89.93	90.88	94.14	91.46
Quickrand	76.60	71.73	80.85	76.61	80.85	76.61
Tomcat	79.05	80.26	91.38	88.17	94.80	89.99
Whetstone	81.62	67.27	91.39	77.76	94.49	80.00
Average	80.13	76.34	89.34	87.98	91.92	89.71

Table 6.2: Effectiveness of the short-live-range analysis and the architecture support for reducing useless commits

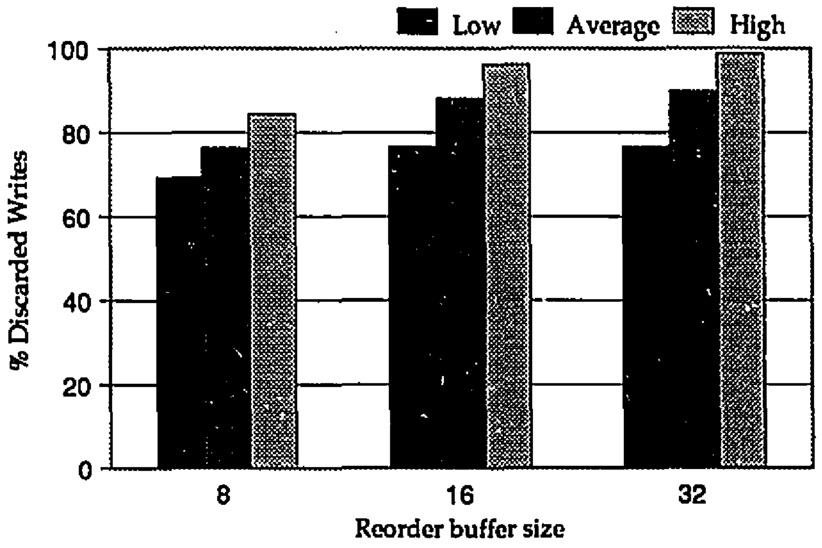


Figure 6.1: Percentage of discarded writes to the register file

the percentage of discarded writes is very high, more than 98%. On the other hand, the lowest values are still considerably high. For the base model, the lowest percentage is 76%. These results are substantial. They show that, if the proposed mechanisms are used in the design of a superscalar processor, we could implement a processor that is able to issue N instructions, but only requires a minimum of $\lceil (1 - 0.88) * N \rceil$ write ports to the register file since we are discarding 88% of the writes to the register file[1]. As mentioned in Section 2.1.2, the complexity of implementation of the register file is a very important factor in processors that are able to execute several instructions per cycle.

Moreover, if we compare the results of this table to the results presented in Table 3.1, we can see that our analysis is able to detect a great majority of the useless commits. For reorder buffers of sizes 16 and 32, the difference is less than 7%. For a reorder buffer with 8 entries, the difference is less than 14%.

6.5 The Effect of Reducing the Number of Register Ports Needed

Since the percentage of commits that can be discarded is large, we decided to measure the loss of performance of the proposed (optimized) model when the number of write ports per register file is restricted, and compare it to the loss of performance for the base model under the same restriction. The results of the comparisons are tabulated in Table 6.3 and illustrated in Figure 6.2. The left side of Table 6.3 shows the relative performance obtained by varying the number of register write ports when the compiler and architecture optimization proposed in this thesis is not applied. We report the performance of variations of the base model when the number of write ports to the register files is restricted to 1, 2 or 3 ports per register file compared (in normalized form) to the performance of the base model when the number of ports is 4. Note that, since the number of instructions decoded by cycle in the base model is also 4, we call this the *not restricted* model. Figure 6.2 shows in a bargraph the performance of the base and optimized models normalized to the not restricted model. From this figure, it can be seen that restrictions on the number of ports to the register file can seriously affect the performance of the base model. Using, for example, only one port per register file can degrade the performance of the base model by 55%. If we increase the number of ports to two per register file we still degrade performance by 18%.

[1]This is an approximation based on an average measurement. In order to be more cautious, a designer could use an approximation based on the lowest percentage of discarded writes, i.e., decide to use a minimum of $\lceil (1 - 0.76) * N \rceil$ write ports.

Benchmark	Base Model Write Ports 1	2	3	Optimized Write Ports 1	2	3
Alvinn	0.46	0.82	0.97	1.00	1.00	1.00
Bubble	0.38	0.74	0.94	1.00	1.00	1.00
L8	0.34	0.69	0.97	1.00	1.00	1.00
L14	0.44	0.85	1.00	0.96	1.00	1.00
L8unroll	0.34	0.68	0.96	1.00	1.00	1.00
L14unroll	0.44	0.87	1.00	0.96	1.00	1.00
Linpack	0.45	0.86	1.00	1.00	1.00	1.00
Quickrand	0.66	0.95	1.00	1.00	1.00	1.00
Tomcat	0.43	0.80	0.99	0.99	1.00	1.00
Whetstone	0.55	0.96	1.00	0.99	1.00	1.00
Average	0.45	0.82	0.98	0.99	1.00	1.00

Table 6.3: Performance effect of the number of write ports on the base and optimized models

