Region-based Register Allocation for EPIC Architectures

by

Hansoo Kim

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

January 2001

Dissertation Advisor

Acknowledgments

I would like to thank my supervisor, Krishna Palem of the Computer Science Department at New York University. His broad knowledge, continued support and sharp advice made the completion of this thesis possible. I would like to greatfully acknowledges my thesis committee: Jack Schwartz, Uzi Vishkin, Vinod Kathail, Ben Goldberg, and Ralph Grishman for spending their precious time. Especial thanks to Vinod Kathail and Bhagi Narahari for his direct advice and encouragement.

Kanchi Gopinath and Zvi Kedem helped me a lot as unofficial members of my committee; their interest and criticism is greatly appreciated. This thesis benefited from discussions with many people. In particular, I would like to express my thanks to Bob Rau, Ken Lueh, Weng-Fai Wong and Scott Mahlke.

I also wish to thank the members of the Trimaran Project at New York University for their technical conversations and sharp questions: Allen Leung, Suren Talla, Igor Petchanski, Jinwoo Kim and Rodric Rabbah.

For the encouragement of my family and my wife's family I am very grateful. Finally, my deepest thanks go to my wife, Sunghee Ha, for her unlimited sacrifice, encouragement and support. I am especially indebted to her for taking care of our two sons, William and Lewin, providing me the time to work.

Abstract

Instruction-level parallelism(ILP) refers to a family of processor and compiler design techniques that speed up execution by allowing individual machine operations to execute in parallel. *Explicitly Parallel Instruction computing (EPIC)* processors evolved in an attempt to achieve high levels of ILP without significant hardware complexity. To take advantage of higher level of ILP in EPIC, the ILP compiler must use aggressive and expensive optimization techniques leading to increased compilation time.

If the size and shape of the compilation unit is limited, the compilation time can be reduced. But this limited scope of compilation may restrict the scope of the optimization, thus limiting the amount of performance improvement that can be achieved. As a result, the compiler may generate less efficient code. Region-based compilation has been proposed as an approach for coping with this problem, namely containing compilation cost without compromising execution performance. In region-based compilation, execution frequencies are used to guide compiler optimizations, with more attention given to the regions of the program with a higher frequency of execution, thus achieving greater overall performance improvements for the same compilation cost.

In this thesis, we address the problem of the compilation time and execution performance trade-off in region-based compilation, within the context of the key

optimization of *register allocation*. We demonstrate that schemes designed for region-based allocation perform as well as or even better than schemes designed for global register allocation while having much smaller compilation time. To achieve this goal, we innovated novel techniques which form the core of this thesis.

We show compilation time savings of 40% on the average, with comparable execution time performance, by synthesizing our techniques in a region-based register allocation. We also explore the relation between the performance of the register allocation and the region size and quantify it. Our research shows that selecting the right sized region has an impact on the performance of register allocation. Based on this observation, we propose the concept of restructuring the regions based on register pressure, and develop techniques for estimating the register pressure in order to improve compilation time while maintaining the execution time.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Instruction-level parallelism(ILP) is a family of processor and compiler design techniques that speed up execution by allowing individual machine operations, such as memory loads and stores, integer additions and floating point multiplications, to execute in parallel. The operations involved are normal RISC-style operations, and the system is handed a single program written with a sequential processor in mind. An important feature of these ILP techniques is that, unlike traditional multiprocessor parallelism and massively parallel processing, parallelism is largely transparent to users. *Explicitly Parallel Instruction computing (EPIC)* [1] processors evolved in an attempt to achieve high levels of ILP without the hardware complexity that is required in superscalar processors. In

[1]The joint program between Intel and HP has announced products such as Itanium in the IA-64 family

EPIC processors, most of the functions to extract ILP are performed by the compiler, while these are done at run-time by the hardware in superscalar processors. The compiler for an EPIC processor must specify exactly which functional unit each operation should be executed on and exactly when each operation should be issued - thus, the compiler plays a more critical role than ever [42] [44]. With EPIC making inroads into general purpose computing, compilers will play an ever-increasing and central role in optimizing the performance of applications.

The implementation of EPIC design philosophy enables new levels of parallelism and redefines the sequential execution model that exists in traditional architecture. For example, the innovative use of predication and speculation uniquely combined with explicit parallelism by the compiler, allows EPIC to progress well beyond the limitations (like mis-predicted branches and memory latency) of traditional architectures. To extract a higher level of ILP in these architectures, the ILP compiler must use aggressive ILP optimization techniques. To satisfy the need for more ILP, many techniques have been designed including inter-procedural optimization and function inlining [1] [39] [25].

However, this opportunity for improved performance comes at the price of increased compilation time. For example, function inlining [2] results in very large function bodies that makes aggressive global analysis and transformation tech-

[2]The function inlining technique replaces a function call with the function body.

niques, such as global data-flow analysis and register allocation inefficient and intractable [22]. The size of the compilation unit has a positive correlation to the amount of ILP we can extract, but it incurs a trade off in compiler performance in terms of compile time and memory utilization. In situations in which the compiler time and memory usage becomes too large, the aggressiveness of the applied transformations must be scaled back to avoid excessive compilation cost.

To help overcome this problem, *region-based compilation* [23] has been proposed as a solution in which the fundamental compilation unit is controlled by the compiler completely. Essentially, the compiler is allowed to repartition the program into a new set of compilation units, called regions. The benefits of region-based compilation is that the compiler can select compilation units that reflect the dynamic behavior of the program and reduce the complexity of compilation by focusing the optimization effort on regions with a very high payoff—typically regions that are visited with high frequency. The benefits of region-based compilation can also be applied to register allocation.

A significant challenge, therefore, is to get as much performance from region-based approaches as can be obtained from their classical "global counterparts". It is of course more desirable to get performance that is even better than conventional global schemes while paying a lower price in terms of compilation time. For instance, the relatively smaller scope of region-based compilation limits the aspects

of the register allocation process as compared to global register allocation. Under the region-based framework model [23], each region may be compiled completely before compilation proceeds to the next region, and register allocation is also performed in each region independently. As a consequence, register mismatches (*i.e.* different registers allocated to the same variable in different regions) between region boundaries can occur in this approach, and these may degrade both register allocation time and execution performance seriously.

1.1 Scope and Objectives

In this thesis, we address the problem of the compilation time and execution performance trade-off in region-based compilation within the context of the key optimization of *register allocation*. Specifically, we demonstrate that schemes designed for region-based allocation perform as well as, or even better than schemes designed for global allocation while taking smaller compilation time. To achieve this goal, we propose several innovative techniques which form the core of this thesis. Our innovations are centered on knowing and using the *execution frequencies* of the program units (such as basic-blocks, super-blocks and hyper-blocks) to guide our register allocation steps. This approach is attractive for two reasons. First, execution frequencies are an inherent part of many region-based approaches to compilation and hence will be available quite naturally. Second, they help guide

optimization decisions using a firm quantitative framework, when compared to the alternate choice of using ad-hoc approaches.

Another aspect that should be considered in modern EPIC compilers is architectural innovations, including predication. To support even higher levels of ILP, it has been announced [43] that commercial EPIC architectures will include predicate execution support. The predicated code provides many challenges and opportunities to the various phases of optimizing compilers including register allocation. For example, many blocks from different control flow paths can be grouped into a single sequence of predicated instructions for compiler optimizations and instruction scheduling - these regions can be formed into *hyper-blocks*. We also study the impact of the predicated code in our work of register allocation.

1.2 Main Contributions

1.2.1 Live Range Splitting

Live range splitting divides a single live range into several pieces and expects the new, smaller live ranges to have a reduced degree of interference, thereby facilitates an efficient register allocation. Live range splitting is the basis of Chow's priority based coloring approach which avoids spilling when splitting is possible. Unfortunately, previous approaches are performed in an ad-hoc manner or use

program structures, but do not consider execution frequency. In Chapter 4, we introduce a new live range splitting technique based on execution frequency and explore it further with rematerialization. We also studied the problems related to predicated codes and the splitting of predicated live ranges followed by proposed solutions.

1.2.2 Register Assignment

The limited scope of region-based register allocation has many potential problems. When a variable is live across many regions, coloring each region independently may be suboptimal as each live segment may be assigned with a different register, thus resulting in a register mismatches. In this case, we need some extra operations to reconcile these miss-matches. The eventual assignment of live-ranges to registers might have to be made globally since, sometimes, a live-range with a lower priority within a region might have a very high priority when viewed across all the regions that it traverses. We introduce the notion of *delayed binding* and *propagation based on frequency* to help overcome this challenge. We proposed three techniques to reduce the possibility of the miss-matches we explained above. All three techniques are based on execution frequency.

1.2.3 Priority Function

The order of coloring is one of the most important factors in register allocation, since the quality of register allocation is governed mostly by it. The base of coloring order is the priority function which is based on the amount of spill code that can be saved by allocating a variable to a register as opposed to spilling the live range of the variable. In the region-based approach where many variables are live-in and live-out, the priority function should consider the effect of the register bindings of these values. Codes inside a hyper-block provides another challenge to the priority-based approach. Unlike the operations in a basic block where the execution frequency of every operation is the same, the operations in hyper-block varies by the multiple branch points in a block or the probability of the predication. In Chapter 6, we propose an extension to the priority function which considers all these effects and analyzes them.

1.2.4 Region Restructuring

Our experiments show considerable compilation time savings with comparable execution time performance using our region-based register allocation which synthesizes our techniques. Our region-based register allocation is not limited to any specific type of region and can be used for any regions passed to the register allocation phase. To avoid the extra compiler time of region formation, the region-based

register allocation approach may use regions formed in earlier phases of compilation like instruction scheduling. But our extensive experiments show that the performance of the register allocation varies according to the region size. The lack of register pressure in previous region formation causes register allocation to have excessive live range splitting or propagation according to the region size. If the size of each region is too large, each region may have register pressure higher than the available number of registers and many of the live ranges are forced to spilt. If the size of each region is too small, the compiler needs to spend more time for propagation than larger regions, and may add unnecessary shuffle code caused by color miss-match. In Chapter 8, we propose the concept of region restructuring based on estimated register pressure. Our approach simply re-groups the regions constructed in previous phases into new regions based on the register pressure. We demonstrate that this linear time region restructuring effectively re-groups the regions and provides more appropriate size of the region than original regions, therefore both the compilation and the execution time can be reduced in all of our benchmarks.

1.3 Outline

This thesis presents a series of techniques to the register allocation in the scope of region-based compilation and predicated execution based on hyper-blocks. Our

approach builds on the notion of a priority based global register allocator originally introduced by Chow and Hennessy [11]. Chapter 2 presents an introduction to the region formations explored for ILP compilers and Chow and Hennessey's model for register allocation. Chapter 3 presents an overview of the organization and operation of the Trimaran compiler which infrastructure is the platform for our implementations and experiments. The main contributions of the thesis are contained in Chapter 4 through Chapter 7. Chapter 8 describes the framework used to compare function-based approach and region-based approach, and evaluates the performance of the techniques with respect to region size. As a result of our analysis, we then propose the concept of restructuring the regions, previously constructed by the instruction scheduler, based on register pressure[3].

[3]The number of registers required in any compilation unit without spilling variables into memory

Chapter 2

Background

The compiler performs its analysis required for optimization and optimization itself in a certain scope of the program called *compilation unit*. The size of compilation units is closely related to both compilation time and the quality of the code that can be obtained. Traditionally the compiler process has been built using the function as a compilation unit, because the function provides a convenient partition of the program. But the size and the contents of a function may not provide a desirable scope of units to the compiler.

This chapter will explain the importance of the region-based compilation approach [23] and several historic region formations especially explored in ILP compilers. We also explain the background of graph coloring register allocation and Chow's priority based approach on which our work is based.

2.1 Region-Based Compilation

The traditional approach of function based compilation, where a compiler manager uses a function as a unit of compilation, has many limitations such as the scope of compiler optimization is limited to a function of critical paths spanning procedures cannot be considered. Modern compilers include aggressive transformations like function inlining and loop unrolling to exploit ILP well beyond a single basic block. However, aggressive transformation often results in excessively large function bodies that make aggressive analysis and transformation technique ineffective and intractable.

Region-based compilation has been proposed as a solution to this problem, because it provides many benefits to aggressive ILP compilers such as

1. The compiler has complete control over the size and contents of the compilation unit.

2. The size of the compilation unit is typically smaller than the function, thereby reducing the impact of the algorithmic complexity

3. Selecting regions allows the compiler to select compilation units that more accurately reflect the dynamic behavior of the program and allows the compiler to produce more compact and optimized code.

4. Each region may be compiled completely before compilation proceeds to

the next region, so all function-oriented compiler transformations may be applied.

In terms of register allocation process, region-based register allocation can have two specific advantages: (I) High levels of ILP and aggressive transformations for EPIC architectures increase register pressure dramatically. The size of the required analysis and data structures for register allocation become much smaller in region-based register allocation. (II) The register allocation phase can be sensitive to the other compiler optimization phases. For instance, if the scheduler optimizes a region aggressively, register allocation may use the information obtained by the scheduler.

2.2 Region formation for ILP compilers

A *basic block* is a sequence of consecutive instructions in which flow of control enters at the beginning and leaves at the end without the possibility of branching except at the end. The optimizations in basic blocks are well understood and can be easily implemented, however they have limited effectiveness due to limited parallelism available for extraction within a basic block. As increasing amounts of ILP become available in microprocessors, instruction scheduling inside basic blocks is not enough to find data-independent operations to utilize the CPU optimally.

To keep the CPU busy enough, operations must be moved from one basic block to another, and in some cases operations must be moved up ahead of conditional jumps. Many different types of regions have been devised. In this section, we will review several types of region formation methods which have been explored for ILP compilers based on execution frequencies.

2.2.1 Trace Scheduling

The trace scheduling algorithm by Fisher [18] was one of the first approaches to instruction scheduling that transcended the level of the basic block. A trace, in this context, is a sequence of basic-blocks without internal loops and trace scheduling has been extended by many researchers. We present an overview of the trace scheduling process as it applies to the thesis. Two key steps in trace scheduling are (1) trace selection and (2) shuffle code insertion.

2.2.1.1 Trace Selection

An ordered sequence of basic-blocks for a trace is constructed first. Using profile information, the scheduler selects a trace from the unscheduled code. Of the unscheduled blocks, the one with the highest execution frequency is chosen; this block is called the *seed* and denoted s. Starting from the seed s, they grow the trace forward and backward in the control flow graph, picking the unscheduled

successors or predecessors of the s with the highest branch frequency as long as the inclusion does not introduce an internal cycle. This inclusion is repeated as long as the following two conditions are satisfied. Here $W(x)$ is the frequency for region R and $W(x \rightarrow y)$ is the frequency for the edge from region x to y respectively and T_e and T_s are certain specified thresholds.

$$\frac{W(x,y)}{W(x)} \geq T_e \tag{2.1}$$

$$\frac{W(y)}{W(s)} \geq T_s \tag{2.2}$$

In figure 2.1 we show the result of a trace formation process. This example flow graph is annotated with branch probabilities. Here, block **F** is the basic block with the highest execution frequency, and is used as the seed. Using the trace picking algorithm, the blocks **B, C, G** and **H** are included in the trace.

2.2.1.2 Shuffle Code

If code is moved across a block boundary while scheduling the trace, the compiler sometimes has to add copies of operations it has scheduled on the trace to off trace. This occurs because some operations, which previously preceded a conditional jump, would have been scheduled after the conditional jump and thus, must be duplicated before the off-trace target of the jump.

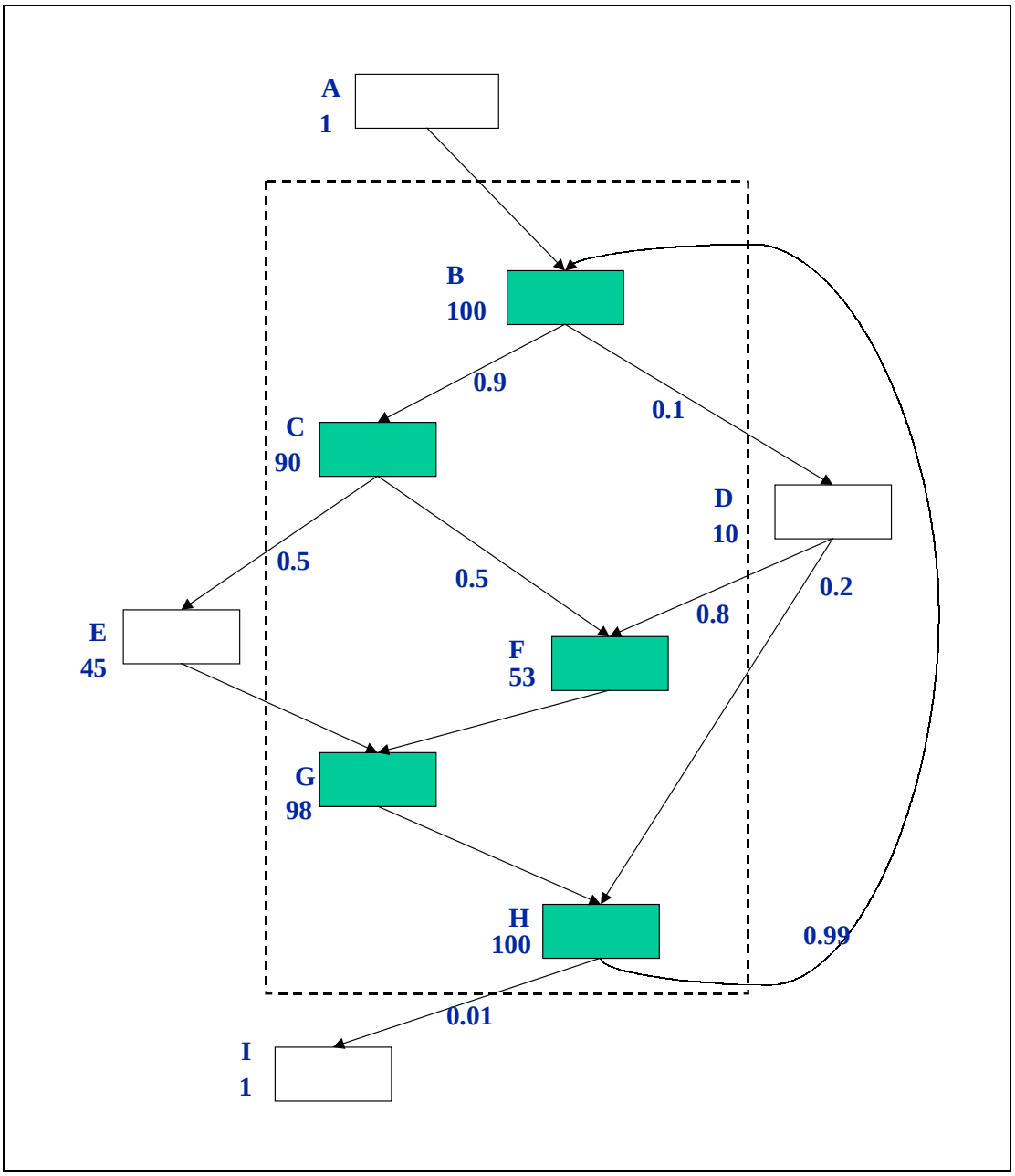


Figure 2.1: Trace Formation

For instance, if an operation $op1$ in $B1$ is scheduled below the conditional jump, say in $B2$, a new operation needs to be copied to $B3$, as in Figure 2.2 (a). Similarly, if an operation $op2$ in $B4$ is scheduled in $B2$, a new operation is copied to $B3$ as in Figure 2.2 (b). Code motion past side entry or side exit is more complicated. If an instruction i is moved from below to above a branch operation as in Figure 2.2 (c), i may kill the value live-in to $B1$ or defined in $B1$. To avoid this problem, a false dependence may be added between the branch operation in $B1$ and $op3$; alternatively, register renaming can be used. Likewise, code motion from $B3$ to $B4$ as in Figure 2.2 (d) may kill the value live-in from $B2$. Again, there are two possible approaches; creating a false data dependency, or copying $B4$ into a new block $B4'$ and redirecting $B2$ to $B4'$ instead of $B4$.

2.2.2 Super-block Scheduling

The complexity of trace scheduling by inserting shuffle code caused by the side entries is simplified in super-block scheduling [24]. A super-block is a trace with a single entry but potentially many side exits. Since side entries do not exist, upward movements past a side entry within a super-block are pure speculation, and no replications in the off-trace code are required. Similarly, any movement from above to below a side exit is pure replication. In the super-block algorithm, we allow an instruction i to move past a side exit E only if it is not live-out on

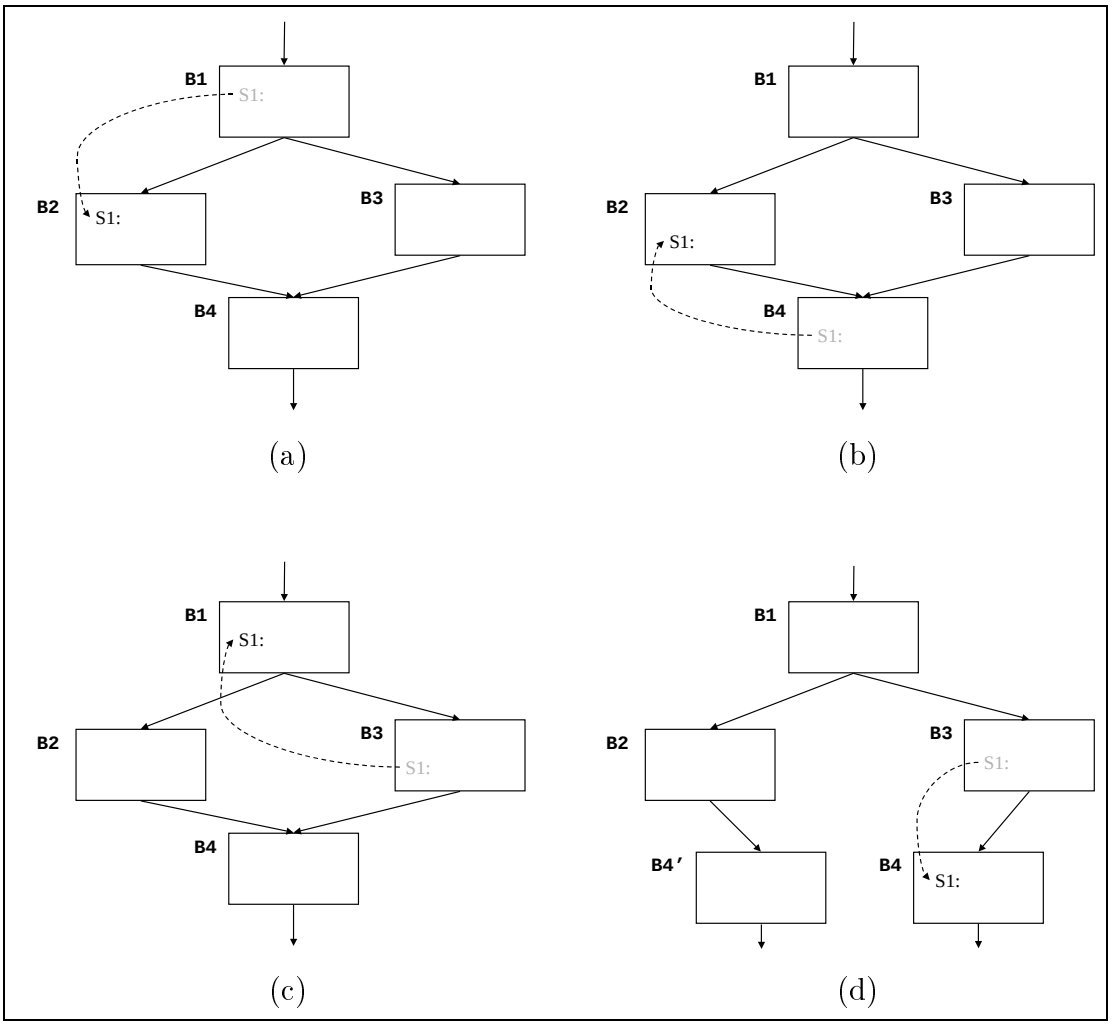


Figure 2.2: Code Shuffle for Trace Schedule

E. This way, downward movements do not require any sort of shuffle code to be inserted into the program after scheduling.

Super-blocks are formed in two steps. First, frequently executed traces are identified using techniques similar to trace scheduling. To restructure a trace into the form of a super-block, the *tail duplication* transformation is used. Tail duplication first duplicates each block A with a side entry, into the blocks A and A'. The side entry is then redirected to the off-trace block A'. Repeated applications of this transformation can be used to turn a trace into a super-block.

In Figure 2.3 we show the use of tail duplication on the trace. The side entry into block D is first tail duplicated; this results in the duplicate block D'. Then the side entry into block G is removed by tail duplicating G, resulting in the duplicate block G'. Finally, the blocks A, B, D, E and G can be merged. As before, block C, D', F and G' are selected for a another super block and block F is duplicated to F'.

Tail duplication can be seen as a program *specialization* transformation. The result of this specialization can in turn enable more traditional compiler optimizations, such as constant propagation, value congruence, partial redundancy elimination, dead-code removal, branch removal etc. A compiler using the super-block compilation model will need to iterate on these optimizations in order to fully take advantage of the super-block transformation.

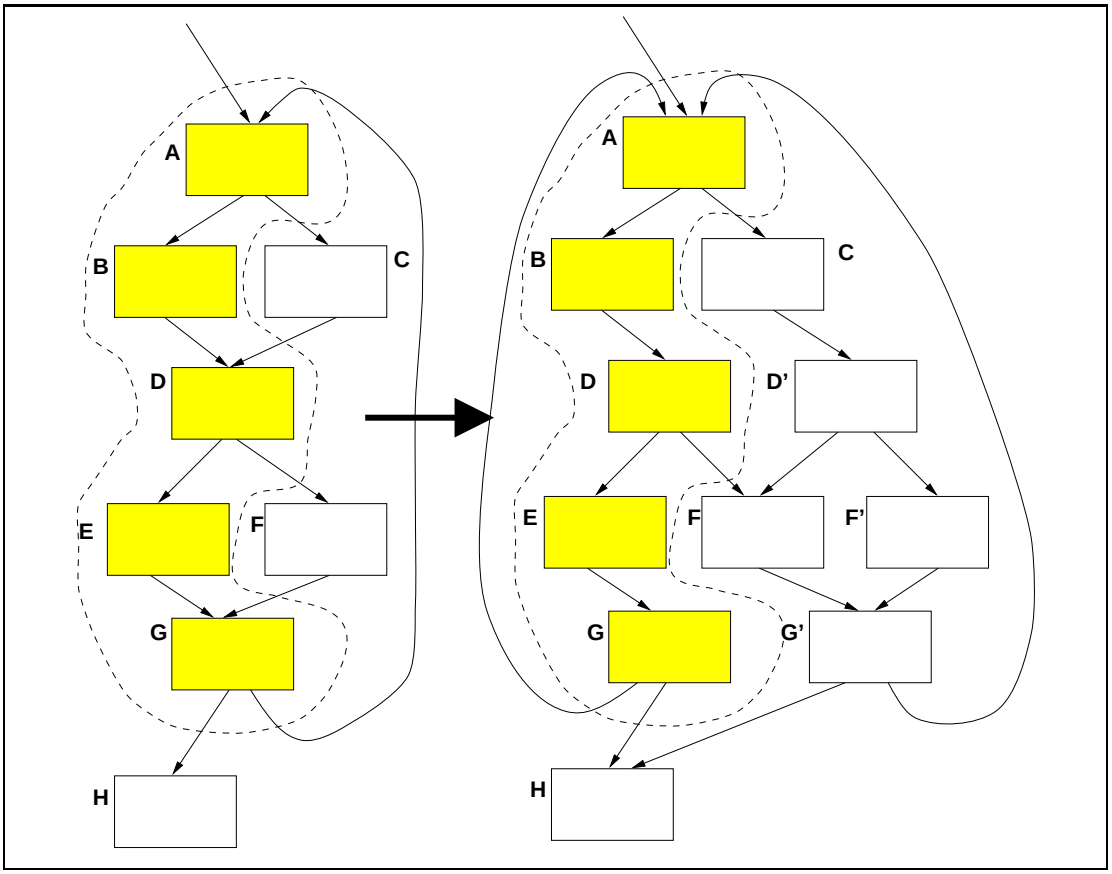


Figure 2.3: Tail duplication.

2.2.2.1 Super-block Formation

Hwu *et al.* [24] describe various techniques of enhancing transformations to increase the available parallelism in super-blocks as follows:

branch target expansion Branch target expansion replicates the target super-block of a forward branch at the end of a super-block if the branch is likely to be taken.

loop peeling A super-block loop is a super-block which ends with a likely control transfer back to its entry. Loop peeling replicates the body of a super-block loop N number of times, where N is its expected number of iterations. The remaining iterations of a loop are then moved off-trace.

loop unrolling A super-block loop that is to be executed N times is unrolled to increase the size of the basic-block. The control flow instructions between the unrolled loop bodies are eliminated.

register renaming Register renaming assigns each definition a new virtual register. This removes false dependences from the flow graph, which can improve the ILP.

operation migration Operation migration moves instructions off trace if their values are unlikely to be needed. This can be seen as a form of re-scheduling replication in which an instruction i is moved to all its use targets.

induction and accumulator variable expansion Induction and accumulator variable expansion can be used in conjunction with loop peeling and loop unrolling. These transformations assign a different virtual register to each induction (or accumulator) variable in the different copies of the replicated loops. Using this transformation, loop-carried dependencies between different iterations of the loops on these variables can be reduced.