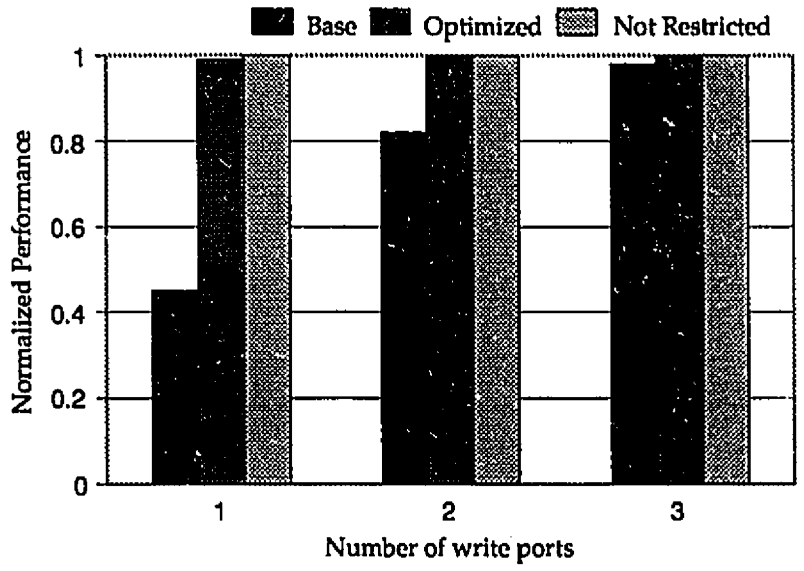


Figure 6.2: Performance of base and optimized models when restricting the number of write ports

The reader might be surprised by the variations in the performance loss of the different benchmarks when the number of register ports is reduced to one per register file (first column of Table 6.3). These variations are due to the different nature of the benchmarks used. Benchmarks with little parallelism (e.g., Quickrand) will be less affected than benchmarks with enough parallelism (e.g., Tomcat, Alvinn) to fully utilize the resources provided in the base model. Another factor that affects this measurement is the instruction mix for each benchmark. Since we are restricting the number of write ports to each register file and our model provides a register file for integer operations and another register file for floating point operations, the benchmarks in which the mix of instructions is mostly integer (e.g., Bubble) will be affected more than the benchmarks where the distribution of instructions is more even (e.g., Whetstone)

We applied the proposed optimization to the base model and performed the same measurements. The results, on the right side of the Table 6.3, show that the loss in performance is very small, as it was expected. Around 1% when using only one write port to each register file. This can be easily noticed in Figure 6.2 because the height of the bars for the optimized and the not restricted models is almost identical.

6.6 Effect of the Allocation of Short-Lived Variables

Table 6.4 and Figure 6.3 show the improvement obtained in the overall execution time of the benchmarks by using the short-lived variables allocation method explained in Chapter 5. To show the effectiveness of the method, we vary the number of registers available for the register allocation process (4, 8 and 16), and the size of the reorder buffer (8, 16 and 32) from the base model. The performance improvement is calculated with the formula $(cycles(traditional) - cycles(proposed))/cycles(traditional)$. Where *traditional* refers to the Chaitin like allocation method and *proposed* refers to our improved allocation scheme.

It can be seen from the Figure 6.3 that the higher the register pressure, the higher the improvement obtained by our method. When the register pressure is high (only 4 registers are available), the improvement is significant: over 22% (22.35%) for a reorder buffer of size 16, and 15.96% and 26.43% when the reorder buffer size is 8 and 32 respectively. As explained in Chapter 5, this improvement is obtained because the use of symbolic registers increases the number of registers available for the allocation of long-lived variables, and because the variables that would be spilled by the traditional allocator can be allocated instead to the symbolic registers thus reducing the possibilities of requiring the introduction of more spill code. It can also be seen from this figure that for a given level of register pressure (number of registers available) the improvement that can be obtained depends on the size of the reorder buffer. The reason for this is that with larger reorder buffers there are better chances to find more short-lived variables and, therefore, further reduce the register pressure.

It is important to note that we are taking all of our measurements assuming a perfect cache and that we are making use of large load/store buffers. Without the use of these features, the improvements obtained by our method would be even greater. That is, the results reported in this section are somewhat conservative and can be used as a safe estimate of the potential improvement gained from our method. In fact, in a previous technical report [LCG94], we presented these same measurements but without using the load forwarding mechanism in the load/store buffer. In that case, the improvements obtained by our mechanism were even higher. The reason for the decrease in performance improvement obtained is that the load forwarding feature helps to reduce the effect of the introduction of spill code. Since we are using large load/store buffers, there is a better chance that the loads introduced by the register allocator find the data they require in the load/store buffer. Thus, in these cases, the loads do not have to wait until the data is fetched from memory, and, therefore, the benefits of our scheme are partially hidden.

	Number of registers								
	4			8			16		
	Reorder Buffer			Reorder Buffer			Reorder Buffer		
Benchmark	8	16	32	8	16	32	8	16	32
Alvinn	21.56	16.81	15.44	0.33	0.22	0.60	0.00	0.01	0.01
Bubble	11.00	19.42	30.52	0.04	0.06	0.04	0.04	0.06	0.04
L8	15.50	28.24	31.97	5.18	6.48	6.52	0.03	0.05	0.07
L14	14.83	20.51	25.65	3.31	0.11	6.84	2.18	2.16	2.72
L8unroll	25.68	36.57	42.01	5.27	9.11	11.23	0.65	1.63	2.00
L14unroll	20.60	27.76	26.70	5.89	9.34	16.57	6.43	6.30	4.88
Linpack	12.92	15.30	27.08	0.15	1.80	1.11	0.70	1.58	1.67
Quickrand	15.89	26.94	32.50	0.23	3.59	5.56	0.00	2.26	4.17
Tomcat	11.81	16.15	14.69	3.35	8.32	7.63	1.73	3.41	4.07
Whetstone	9.76	15.83	17.73	5.27	13.10	14.29	7.64	16.20	15.88
Average	15.96	22.35	26.43	2.90	5.21	7.04	1.94	3.37	3.55

Table 6.4: Percentage of improvement obtained by allocation of short-lived variables

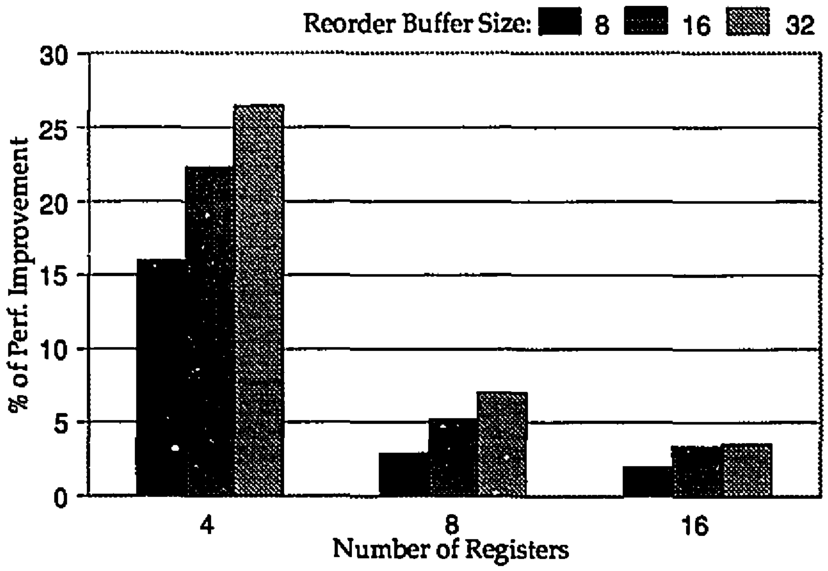


Figure 6.3: Average improvement obtained by the allocation of short-lived variables

Chapter 7

Related Work

In our research work we have discussed issues like the reduction of ports to the register file, the exposition to the compiler of the renaming capabilities provided by the reorder buffer, and modifications to the register allocation process to exploit the occurrence of short-lived variables. In this chapter, we briefly describe the work related to these issues. For each study we give a brief description and, whenever applicable, we compare it against our research.

In [PS88], Pleszkun and Sohi remark upon the occurrence of the useless commit of instructions to the register file when using state maintenance mechanisms like the reorder buffer or the Register Update Unit – RUU [SV87]. As mentioned in Section 2.1.1, the RUU is an implementation of a centralized instruction window that also supports the features of the reorder buffer. The authors study, among other factors, the performance increase obtained by the out-of-order issue of instructions with the support of the RUU. They briefly discuss the complexity of the implementation of the buses between the functional units, the RUU and the register file. They conclude that because of the RUU capability of forwarding the generated values to other instructions in the RUU, the number of read ports from the register file can be reduced thus reducing the complexity of the buses. They also notice that a register update is not necessary if there is a succeeding instruction in the RUU unit that updates the value of the same register. Based on this, they suggest that the number of write ports to the register file could also be reduced without affecting performance. However, they do not perform any experiments on these observations. The authors do not mention how the unnecessary writes to register can be avoided in the presence of speculative execution and interrupts.

In [PGH+87], Pleszkun et al. propose exposing the structure of the reorder buffer to the compiler to improve the performance of the processor. In their model the compiler can use the reorder buffer in 3 ways: 1) to access old register values while new ones are being computed, 2) to access values that been generated but have not been committed to the register file, and 3) to select instructions to be discarded after they have been issued but before they have committed. With these features, the compiler is able to improve the scheduling of the instructions and provide speculative execution. As in our research, the authors also face the problem of determining at compile time which instructions are present in the reorder buffer when another one is decoded. To solve this problem they propose to keep the number of instructions in the reorder buffer constant by allowing one instruction to commit only if another one has been put at the end of the buffer. This research work also proposes to handle the interruptions by saving the state of the reorder buffer and they note that this process can be expensive. Our work has been influenced by theirs, but our objectives are different. Their work does not make explicit use of the renaming capabilities of the reorder buffer. Also, since in their model at most one instruction is committed each cycle, they do not have to consider the problem of reducing the complexity of the register file and the associated busses.

In [FS92], Frankin and Sohi make an extensive analysis of the characteristics of the communication between instructions through registers. In their study they conclude that many of the values generated are used only once and that most of the values are dead soon after they have been created, i.e., between 30-40 instructions later. They also observe that, by using a mechanism to buffer 30 or more instructions, at least 80% of the writes to the register file are unnecessary. These observations resulted from analyses of code produced by two separate commercial compilers. Moreover, the authors briefly discuss compiler mechanisms that could be used to reduce the number of writes to the register file. However, in the mechanisms proposed they do not consider the effect of speculative execution or the problem of handling interrupts. Our experimental results reported in Section 3.2 and Chapter 6 were derived independently. The observation that most of the short-lived variables have live ranges within a basic block is new. Also, our solution strategy and implementation scheme are novel, and can effectively work in the presence of speculative execution. Finally, the application of their observations is different from ours: they use their results in the design of the Multiscalar processing paradigm in order to reduce the traffic between the elements of the distributed register file that this model uses.