2.2.3 Hyper-block Scheduling

A hyper block is a set of predicated basic-blocks in which control may only enter from the top, but may exit from one or more locations. The motivation behind *hyper-blocks* is to group many basic-blocks from different control flow paths into a single manageable block for compiler optimization [35]. To exploit the high level of ILP in the architectures with predicated codes, hyper-blocks are often used as the unit of program presentation. Hyper-blocks provide a very natural unit for instruction scheduling, as well as register allocation as we will see soon.

A common problem of all optimization and instruction scheduling is conditional branches in the target application. The process of eliminating conditional branches from a program in order to utilize predicated execution support is referred to as *if-conversion*[48] [27] [38]. If-conversion replaces conditional branches in the code with comparison instructions which set a predicate. Instructions which

are control-dependent on the branch are then converted to predicated instructions dependent on the value of the corresponding predicate. If-conversions can eliminate all loop backward branches from a program.

2.2.3.1 Region Selection

The first step of hyper-block formation is to decide which basic-blocks in a region to include in the hyper-block. Unlike the super-block, which contains only basic blocks from a single flow of control, a hyper-block may include basic blocks from mutually exclusive branches of a region. Typically, hyper-blocks are formed from blocks that fall within an innermost loop and conventional techniques for if-conversion predicate all blocks with a single-loop nesting region together.

2.2.3.2 Hyper-block Formation

We need two conditions in order to select basic-blocks and convert them into a hyper block.

1. There exist no incoming control flow graph arcs from outside basic-blocks to the selected blocks except the entry block.

2. There is no nested block in a selected region.

When these conditions are met, we do the following for formation:

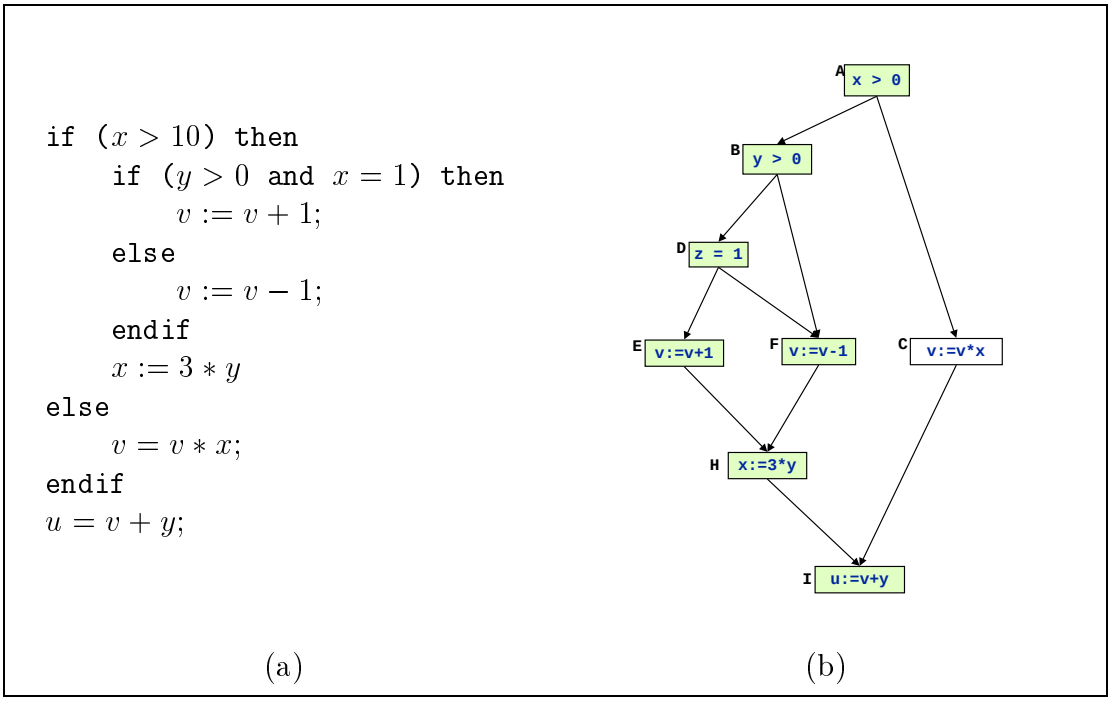


Figure 2.4: Hyper-block formation: Control Flow Graph

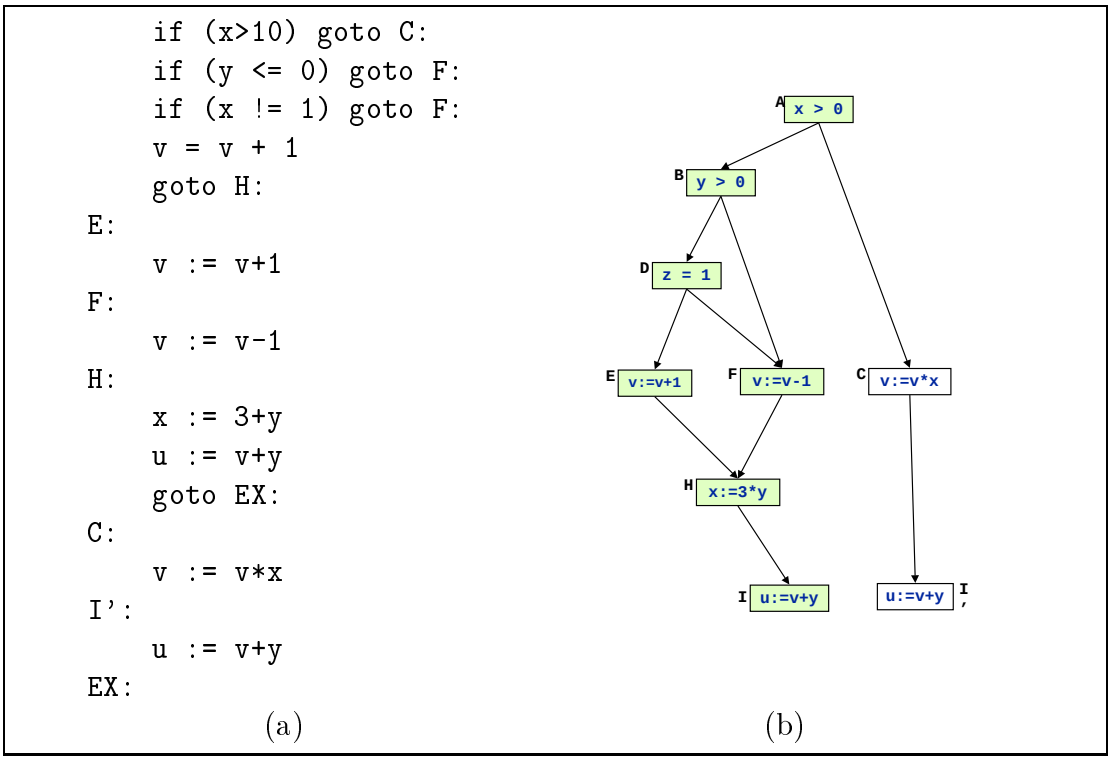


Figure 2.5: Hyper-block formation: Tail duplication

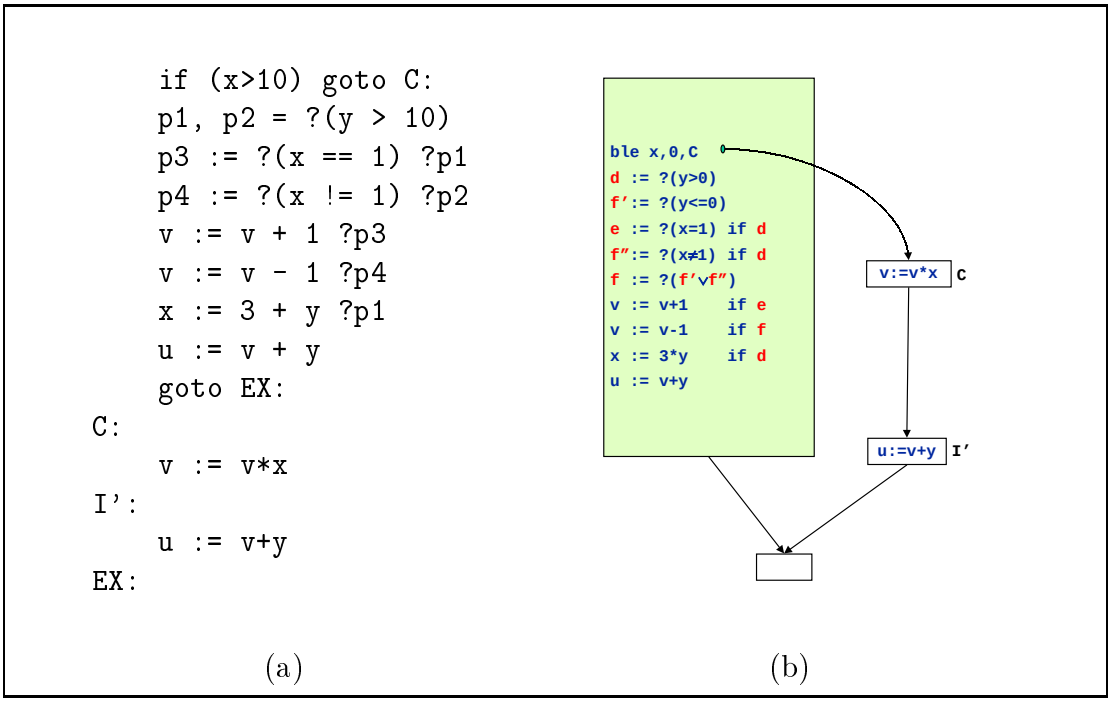


Figure 2.6: Hyper-block formation: If-Conversion

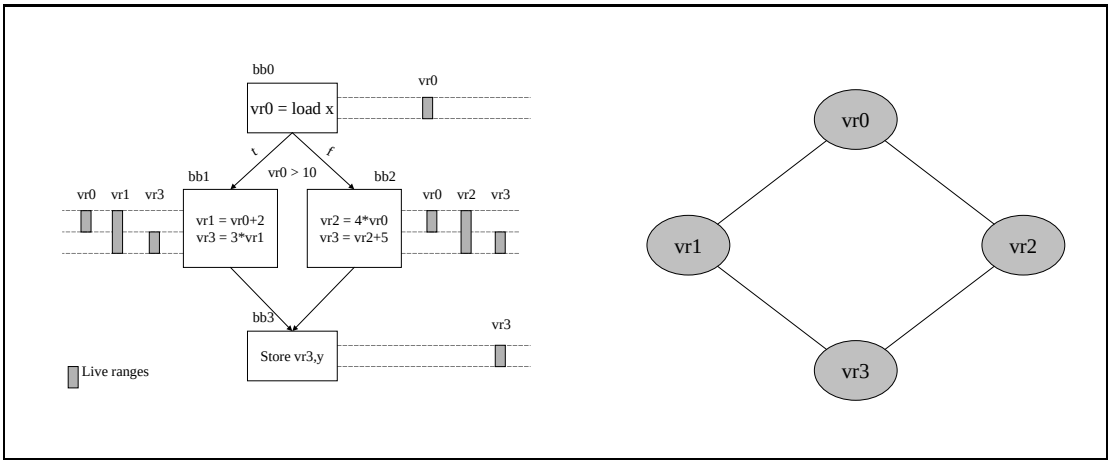


Figure 2.7: Register Requirements before If-conversion

- *Tail duplication* Tail duplication is used to remove control flow entry points into the selected blocks from outside the region. This is similar to super-block formation.

- *Loop peeling* For loop nests with inner loops that iterate only a small number of times, efficient hyper-blocks can be formed by including both outer and inner loops within a hyper block.

- *Node splitting* Node splitting may be used on the selected blocks to eliminate dependencies created by control path merges.

- *If-conversion* If-conversion substitutes a single block of predicated instructions for a set of basic-blocks containing a conditional control-flow between the blocks.

Figure 2.4 shows a sample program we use for the example of the hyper-block construction and its control flow graph. The shaded blocks are the regions we choose for hyper-block and please note that block C is intentionally excluded for demonstration purpose. This exclusion will be happened if the frequency of the block C or the edge frequency between A and C is smaller than the threshold.

Figure 2.5 shows the pseudo assembly codes and control flow graph after tail duplication is performed from the program in 2.6 (a). To avoid a side entry from region C to I, a duplicated region of I is created as a block I'. Finally, if-conversion

creates a hyper-block with region A, B, D, E, H and I as in Figure 2.6 (b).

2.3 Overview of Trimaran Compiler

The Trimaran compiler infrastructure has been developed for supporting state of the art research in compiling Instruction Level Parallel architectures, especially those based on Explicitly Parallel Instruction Computing (EPIC) and for research in instruction scheduling, register allocation and machine dependent optimizations.

Trimaran provides advanced capabilities and support for experimenting with innovative, forward-looking ILP architectures and the compiler modules needed to generate high-performance code for these architectures.

2.3.1 HPL-PD: A parameterized ILP architecture

HPL-PD [28] is designed to support the advanced features of EPIC style architectures and it was conceived for research in instruction level parallelism. The advanced features of the architecture include:

1. Control speculative execution

2. Predicated Execution

3. Compiler control of the memory hierarchy

4. Data speculation

2.3.2 MDES: High level machine description language

Trimaran uses a machine description database to represent the properties of the target HPL-PD processor. A high-level textual language, called HMDES, is used to describe the machine database so that a new HPL-PD processor can be created by changing the machine description database.

The Trimaran compiler is fully parameterized with respect to the target machine information. The compiler modules are allowed to make a fixed set of queries to the database through an *mdes query system (mQS) interface*. The mQS provides basic information about the machine, including opcodes, register files, operation latencies and resources and resource tables. For example, the scheduling interface provides latencies for dependence edges in a data dependence graph. There is support for a broad space of realistic machines including machines with non-unit latencies and complex reservation tables. Both equals(EQ) and less-than-equals(LEQ) latency models are supported.

2.3.3 Trimaran Modules

2.3.3.1 IMPACT: A Compiler Front End

Trimaran includes the IMPACT front-end for C with the following modules

- K&R/ANSI-C parser built upon EDG
- Variable and Function name renaming and complex expression flattening
- Control flow profiling
- C source file splitting and function inlining
- Classical basic block and function level optimization
- Super-block and hyper-block formation
- ILP transformations including loop unrolling, register renaming with copy, induction variable expansion and predicate promotion.

2.3.3.2 ELCOR: An optimizing compiler back-end optimizer

ELCOR is the compiler back-end for Trimaran and performs instruction scheduling, register allocation, and other machine-dependent optimizations.

2.3.3.2.1 Data-flow Analysis The Trimaran back-end provides many types of data flow analysis including variable liveness analysis, reaching definitions analysis, available expression analysis and predicate analysis.

Live variable analysis operates on a region of code and annotates information on the control flow edges between basic-blocks and hyper-blocks. This module can also compute the upward-exposed define variables, downward exposed used

variables, and downward exposed defined variables. Reaching definitions analysis computes def-use chains for a region of code and annotates the information on the region. Available expression analysis results are also annotated on the region being analyzed, and expression availability can be queried at any point in the control flow graph of this region. The Trimaran also provides a uniform way for modules to ascertain the relationship between predicate expressions through the *Predicate Query System (PQS).*

2.3.3.2.2 Control Flow Analysis Control flow analysis modules operate on a control flow graph where all nodes of the graph are basic blocks. The Elcor provides three control flow analysis: dominator and post-dominator analysis, control dependence analysis, and loop detection. Each analysis module has a data structure that encodes the analysis results. The loop detection module also identifies basic induction variables in a loop.

2.3.3.2.3 Dependence Graph Construction Dependence graph construction is a scheduling-specific phase of Elcor that introduces dependence edges in an operation graph. The edges represent flow dependences, anti-dependences and output dependences for registers as well as memory dependences and control dependences that restrict code motion across branch operations. In addition to its use for scheduling, the data dependency graph is useful for guiding optimizations

that consider critical path lengths in a program.

2.3.3.2.4 Modulo scheduling and Rotating register allocations The loop scheduling consists of two modules: the modulo scheduler and stage scheduler. The modulo scheduler [41] [50] allocates resources for the loop kernel subject to an initiation interval. The stage scheduler moves operations across stages in order to reduce register usage in the loop. When a loop is modulo scheduled, some of the virtual registers in the loop are designated as rotating registers. Rotating register allocation maps such registers to the rotating register files immediately after modulo scheduling. Stage scheduling can be used to reduce the rotating register requirements in a loop after modulo scheduling. The remaining registers are allocated to the static register file after scalar scheduling of the rest of the program.

2.3.3.2.5 Acyclic scheduling Trimaran employs three variations of acyclic scheduling: cycle-scheduler, backtracking scheduler and meld-scheduler. The cycle scheduler generates a schedule by constructing instructions from operations for each issue cycle in order. The backtracking scheduler is a modified versions of the cycle scheduler which can either do limited backtracking to support scheduling of branch operations with branch delay slots, or do unlimited backtracking. The meld scheduler is a modified version of the cycle scheduler that can propagate

operation latency constraints across scheduling region boundaries.

2.3.3.3 Simulator

Trimaran has a cycle level simulator for the HPL-PD architecture which is configurable by a machine description language called HMDES. The simulator provides run-time information on execution time, branch frequencies, and resource utilization. This information can be used for profile-driven optimizations as well as to provide validation of new optimizations.

2.3.3.4 Register Allocation

A diagram of the Trimaran compiler is presented in Figure 2.8. The compiler consists of three major components: the IMPACT front-end, the Elcor back-end and a cycle level simulator.

Thanks to the modular design of Trimaran, register allocation can be implemented as a separate module and can be plugged in between any modules. Our work is implemented after the first acyclic scheduler (pre-pass scheduler) and is followed by another pass of scheduling (post-pass scheduler). Spill codes inserted by register allocation may lead to inadequate scheduling for EPIC processors, since STORE or LOAD operations inserted by register allocation generate many idle cycles. So it is necessary to schedule the spill code which were inserted during

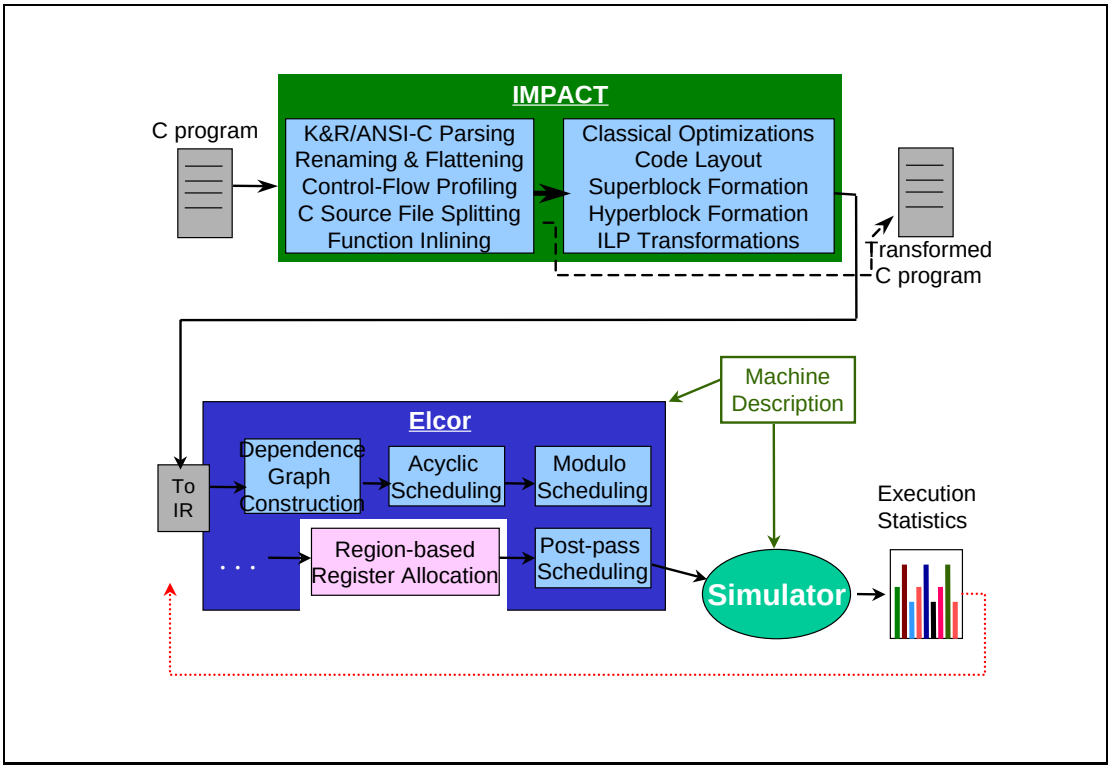


Figure 2.8: Trimaran Compiler Infrastructure

register allocation. For certain types of scheduling problems, when using certain scheduling algorithms, it is possible to allocate the registers and introduce the spill code during scheduling [12]. This is known as combined instruction scheduling and register allocation. In such cases, one can schedule the spill code as it is generated.

2.3.4 Experiment Framework

To validate the effectiveness of our algorithms, we also have conducted experiments on the Trimaran compiler [49] for selected Unix programs and Spec 92 and Spec 95 benchmarks. We simulate the execution of these programs on a 4-way ALU EPIC machine with 2 floating-point units, 2 memory units and 1 branch unit. In our experiments we varied the register file size. The following configurations were tested:

- 32 general purpose registers (GPR) and 32 floating point registers (FPR)
- 64 general purpose registers (GPR) and 64 floating point registers (FPR)
- 96 general purpose registers (GPR) and 96 floating point registers (FPR)

In all experiments we used 32 branch target registers (BTR) and 128 predicate registers so that these registers would not be a bottleneck.

Chapter 3

Overview of Region-based Register Allocation

3.1 Coloring Approach of Register Allocation

Graph coloring is a well established approach to register allocation and has been used as the central paradigm for register allocation in modern compilers.

A *coloring* of a graph $G = (v, e)$ is an assignment of a color to each node of the graph such that any two adjacent nodes do not have the same color. Informally, a live range is a collection of operations (or basic blocks in some approaches) where a particular definition of a variable is live. For register allocation, a graph called the *interference graph* is constructed from the program in the following way.

Each node v_i in the interference graph G represents the *live range* of a program data value which is a candidate to reside in a register. Two nodes in the graph are connected by an edge e if the two data values corresponding to those nodes interfere with each other in such a way that they can not reside in the same register. The coloring approach to register allocation seeks to map a fixed set of colors to all the nodes of the interference graph. In coloring the interference graph, the number of colors used corresponds to the number of registers available for use in register allocation.

Chaitin's approach forms the basis of many register allocation schemes [8]. His approach consists of two major phases, *simplification* and *register assignment*. Simplification is based on the observation that if node V has degree less than the number of available registers N, then the register assignment phase is guaranteed to find a color for V regardless of the colors assigned to V's neighbors. Even if each of V's neighbor node is assigned a different color, at most $N-1$ colors are used, and there is an available color for V. The main point is that the graph is still *N-colorable* if this V is removed from the graph G when G is *N-colorable*. This process of removing nodes that have degrees less than N is called simplification. Simplification is blocked when all live ranges of the graph have a degree of at least N. Then, Chaitin's approach chooses a live range from the graph for spilling. Spilling a variable means keeping it in memory, rather than in a register. In this

case, the live range corresponding to the spilled variable can be deleted from the interference graph. This deletion will reduce the degree of G and this process can be repeated until the graph is colorable. The selection of live ranges for spilling is based on certain heuristics related to the cost of spilling. Register assignment follows the simplification phase and the physical registers are assigned to the live range in the reverse order in which they were removed during simplification.

3.2 Register Allocation Steps in the Chow and Hennessy model

An alternative to Chaitin's approach is the register allocation via priority based graph coloring introduced by Chow and Hennessy [11]. They also introduce the concept of live range splitting as an alternative to the spilling techniques used by Chaitin et al.

Chow and Hennessy noticed that register allocation should use cost and benefit analysis for better performance. The value of assigning a given variable to a register depends on the cost of the allocation and the resultant savings. The cost is a function of the need to introduce register–memory transfer operations, which put the variable in a register, and later update its home location. Greedy coloring assigns colors to live ranges in a heuristic order determined by priority function.

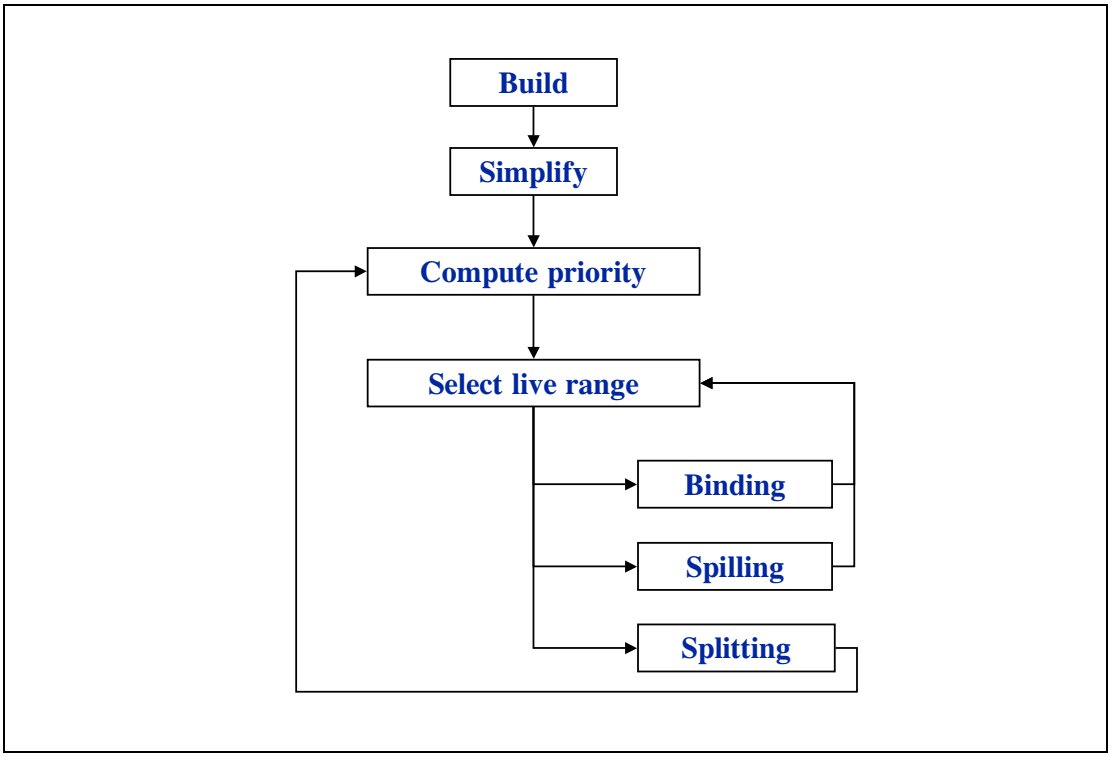


Figure 3.1: The framework of Chow and Hennessy register allocation

The priority function captures the savings in memory accesses which results from assigning a register to a live range as opposed to keeping the live range in memory.

This thesis builds on the work of Chow and Hennessy in so far as region itself provides natural live range splitting in our region-based approach, and frequency information used in region formation can also be used in priority function. Therefore, it is important to understand their work.

Their framework, illustrated in Figure 3.1, has the following steps.

1. Build the interference graph.

2. Prioritize each live range according to some heuristic function.

3. Color the live-range with the highest priority. Repeat from (3) whenever there remains an uncolored live range.

4. If there is a live range that cannot be colored, reduce the interference of the live range by splitting it into two or more smaller live ranges and repeat from (3). To connect split live ranges, shuffle code is inserted at splitting points.

3.2.1 Live Range Construction

A live range is an isolated and contiguous group of nodes in the control flow graph. Live ranges are discovered by finding connected groups of *def-use chains*. A single def-use chain connects the definition of a virtual register to all of its uses. In the Chow and Hennessy approach, live ranges are determined by data-flow analysis – live-variables analysis and reaching definition analysis. A variable is live at the entry of block i if there is a direct reference of the variable in block i or at some point reachable from block i not preceded by a definition. The reaching attribute is solved by forward iteration through the control flow graph. A variable is *reaching* in block i if a definition or use of the variable reaches block i. The live range $lr(v)$ of a variable v is constructed as $live(v) \cap reach(v)$.