Uht and Johnson [UJ92] discuss different issues related to the complexity of the data path in a highly concurrent machine. Their execution model has 32 processing elements and provides special hardware support for loop execution. Their model also supports register renaming by using a mechanism called the "shadow sink matrix". The shadow sink matrix provides a different set of renamed registers for each iteration of the loop. They observe that building a data path for such a highly concurrent machine can be extremely difficult or even impossible. To reduce the complexity of the data path they propose to duplicate the sink matrix so that each processing element can read values from their own matrix. Also, they propose to extend the model so that writes to scalar variables can be eliminated if they are superseded by writes to the same variables in succeeding loop iterations. They report that with this model they can eliminate 72% of the register writes thus allowing further simplification of the data path. Their mechanism is quite interesting, but their architecture model is very different from ours.

Finally, in [HC94], Hoogerbrugge and Corporaal study the register file port requirements of Transport Triggered Architectures (TTA). In this type of architecture, the processor is not programmed by specifying the operations to be executed, which causes implicit data movement between the register files and the functional units. Rather, the program explicitly specifies the data movements. Therefore, an instruction of the form:

```
add r1, r2, r3
```

is converted to these three instructions:

```
r2 -> alu1.add_o; r3 -> alu1.add_t; alu1.add_r -> r1
```

which establish the movement between registers and specific functional units. In this small example, `r2` is moved to the functional unit 1 (alu1) as an operand for the add operation, `r3` is moved to the same functional unit as the trigger for the add, and, finally, the result of the add is sent from the functional unit to `r1`. The researchers show that this type of architecture requires less ports to the register file. The reason is that their instructions can be used to forward a value between two functional units thus avoiding the access to the register file. Their experiments show that this method can eliminate 50% of the read traffic and 65% of the write traffic to the register file. Our hardware/software scheme can eliminate a larger percentage of writes to the register file because of the buffering of instructions in the reorder buffer. This buffering allows the forwarding of values between instructions that are not as close together as it is required in the TTA forwarding mechanism.

Although the works listed here are related to some of the issues we have examined in this thesis, we are not aware of any research that explicitly shows how to modify the register allocator so as not to assign short-lived variables to physical registers, thus reducing register pressure and spill cost.

Chapter 8

Conclusions and Future Work

Modern commercial superscalar processors are now able to use features that for years were exclusively used by supercomputers. Hardware features like branch prediction and dynamic scheduling supported by mechanisms like the reorder buffer or the Register Update Unit are becoming standard elements in the design of current and future microprocessors. Moreover, the trend of introducing more hardware features on a single chip is expected to increase as the transistor densities continue to climb [FPT94]. Accordingly, in the near future we expect to see superscalar processors with larger on-chip caches, larger register renaming buffers and larger windows for dynamic scheduling. The introduction of these features poses new challenging problems for the compiler.

In this thesis we have seen that by allowing the compiler to access the different resources used by the architecture, the implementation of some features in hardware can be simplified and a better utilization of the provided resources can be obtained. In particular, we have demonstrated how compiler and architecture techniques can be combined to take advantage of the fact that many program variables are short-lived. Our implementation and experimental results have provided ample evidence that the proposed optimizations can be effective and may lead to significant performance improvements with relatively few architectural modifications.

One possible direction for future research is to continue improving the scheme for the allocation of the short-lived variables to the locations provided by the reorder buffer. One way to do this is by using a modified version of live range splitting [Bri92]. Our current implementation of the register allocator naively introduces loads/stores for each use/definition of a spilled variable. We believe this can be improved by splitting the live ranges of spilled variables in such a way that the resulting range fragments are short enough to be considered short-live ranges, but long enough to span several uses of the same variable. In this way, the new live ranges could still be allocated to the locations in the reorder buffer and we would further reduce the number of load/stores introduced in the spilling process.

In this thesis we have studied how the register allocation process can take advantage of the exposure of hardware mechanisms like the renaming buffers. Another direction of investigation would be to analyze whether other compiler optimizations could benefit from the knowledge of mechanisms like the reorder buffer. In particular, it would be interesting to study the effect on the problem of instruction scheduling. If the scheduler had knowledge of the size of the reorder buffer if could, for example, decide in some cases not to separate the producer and the consumer of a value by a number of instructions greater than the size of the reorder buffer. In this way, it is possible that the live range associated to the value becomes a short-live range and we could use our allocation scheme to allocate this range into the locations provided by the reorder buffer thus decreasing the register pressure.

Bibliography

[AA93] Donald Alpert and Dror Avnon. Architecture of the Pentium Microprocessor. *IEEE Micro*, 13(3):11–21, June 1993.

[AAD+93] Tom Asprey, Gregory S. Averill, Eric Delan, Russ Mason, Bill Weiner, and Jeff Yetter. Performance Features of the PA7100 Microprocessor. *IEEE Micro*, 13(3):22–35, June 1993.

[AH82] Marc Auslander and Martin Hopkins. An overview of the PL.8 compiler. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 22–31, Boston, Massachusetts, June 23–25, 1982. ACM SIGPLAN. *SIGPLAN Notices*, 17(6), June 1982.

[AKC86] R. D. Acosta, J. Kjelstrup, and Torng H. C. An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors. *IEEE Transactions on Computers*, 35(9):815–828, September 1986.

[ASU88] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers — Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, Massachusetts, corrected edition, 1988.

[BCKT89] Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Coloring heuristics for register allocation. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 275–284, Portland, Oregon, June 21–23, 1989. ACM SIGPLAN. *SIGPLAN Notices*, 24(7), July 1989.

[BP93] Michael Butler and Yale N. Patt. A Comparative Performance Evaluation of Various State Maintenance Mechanisms. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 70–79, Austin, Texas, December 1–3, 1993. IEEE-CS TC-MICRO and ACM SIGMICRO.

[Bri92] Preston Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, Houston, Texas, April 1992. Published as Rice COMP TR92-183.

[BW90] H. B. Bakoglu and T. Whitside. RISC System/6000 hardware overview. In Mamata Misra, editor, *IBM RISC System/6000 Technology*, pages 8–15. International Business Machines Corporation, first edition, 1990. Order No. SA23-2619.

[CAC+81] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, January 1981.

[CDN92] Andrea Capitanio, Nikil Dutt, and Alexandru Nicolau. Partitioned register files for VLIWs: A preliminary analysis of tradeoffs. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 292–300, Portland, Oregon, December 1–4, 1992. ACM SIGMICRO and IEEE-CS TC-MICRO.

[CF87] Ron Cytron and Jeanne Ferrante. What's in a name? or the value of renaming for parallelism detection and storage allocation. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 19–27, St. Charles, Illinois, August 17–21, 1987.

[CH90] Fred C. Chow and John L. Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, October 1990.

[Cha82] G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 98–105, Boston, Massachusetts, June 23–25, 1982. ACM SIGPLAN. *SIGPLAN Notices*, 17(6), June 1982.

[CK91] David Callahan and Brian Koblenz. Register allocation via hierarchical graph coloring. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 192–203, Toronto, Ontario, June 26–28, 1991. ACM SIGPLAN. *SIGPLAN Notices*, 26(6), June 1991.

[DA92] Keith Diefendorff and Michael Allen. Organization of the Motorola 88110 superscalar RISC microprocessor. *IEEE Micro*, 12(2):40–63, April 1992.

[DFL72] J. B. Dennis, J. B. Fosseen, and J. P. Linderman. Data flow schemas. In *International Symposium on Theoretical Programming*, number 5 in Lecture Notes in Computer Science, pages 187–215. Springer-Verlag, Berlin, 1972.

[DHB89] James C. Dehnert, Peter Y.-T. Hsu, and Joseph P. Bratt. Overlapped loop support in the Cydra 5. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–38, Boston, Massachusetts, April 3–6, 1989. ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society. *Computer Architecture News*, 17(2), April 1989; *Operating Systems Review*, 23, April 1989; *SIGPLAN Notices*, 24, May 1989.

[DOH94] Keith Diefendorff, Rich Oehler, and Ron Hochsprung. Evolution of the PowerPC Architecture. *IEEE Micro*, 14(2):35–49, April 1994.

[Don94] Christopher M. Donawa. The design and implementation of a structured backend for the McCAT C compiler. Master's thesis, McGill University, Montréal, Québec, March 1994.

[FERN84] Joseph A. Fisher, John R. Ellis, John C. Ruttenberg, and Alexandru Nicolau. Parallel processing: A smart compiler and a dumb machine. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 37–47, Montréal, Québec, June 17–22, 1984. ACM SIGPLAN. *SIGPLAN Notices*, 19(6), June 1984.

[Fis83] Joseph A. Fisher. Very long instruction word architectures and the ELI-512. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 140–150, Stockholm, Sweden, June 13–17, 1983. *Computer Architecture News*, 11(3), June 1985.

[FPT94] Matthew Farrens, Andrew R. Pleszkun, and Gary Tyson. A Study of Single-Chip Processor/Cache Organizations for Large Number of Transistors. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 338–347, Chicago, Illinois, April 18–21, 1994. IEEE Computer Society and ACM SIGARCH. *Computer Architecture News*, 22(2), April 1994.

[FR91] Joseph A. Fisher and B. Ramakrishna Rau. Instruction-level Parallel Processing. *Science*, 253:1233–1241, September 1991.

[FS92] Manoj Franklin and Gurindar S. Sohi. Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 236–245, Portland, Oregon, December 1–4, 1992. ACM SIGMICRO and IEEE-CS TC-MICRO.

[GSS89] Rajiv Gupta, Mary Lou Soffa, and Tim Steele. Register allocation via clique separators. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 264–274, Portland, Oregon, June 21–23, 1989. ACM SIGPLAN. *SIGPLAN Notices*, 24(7), July 1989.

[HC94] Jan Hoogerbrugge and Henk Corporaal. Register file port requirements of Transport Triggered Architectures. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, San Jose, California, November 30–December2, 1994. ACM SIGMICRO and IEEE-CS TC-MICRO. To appear.