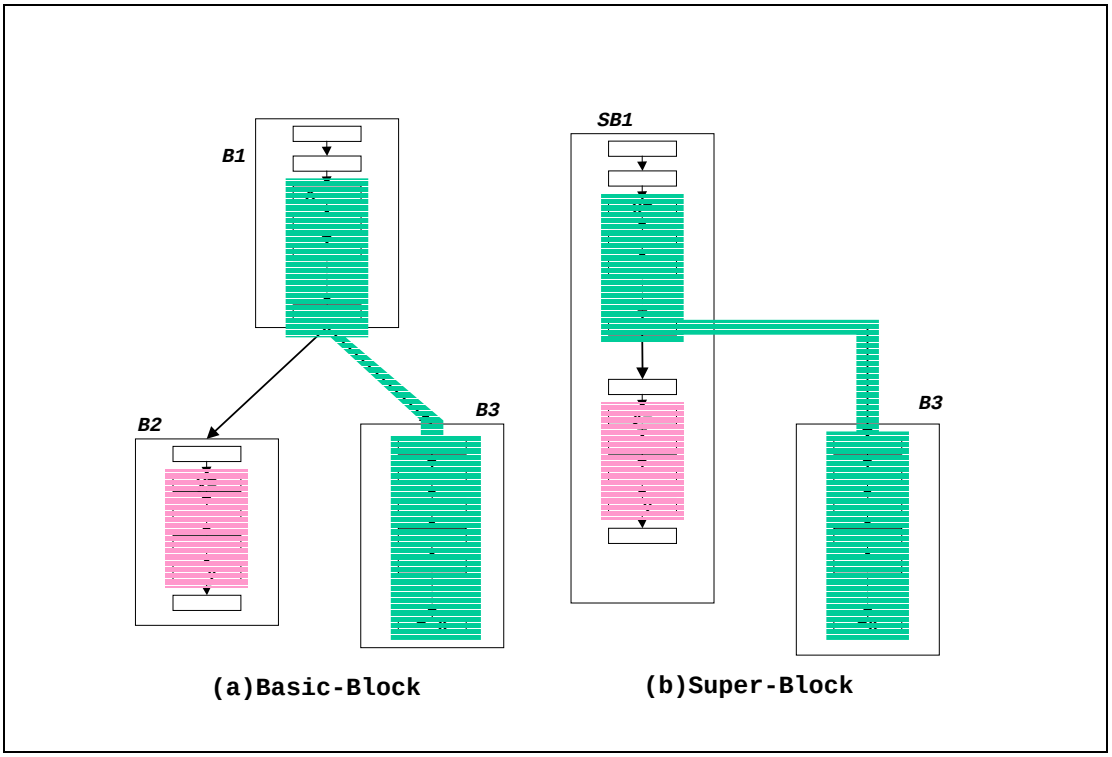


Figure 3.2: Interference of live ranges with BB and SB

Chow and Hennessy uses the basic-block as the unit of coloring. The granularity of the basic block representation is used in their approach for faster compilation. They noticed that the coarser interference graph may result in less efficient allocation but they claimed that the penalty of coarse-grained live ranges is minimal since the size of basic blocks is typically small.

For region-based compilation using hyper-blocks or super-blocks, this penalty can be greater for two reasons. First, the register pressure in a basic block with coarse grained live ranges is not as large as in HB since the size of a basic block is

Benchmark	Coarse	Fine	Reduction
a5	3988	2354	40.97%
fir	716	684	4.47%
idct	770	730	5.19%
idea	5114	3682	28.00%
nbradar	4946	4240	14.27%
paraffins	5472	5278	3.55%
polyphase	6390	5250	17.84%
qsort	1056	986	6.63%
strcpy	104	84	19.23%
wave	238	228	4.20%
wc	948	894	5.70%

Table 3.1: The number of edges in interference graph for coarse grained (basic block) live ranges and fine grained (operation) live ranges

usually smaller. Second, a coarse grained live range may have extra interferences in super-blocks or hyper-blocks not present in basic blocks. In Figure 3.2 (a), variable **x** is live in blocks **B1** and **B3** only. Hence, it does not interfere with variable **y** which is live in **B2**, even if we use a basic block as the unit of a live range. But if we create a super-block **SB1** from blocks **B1** and **B2**, a coarse grained live range construction will report interferences between **x** and **y**.

By constructing live ranges recursively from sub-live ranges, which can go all the way down to operation level, we have been able to compare the effect of granularity of live ranges for many different types of regions including basic blocks and hyper-blocks. Table 3.1 and Table 3.2 show the difference between basic block based live ranges and operation based live ranges by using the total number of

Benchmark	Coarse	Fine	Reduction
a5	8190	3362	58.95%
fir	10696	7224	32.46%
idct	7876	6672	15.29%
idea	8642	4614	46.61%
nbradar	130894	85154	34.94%
paraffins	21332	15406	27.78%
polyphase	74676	48636	34.87%
qsort	20282	12002	40.82%
strcpy	4610	3342	27.51%
wave	6444	4740	26.44%
wc	4600	2320	49.57%

Table 3.2: The number of edges in interference graph for coarse grained (hyper-block) live ranges and fine grained (operation) live ranges

interference edges from several benchmarks chosen from digital signal processing programs and common Unix utilities. For the hyper-blocks, the weighted average of coarse-grained live ranges has about 54% more interference edges than fine-grained live ranges where as the average for basic-blocks is is about 22%. This reduction makes for faster register allocation and hence a reduction in compilation time.

3.2.2 Interference Graph Construction

Intuitively, two live ranges interfere with each other if the allocation to the same register changes the meaning of the program. In Chow's approach, two live ranges are said to interfere if the intersection of their live ranges is not empty.

We however, use Chaitin's definition of liveness. Two live ranges interfere if one of them is live at the definition point of the other. This is necessary as our approach is based on live units that can be as small as an operation. Hyper-block formation substitutes a single block of predicated instructions for a set of basic blocks containing conditional control-flow between those blocks [35]. Central to the construction of predicated code is the computation of accurate liveness information in order to build the sparsest possible interference graph. This is essential for reducing compilation time, but even more crucial for *execution performance.*

Consider the control-flow graph and live ranges in Figure 3.3 and the corresponding interference graph. In **B3** for example, **x** is live when **y** is defined. Therefore, there is an edge between vertices **x** and **y** in the interference graph. Notice that there is no edge between vertices **y** and **z** since the thread of execution moves exclusively to one of the two basic-blocks **B2** or **B3**. The accurate interference graph can be colored by no more than 2 colors. Using if-conversion, the four basic-blocks of Figure 3.3 are merged into the single predicated block of Figure 3.4. The operations previously executed in **B2** and **B3** are guarded by the predicates **p** and **q** respectively.

An interference graph using traditional live ranges is also shown in Figure 3.3. This interference graph contains an edge between **y** and **z** because **y** and **z** are

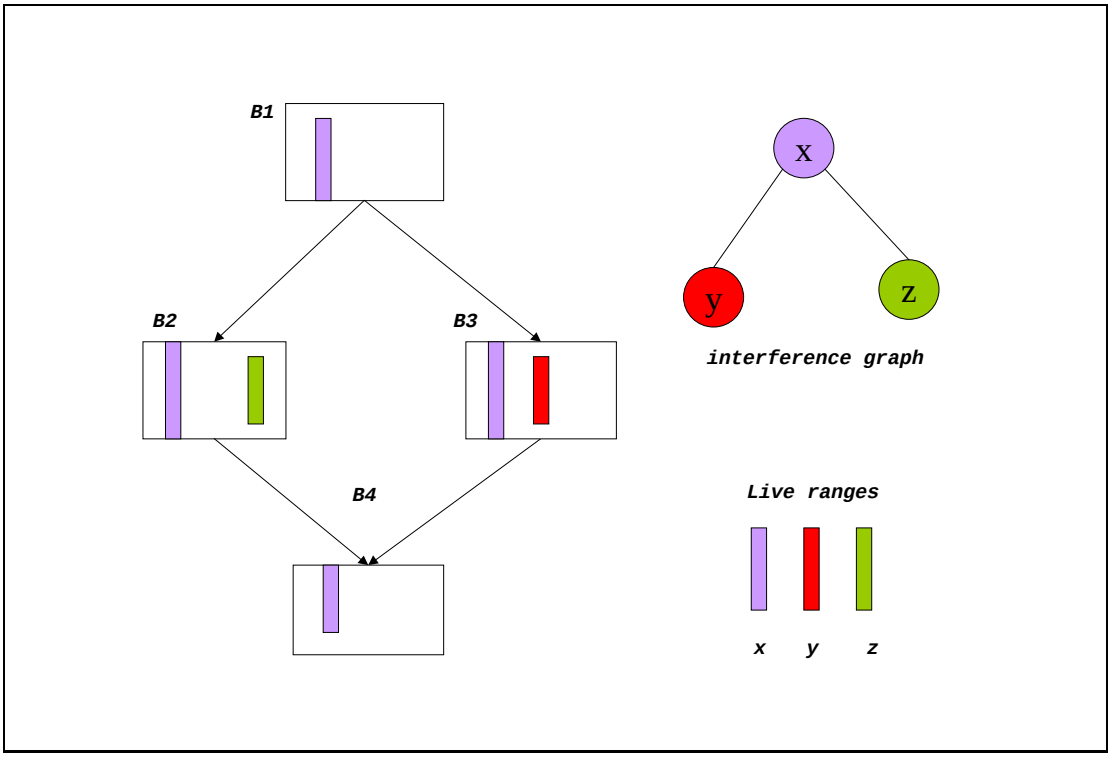


Figure 3.3: Interference Graph with Basic blocks

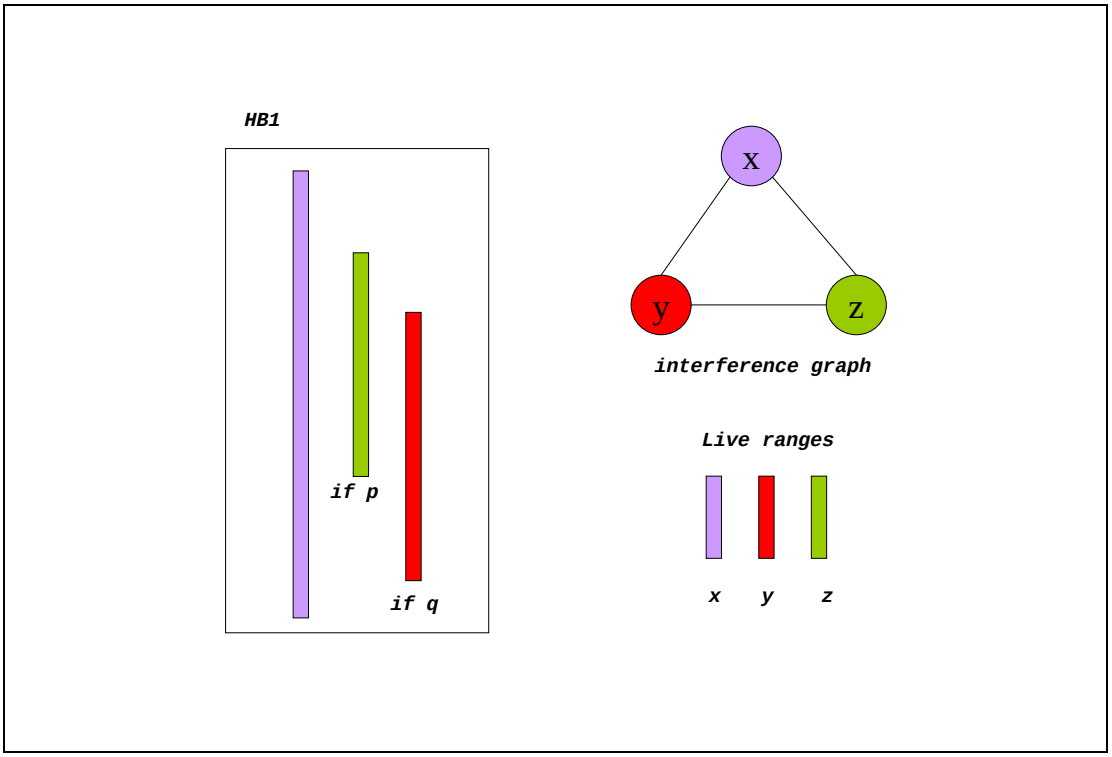


Figure 3.4: Interference Graph with Hyper-blocks

considered to be live simultaneously, as traditional analysis does not examine the predicate expressions. As a result, the number of registers required by this predicated block increase from 2 to 3. This increase in the register requirements is mainly due to two reasons. First, without knowledge about predicates, the data flow analysis must make conservative assumptions about the side effects of the predicated operations. Second, the solutions of the data flow analysis rely heavily on the connection topology among basic-blocks in the flow graph, which is altered by the if-conversion process used to construct hyper-blocks.

The problem of predicate-aware liveness computation has been studied extensively in the past [20] [16]. Eichenberger and Davidson[16] consider register allocation for predicated code. "P-facts" are defined to capture logically invariant expressions in straight line code (using both the predicated expressions in instructions and branch conditions) and analysis of these are used to define liveness. A more complex region like a loop is considered through "bundling" of virtual registers. A predicate insensitive register allocation is then used. Their framework does not consider spilling and emphasis is therefore placed on minimizing register requirements in the context of modulo-scheduled loops [50], rather than on good register allocation with a fixed number of registers. Their data reports show the reduction of registers needed for the Livermore kernel loops.

Gillies, Ju, Johnson and Schlansker[20] also consider predicated register allocation but hyper-blocks are not used in their framework for region construction. Global analysis (on whole procedures) is performed on the standard control-flow graph (CFG) with BB's having predicated code, and control predicates representing the branch component from the CFG. The basis of a BB is the set of predicates used in the BB code stream. The basis of BB within a global scheduling region (GSR) are recursively merged to form a single common basis for data flow analysis (DFA). Interval analysis is used to guide this process. To avoid expensive global analysis, a limit of 32 is imposed on the size of the basis during the recursive

merging. Our approach uses a similar predicate analysis but HBs are constructed taking into account the frequency of execution.

Our analysis differs in that we use a *predicate query system(PQS)* [26] rather than eagerly computing all the implications (the set of P-facts), which can be very expensive in larger regions. We address live range interferences in the context of region-based compilation by performing hyper-block formation. We perform local predicate analysis using a PQS in our region-based register allocation. Due to the the artificial limit of the size of the basis for each interval or the simplification used in PQS, the rest of the predicate expressions have to be approximated to what is in the basis or to TRUE.

Our study (see Table 3.3) shows that region-based register allocation with local predicate-aware liveness analysis gives limited performance improvement. While some procedures improved with predicated analysis, the performance improvement is either small or none in many of our test cases.

This can be explained in several ways. First, hyper-block formation is not undertaken if there exists nested inner loops inside the selected region. So only programs without inner loops but with many conditional branches and many local variables to be colored will benefit from predicate-aware liveness analysis. Second, the hyper-block construction maximizes the potential of instruction scheduling and this disables many of the careful interference calculations of the predicate-

Benchmark	Predicate -Unaware(U)	Predicate Aware(A)	Number of Functions(N)	Saved (U-A)/N
008.espresso	1729866	1729014	361	2.36%
023.eqntott	276686	276456	62	3.70%
072.sc	484094	483380	179	3.98%
085.gcc	5682058	5678266	1451	2.61%
124.m88ksim	465746	465518	252	0.90%
129.compress	155766	155744	24	0.91%
130.li	82648	82524	357	0.34%
132.ijpeg	1641726	1640956	473	1.62%
cccp	373380	373200	63	2.85%
cmp	4376	4376	5	0.00%
eqn	128276	128148	59	2.16%
lex	726564	726390	60	2.90%
tbl	421176	420708	89	5.25%
yacc	841108	840938	49	3.46%
AVERAGE				2.36%

Table 3.3: The number of edges in the interference graph for predicate-aware and predicate-unaware liveness analysis

aware register allocation. For example, if a branch is heavily weighted in one direction, the block on the opposite side may not be included in the hyper-block, so as to maximize speculation. Third, if a block contains a function call with side effects, the block will not be included in a hyper-block, as this prevents code motion and instruction scheduling suffers. Last, many predicate conditions are promoted to TRUE in our hyper-block construction in order to get a good instruction schedule, and this gives register allocation little chance for accurate interference graph construction.

3.2.3 The Coloring Algorithm

The first step of the coloring process is to remove *unconstrained* live ranges from the interference graph. Chow and Hennessy distinguished between *constrained* live ranges and *unconstrained* live ranges in the coloring process. Unconstrained live ranges have a number of neighbors in the interference graph which are less than the original number of registers available. These live ranges are not colored until the very end, since it is certain that some unused color can be found for them.

Then each live range selected according to the order of priority given by the *priority function* is assigned a color. If there is no available color, an attempt is made to split the live range. The process of selecting the highest priority live range with a possibility of live range splitting is repeated until the interference graph is empty.

3.2.4 Priority Function

As stated earlier, Chow and Hennessy noticed that register allocation should use a cost and benefit analysis. The value of assigning a given variable to a register depends on the cost of the allocation and the resultant savings.

The priority function captures the savings in memory accesses from assigning a register to a live range rather than keeping the live range in memory. It is

proportional to the total amount of execution time savings, $S(l_i)$, gained due to the live ranges residing in registers. First, the register binding benefit s_i is computed for every live unit of the variable i in each basic block as follows:

$$s_i = \text{LOAD\_SAVE} \times u + \text{STORE\_SAVE} \times d - \text{MOVE\_COST} \times n \quad (3.1)$$

where u denotes the number of uses and d denotes the number of definitions and n denote the number of register moves needed in that live unit. Then, $S(l_i)$ is computed by summing s_i over the every live unit i in the live range lr weighted by the execution frequency w_i in the individual basic-blocks:

$$S(l_i) = \sum_{i \in lr} s_i \times w_i \quad (3.2)$$

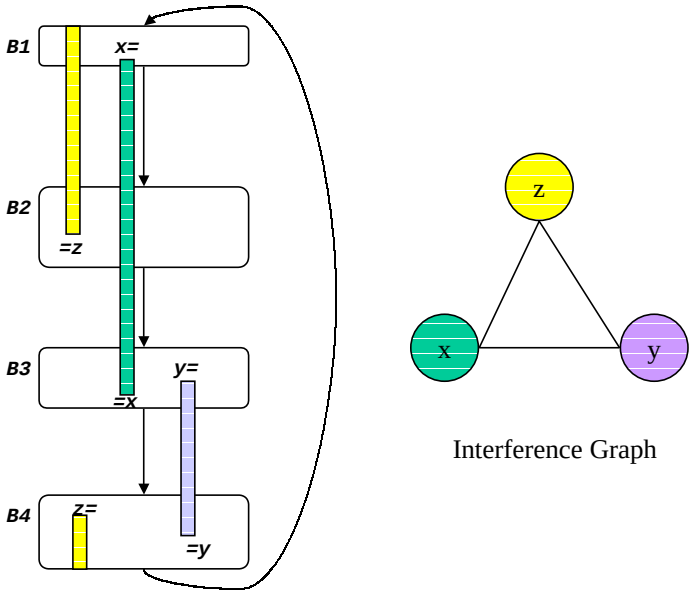
The last factor to consider is $P(l_i)$, the size of the live range, which is approximated as the number of live units, N in the range. A live range occupying a larger region of code takes up more register resources if allocated to a physical register. The total savings are therefore normalized by N so that smaller live ranges with the same amount of total savings will have higher priority. Thus, the priority function is computed as

$$P(lr) = \frac{S(lr)}{N} \quad (3.3)$$

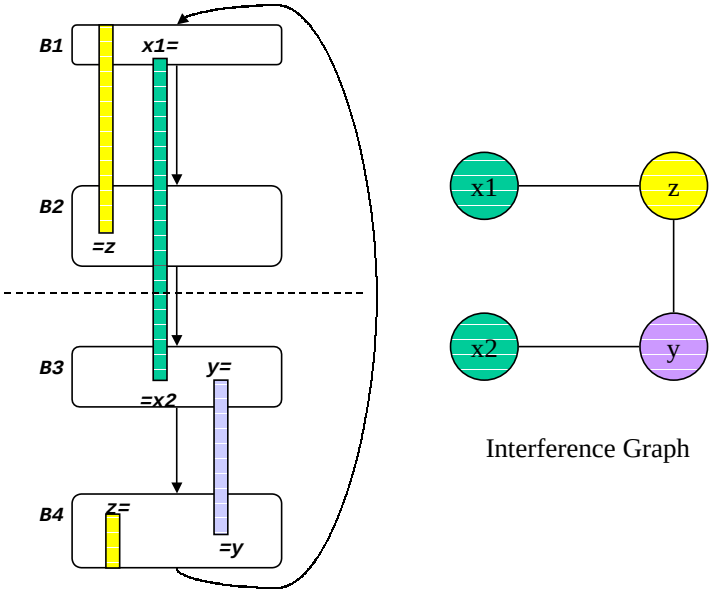
3.2.5 Live Range Splitting

Color assignment is blocked when no legal color exists for the next live range lr_i to be colored. *Live range spilling* is one solution where a live range lr_i is assigned a location in memory, and all references to lr_i are done by memory access operations like STORE/LOAD codes. These are referred to as *spill code*. *Live range splitting* is an alternative to the spilling when no registers are available. The basic purpose of splitting is to reduce the degree of interference by segmenting a long live range into smaller live ranges. Although the number of live ranges is increased, each smaller live range usually interferes with fewer other candidates because the live ranges occupy smaller regions and the probability of overlaps with other live ranges is thereby reduced. This is illustrated in Figure 3.5. In Figure 3.5 (a), all three variables x, y and z are interfered each other and we have 3-colorable graph. If we split the live-range of x and create two smaller live-ranges $x1$ and $x2$, the required number of registers is reduced to two. In this case, we may need an extra copy code at the boundary of $x1$ and $x2$ as we will explain in following section. But this can be reduce many memory access code by spilling the show live range of x.

In splitting a live range, Chow and Hennessy separate out a component from the original live range, starting from the first live unit which has at least one reference to the variable. This component is made as large as possible to the extent that its basic blocks are connected. This has the effect of avoiding the



(a) Interference graph before live-range splitting



(b) Interference graph after live-range splitting

Figure 3.5: Live range split

creation of too small a *live range segment*. The live range splitting steps for the live range lr can be summarized as follows:

1. Find a live unit lu in lr in which the first appearance is a definition. If this cannot be found, then start with the first live unit. Add lu to a the live range lr'.

2. For each successor lu of the live units in lr in breadth-first order, add lu into lr' as long as lr' is colorable.

3. Update the interference graph and priority of lr and lr'

4. If lr and/or lr' become unconstrained after the split, move them from the constrained pool to unconstrained pool.

3.2.6 Spill code and shuffle code insertion

Spilling a live range lr assumes that the value of lr resides in memory and it is necessary to insert a store operation after every definition of lr and a load operation before every use. The costs due to these operations are called *spilling cost*.

When a live range is split into a number of smaller live segments and split live ranges are bound to different registers, *shuffle code* (also called *patch up code* or *compensation code*) may be needed to move the data from one live range to

the next. Figure 3.6 shows several different examples of shuffle code insertion. When a preceding live segment is bound to a register while the following segment is spilled, a store operation is required (Figure 3.6(a)). Likewise, if the preceding live segment is spilled and the following live segment is bound to a register, a load operation is required for the boundary as the Figure 3.6(b) shows. The Figure 3.6(c) shows that if two adjacent live segments are bound to different physical registers, a move operation is needed. In region-based register allocation, region boundaries provide implicit and natural splitting points to the compiler. In this case, the shuffle code for the region boundaries are identical to the shuffle code required in live range splitting.

Other operations are also inserted during register allocation in order to comply with the calling convention. Many compilers divide the registers into two sets, *caller-save* and *callee-save* registers, and we will cover the detailed issues about them in a later chapter.

The register allocation cost is the total overhead due to register allocation, as compared to the ideal case where all variables can be assigned to the physical registers. As previously explained, this cost is the sum of three components:

1. Spill Cost to move data to and from memory when no register is assigned to a live range

2. Call Cost to store and load registers upon procedure entry and exit

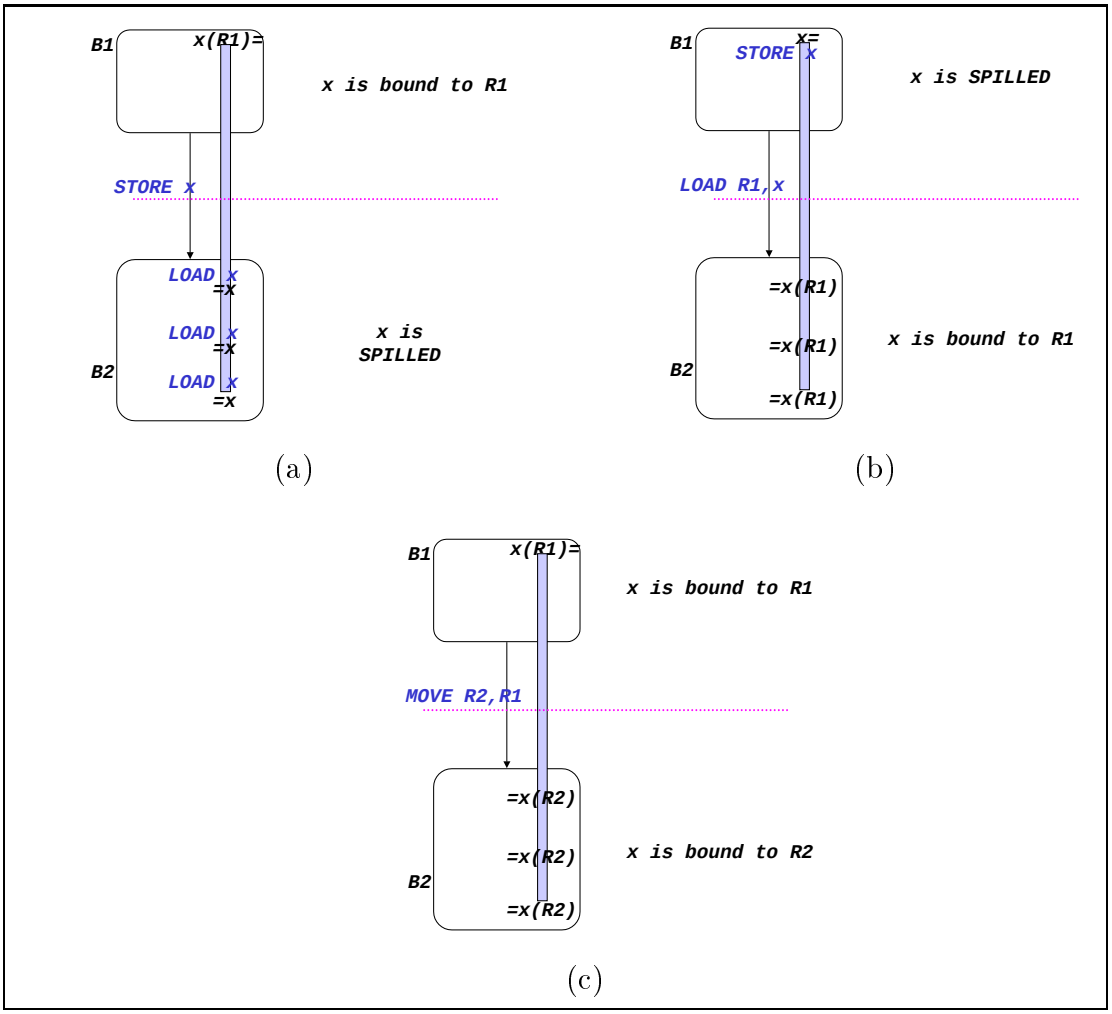


Figure 3.6: Live range split and shuffle code

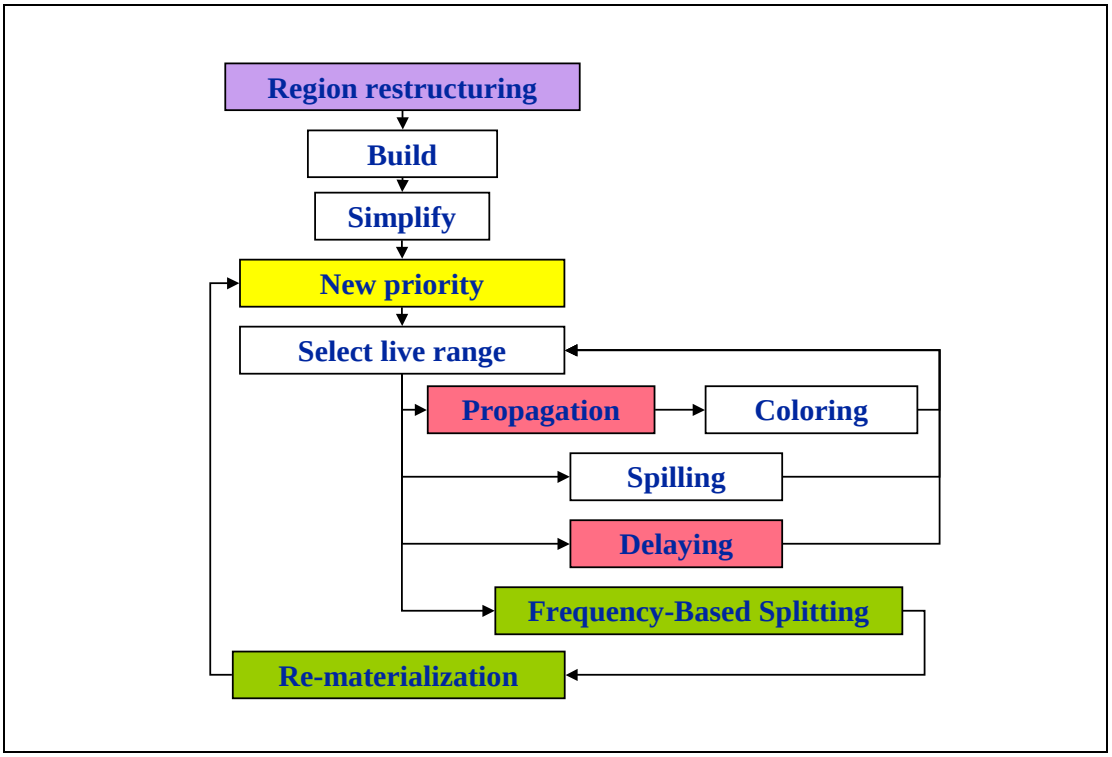


Figure 3.7: The framework of region-based register allocation

3. Shuffle Cost to move values from one live range segment to another

3.2.7 Region-Based Register Allocation Framework

Our region-based register allocation framework is shown in Figure 3.7 based on Chow and Hennessy framework. To overcome the problems of suboptimal coloring by limited scope of region-based register allocation, we introduce the notion of *delayed binding* and *propagation* based on frequency. The coloring of register allocation works collaboratively with these. We introduce a new live range splitting

technique based on execution frequency and explore it further with rematerialization. We will also study the problems related to predicated codes and the splitting of predicated live ranges followed by proposed solutions. Our priority function is refined for the region-based approach where many variables live-in and live-out are considered into the priority function. For the codes inside a hyper-block, we reflect the effects that the operations in hyper-block varies by the multiple branch points or the probability of the predication. Even though our basic register allocation framework is not limited to any specific type of region, we propose the concept of region restructuring based on estimated register pressure to improve the performance of region-based register allocation.