[HDE+92] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. In Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, number 757 in Lecture Notes in Computer Science, pages 406–420, New Haven, Connecticut, August 3–5, 1992. Springer-Verlag. Published in 1993.

[HGAM93] Laurie J. Hendren, Guang R. Gao, Erik R. Altman, and Chandrika Mukerji. A register allocation framework based on hierarchical cyclic interval graphs. *The Journal of Programming Languages*, 1(3):155–185, 1993.

[HP87] Wen-mei W. Hwu and Yale N. Patt. Checkpoint Repair for Out-of-order Execution Machines. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 18–26, Pittsburgh, Pennsylvania, June 2–5, 1987. IEEE Computer Society and ACM SIGARCH. *Computer Architecture News*, 15(2), June 1987.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

[Hsu94] Peter Hsu. Designing the TFP Microprocessor. *IEEE Micro*, 14(2):23–33, April 1994.

[Joh91] Mike Johnson. *Superscalar Microprocessor Design.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1991.

[Jol91] Richard D. Jolly. A 9-ns, 1.4-Gigabyte/s, 17-ported CMOS Register File. *IEEE Journal of Solid-State Circuits,* 26(10):1407–1412, October 1991.

[Kel75] Robert M. Keller. Look-ahead processors. *ACM Computing Surveys,* 7(4):177–195, December 1975.

[KH93] P. Kolte and M.J. Harrold. Load/Store Range Analysis for Global Register Allocation. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation,* pages 268–277, Albuquerque, New Mexico, June 23–25, 1993. ACM SIGPLAN. *SIGPLAN Notices,* 28(6), June 1993.

[LCG94] Luis A. Lozano C. and Guang R. Gao. Effective utilization of the reorder buffer for short-lived variables. ACAPS Technical Memo 86, School of Computer Science, McGill University, Montréal, Québec, July 1994.

[LS84] J. K. F. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *Computer,* 17(1):6–22, January 1984.

[McG90] Steve McGeady. Inside Intel's i960CA superscalar processor. *Microprocessors and microsystems,* 14(6):385–396, July 1990.

[Mou93] Cecile Moura. SuperDLX – a generic superscalar simulator. ACAPS Technical Memo 64, School of Computer Science, McGill University, Montréal, Québec, April 1993.

[NP94] Cindy Norris and Lori L. Pollock. Register allocation over the Program Dependence Graph. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation,* pages 266–277, Orlando, Florida, June 20–24, 1994. ACM SIGPLAN. *SIGPLAN Notices,* 29(6), June 1994.

[PF92] Todd A. Proebsting and Charles N. Fischer. Probabilistic Register Allocation. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation,* pages 300–310, San Francisco, California, June 17–19, 1992. ACM SIGPLAN. *SIGPLAN Notices,* 27(7), July 1992.

[PGH+87] A. R. Pleszkun, J. R. Goodman, W.-C. Hsu, R. T. Joersz, G. Bier, P. Woest, and P. Schechter. WISQ: A restartable architecture using queues. In *Proceedings of the 14th Annual International Symposium on Computer Architecture,* pages

290–299, Pittsburgh, Pennsylvania, June 2–5, 1987. IEEE Computer Society and ACM SIGARCH. *Computer Architecture News*, 15(2), June 1987.

[PS88] A. R. Pleszkun and G. S. Sohi. The performance potential of multiple functional unit processors. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 37–44, Honolulu, Hawaii, May 30–June 2, 1988. IEEE Computer Society and ACM SIGARCH. *Computer Architecture News*, 16(2), May 1988.

[PSS+91] V. Popescu, M. Schultz, J. Spracklen, G. Gibson, B. Lightner, and D. Isaman. The Metaflow architecture. *IEEE Micro*, 11(3):10–13, June 1991.

[RF93] B. R. Rau and J. A. Fisher. Instruction-level parallel processing: History, overview and perspective. *Journal of Supercomputing*, 7:9–50, May 1993.

[SD94] Peter Song and Marvin Denman. *The PowerPC 604 RISC Microprocessor*. Motorola; IBM Corporation, 1994.

[Smi89] James E. Smith. Dynamic Instruction Scheduling and the Astronautics ZS-1. *Computer*, 22(7):21–35, July 1989.

[SP88] James E. Smith and Andrew R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, 37(5):562–573, May 1988.

[SV87] Gurindar S. Sohi and S. Vajapeyam. Instruction Issue Logic for High-Performance Interruptable Pipelined Processors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 27–34, Pittsburgh, Pennsylvania, June 2–5, 1987. IEEE Computer Society and ACM SIGARCH. *Computer Architecture News*, 15(2), June 1987.

[SW94] James E. Smith and Shlomo Weiss. PowerPC 601 and Alpha 21064: A Tale of Two RISCs. *Computer*, 27(6):46–58, June 1994.

[Tho64] J. E. Thornton. Parallel operation in the Control Data 6600. In *Proceedings of the AFIPS Fall Joint Computer Conference*, 1964.

[Tom67] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, January 1967.

[UJ92] August K. Uht and Darin B. Johnson. Data path issues in a highly concurrent machine. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 115–118, Portland, Oregon, December 1–4, 1992. ACM SIGMICRO and IEEE-CS TC-MICRO.

[Wal86] David W. Wall. Global register allocation at link time. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 264–275, Palo Alto, California, June 25–27, 1986. ACM SIGPLAN. *SIGPLAN Notices*, 21(7), July 1986.

[WHKM93a] Steven W. White, Phil D. Hester, Jack W. Kemp, and G. Jeanette McWilliams. How Does Processor MHz Relate to End-User Performance. *IEEE Micro*, 13(4):8–15, August 1993. Part 1: Pipelines and Functional Units.

[WHKM93b] Steven W. White, Phil D. Hester, Jack W. Kemp, and G. Jeanette McWilliams. How Does Processor MHz Relate to End-User Performance. *IEEE Micro*, 13(5):79–88, October 1993. Part 2: Memory Subsytem and Instruction Set.

[YP93] Tse-Yu Yeh and Yale N. Patt. A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 257–266, San Diego, California, May 17–19, 1993. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News*, 21(2), May 1993.

Appendix A

The McCAT testbed

All of our analyses and experiments were carried out using the McCAT testbed. The McGill Compiler and Architecture Testbed (McCAT) was developed to test different compilation techniques on different architecture targets. It was designed with two objectives in mind: first, build a compiler that supports both high level and intermediate representations in order to facilitate different analyses and transformations; and second, build architecture simulator tools to test the output of the compiler, experiment with different combined hardware-software mechanisms and produce various performance results.

A.1 The McCAT C Compiler

As shown in Figure A.1, the McCAT compiler is an optimizing/parallelizing C language compiler based on three structured intermediate representations [HDE+92]. This family of representations supports pervasive flow information, that is, flow information obtained from analyzing one intermediate representation can be utilized by lower intermediate representations to perform their analyses. FIRST, a modified version of the GNU GCC compiler front-end, separates the front-end processing of parsing and type checking from the back-end phase of analysis, transformations and code generation. In the second phase of the compiler, FIRST is transformed to SIMPLE, an abstract syntax tree (AST) suitable for high level analyses like the alias and dependence testing analyses. In the next phase, LAST – a low level AST representation – is obtained from SIMPLE. In LAST, low-level

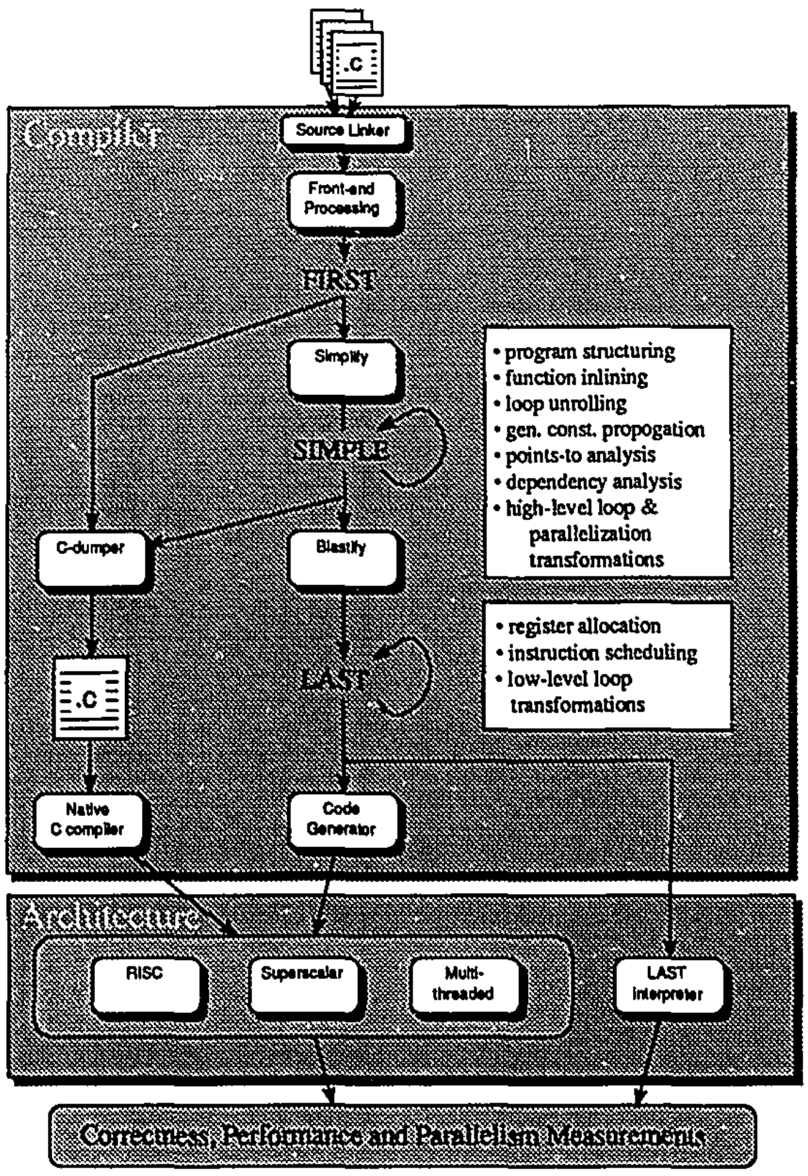


Figure A.1: The McCAT compiler

architectural details are exposed so that optimizations like register allocation and instruction scheduling can be performed. The LAST intermediate representation is very close to assembly language. In most cases, there is a one-to-one correspondence between LAST statements and assembly language instructions. However, high-level constructs such as For and While loops are still represented in order to maintain the structured representation of the program. Using LAST, code can be generated for different architectures during the last phase of the compiler. Currently, we have implemented a code generator for the DLX architecture [HP90], and code generators for SPARC, RS6000 and MIPS architectures are under development.