3.3 Other Region-Based Register Allocation

Callahan and Koblenz [7] partition the register allocation of a function by defining a hierarchical tiling based on the control flow graph. In the first phase, the tiles are colored individually in a bottom-up fashion and each virtual register is mapped to pseudo-registers. In a second, downward pass the pseudo-registers are mapped to physical registers. Norris and Pollock propose a similar hierarchical approach; however, the partition is based on the program dependency graph(PDG) [36]. The register allocation is done in a hierarchically in a bottom-up manner also, and Chaitin's algorithm is used at each region node. These two region-based

approaches using program structures would tend to allocate the highest frequently executed regions first, since leaf nodes of the hierarchical graph or the PDG-based regions tend to correspond to the innermost loop bodies. But such region formation is not sensitive to actual patterns of execution and frequencies of the region boundaries where the shuffle codes are inserted. Each control dependent subset of a given node that is common with those of another node is factored out, and a region is created to represent it. In this technique, the scope of register allocation tends to be too small, on the order of one C statement, resulting in unnecessary spill code [22]. In a branch intensive code, the regions formed by this method will not typically perform better than a basic block. Norris and Pollock report only a 2.7% improvement over a standard global register allocator.

Gupta, Soffa and Ombres[45] use Tarjan's result on colorability of a graph by decomposing it into subgraphs using clique separators. The result states that if each subgraph can be colored using at most k colors, then the entire graph can be colored in k colors by combining the coloring of subgraphs [46]. A *clique separators* is a completely connected subgraph whose removal disconnects the graph [19]. Each subgraph includes the clique separator, and a renaming of registers for the clique separator nodes might be needed when merged with other subgraphs. In Tarjan's work, the entire interference graph is constructed all at once, and clique separators identified later. In Gupta *et al.*, clique separators are identified by

examining code and interference graphs, then we construct one subgraph at a time for space efficiency. A program is partitioned by selecting traces (paths) through the CFG and finding clique separators for each trace. If a variable is live on multiple traces, renaming may be necessary at branch or join points. The maximum number of cliques, chosen as separators, in which a live range can occur is fixed to a small constant, e.g., 2. Though traces use frequency information, the clique separators (and hence the regions for register allocation) are determined by starting and stopping of live ranges rather than by groups of traces that are executed together frequently. The time complexity of register allocation with m-clique is $O(n^2 \div m)$, but the overhead of determining the clique separators is significantly larger than the benefit in register allocation time and results in significantly longer overall time.

The Multiflow compiler [12] employs trace scheduling as a framework for both register allocation and scheduling. The trace scheduler picks a trace and then passes it to the instruction scheduler. Rather than applying global register allocation after instruction scheduling, combined register allocation and instruction scheduling are performed. The instruction scheduler keeps the records of register binding and this information is maintained for each exit and entry point of a region through a data structure called a *Value Location Mapping (VLM)* as we will describe it further in Section 5.2. This information in VLM is used for the register

allocation of the subsequent regions for register biding decision, and the shuffle code between traces when a variable is not bound to the same register among the traces. In the IMPACT compiler, Hank uses a approach similar to the Multiflow compiler [22]. The compiler selects a region and performs classical global optimizations, ILP optimization, instruction scheduling and register allocation.

Proebsting and Fischer use probabilistic register allocation [40] which is a hybrid of the priority-based and program structure-based approaches. The approach consists of three steps, local register allocation, global register allocation, and register assignment. The global register allocation step partitions a program into regions based on the loop hierarchy and proceeds from the inner most loops to the outermost loops. When a variable is assigned to a register, the shuffle code is placed at the entry and exit points of a loop. Like previous structure-based region formations, this is not sensitive to actual patterns of execution and frequencies of the region boundaries.

Lueh [33] uses graph fusion in his fusion-based register allocation. His approach starts with an interference graph for each region of the program where a region can be basic block, super-block, a loop nest, or some combinations of these. Regions are merged in a bottom-up manner by the order of frequency of edges between regions, and the interference graph is fused and the live range is again simplified as in Chaitin's approach. His approach is closest to ours in the sense that frequency

information is used for fusion and that shuffle code are likely to be inserted at less frequent points. Live-range splitting is implicitly done at region boundaries. However, it does not split large super-blocks or hyper-blocks, even if the register pressure is very high inside a block.

Chapter 4

Live Range Splitting

In region-based register allocation where the regions are selected before the coloring process, the regions also provide implicit live range splitting points. However, inter-region live range splitting still plays an important role when the regions are constructed aggressively to obtain the higher level of ILP. These regions may need further splitting since selected regions are usually large and there is high register pressure. In this chapter we explore two important strategies for improving the effectiveness of graph coloring register allocation; live range splitting and rematerialization. Splitting divides an uncolorable live range into several smaller and potentially more easily colorable live ranges, sometimes called *live segments* specifically. Since smaller live ranges usually have less interferences than larger live ranges, splitting can frequently reduce a live-range into two colorable halves.

To preserve the semantics of the program, the shuffle code is inserted at the split points to connect the flow of values between the split live ranges. Rematerialization is another spill code saving technique. Rematerialization recomputes a value by need instead of reloading from memory, whenever this is possible and more profitable than spilling. We also explore the problem and requirement of the live range splitting for predicated code.

4.1 Introduction

There are two main issues in live range splitting and rematerialization: (i) how to choose the right live ranges to split and to rematerialize, and (ii) how to find the right places to split and to rematerialize. These decisions can drastically affect the quality of the generated code. The priority function, as we explained in 1.2.3, can guide the register allocator to find the live range to be split. It is important to understand that two split live segments are colored differently and the splitting points tend to be the places where the shuffle code is placed. Split live ranges tend to be colored with different physical registers since the bigger live was split into two segments when it could not be colored with the single register. If there is a register available through out the whole live range, then we would not need to split this live range into the smaller segments.

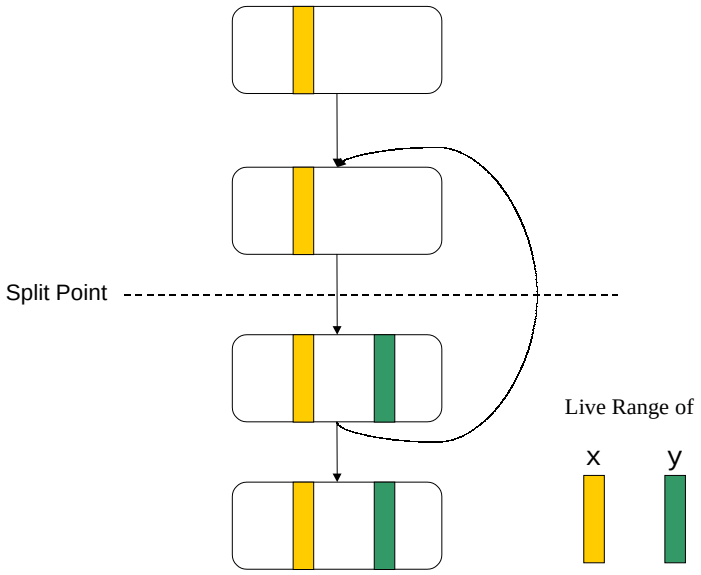
In particular, if the splitting point is located in a hot spot of the program, then

the shuffle code will be executed often, which may lead to a slow program execution. When we split a live range, it is important for the execution performance to select split point at points of low execution frequency, thereby reducing the shuffle costs. Figure 4.1 shows two different examples of two possible live range splitting when the live range of x is uncolorable, where Figure 4.1 (b) has lower shuffle costs than Figure 4.1 (a) has. In this chapter, we explore the limitations of the previous proposed live range splitting algorithm and present the new live range splitting algorithm based on frequency, called *Frequency Based Splitting (FBS)*.

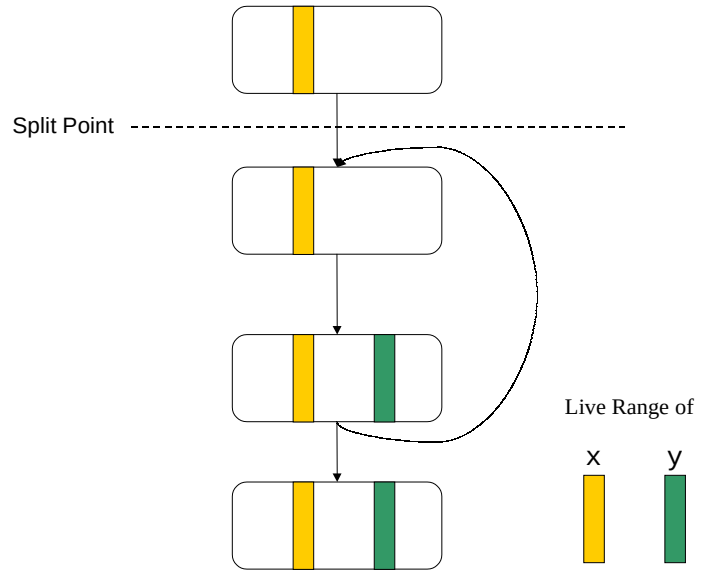
Chaitin *et al.* point out that certain values can be recomputed in a single instruction and that the required operands are always available for the recomputation. They call these always available values *never-killed* and there are many cases of this, like:

- integer constants or floating-point constants
- constant offset from the frame point or the static data pointer
- data loads from stack frame or the static memory address
- constant literal or string

The recomputation of a value from these never-killed is cheaper than storing that value to memory and loading it back when necessary. This technique



(a) Possible Live Range Split by Chow and Hennessy approach



(b) Preferred live range split

Figure 4.1: Live range splitting

is called *rematerialization* [5][3]. In other words, rematerialization is a spill reduction technique of replacing spills and reloads with instructions that recompute values that have been evicted from the registers, whenever this is possible and profitable. Figure 4.2 shows an example of rematerialization when there is no register available for the variable x. The left figure illustrates the code that would be produced when entire the live range of the variable x is spilled to memory such that load code is required before every use of the variable and store code is inserted after its definition. The right figure shows the code with rematerialization of x. The rematerialization codes are inserted before the uses and the original definition can be removed. This thesis presents the technique of *frequency based rematerialization*(FBR) based on FBS, used in our region-based register allocator for predicated VLIW/EPIC architectures. FBR is the natural extension of live range splitting and rematerialization is obtained only at the splitting points.

4.2 Frequency Based Splitting

The previous priority based approach by Chow and Hennessy [11] does not take account execution frequency when it splits the live ranges. Their approach uses a simple heuristic to split live ranges:

1. The split component of the original live range should be as large as possible

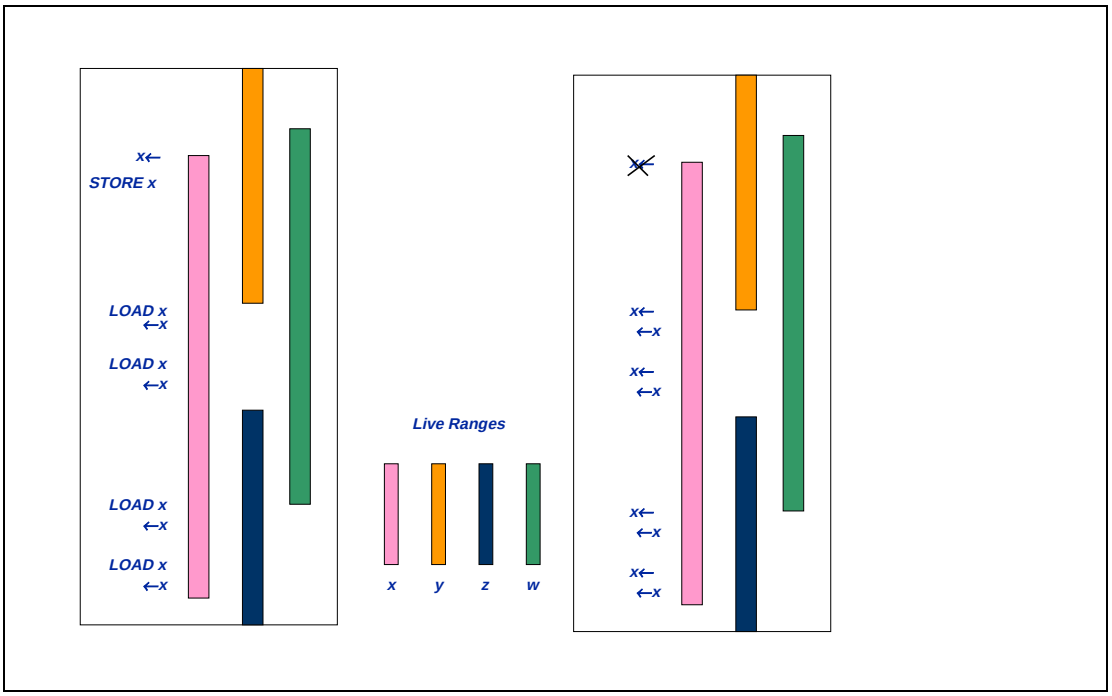


Figure 4.2: Examples of live range spilling and rematerialization

to the extent that the operations are connected. This avoids the creation of too many small fragments.

2. The split segment should be small enough to have fewer interfering live-ranges than the number of available registers, so at least one of the split live range is colorable.

The algorithm by Chow and Hennessy for splitting finds the first appearance of the variable and extends it as far as it is colorable. While their approach uses execution frequencies in the priority function, it does not use any frequency information in searching for splitting points; it finds splitting points in breadth first search(BFS) order with the highest register pressure. The region-based register allocator proposed by Hank [22] does not split live ranges in a region but instead spills them if they are not colorable. The problem with these approaches is that the split points of two divided live segments may have the highest frequency in many cases. Figure 4.3 shows an example of splitting a live-range with these two different schemes. If the split points are searched by BFS order and the search algorithm starts from the first region by the control-flow, then splitting may be done on a loop back edge. Figure 4.3 (a) illustrates this case where the splitting starts from the region $B0$ and the splitting point is between the region $B3$ and $B4$. Figure 4.3 (b) shows an example of an alternative way of splitting which may result from using frequency information to determine the split point (where

the live range in the loop is not split into two segments). In this case, the shuffle code between two live segments can be move to the less frequent point, the edge $E(B3, B4)$, than $E(B2, B3)$.

Briggs [3] also introduced live range splitting into Chaitin style register allocation. The splitting points are determined using *static single assignment* (SSA) [15] representations of programs and the loop boundaries [14] [13] [4]. Since live range splitting is performed prior to the coloring phase, the decisions regarding splitting points and live range selections for splitting are made prematurely. Live ranges may be split unnecessarily, resulting in execution performance degradation by extra shuffle code. Many heuristics are used to eliminate this extra shuffle code, including *biased-coloring* and *conservative coalescing*. Briggs conceived that his approaches often produce significant losses due to excessive copies, but he still did not provide a solution. The live range splitting based on SSA or loop structures can be used in priority based register allocation framework. But the splitting points by these approaches are not necessarily the places of lower execution points. For example, in a control-dependent program in which there is high register pressure inside a loop, a loop-based split may not be appropriate.

Live range splitting before the coloring process has many similarities to region-based register allocation where region formation is also performed before coloring. We will explain many region-based approached in Chapter 8 in details.

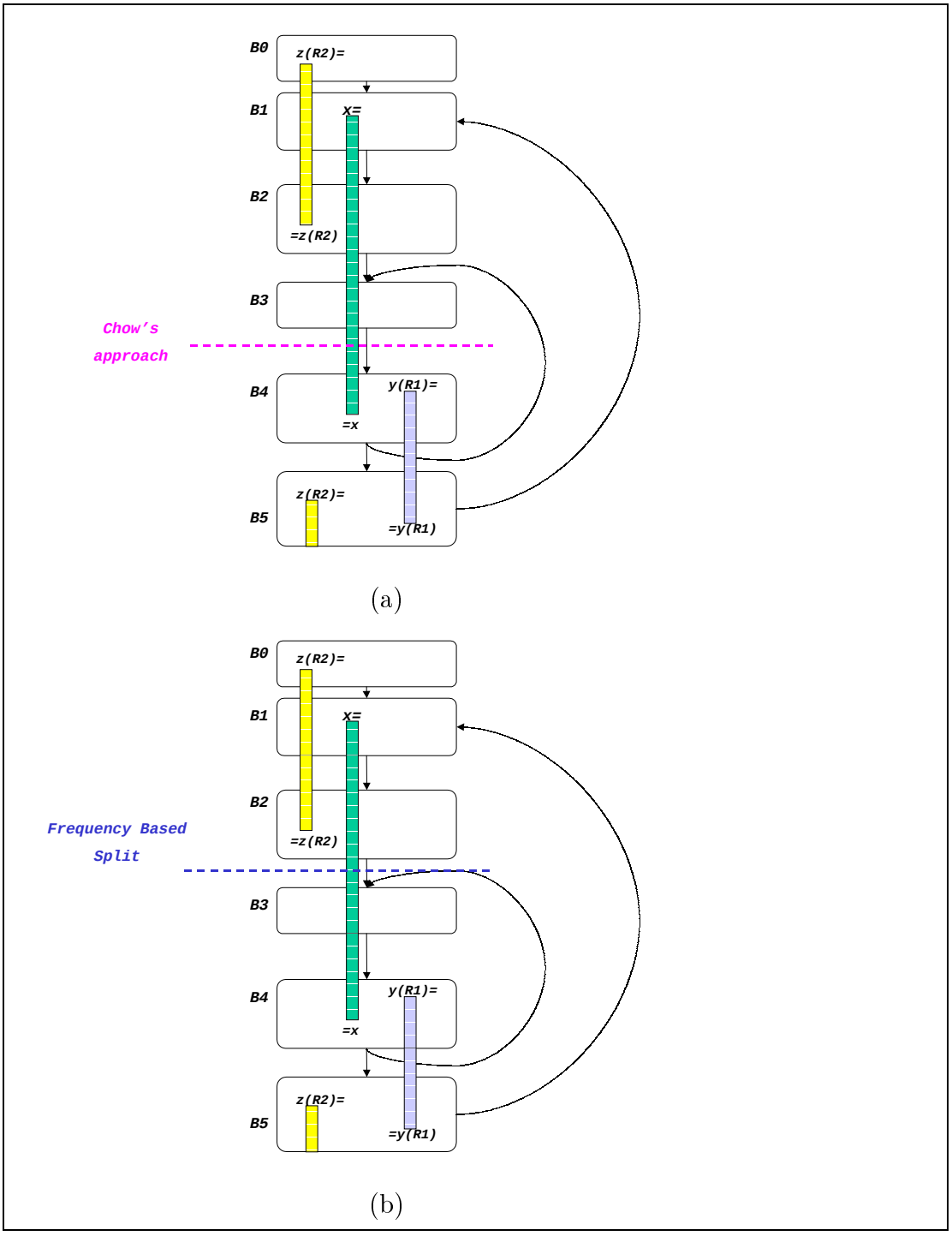


Figure 4.3: Possible Live Range Split by (a) BFS order (b) Frequency order

4.2.1 Algorithm

This thesis proposes the frequency based splitting (FBS) algorithm, which uses frequency information to guide live range splitting and attempts to split along the edges where the frequency is the lowest in the region. Our splitting algorithm is solely based on execution frequency and the order of finding splitting points is driven by the control flow frequencies between each base region (*i.e* basic blocks, super-blocks and hyper-blocks). From all the blocks considered in a region, the highest frequent block, called the *seed*, is selected. The scope of the first part of the splitting region is expanded to its successors or predecessors by the order of the frequency of the control flow arc as long as following conditions are met:

1. Expanded live-range is still colorable

2. Connected edge is the highest entry/exit edge of the neighboring region

The first condition guarantees that the new live range prevents too many splits from happening. Figure 4.4 shows examples of frequency-based splitting and splitting based on program structure (loop based splitting). The variable x is live in all regions and no register is available for x while the variable y is bound to $R1$ and the variable z is bound to $R2$. Our algorithm may choose $B4$ as a seed (the highest frequency block) and expand in the order of the region $B3$, $B1$, $B0$ and $B5$ by using edge frequency. Thus, it chooses the lower frequency edges

$e2$ and $e3$ as split points and minimizes the cost of inserting shuffle codes. If a split point decision is made by the program structure, as in loop based splitting, the live range of x is split at $e1$ and $e4$ and the shuffle cost is higher than in our frequency based splitting.

The second condition is necessary to avoid cases when splitting expands too aggressively and the splitting points are chosen at the higher frequency edges. Figure 4.5 shows an example of how standard frequency based splitting can be different from a conservative approach. In Figure 4.5 (a), the first splitting approach will choose $B2$ as the seed block and expand to region $B1$ and $B2$ until $e1$ and $e4$ are chosen as split points. If we choose $e2$ and $e3$ as split points instead, the shuffle cost will be reduced. By the second condition of our splitting algorithm, the region splitting starting from the region $B2$ will not be expanded to either the region $B1$ or $B4$, since $B1$ or $B4$ has other outgoing (or incoming) regions with higher frequencies between them. The detailed algorithm is described in Figure 4.6.

4.3 Rematerialization with Live Range Splitting

Frequency based rematerialization (FBR) is a natural extension of FBS by incorporating rematerialization in the splitting process. The main observation is that splitting points computed by FBS are also good places to insert rematerialization

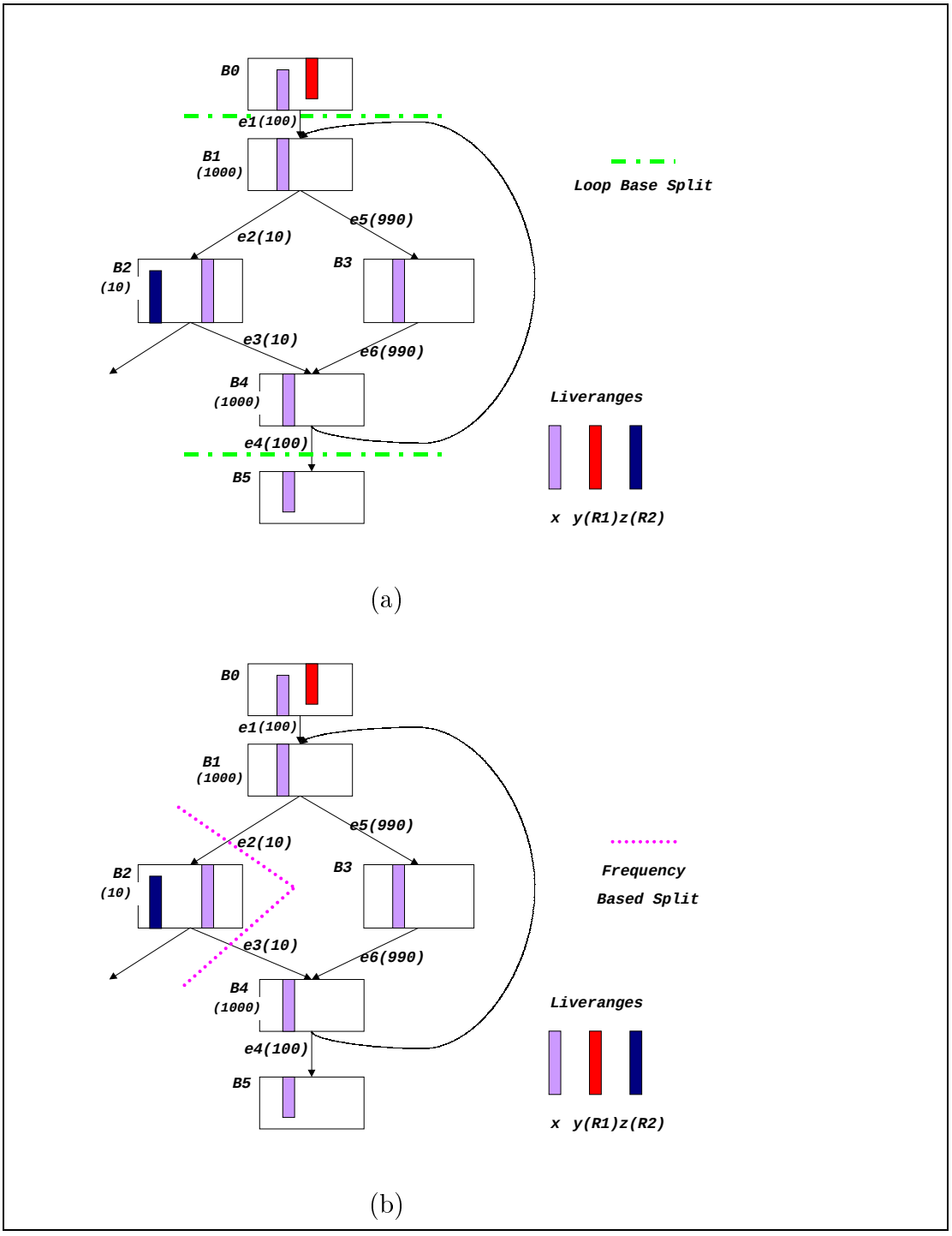


Figure 4.4: Live range splitting (a) by program structure (b) by frequency

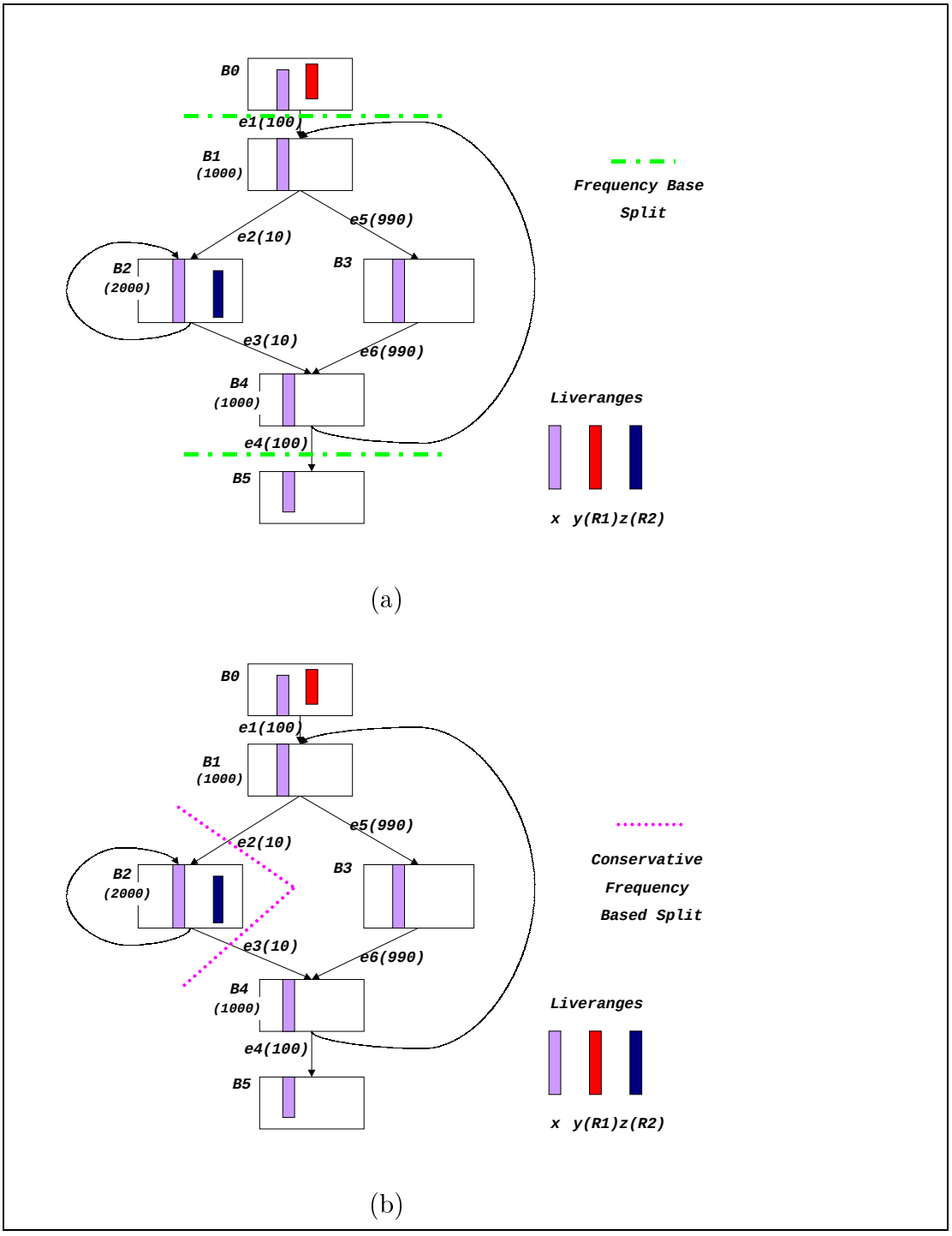


Figure 4.5: Improving Frequency Based Split

```
func split(live-range LR) {
    SEED = the highest frequent sub-region in LR;
    LR1 = {SEED};
    LR2 = LR - {SEED};
    EDGES = the cross edges between LR1 and LR2;
    while (EDGES) {
        pop highest frequency E from EDGES;
        NEXT = live segment in LR2 connected by E;
        if (E is the highest entry/exit edge compare to NEXT's
            corresponding entry/exit edges to LR2)
            if (LR1+{NEXT} is colorable) {
                LR1 += {NEXT};
                LR2 -= {NEXT};
                update EDGES as cross edges between LR1 and LR2;
            }
    }
    return (LR1, LR2);
}
```

Figure 4.6: Frequency Based Live Range Splitting Algorithm

code, for these reasons:

- First, FBS splitting points are, by definition, the program points of low execution frequency, and rematerialization code that is inserted at these places will also be infrequently executed.