A.2 The SuperDLX Superscalar Simulator

SuperDLX is one of the simulation tools developed for McCAT. This simulator was developed by Cecile Moura as part of her Master's project and is described in [Mou93]. Figure A.2 gives a block diagram of the simulated architecture, which is a superscalar version of *dlxsim*, the simulator for the DLX architecture [HP90] developed at the University of California. SuperDLX simulates many of the features described by Johnson [Joh91] including dynamic scheduling, branch prediction, speculative execution, register renaming and load bypassing and forwarding. The simulator is completely configurable and provides a statistics module which allows performance and resource usage analyses.

Some of the configuration parameters that can be given to the simulator are:

- Number of entries in the reorder buffer, the instruction window and the load/store buffers.
- Number of functional units and latency of execution for each type of unit.
- Maximum number of instructions fetched, decoded and committed each cycle.
- Latency of memory operations and number of simultaneous memory accesses.
- Use of branch prediction and size of the branch target buffer.

As part of this thesis, new features were added to the simulator in order to carry out our experiments. These features include:

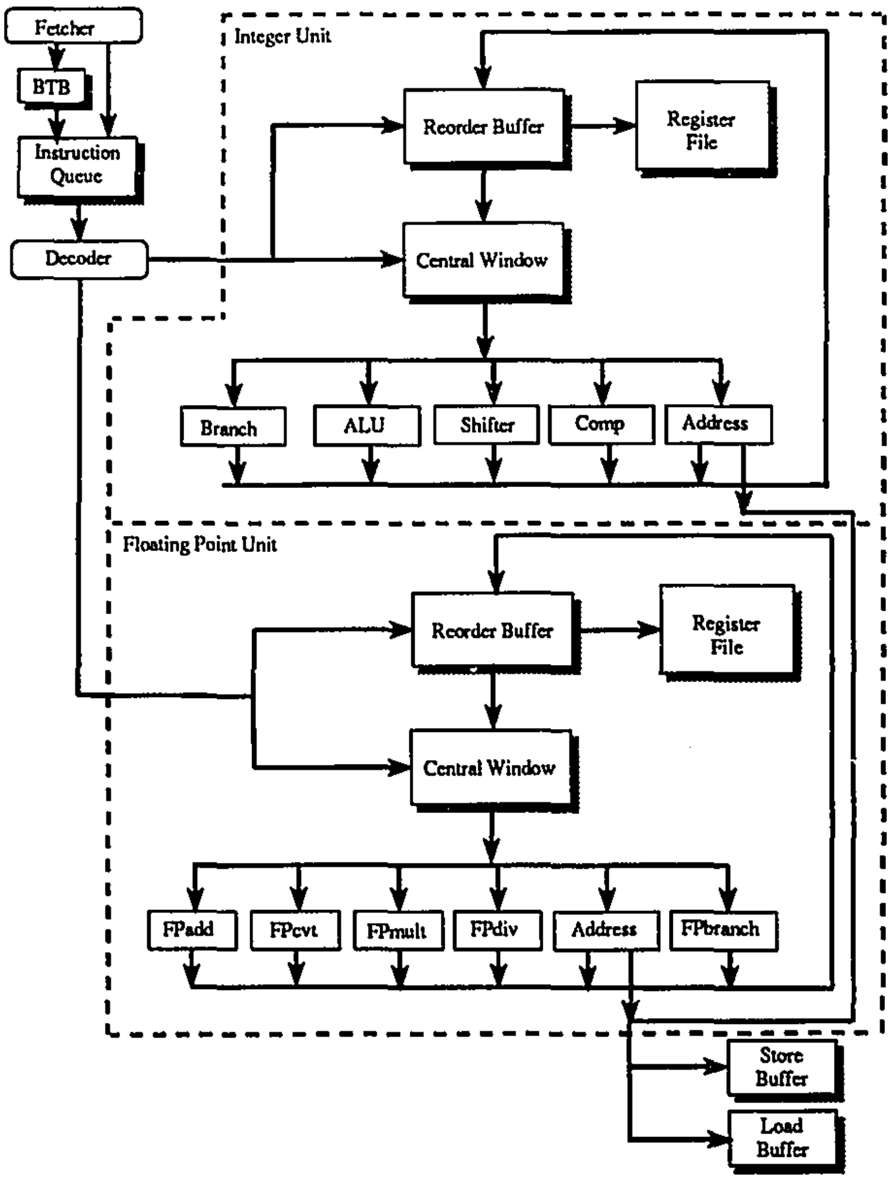


Figure A.2: The SuperDLX simulator

- Support for the execution of code with filled delay slots.
- A mechanism to control the maximum depth of speculation.
- The implementation of the mechanism for load bypassing.
- A mechanism to restrict the number of read and write ports to the register file.
- A mechanism to keep the reorder buffer filled with a minimum number of instructions.
- Statistics to measure the number of useless writes to the register file, and a mechanism to find the value of this measurement when the program is executed on a basic block by basic block basis (used for the measurements performed in Section 4.2).