- Secondly, FBS always splits a live range into two live ranges such that at least one is colorable. Rematerialized code inserted in the colorable live range is guaranteed not to be spilled.

- Finally, rematerialization can dramatically reduce the sizes of split live ranges over that of FBS, making it more likely that these live ranges can be colored as well as other live ranges.

Briggs uses SSA in his implementation of rematerialization [5]. By examining the defining instruction for each value, he recognizes never-killed values and propagates this information through the SSA graph. *Sparse simple constant algorithm* by Wegman and Zadeck [52] is used to propagate never-killed information .

4.3.1 The FBR Algorithm

The FBR algorithm is described in Figure 4.7. The basic framework of FBR is very similar to FBS. We first perform the splitting step and partition the live range `LR` into two live ranges `LR1` and `LR2`. We then check all live range splitting

```
function FBR(live range LR of x)
  (LR1,LR2) = FBS(LR);
  for all cross edges E between LR1 and LR2 do
     if a value v crosses E and
        v is live and
        v is rematerializable then
        rematerialize(v,x,E)
```

Figure 4.7: The Frequency Based Rematerialization Algorithm

```
function rematerialize(v,x,E)
   Insert code to recompute v at E
   /* Recompute the live range after rematerialization */
   Traverse the control flow graph from E
                        in the backwards direction.
   Incrementally recompute the live ranges.
   Remove dead definitions of x in the process.
```

Figure 4.8: The Rematerialize Algorithm

points, represented as cross edges `E` between `LR1` and `LR2`. For each edge `E`, we check whether the value crossing `E` is rematerializable. If it is, we insert the rematerialized code and update the new live range information.

The auxiliary routine `rematerialize` shown in Figure 4.8 is responsible for updating the live ranges after a value has been rematerialized. Unlike splitting, which introduces a new definition and a new use (in the shuffle code), rematerialization introduces a definition but does not introduce a new use.

The example in Figure 4.9 illustrates the working of the FBR algorithm. We

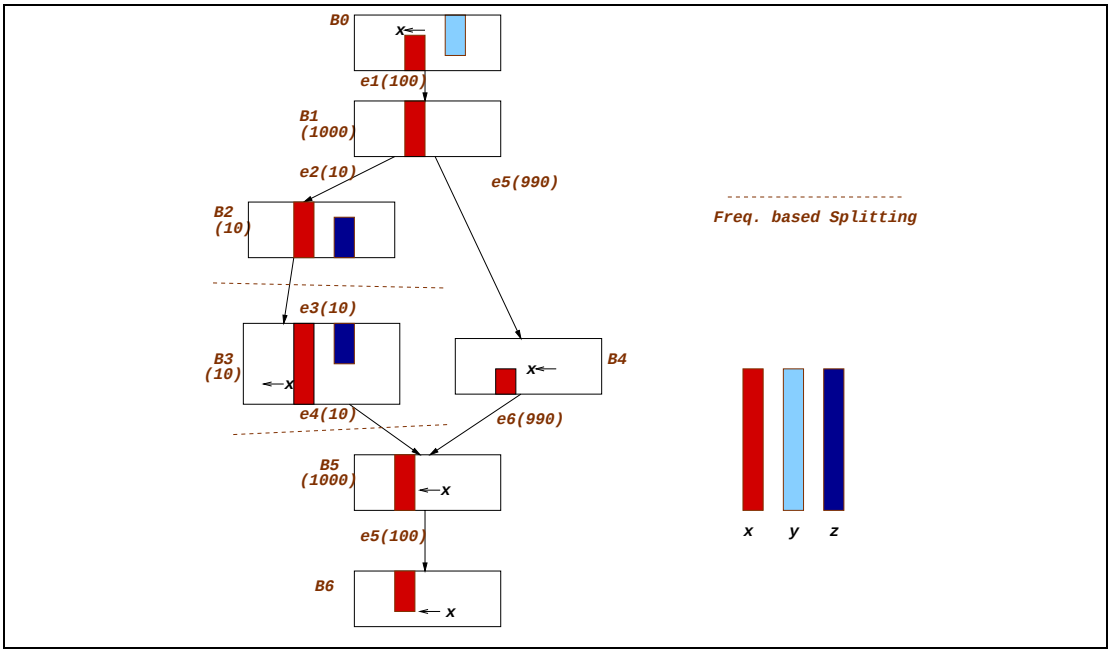


Figure 4.9: A Rematerialization Example.

first use FBS to split the live range corresponding to the variable x. FBS has determined that edges $e2$ and $e4$ are the best split points of the program. The result of the splitting divides the live range into two live ranges spanning blocks $B0, B1, B2, B4, B5, B6$ and block $B3$. Let us suppose that the definition of x in block $B1$ is rematerializable. FBR will then proceed to insert rematerializable code at the split edges $e2$ and $e4$. The result of this rematerialization is shown in Figure 4.10(a).

Finally, FBR reconstructs the new split live ranges by performing an incremental liveness analysis by walking the program in the reverse direction of the

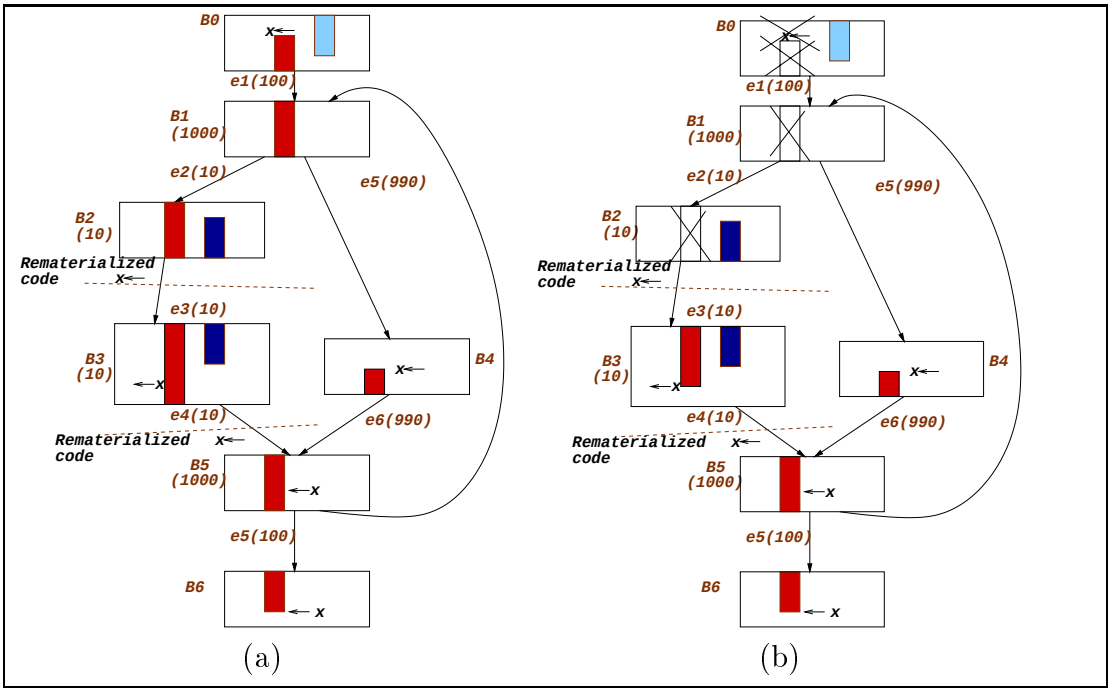


Figure 4.10: Rematerialization Steps.

control flow graph. We terminate this traversal as soon as we encounter use of x^1. In this example, the definition of x in block $B0$ can be deleted, since it is no longer live by the virtue of the rematerialized code in $e2$. The result is that we can delete the portion of the live range in block $B0, B1$, and $B2$, leading to a smaller interference graph. The result of this transformation is shown in Figure 4.10(b).

4.3.2 Finding Rematerializable Code

One decision that every rematerializing allocator must consider is to determine what live ranges in a program are rematerializable. Briggs [3] suggests that each live range should be decomposed into its component values, according to the structure of the SSA [15] graph. To discover rematerializable values, a simple forward propagation algorithm similar to constant propagation [51] can be used.

In Briggs' scheme, the following values are considered to be rematerializable:

- All constants,
- All address arithmetic expressions that depend only on constants and dedicated registers such as the frame pointer and a global data pointer.
- Loads from constant offset in the stack frame or in the static data area.

[1] All live-out values are considered to be uses.

To deal with predicated instructions, our algorithm extends Briggs' scheme in the following ways:

- Since instructions may be predicated in a hyper-block, a value is only considered to be rematerializable if its predicate is also rematerializable. This is because the rematerialized code must be predicated under the same condition as the original instruction.

- Predicate computations that depend on rematerializable value can be considered to be rematerializable.

In our implementation, we deviate from Briggs' scheme in not using SSA to discover rematerializable values. Instead, we maintain a set def/use chains and use these to discover rematerializable values by demand. We arrive at this implementation decision due to two important reasons: *(i)* SSA, without extensions, does not deal with predication in a clean way, and *(ii)* because of the use of region-based register allocation, our def/use chains are relatively compact, so the sparseness of SSA is not as big an advantage as in traditional global algorithms.

4.3.3 Optimistic Rematerialization

It is informative to compare FBR with Chaitin-style (and Briggs-style) rematerialization [8, 9, 3]. In these earlier frameworks, the decision to spill occurs relatively

late in the allocation process, and because of this, the placement choice of rematerialization code is restricted. In general, rematerialization code is inserted at where a reload may occur, i.e before the use of a value. This implies that if a rematerialized use site appears in the middle of an inner loop, the rematerialized code will be executed often. While global (post-pass) scheduling (e.g. [18, 24, 35]) can be used to hide the latency of these rematerialized code, most scheduling schemes stop at loop boundaries and thus they cannot hoist rematerialized code out of loops.

In contrast, the FBS heuristic decides to split and rematerialize live ranges relative early in the coloring process. Since we are have fine control on where splitting and thus rematerialization can occur, we can use this opportunity to minimize the cost of rematerialization.

Consider the example hyper-block in Figure 4.11(a) where the live ranges y, z and w have already been colored and there are no more colors available for x[2]. When the live range x is rematerialized in Chaitin/Briggs approach, rematerialization code in necessary before each use. The result of this is shown in Figure 4.11(b).

In the FBR algorithm we insert rematerialization code at split points even if we are not guaranteed that the rematerialized code can be colored. We call

[2]Other live ranges are not shown.

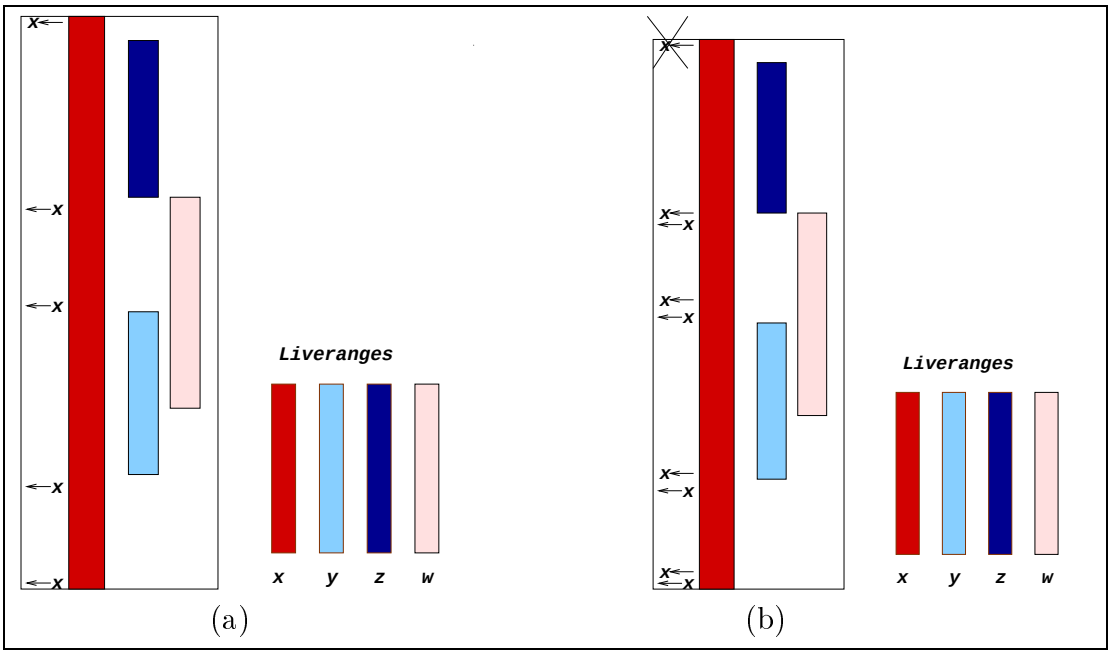


Figure 4.11: Chaitin-style Rematerialization

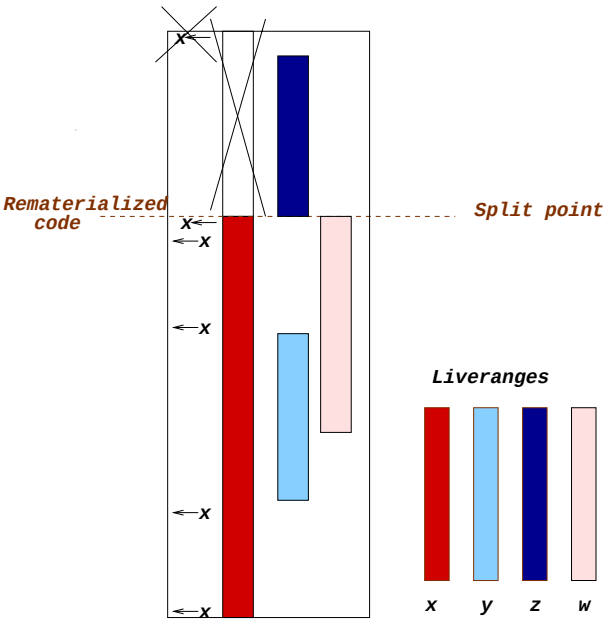
this *optimistic rematerialization*[3]. Note that since rematerialized code is itself rematerializable by definition, the result of this optimistic approach has a few interesting consequences:

- If the rematerialized code is itself colorable, then it will be colored as usual.
- If it is not colorable, we may recursively apply the FBR algorithm and split the (split) live range into smaller live ranges. As a side-effect of the FBR algorithm, dead rematerialization code inserted earlier will be deleted and new rematerialization code will be inserted at new split points.

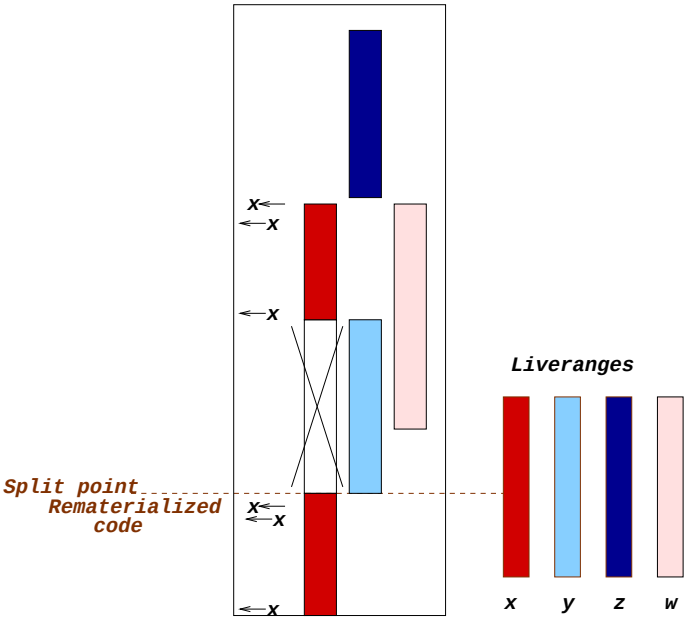
Figure 4.12 shows the operation of the FBR algorithm based on the live ranges in Figure 4.11(a). Here, an initial splitting point is chosen immediately after the the live range of z ends. Since the original definition of x is now dead after the rematerialized code is inserted, it can be removed. In Figure 4.12(b) we re-apply the FBR heuristic on the remaining live range of x, since it is still not colorable. After rematerialization is applied again, the top half portion of the split live range can be shrunk further. The resulting live ranges can then be colored.

Note that while live range splitting by itself reduces the register pressure by decreasing the degree of interference graph, the size of a split live range itself is not reduced. Using rematerialization, we can actually reduce the size of split live

[3]An alternative approach is to insert rematerialize only if we are guaranteed that the resulting code can be colored.



(a) Initial live range splitting



(b) x needs further splitting

Figure 4.12: Optimistic Rematerialization

ranges by removing unreferenced definitions.

4.4 Experiment

Figure 4.13 summarizes the performance of our frequency based split algorithm and rematerialization for two different sets of benchmarks: common Unix benchmarks and, selected Spec Int92 and Spec Int95 programs. We show three different sets of data for each benchmark, each compares to the total dynamic execution cycles of Chow's style of live range splitting: frequency based live range splitting (FBS), rematerialization based on splitting (FBR) and rematerialization with frequency based live range splitting (FBS+FBR).

In Figure 4.13 we compare the performance improvement of Frequency Based Splitting (FBS) and Frequency Based Rematerialization (FBR), using the dynamic cycle count as the basis. In this experiment, region-based register allocation is performed, and the granularity of each region is a hyper-block.

The target machine used in this experiment has 4 integer units, 2 floating-point units, 2 memory units and 1 branch unit, and it has 32 GPRs and 32 FPRs. The scope of the register allocation is the function based region.

Our experiment shows FBS and FBR achieves better quality of code and reduces the execution time by 10% and 7% respectively. Furthermore, a combination of these two techniques improves execution performance even better by as much

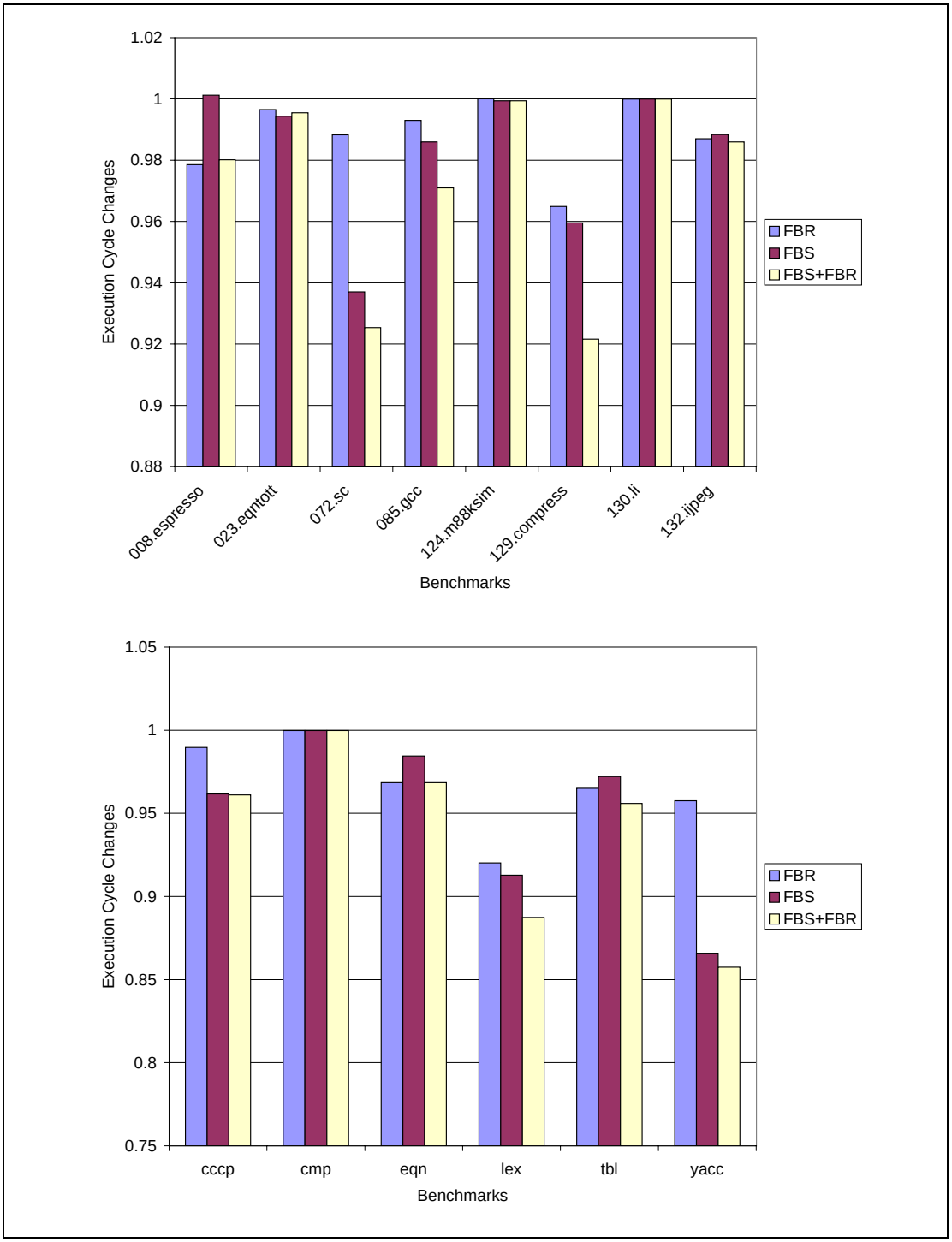


Figure 4.13: Frequency based splitting and rematerialization performance

as 12%. Most of the performance improvement of FBR can be attributed to rematerialization of function call addresses, branch target addresses, and address offsets from the stack pointer. Our conjecture is that as we increase the size of the hyper-blocks, the regions will have more branches and function calls and thus will result in even greater opportunity for rematerialization.

4.4.1 Sensitive to Register Pressure

The performance of live range splitting is closely related to the register pressure. As register pressure increases, more live ranges need to be split and different techniques of live range shows more difference. Register pressure is sensitive to two major factors in ILP compilers: the size of register file and the number of functional units. It is not hard to see that register pressure decreases as register file size increases. In ILP compiler, more instruction can be scheduled in parallel as the number of functional units increases.

Table 4.1 shows the number of live range splitting required for our register allocation for the different size of register files. For this experiment, we use the machine has 4 integer units, 2 floating-point units, 2 memory units and 1 branch unit. The scope of the register allocation is the region based on a function. The count of live range splitting decreases dramatically after 32GPR+32FPR set. In the machine model with ILP scheduling we use 64GPR+64FPR register seems to

Benchmark	32GPR+32FPR	64GPR+64FPR	96GPR+96FPR
008.espresso	1487	90	58
023.eqntott	359	16	8
124.m88ksim	83	23	15
129.compress	193	13	4
130.li	25	15	15
132.ijpeg	1079	224	92
cccp	268	13	4
cmp	18	2	2
eqn	53	9	8
tbl	654	67	42
lex	1048	88	37
yacc	1486	50	44

Table 4.1: Live Range Split Count

be enough. This becomes more clear in the following test.

We also vary the size of functional units to increase ILP factor and register pressures. In our experiments, we increase the number of integer units to 6 and floating-point units to 4, but this did not improve the execution performance significantly. To explore this behavior further, we conduct the experiment of register allocation with an infinite number of registers, so no register will be spilled. Table 4.2 shows the total dynamic execution cycles and dynamic number of operations with ILP factor for our selected benchmarks. Our experiment shows that the average number of instruction per cycle is no more than 3. This means the number of functional units in our experiment was not what prevented us from obtaining higher ILP. The instruction scheduling and region formation

Benchmark	Dynamic Execution Cycles (C)	Dynamic Operations Count (O)	ILP factor (O/C)
008.espresso	218322683	760693506	3.48
023.eqntott	361079842	1555775187	4.31
072.sc	265024760	636878163	2.40
085.gcc	84275725	183525339	2.18
124.m88ksim	56474256	123990825	2.20
129.compress	14564842	42110391	2.89
130.li	105162363	242728632	2.31
132.ijpeg	569619724	1617547655	2.84
cccp	3262669	9480713	2.91
cmp	1275577	2196090	1.72
eqn	24063896	68600191	2.85
tbl	1689261	4862907	2.88
yacc	19057501	61774019	3.24

Table 4.2: Execution Data and ILP factor

used does not fully expand ILP currently, which may be at least partially the reason that most of benchmarks do not require more than 64GPR+64FPR. Under current circumstance, our experiment is conducted mostly with 32GPR+32FPR, 48GPR+48FPR, and 64GPR+64FPR, if it is not specified.

4.4.2 Interference Degree Changes

To validate the effectiveness of the FBS and FBR algorithm, we also have conducted an experiment of the changes of the live range splitting. Our experiment shows that FBS can reduce live range splitting by 50% on average, compared to Chow's approach. The result is illustrated in the second column and the third

column of the Table 4.3. In splitting a live range, Chow separates out a component of the original live range that is as large as possible to the extent that it is colorable. In other words, the counterpart of the split live range may not be still colorable and may need further live range splitting. This counterpart of live range splitting, when the structure of the program is not considered, tends to need further live range splitting.

As we explained earlier, rematerialization can reduce the degree of interference graph by reducing the size of live range rematerialized. This leads register allocation to decrease register pressure on many other live ranges and the number of live range splitting required can also be decreased. In Table 4.3, we show the number of live range splitting required in selected benchmarks. Column 3 and 4 shows the live range splitting count for our base case which is frequency live range splitting (FBS) without any rematerialization and with rematerialization based on live range splitting (FBS+FBR). The percentage of savings are showed in the last column.

4.5 Live range split for predicated codes

Predication[27] has been included in EPIC-style architectures and provides many opportunities of ILP optimization to the compiler. It enables modulo-scheduling[41] to reduce code expansion and to be scheduled with kernel-only codes. More com-

Benchmark	BASE	FBS	FBS+FBR	1-FBS/BASE	1-FBS+FBR/FBS
008.espresso	1487	744	717	49.97%	3.63%
023.eqntott	359	251	239	30.08%	4.78%
072.sc	352	115	115	67.33%	0.00%
085.gcc	7431	2312	2078	68.89%	10.12%
124.m88ksim	83	30	30	63.86%	0.00%
129.compress	193	113	94	41.45%	16.81%
130.li	25	15	15	40.00%	0.00%
132.ijpeg	1079	668	655	38.09%	1.95%
cccp	268	140	113	47.76%	19.29%
cmp	18	9	9	50.00%	0.00%
eqn	53	30	25	43.40%	16.67%
lex	1048	505	387	51.81%	23.37%
tbl	654	253	228	61.31%	9.88%
yacc	1486	621	605	58.21%	2.58%

Table 4.3: Comparison of live range split count for FBS and FBS+FBR

monly, predication handles branch intensive programs since trace scheduling or super-block scheduling cannot handle clusters of traces that should be considered together. However, the predicated instructions in hyper-blocks [35] pose many interesting problems to many phases of the compiler including register allocation.

Predicate analysis for liveness we explained in Section 3.2.2 was one approach to cope with predicated code. Live range splitting for predicated code poses another problem related to shuffle code. If the interference graph construction uses liveness analysis with predication, the shuffle code must be predicated also with the same predicate condition of variable liveness at the split point. Predication for shuffle code makes post-pass scheduler (which comes after register allocation) to

schedule this code efficiently and can reduce the execution time. More importantly this is required to reserve the semantic of the program.

Consider the example in Figure 4.14 where the variable x is defined and used under the condition p, and the variable y is defined and used under the condition q where the predicate value p and q are exclusive to each other. If variable x was split and two segments of x were bound to register $R1$ and $R2$ respectively, we need to guard the shuffle code with the same predication condition of the liveness of variable x, i.e. p. If the predication guard is missing for the shuffle code, the value of y guarded by the predicate q can be corrupted by the shuffle code when q is true.

It is important to understand that predicate conditions for the shuffle code must be exactly the same as the liveness condition used in the phase of the interference graph construction. Please consider the example of liveness condition of live range x in Figure 4.15 (b) which is obtained from control flow graph in Figure 4.15 (a). At the split point, the variable x is live under the predicate condition of $p \wedge (s \vee u)$. There are a few obstacles in making this shuffle code with the predication. First, some conditions for the shuffle code may not yet be defined at the split point. In this example, predicate variable for the condition s or u may be defined somewhere between the split point and the use operations. This problem can be solved by refining the live range split carefully. We always

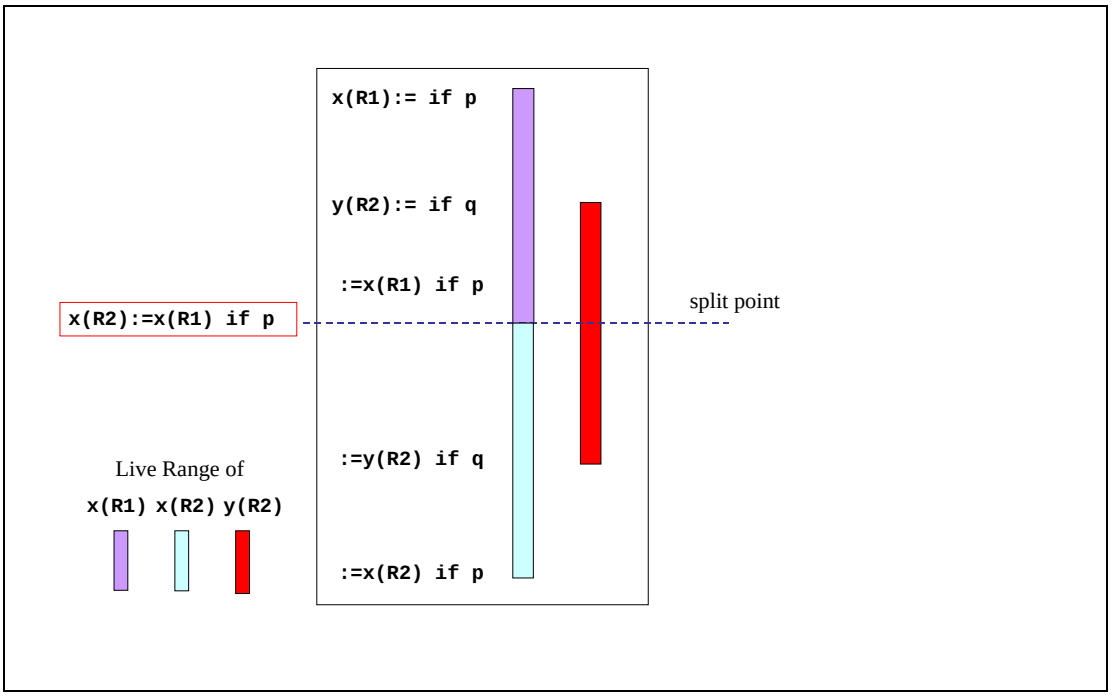
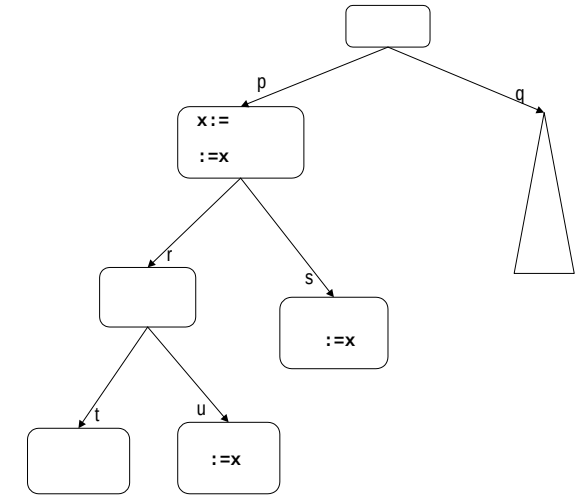
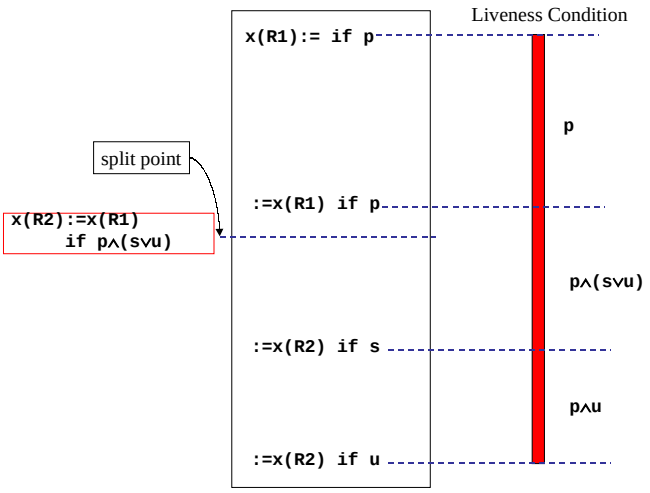


Figure 4.14: Predicate guard of shuffle code



(a) Predicate partition graph with code fragment



(b) Hyper-block codes and liveness condition

Figure 4.15: Predicate partition graph

make the splitting point to be after the definition of predicate variables needed for the shuffle code. From the end of live range, we scan the operation in reverse order of execution as long as it it colorable as in Chow's approach. We may select splitting points in the following ways:

- if we encounter an operation which makes the new live range uncolorable
- or if we encounter the predicate definition which the variable refers to.

The second problem is the predicate expression required for the shuffle code may be difficult to obtain and may need too many operations. As in Figure 4.15 (b), the program did not define the predicate variable for predicate expression $p \wedge (s \vee u)$ as a single predicate variable. We may need many instructions to derive the desired predicate value from the existing predicate expression. In our example, we need to define the new predicate variable v for $p \wedge (s \vee u)$ and guard the shuffle code. The features of EPIC architecture make it possible to optimize such a way that this shuffle code can be scheduled efficiently, if the machine supports simultaneous writing to the register. Some EPIC architecture design like HPL-PD permits multiple operations to simultaneously write a value into a register provided all such operations write the same value. In this case, the result stored in a register is well-defined and is the value being written into the register, so the post scheduler can schedule the shuffle in the same clock cycle. Figure 4.16 shows

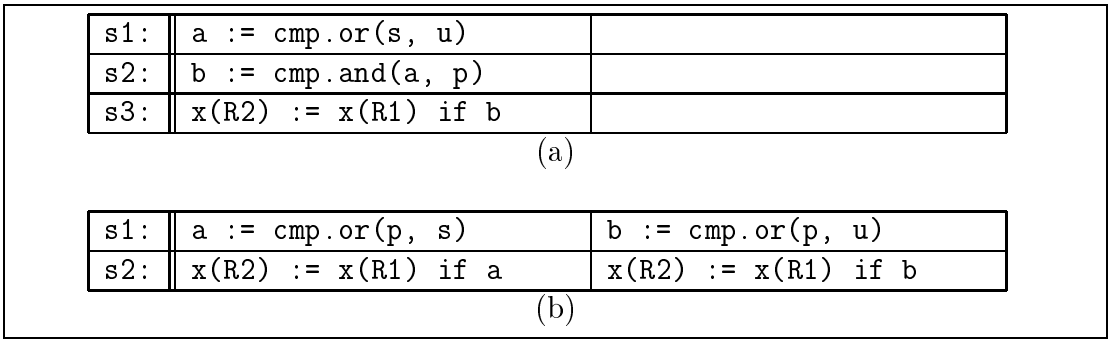


Figure 4.16: Shuffle code for the hyper-block in Figure 4.15. Predication condition $p \wedge (s \vee u)$ is promoted to $s \vee u$. (a) Predicate variable w is defined and used for shuffle code. (b) Parallel shuffle code where simultaneous writing is performed.

an example code of the difference between the simultaneous writing approach and the non-simultaneous approach.

The last problem can occur if there is high register pressure for predicate registers and new predicate registers may not be available. In our implementation of the liveness analysis used in the interference graph construction, the predicated expression is promoted to the lowest upper bound predicate value defined in the program by the conservative approximation based on the predicate partition graph [20][16]. So in our previous example in 4.16, the predicate expression of $p \wedge (s \vee u)$ will be promoted to its smallest superset as p and this is the predicate guard used in interference graph construction. In this approach, the predicate guard for shuffle can be expressed in conservative simple form and we do not need any extra predicate register. To guarantee the simple predicate variable is live at

the split point, we again check the liveness of that predicate register at splitting time.

Chapter 5

Register Assignment

region-based register allocation has potential weakness when compared to the global register allocation because of its limited scope of region as a compilation unit. By using region formation, one large live range can be divided into several live segments and the register allocator in a region may be performed fully independently before a compilation scope moves to the next region. Coloring each region independently may be suboptimal as each live segment may be assigned a different register, resulting in many patch up codes at each live segment boundary. When each live segment is colored independently, the register allocator must ensure that the necessary shuffle code is inserted to handle the different register assignment between regions, and this shuffle code increases execution time in many cases.

In this chapter, we explore the problems of the register assignment in region-based register allocation and propose several innovations to overcome this potential disadvantage of the region-based approach, thereby reducing shuffle cost. All of our innovations are also centered on knowing and using the *execution frequencies* of the program units such as basic blocks, super-blocks, hyper-blocks and others.

5.1 Issues Related to Register Assignment

According to how two adjacent regions are assigned to the registers, shuffle code may be required to reconcile those two regions. Ideally, if both adjacent regions are assigned to the same physical register or both spilled, no shuffle code is required. If a variable is allocated to different physical registers in different regions, then shuffle code is required. Figure 5.1(a) shows the cases where store and load shuffle are required. The variable x is allocated to a physical register $R1$ in the region $B1$ and the region $B3$ while x is spilled in the region $B2$. The variable x needs to be stored into memory after $B1$ but before $B2$, and needs to be loaded back to the register $R1$ before the region $B3$. If two adjacent live segments are colored with different colors in each region, copy operations are required. In Figure 5.1(b), the variable x is assumed to be allocated to different physical registers; the register $R1$ in the region $B1$, the register $R2$ in the region $R2$ and the register $R3$ in the

region $B3$ respectively. This requires copy operations at each region boundary as $R1$ is copied to $R2$ between $B1$ and $B2$, and $R2$ is copied to $R3$ between $B2$ and $B3$.

Please note this shuffle code is very similar to the code required in live range splitting. region-based register allocation has a meaning of aggressive live range splitting which splits the live ranges before the coloring process [6]. So it is natural that shuffle code requirements for live range splitting and region-based register allocation are the same. We will explore and compare in detail dynamic live range splitting (live range splitting is performed as coloring process is performed) and aggressive live range splitting (live range splitting is performed in advance) in chapter 8, where we cover the comparison of region-based approach and global register allocation.

To implement inserting shuffle code between two adjacent regions, another region may be required. Creating a new block degrades the runtime performance in two ways. First, the new block needs extra branch operation above the shuffle code itself, so we extra extra branch latencies. Second, the ILP factor of the shuffle block is relatively low since there tends to be not many operations in it. In particular, if the machine architecture does not support indirect memory access, the stack address for memory access operations needs to be stored in some registers. This may require special registers for stack address, or consumes

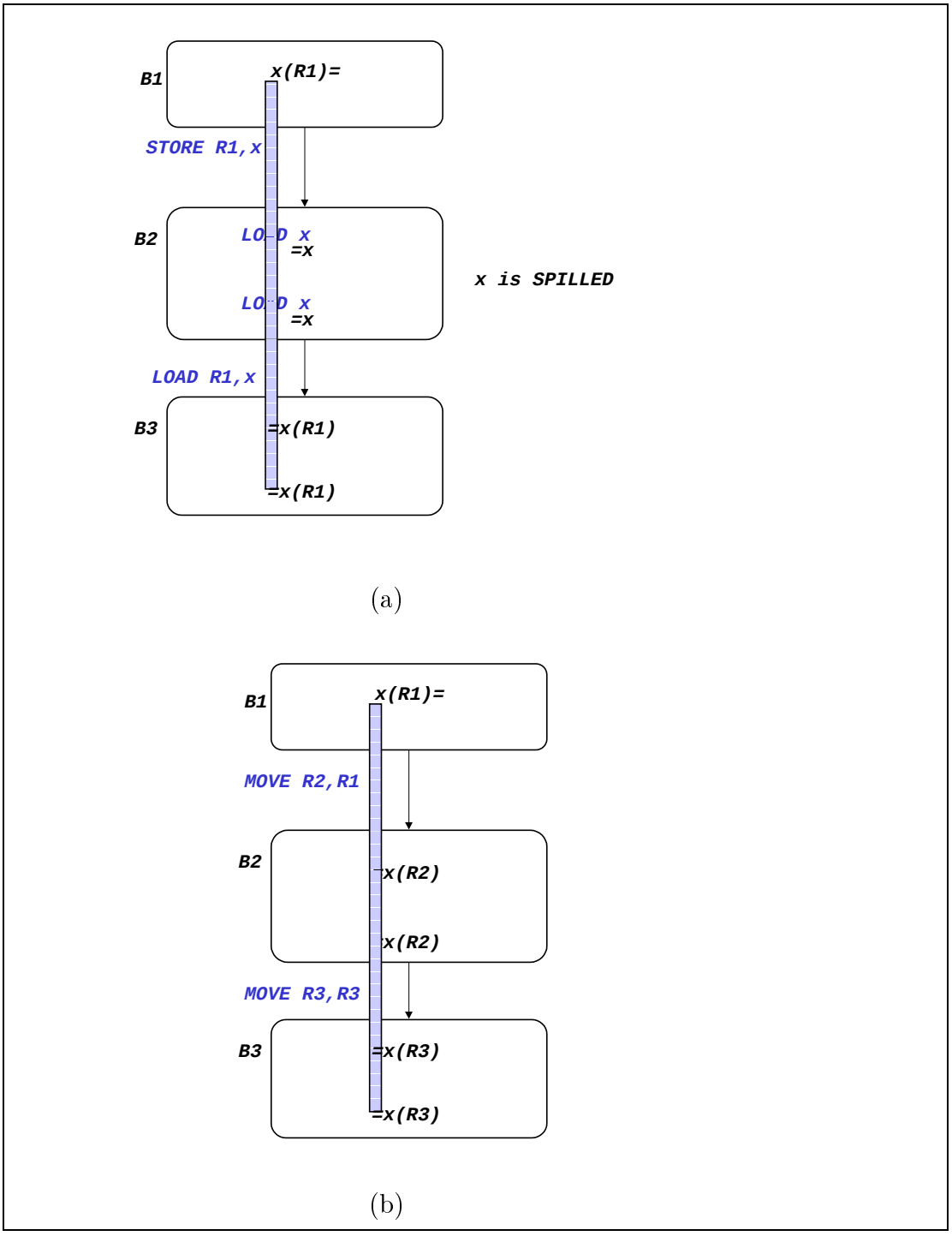


Figure 5.1: Region Reconciliation

general registers. So there can be false dependency between those memory address operands, and these operations cannot be parallelized.

In many cases, fortunately, shuffle code can be moved to either the source or the destination region. In two adjacent regions, if the only connecting edge is the single exit edge of its source region and the single entry edge to the destination region as in Figure 5.2 (a), the shuffle code can be moved to the either regions. Figure 5.2 (b) shows a case where the load shuffle code can be merged into the destination region but not to the source region. It is assumed that the variable x is live in the region $B1$ and the region $B3$, and it is bound to the register $R1$ in $B3$ while it is spilled in $B1$. The shuffle code can be merged into $B3$, but this code cannot be merged to $B1$ since it may change the semantics of the program. In this example, if the LOAD operation is moved to the region $B1$ and the control path to $B2$ is taken, the moved LOAD operation will over-write the value of variable y which was also assigned to $R1$. Figure 5.2 (c) shows a reverse case where the shuffle code can be merged into the source region only. It is assumed that the variable x is bound to different registers in each region, so shuffle codes are required in each region boundary. Neither shuffle code from $B1$ or from $B2$ can be merged into $B3$ while they can be moved into the $B1$ or $B2$ respectively. If any of shuffle code is moved to $B3$, it may change the meaning of the program when the execution control is coming from the other region. For instance, if the

left side MOVE operation is merged into $B3$ and the control is from $B2$ to $B3$, the live-in value of x will be overwritten by the value of y which is in the register $R1$. Finally, Figure 5.2 (d) shows a case where the shuffle code cannot be merged into either the source region or the destination region, and a new region for shuffle code is necessary. For the same region as we explained in the previous two examples, the shuffle code between $B2$ and $B4$ cannot be move into either the source or the destination region.

5.2 Frequency Based Propagation and Delayed Binding

Propagation is the concept where coloring information of neighboring segments is used, *i.e.,* propagated, in register binding for the region currently being colored. Propagation has been shown to decrease the shuffle cost by decreasing the shuffle code. Whenever possible, our frequency based propagation algorithm searches for a register from neighboring live ranges by the frequency order of control flow edges.

A segment of the live range in a region may not have any references to that variable (no defs and no uses), even if the value is live-in and live-out through the region. We call this a *pass-through live range*. Pass-through live ranges present

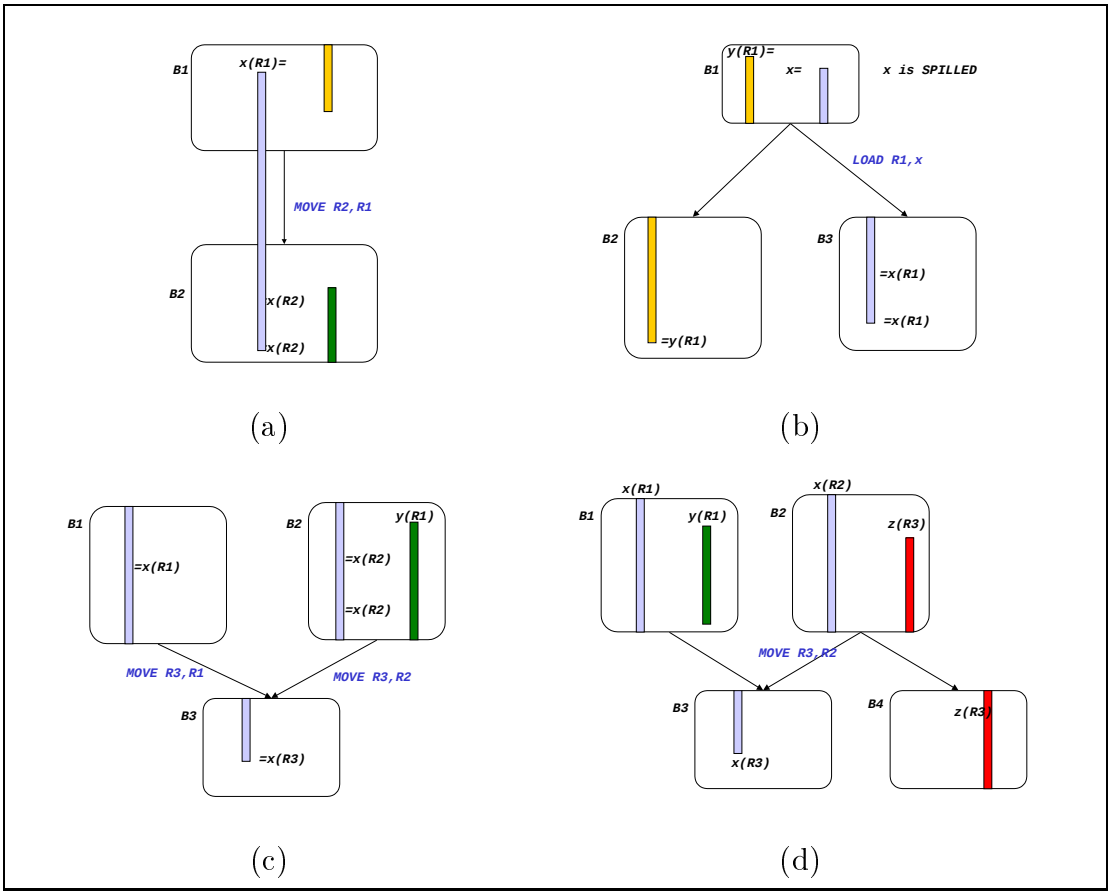


Figure 5.2: Shuffle Code Merge

another interesting challenge to register allocation. For pass-through live ranges, it is not clear whether we want to bind to a register in many cases. In the case that several pass-through live ranges exist in a region and the number of available registers is smaller than the number of pass-through live ranges , it is hard to decide that which live ranges can be bound to physical registers since the register binding benefits for all pass-through live ranges are equal to zero for the given region. Let us assume there are M live ranges in a region with N pass-through and $M > R$, $(M - N) < R$ where R is the number of physical registers. All of $(M - N)$ non pass-through live ranges can be bound to registers. From the N pass-through live ranges, only partial live ranges $(R - M)$ can be bound to registers. In this case, it is hard to decide which ones can be bound and which ones can be spilled.

In some cases, even though there are enough registers available for all live ranges, it may be more beneficial to spill a live range. If a live range was spilled in its neighboring region $B2$, it will be more beneficial to spill the pass-through live range in the current region $B1$ in order to avoid more expensive shuffle code between $B2$ and $B1$. When the information of the neighboring region is not known, it is hard to decide register binding for a pass-through live range.

As shown in [12] and [22], by *delaying* register binding decision for pass-through live ranges, we can resolve these problems. In [12], they delay the binding decision

whenever they cannot decide register binding, because no propagation information is available at that time. Whenever one of its neighboring live ranges have propagation information, this information has propagated to the delayed live segments. The simple version of propagation is also used in Hank's approach which is derived from the Multiflow compiler [12] by using VLM(Value-Location Mapping) to reduce shuffle cost problem.

The compilation manager maintains a table of register bindings of all outward-exposed virtual registers at each region entry and exit point of each register allocated region. Through the table of register bindings called VLM, register binding information is propagated from its neighboring regions processed by its region weight. Figure 5.3 shows an example of propagation of register binding information through VLM. The region $B1$ has been allocated and has three outward-exposed variables that live-in at the entry point of the region $B2$. This information can be used for two purpose. First, register allocation for the region $B2$ can use this information and guide the register binding to minimize the shuffle code. Second, this information is used to insert required shuffle code.

Given that regions are being compiled in order of importance (by its frequency order), Hank applied the register binding information of the first bound region to the following regions for propagation. But his approach may have limitations in cases of complex control flow graphs where a region has multiple entry or exit

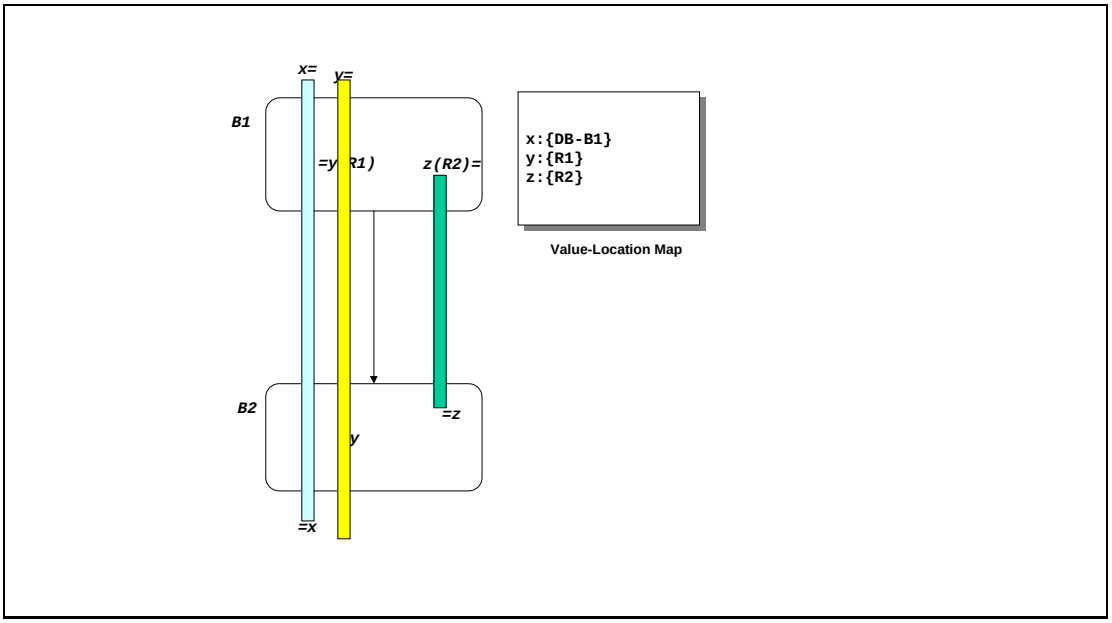


Figure 5.3: Value-Location Map example

edges, and the edge frequency is not exactly match to the region frequency. In Figure 5.4, the frequency of blocks are $B1 > B3 > B2$. If we process register allocation by block frequency order, the variable x in $B3$ can be colored with the register $R2$ while x was bound to the register $R1$ in $B1$ (this can happen, for example, if $B3$ has other live-ranges with higher priority which have been assigned $R1$). If regions are processed by control flow order and region frequency, the variable x bound to $R1$ can be propagated to $B2$. If x in $B2$ is bound a register other than $R1$, the shuffle code will be added to the edge $e(B2, B3)$. If $B2$ has no register pressure, it is more efficient to bind x with $R2$ in the region $B2$ and $B3$ and to put shuffle code in $e(B1, B2)$.

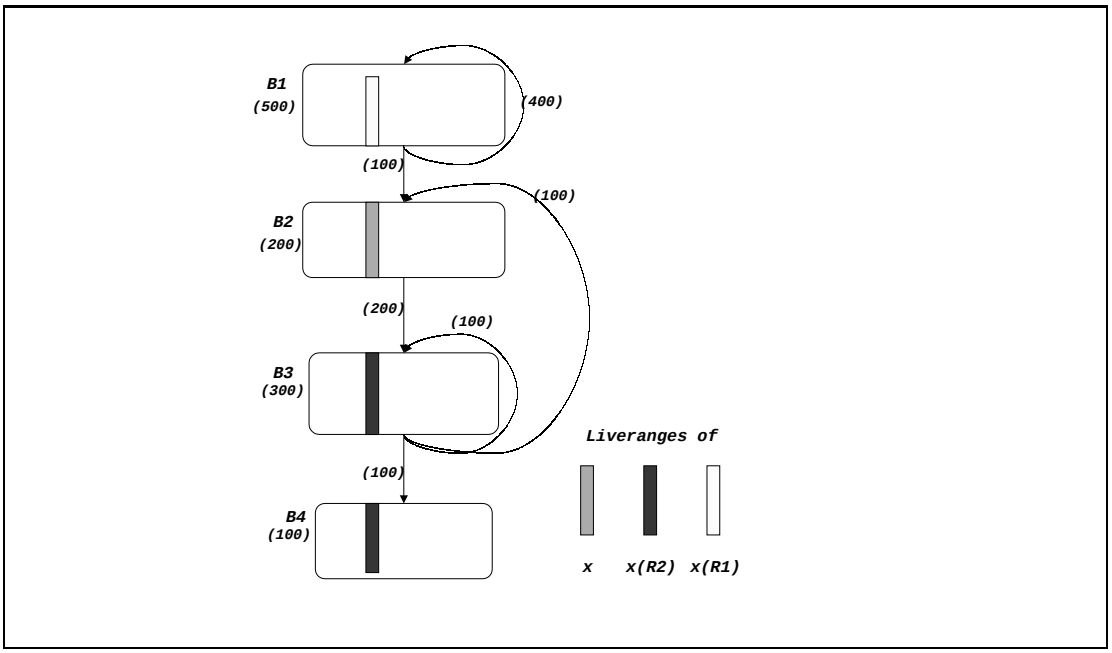


Figure 5.4: Region-Based Register Allocation and Propagation

In our approach, we propagate the potential candidates for binding the pass-through live range by using the adjacent live range information and frequency information. If a pass-through live range does not have any adjacent live range information, register binding for that live range is *delayed* until the register biding information can be propagated properly. The register binding candidates cannot be used when they are forbidden or they are unsuitable when the binding benefit is not beneficial compared to spilling with shuffling (*i.e.* the binding is unsuitable due to the presence of functions). In these cases, the live range is further delayed. Otherwise we find the preferred binding and reconcile. Using adjacent live ranges information, we compute benefits of the neighboring bindings by the frequency order of control flow, skipping only delayed ones. The benefit is calculated as the difference of register binding benefit and weighted shuffle cost at entry and exit points. If delayed bindings also exist on either side of this pass-through, shuffle cost is set to zero with the expectation that the actual shuffle cost will be absorbed by live ranges allocated later.

5.2.1 Algorithm

We now describe a register allocation algorithm that uses frequency information in the propagation process. Whenever possible, our frequency based propagation algorithm searches for a register from neighboring live ranges as outlined in Fig-

```
func coloring(live-range LR) {
    if (LR is pass-through)
        delay_register_ binding(LR);
    else if (LR has neighboring live range)
        coloring_with_propagation(LR);
    else
        allocate_new_register(LR);
}
func coloring_with_propagation(live-range LR) {
    initialize EDGES with set of edges between LR and all
    neighboring live ranges;
    LR.UNAVAILABLE = LR.FORBIDDEN;
    for each E in EDGES by frequency order {
        ADJ = neighboring live segment connected by E;
        if (ADJ is delayed)
            LR.UNAVAILABLE += ADJ->UNAVAILABLE;
        else if (ADJ is spilled)
            continue to next edge;
        else if (ADJ is bound and ADJ.COLOR is unavailable in LR)
            bind LR with ADJ.COLOR;
    }
}
```

Figure 5.5: Frequency Based Propagation Algorithm

ure 5.5. This refers to the register binding of the neighboring live range, in order of control flow edge frequency, to decide register allocation. In the case of the pass-through live ranges, register binding is delayed until the register binding of all of its neighboring live live ranges is finished instead of propagating from its first available neighbor.

Figure 5.6 (a) shows the examples of how our propagation works. It is assumed

that regions are compiled by their frequency orders. The region $B1$ is colored first where the variable x is assumed to be bound to the register $R1$. When the region $B3$ is colored, the register $R1$ is assumed to be used for an another live range, and the variable x is bound to the register $R2$ instead. When we bind the register for x in $B2$, our approach will propagate register binding information from $B3$ before $B2$ by the order of control flow. Figure 5.6 (b) shows a more interesting case with delayed binding. The region $B4$ is assumed to be the most important region and the register binding of the variable x is delayed, since it is a pass-through live range without any propagation information from its neighboring regions. When the region $B2$ is processed and the variable x is bound to the register $R1$, this coloring information can be propagated to the region $B4$. If x is spilled in region $B3$ later (there may be high register pressure in $B3$), the compiler manager should add shuffle code between $B3$ and $B4$. However, if the variable x is spilled in $B4$ rather than bound to any register, we need shuffle code somewhere in the control-flow between $B2$ and $B4$. This approach has lower frequency than allocating a register to x in $B4$. In our approach, the propagation of delayed live range is delayed repeatedly until its neighboring live range connected by the highest frequency control-flow edge has propagation information. For this purpose, we maintain the information of all control flow edges with live-in and live-out information when a region is constructed. In this example, the register propagation for x in the region

$B4$ is delayed until the register binding for x in the region $B5$ is decided.

5.3 Propagation of Unavailable Registers

Register binding for the current segment of a large live range can be constrained by the register which later cannot be used for delayed live range. If the current live range uses a register which was forbidden in delayed segment in the previously region, adding the compensation code is unavoidable. Thus, we prefer to bind the current live range with a register that is also available for the delayed segment in the adjacent regions, and we can avoid extra shuffle code. These concepts are illustrated below.

In figure 5.7, the variable a is bound to the register $R2$, and the variable y is delayed in the region $B3$. When the variable y is colored in the region $B2$, we want to use a register other than $R2$, otherwise shuffle code for y in the edge $E(B2, B3)$ is required. This information of preferred or unpreferred registers needs to be propagated to other regions for further optimization. In figure 5.7, register binding for the variable x will be delayed in $B3$ as well as in the region $B2$ and the region $B4$. When register allocation binds a register to x in the region $B1$, the register allocation should avoid using $R1$ but would also not use $R2$ to prevent any compensation code insertion. In particular, binding the variable x with the register $R2$ may lead to even poorer quality code than using $R1$, since

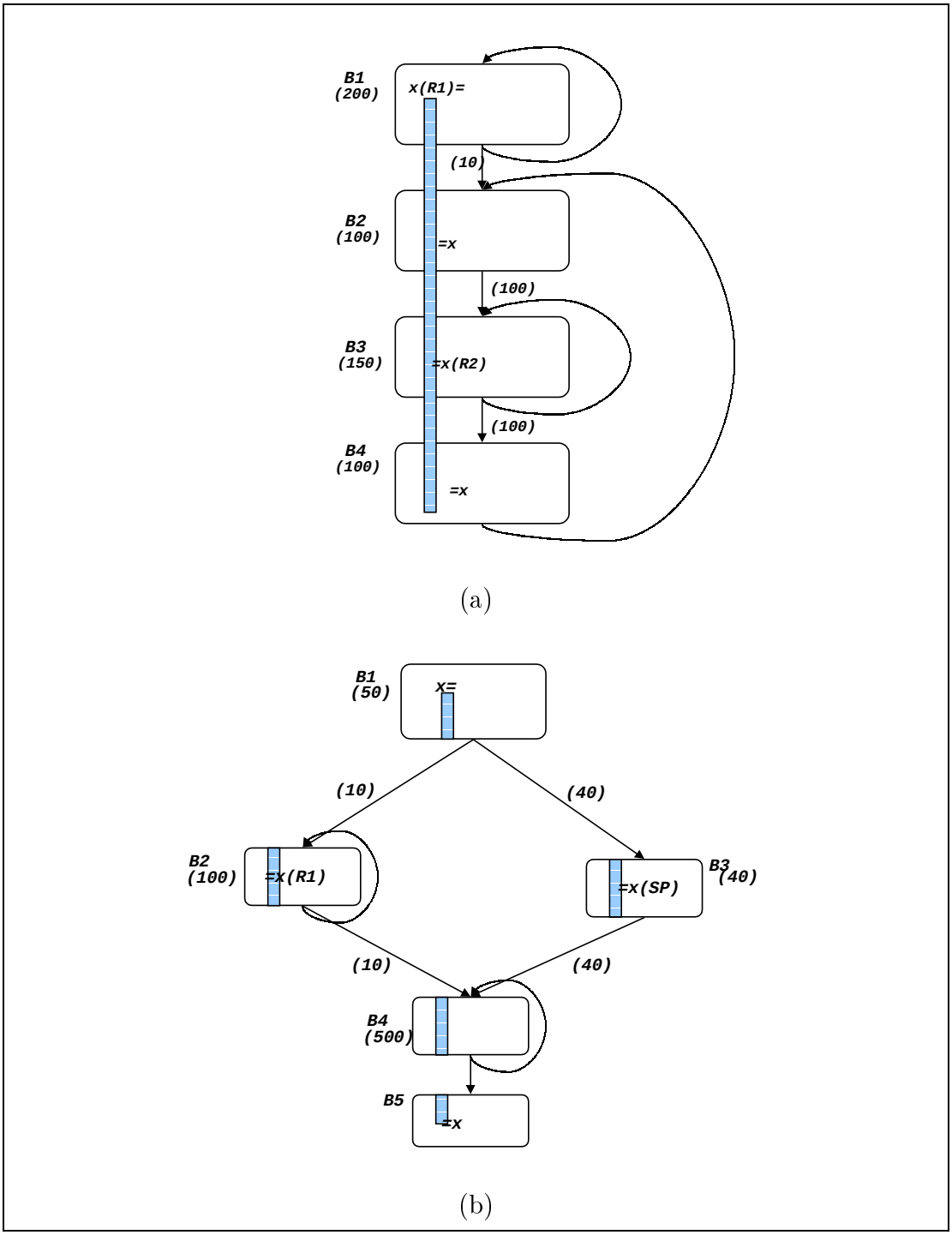


Figure 5.6: Register Binding Propagation

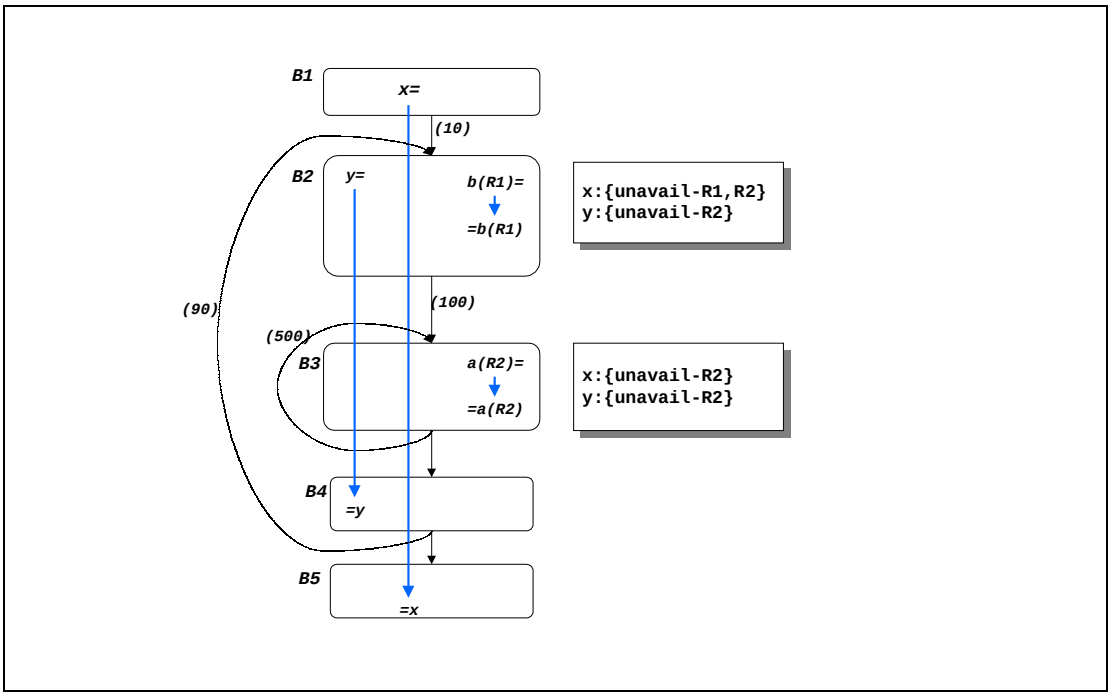


Figure 5.7: Propagation of Unavailable Registers

more expensive shuffle code is required in a higher frequency point as in the edge $E(B2, B3)$.

In our approach, when we delay the binding, the forbidden register information is propagated as *unavailable registers* from a higher frequency region to a lower frequency region. In the previous figure, the register $R2$ is propagated as an unavailable register for x and y from $B3$ and $B2$. When the compilation unit is moved to $B1$ and register allocation scope is in $B1$, both $R1$ and $R2$ will be unavailable for x. If the variable x is still bound to $R1$ or $R2$ (*i.e. because there is high register pressure*), x can be spilled in both $B2$ and $B3$ by unavailable register information.

While this algorithm works well in most cases, it may have drawbacks in some instances as shown in Figure 5.8. Unavailable register information is propagated from region $B3$ (and $B6$) to $B2$, and it is assumed that x is forced to be bound to $R1$ because of register pressure. Since $R1$ is unavailable in $B2$, it may be spilled in region $B2$, $B3$, $B4$, and $B6$ and shuffle code is inserted in $E(B1, B2)$ as before. But if we spill x in $B3$ only, the shuffle code is moved to the lower frequency edge $e1(B2, B3)$. We improve the register binding heuristic further for delayed live segment based on control-flow frequency. When we decide register binding for a pass-through live-range, we exclude unavailable registers only when its region connecting control flow frequency is larger than the frequency of the control flow

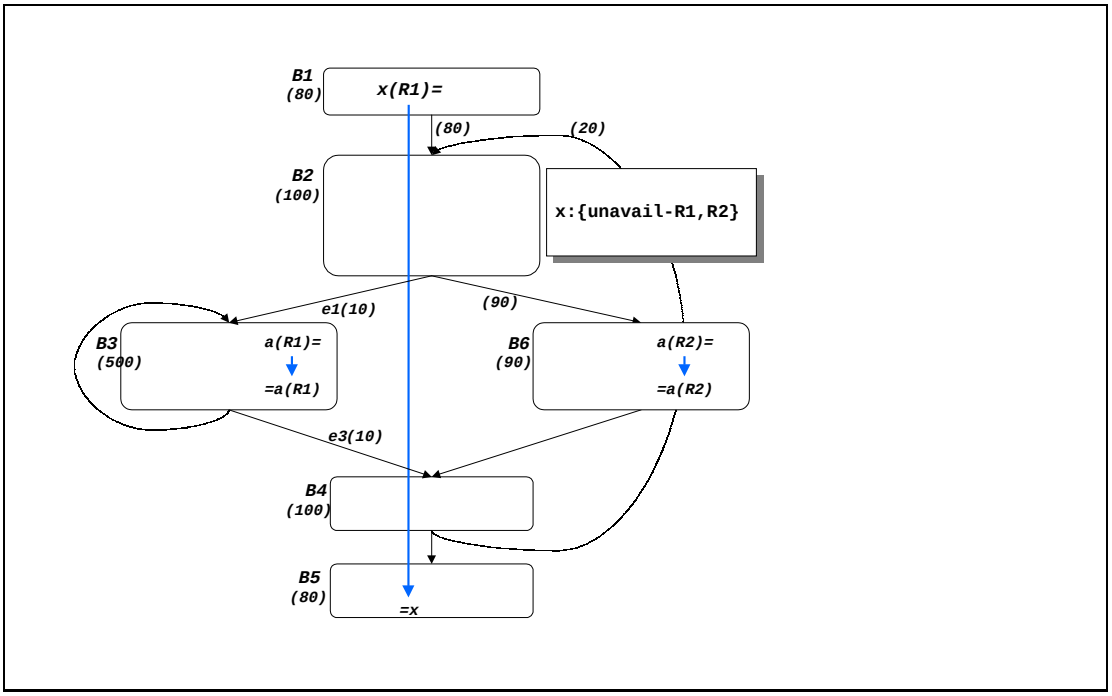


Figure 5.8: Frequency Based Propagation and Register Binding

from the propagating register. We can visit each neighboring live range by edge frequency until we find an available register while updating the current forbidden register list with the unavailable list from the neighbors. For example, in $B2$ in Figure 5.8, the variable x will include $R2$ as a forbidden register (from unavailable register in $B6$) when it finds a color $R1$ from $B1$, but $R1$ is not yet added to the forbidden register list, since control flow frequency of $(B2, B3)$ is smaller than that of $(B2, B6)$. So the variable x in $B2$ can be bound to register $R1$ as propagated from $B1$.

5.4 Clock Hand

In traditional coloring approaches for global register allocation, the registers should be selected to minimize the number of used registers. This approach makes more registers available for the live ranges with higher interference degree. In region-based register allocation with a large number of registers (the likely case with EPIC architectures), it is important to do a high level *global* partitioning of available registers into sets of registers for coloring in each partition.

As in Figure 5.9, consider 3 regions ($B1$, $B2$ and $B3$). Assume that there is a live-range for the variable x that straddles $B1$ and $B2$ while the live range for y and the live range for z straddle $B2$ and $B3$. Assume also that x and y do not interfere in $B2$ but z does as illustrated in Figure 5.9 (a). If we process $B1$ and then $B3$ by their assumed frequency order and if we happen to bind physical register $R1$ for both the variable x and the variable y (even when we have ample registers), we need two registers to color all the live ranges in this example. However, if we apply same approach to the live ranges in Figure 5.9 (b), either variable y or z cannot be colored to the register $R1$ and shuffle code becomes necessary at one set of edges out of $R2$ either for x or y. Note that such "gratuitous" shuffle code may be necessary not only with one region in between $B1$ and $B3$ as just considered but also with many regions in between. Hence there is a need for intelligent precoloring so that "gratuitous" shuffle code does

not result even when plentiful registers are present. The basic problem is that each region may independently allocate registers from the same "subset" of registers even when other "subsets" are available.

Graph-theoretic ideas that have been used in the past for register allocation recursively on subgraphs are clique separators[45] and approximation through multi-commodity flows[29]. In the first approach, the maximum number of cliques chosen as separators decides the regions such a way that a live-range can occur in cliques. In the latter approach, the algorithm finds a minimum balanced directed cut of directed acyclic graph(DAG) to split DAG into roughly equal pieces $G1$ and $G2$ such that a minimum number of live ranges cross the cut and so that there exist edges from $G1$ to $G2$ but not from $G2$ to $G1$. We can now process register allocation or scheduling $G1$ before $G2$.

However, in our case, what we have is a partition into regions based on frequency information which is unlikely to be the same as the partitioning obtained in the graph-theoretic approaches. In addition, such approaches can add to compilation time. Due to the above difficulties, one simple approach we have investigated is the use of a hash function based on variable name and register size to locate a free register. Another approach that we investigate is the *clockhand* algorithm[1]. We maintain a "freelist" of registers with a clockhand. For each region, first com-

[1]term borrowed from OS scheduling where the processing of certain lists is started from where last terminated

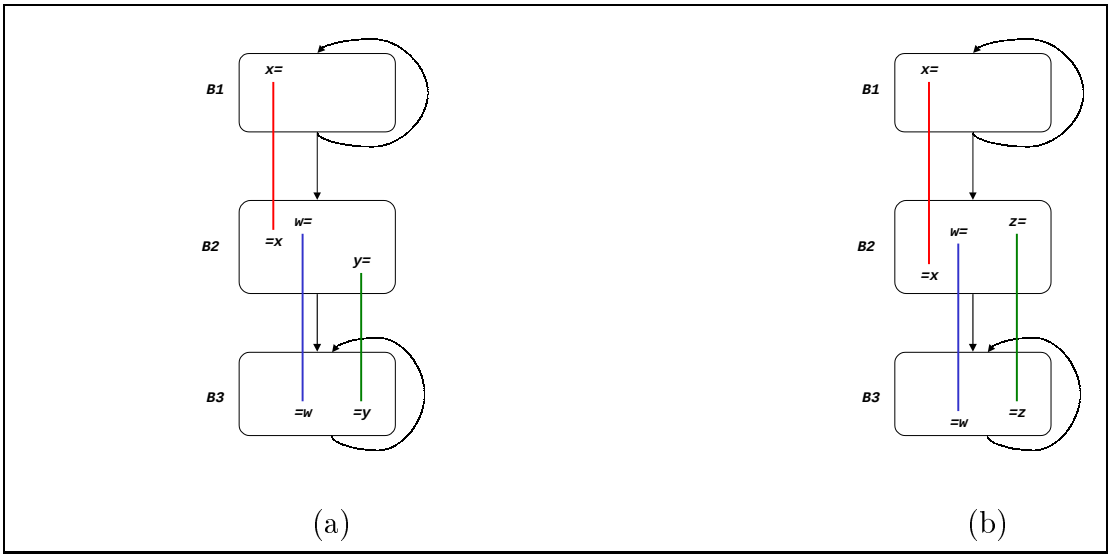


Figure 5.9: Register Selection Problem for Region. (a) x and z do not interfere in $R2$ (b) x and z interfere in $R2$

pute its clockhand (*i.e.* the starting point in the freelist from where registers get allocated) by choosing the best non-interfering clockhand of all neighboring already allocated regions. At the end of register allocation for this region, we store the first clockhand used and the last register allocation to enable clockhand computation by other nearby regions at a later time.

Our experiment shows limited performance improvement when there is high register pressure. "Clockhand" algorithm may not have positive impact in the case of Figure 5.9 (a) where the required number of registers for coloring increases to three. As register file size increases, the clockhand algorithm has positive performance changes in most cases as we will show in the experiment section.

5.5 Experiments

Propagation plays an important roll in region-based register allocation. Without proper consideration of register matching, the shuffle codes between region boundaries is tremendous and makes a region-based register allocation impractical to use. We compare our approaches to traditional propagation approaches used in Multiflow. Figure 5.10 shows the performance gains made by our frequency based propagation method. As seen in figure 5.10, some benchmarks show large performance improvement – this occurs in cases when the spill codes are placed in the outer loop instead of the inner loop. As register size increase, the performance improvement increases up to a certain point and then starts to decrease as shown in **008.espresso**, **072.sc**, **129.compress**. This is because:

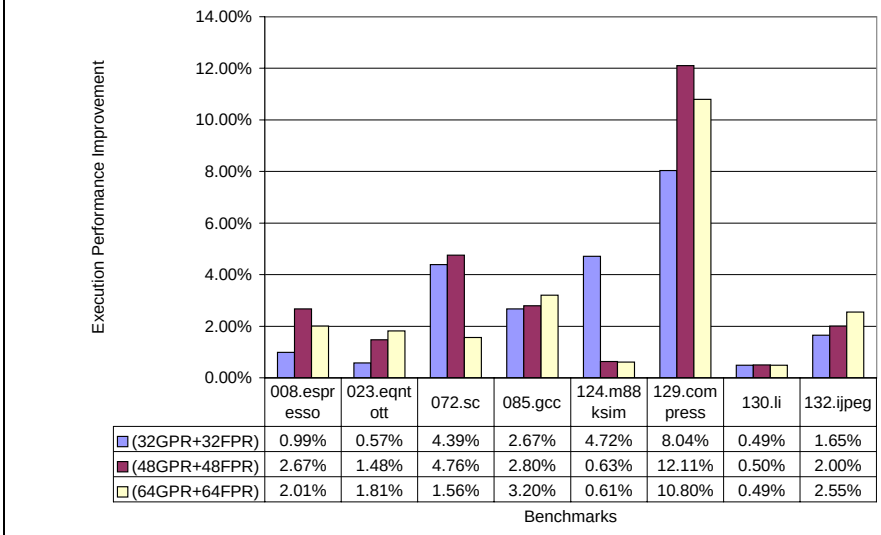
1. The control flow does not change with register size, so the register propagation behavior does not usually change.

2. Under high register pressure (with smaller register size), many live segments are spilled and propagation is not critical.

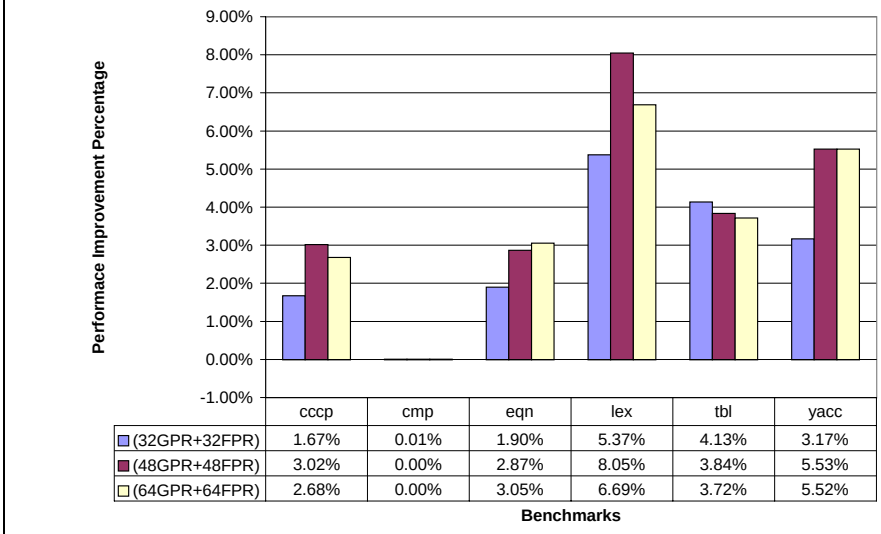
3. As register size increases, some live ranges which were spilled with smaller register sets may now be allocated to registers. Propagation scheme is important to reduce register color mismatch and shuffle code. However, if there are enough registers, each live range can have its own color and even

a simple propagation scheme is enough to reduce the shuffle code.

Performance improvement obtained from the clockhand algorithm is summarized in Figure 5.11. While clockhand algorithm shows performance improvement in most of our benchmarks, it may have some negative impact as in **023.eqntott**. This is because each region is competing to find register when there is high register pressure. If there are more registers available and the register pressure decreases, our approach has better performance improvement in every benchmark by distributing registers for each live ranges as we illustrated in the Figure 5.11.



	008.espresso	023.eqntott	072.sc	085.gcc	124.m88ksim	129.compress	130.li	132.ijpeg
(32GPR+32FPR)	0.99%	0.57%	4.39%	2.67%	4.72%	8.04%	0.49%	1.65%
(48GPR+48FPR)	2.67%	1.48%	4.76%	2.80%	0.63%	12.11%	0.50%	2.00%
(64GPR+64FPR)	2.01%	1.81%	1.56%	3.20%	0.61%	10.80%	0.49%	2.55%



	cccp	cmp	eqn	lex	tbl	yacc
(32GPR+32FPR)	1.67%	0.01%	1.90%	5.37%	4.13%	3.17%
(48GPR+48FPR)	3.02%	0.00%	2.87%	8.05%	3.84%	5.53%
(64GPR+64FPR)	2.68%	0.00%	3.05%	6.69%	3.72%	5.52%

Figure 5.10: Frequency based propagation performance changes

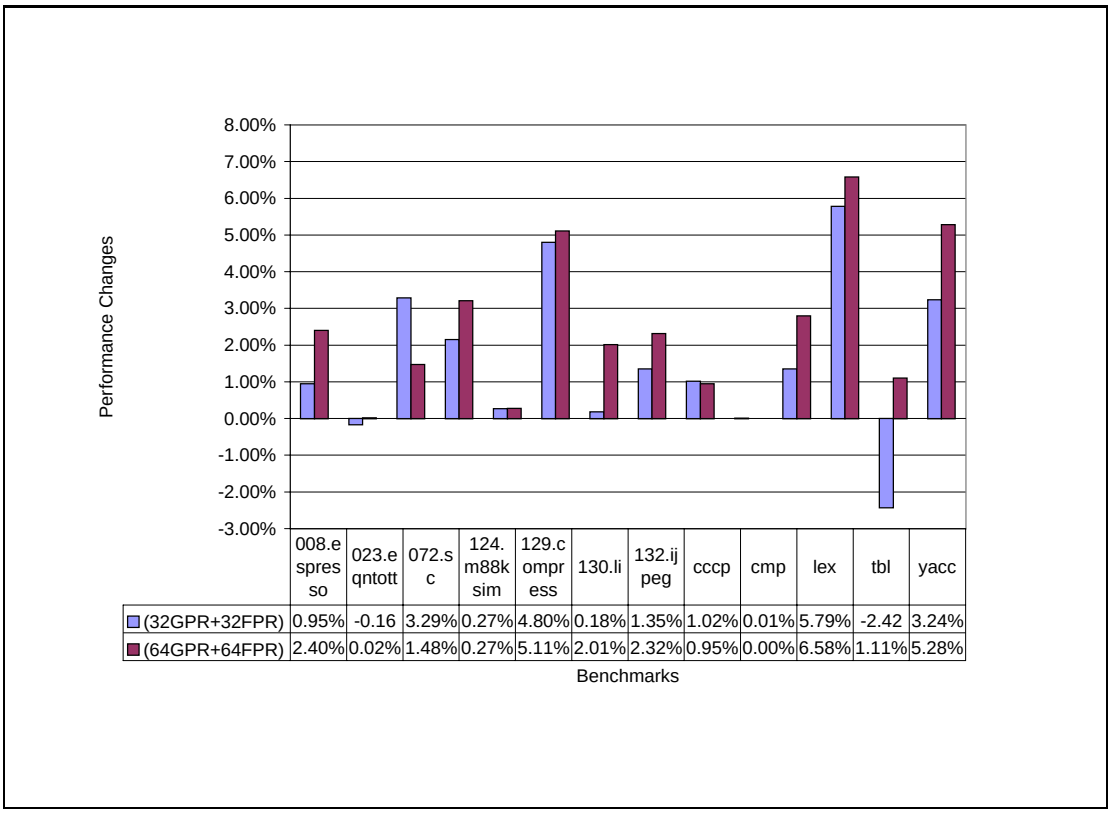


Figure 5.11: Performance comparison: clockhand algorithm

Chapter 6

Priority Function

The order of coloring is an important factor in register allocation since it determines the variables to be spilled, thereby affecting the amount of spill code. The priority function, proposed by Chow and Hennessy [11], is based on the amount of spill code that can be saved by allocating a variable to a register as opposed to spilling the live range of the variable (*i.e.*, storing variable in memory).

Many revisions to the traditional method of priority function are necessary to accommodate region-based register allocation, especially, with predicated code. In this chapter we propose several new approaches to improve priority functions in the scope of register allocation for regions and predicate analysis.

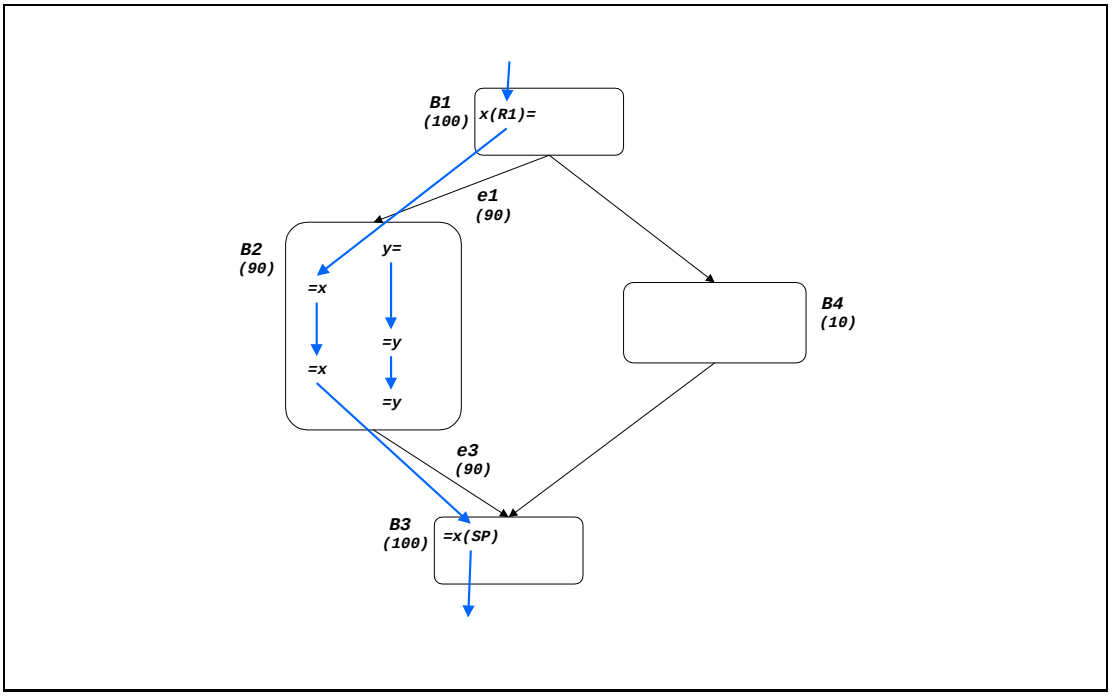


Figure 6.1: Frequency Based Propagation and Register Binding

6.1 Priority function for region-based compilation

For a given live range lr in blocks B_1 to B_i, the priority function can be defined as in equation 3.3. In Figure 6.1, the global priority $PR(x)$ and $PR(y)$ for variable x and y by given Chow's priority function can be defined as

$$PR(x) = STORE\_COST * 100 + (LOAD\_COST * 2) * 90 + LOAD\_COST * 100$$

$$PR(y) = STORE\_COST * 90 + (LOAD\_COST * 2) * 90$$

based on previous function.

If our compiler is region-based and if the region $B2$ is the current scope of register allocation, the priority of x and y in region $B2$ can be derived as below by applying the same function to this region scope;

$$\begin{aligned} PR(x_{B_2}) &= (LOAD\_COST * 2) * 90 \\ PR(y_{B_2}) &= STORE\_COST * 90 + (LOAD\_COST * 90) * 2 \end{aligned}$$

where y ends up having higher priority:

This priority function may not adequately model the register allocation benefit when a live range is a segment of a bigger live range that encompasses many regions. By the region priority above, the variable y has higher priority than the variable x, and x will be spilled in the region $B2$ if there is only one register available. If the variable x is bound to a register in region $B1$ and $B3$, we need a STORE code for the variable x on the entry point and a LOAD code on the exit point of the region $B2$. Therefore, the actual amount we can save by region register allocation may be different according to the register bindings of its neighboring regions. The priority function should be extended to consider these shuffle costs.

One simple approach is to consider every entry point and exit point as implicit definition and use point respectively, and to derive the priority function. However,

in a case when x is spilled in $B1$ or $B3$ and x is also spilled in $B2$, we do not need any shuffle code and there is no extra cost that needs to be considered for spilling in priority function.

We modified Chow's priority function to include the propagation information and edge frequency of live-in/live-out value of a live range. Suppose $B_i(x)$ is the live range of the variable x in the region B_i, $freq(B_i, B_k)$ is the edge frequency between region B_i and B_k, and $freq(B_i)$ is the region weight of B_i. $N_{in}(B_i(x))$ is the set of live segments which precedes $B_i(x)$ and is bound to a register and $N_{out}(B_i(x))$ is the set of live segments which succeeds $B_i(x)$ and is also bound to a physical register. For a given variable x in region B_i, we define our priority function $PR(B_i(x))$ as follows.

$$\begin{aligned} N_{in}(B_i(x)) \;=\; & \{B_j(x) \mid (freq(B_j) > freq(B_i)) \\ & \wedge (B_j(x) \text{ is in the preceding live range of } B_i(x)) \\ & \wedge (B_j(x) \text{ is bound to a register})\} \end{aligned}$$

$$\begin{aligned} N_{out}(B_i(x)) \;=\; & \{B_j(x) \mid (freq(B_j) > freq(B_i)) \\ & \wedge (B_i(x) \text{ is in the preceding live range of } B_j(x)) \\ & \wedge (B_j(x) \text{ is bound to a register})\} \end{aligned}$$

$$
\begin{aligned}
PR(B_i(x)) \;=\; & STORE\_COST * D_{B_i} + LOAD\_COST * U_{B_i} \\
& + \sum_{B_j(x) \in N_{in}(B_i(x))} STORE\_COST * freq(B_j, B_i) \\
& + \sum_{B_k(x) \in N_{out}(B_i(x))} LOAD\_COST * freq(B_i, B_k)
\end{aligned}
$$

The priority of $B_i(x)$ is defined in terms of the number of STORE/LOAD operations that can be saved by binding x to a register rather than spilling as well as shuffle cost savings. If the preceding live segment of $B_j(x)$ is bound to a register, we need a STORE operation on the edge $E(B_j, B_i)$ between the region B_j and B_i when we spill $B_i(x)$. Likewise, we need a LOAD operation on the edge $E'(B_i, B_j)$ between B_i and B_j, when

- $B_i(x)$ is spilled,
- $B_j(x)$ is the successor live segment of $B_i(x)$, and
- $B_j(x)$ is bound to register.

6.2 Priority function with predicate analysis

The use of hyper-blocks to group together traces that are executed frequently (for more effective optimizations, including register allocation) provides challenges to register allocation. The function defined in the previous section captures the priority quite well in basic blocks in which all operations in a basic block have the

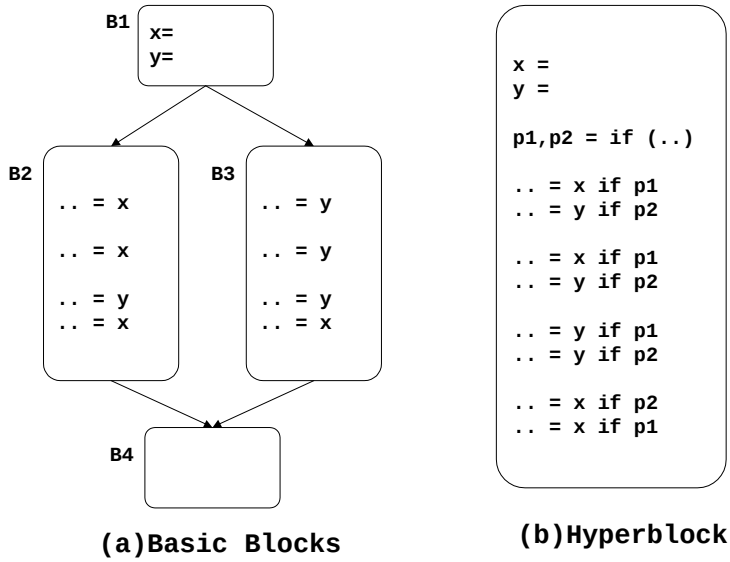


Figure 6.2: Example for priority function with predicate analysis

same weight. As it is explained in the introduction, the ILP compiler uses new styles of blocks like super-blocks or hyper-blocks frequently to explore the higher level of ILP. These regions may have multiple exit points and not every operation in a single region has same frequency. So the priority function designed in 3.3 or 6.1 is not suitable. Especially in a hyper-block, where some operations can be nullified due to predication, these functions do not reflect predicated execution well. In EPIC architectures, the nullified operations are also scheduled by compilers and execution cycles will be spent anyway. But in memory hierarchy model, these operations can have huge performance difference by frequent cache misses. In Figure 6.2, the weights of $B2$ and $B3$ are 90 and 10 respectively. By using the functions in the previous section, $LiveRange(x)$ has a higher priority than $LiveRange(y)$ has in $B2$. In the corresponding hyper-block, as in Figure 6.2 (b), the priority of $LiveRange(x)$ and $LiveRange(y)$ is the same, even if their execution frequency is different. To correct this problem, we have enhanced the priority function further to reflect frequency information using the predicate expression:

$$Priority(lr) = \sum_{lr \in HB_1..HB_n} (D_i * ST\_COST * PR(D_i) + U_i * LD\_COST * PR(U_i)) * w_i$$

where $PR(x)$ is the fraction of the time the variable access actually occurs due to predication.

6.3 Dynamic Priority Function

According to the approach we have used so far, the priority function is primarily governed by the total register savings, and determined by the frequency and the store or load operations that can be saved by register binding. But it is worth noting that there are subtle differences in benefit even though two live ranges have the exact same number of uses and definitions.

In Figure 6.3, the live range of variable w, x, y and z are assumed to have the same benefit for register binding. But if there is only one register available, register assignment for live range of w forces the compiler to spill the other three live ranges while spilling of live range w will make it possible for x, y and z to bind to a register. A live range occupying a larger region of code takes up more register resource if allocated in the register.

Chow noticed that the bigger a live range, the more conflicting live ranges cannot use the same register as the live range. That is, higher register pressure is introduced if bigger live ranges reside in registers. Based on this observation, Chow's priority function is normalized by the size of the region (number of basic blocks in a live range), which is approximated as the number of live units, N, in

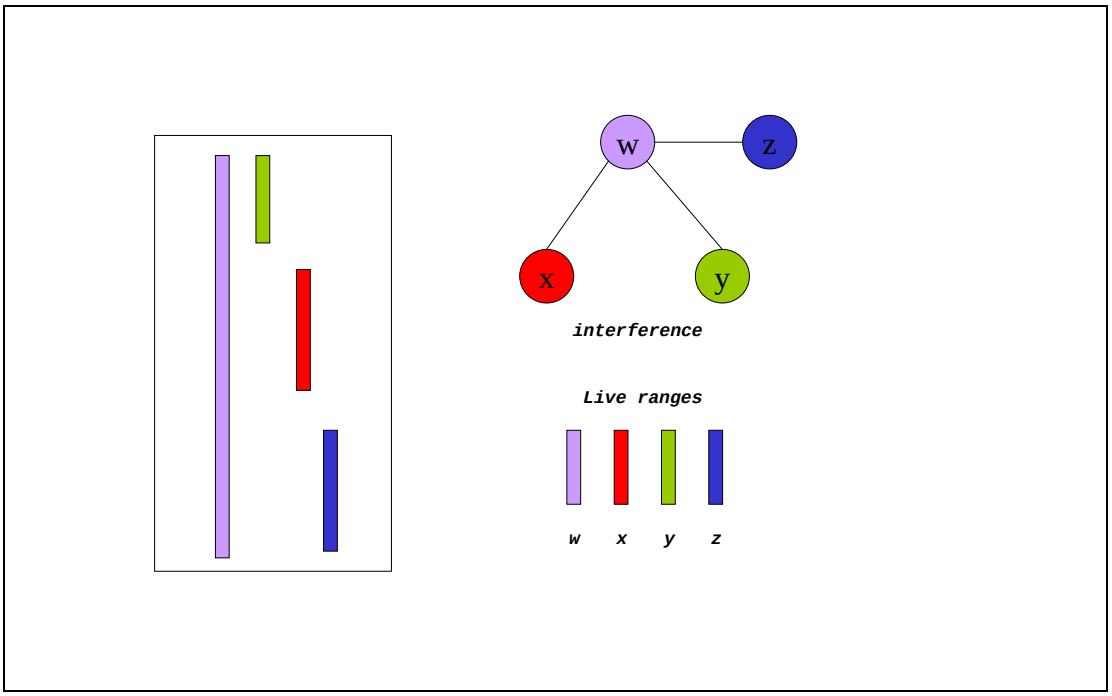


Figure 6.3: Example of Live Range and Interference Graph

the range. So smaller live range with the same amount of total savings will have higher priority. Thus the actual priority we are using is redefined as

$$P(lr) = \frac{S(lr)}{N}$$

as shown in the equation 3.3.

Even though the size of the live unit provides good heuristics for normalization, large size live range does not mean necessarily more register resource consumption. The actual factor for register resource consumption is decided by the degree of the node in the interference graph. The spilling heuristic of Chaitin-style coloring [8] and optimistic coloring [3] uses $\frac{S(lr)}{degree(lr)}$ where $degree(lr)$ is defined as the degree of the interference graph of the node for variable lr.

Later work by Bernstein *et al.* explores other spill choice functions. They present three alternative functions:

$$P(lr) = \frac{S(lr)}{degree(lr)^2} \tag{6.1}$$

$$P(lr) = \frac{S(lr)}{degree(lr)area(lr)} \tag{6.2}$$

$$P(lr) = \frac{S(lr)}{degree(lr)^2 area(lr)} \tag{6.3}$$

In the above equations, $area_n$ represents an attempt to quantify the impact lr

has on live ranges throughput the routine:

$$area(lr) = \sum_{i \in Q(lr)} (5^{depth_i} * width_i) \tag{6.4}$$

where $Q(lr)$ is a set of instructions in live range lr (i.e. instructions where a variable is live), $depth_i$ is the number of loops containing the instruction i, and $width_i$ is the number of live ranges live across the instruction i. The experiment of Bernstein *et al.* shows that no single spill priority function dominates the others. They propose using a "best of 3" technique. They repeat *simplify* three times, each time with a different spill metric, and choose the version giving the lowest spill cost.

The problem with the priority based approach is that the priority function never changes unless it is split through the coloring process. So it may fail to capture the dynamic behavior of the register binding benefit.

Consider Figure 6.4 where $vr1$ has the highest spill cost of 90 followed by $vr4$. If we have two registers available, the live ranges of $vr1$ and $vr4$ are bound to the physical register and total spill cost is 72 by spilling $vr2$ and $vr4$. Normalizing the benefit (spill cost) by interference degree does not help in this case, since $vr1$ and $vr4$ remain the same. If we allocate one register to $vr1$ and the other to $vr2$ and $vr3$ (please note $vr2$ and $vr3$ do not interfere, so they can be bound to the

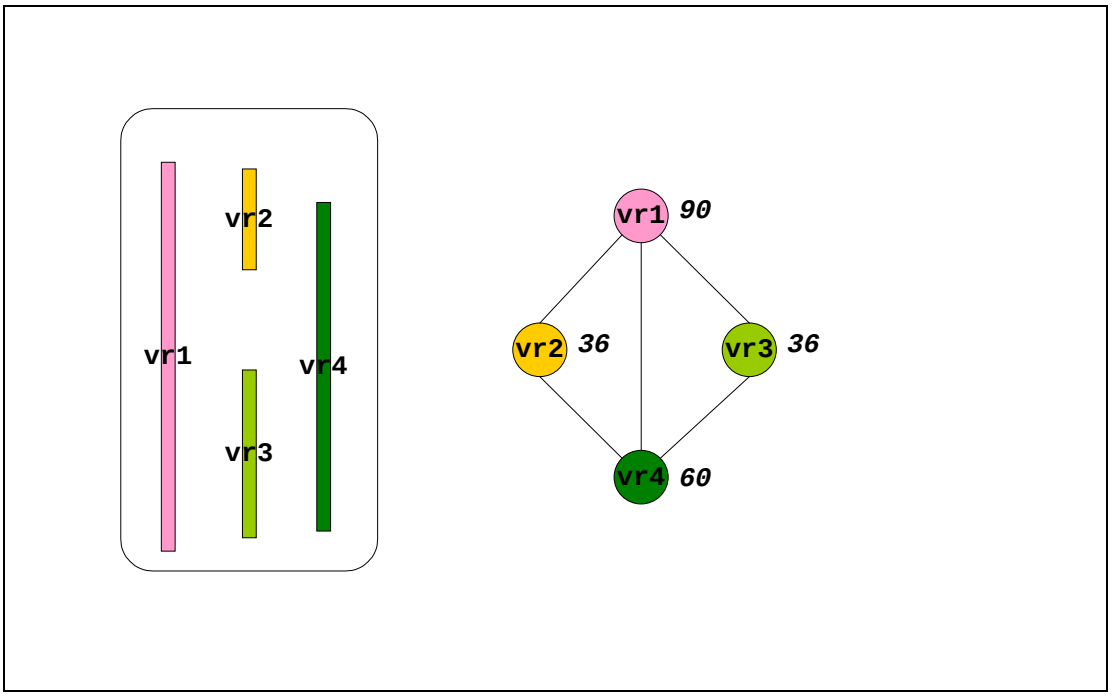
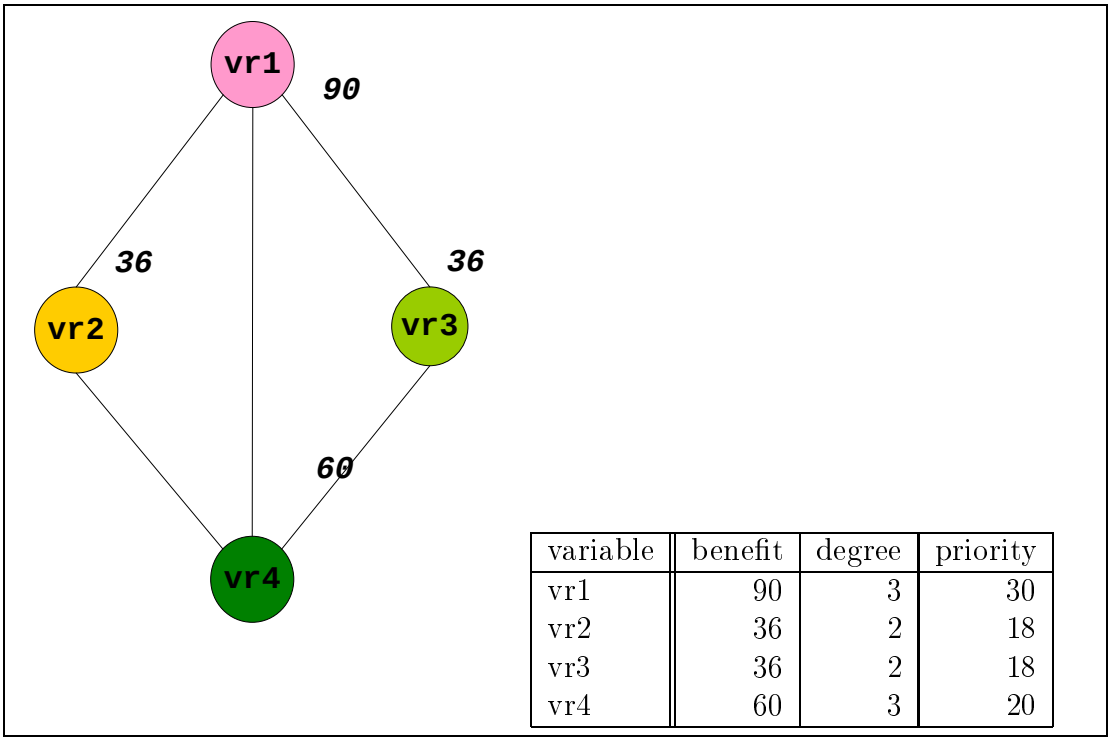


Figure 6.4: Live Range Interference and the Priority

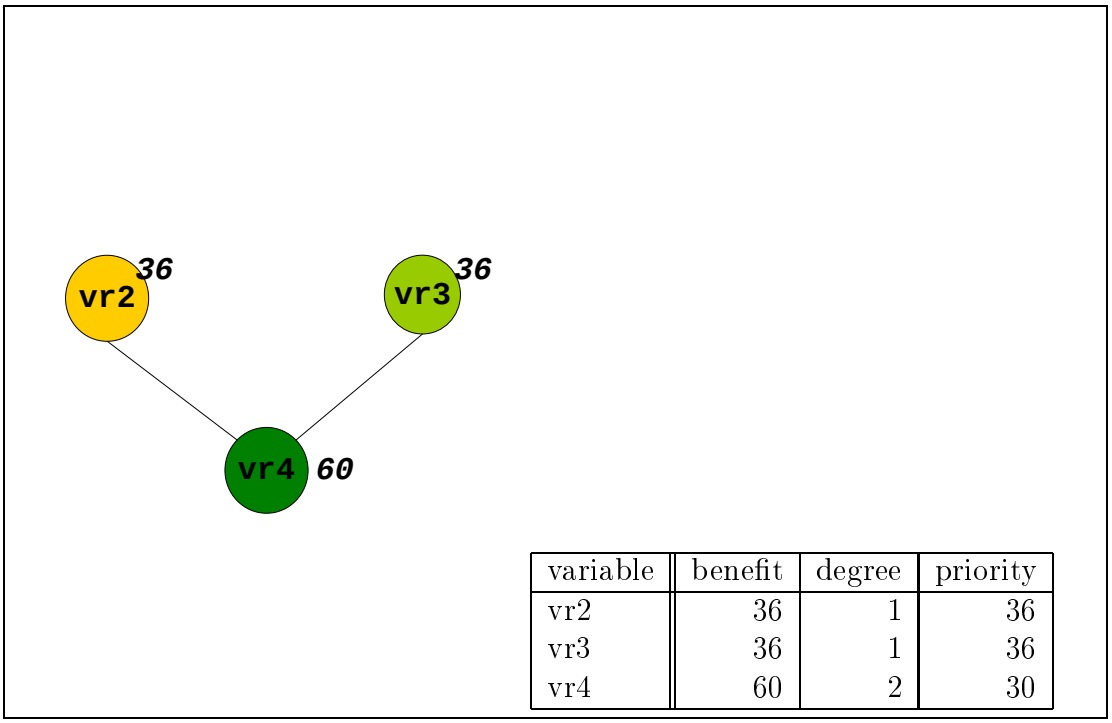


variable	benefit	degree	priority
vr1	90	3	30
vr2	36	2	18
vr3	36	2	18
vr4	60	3	20

Figure 6.5: Dynamic priority based Coloring. Interference graph and priority in initial stage

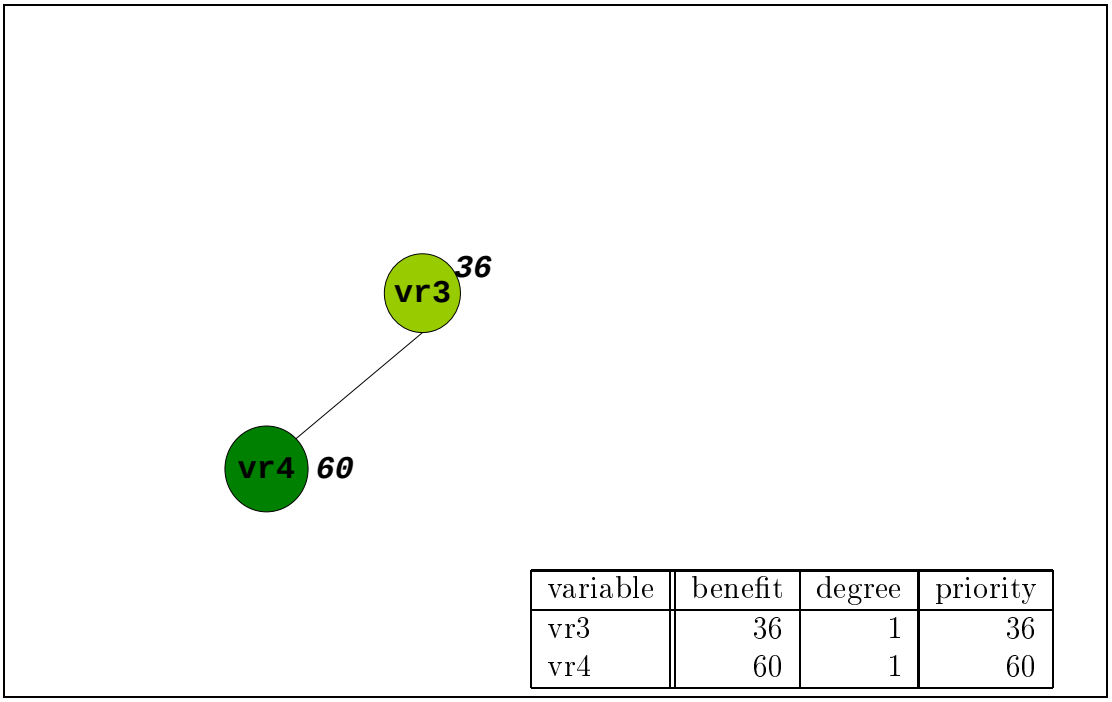
same physical register), the total spill cost is reduced to 60. In our approach, the interference degree of each live range is used as a normalization factor but the priority function is updated dynamically by the interference degree of *uncolored* live ranges as register allocation proceeds.

Figure 6.5 shows an example of our dynamic priority function and coloring based on the interference graph of Figure 6.4. It is assumed again that we have only two available registers. Initially, $vr1$ has the highest priority of 30 and will be allocated to a register. After $vr1$ is processed, every other live range's interference



variable	benefit	degree	priority
vr2	36	1	36
vr3	36	1	36
vr4	60	2	30

Figure 6.6: Dynamic priority based Coloring. Interference graph and priority after $vr1$ is colored



variable	benefit	degree	priority
vr3	36	1	36
vr4	60	1	60

Figure 6.7: Dynamic priority based Coloring. Interference graph and priority after $vr2$ is colored

degree is decreased by one and then $vr2$ or $vr3$ becomes the next highest priority node as in Figure 6.6. Figure 6.7 shows the priorities after $vr2$ is colored where the next highest priority node now becomes $vr4$ and there are no registers available for $vr4$ and $vr3$ is colorable.

6.4 Experiments

Figure 6.8, 6.9, and 6.10 summarize the results of our experiments in evaluating the performance of our priority function in comparison to Chow and Hennessy's priority function in region. For each chart, we denote the reduced number of execution cycles of the benchmarks based on Chow and Hennessy's approach. Priority function with propagation information shows positive performance improvement throughout all of our benchmarks. Except for **cccp**, most of the benchmarks have limited improvement. Our analysis shows this is closely related to the region style we used in our experiment. Most of the related sub-regions of the programs are integrated into one large hyper-block and the live-in or live-out variables across the regions have a small fraction of the control flow graph, therefore the effect of propagation is relatively small. When a basic-block is used as a unit of region, the dynamic priority function shows a much larger effect. Predicate-aware priority function also shows consistent performance improvement on a weighted averaged of 4.87% and the improvement may go up to about 10% as in **008.espresso**. Dy-

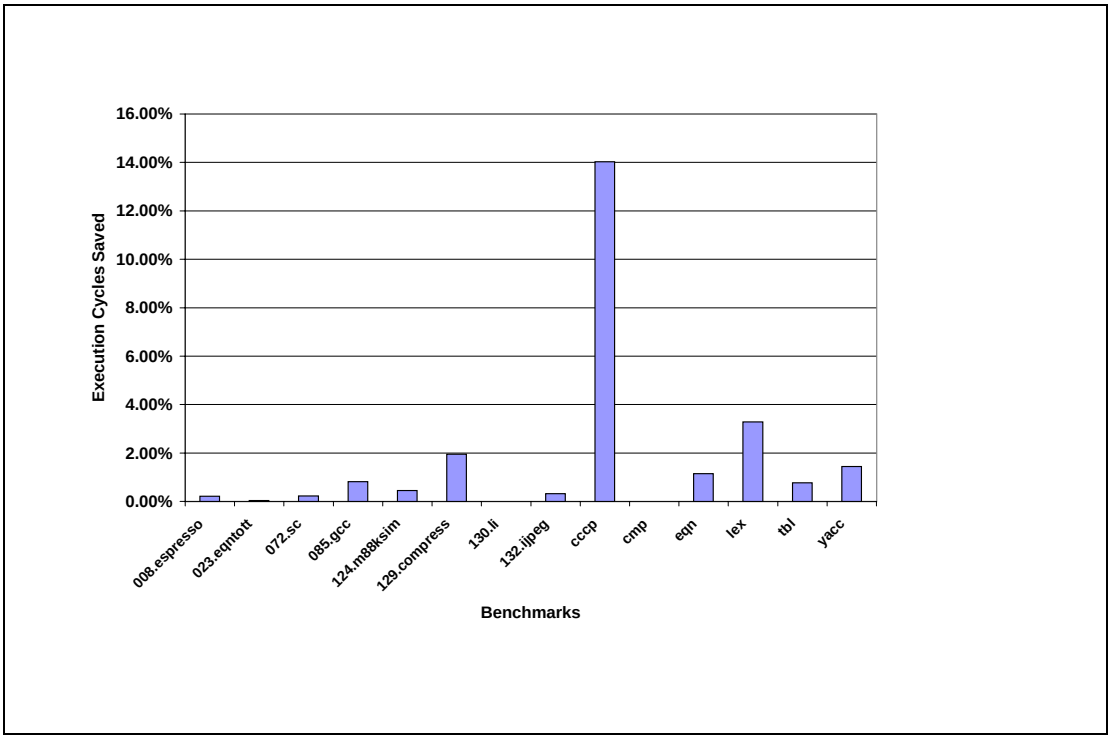


Figure 6.8: Performance improvement by region-based priority function with propagation

namic priority function shows positive performance improvement in most of our tests except **130.li** which seems to be affected by noise.

Figure 6.11 shows the performance changes caused by incremental addition of our algorithms; priority function with propagation (PROPA), dynamic priority function (DYN) and predicate-aware priority function (PRED). As we expected, each benchmark shows better performance with incremental enhancement. The analysis of the three previous charts shows that each different benchmark has its own improvement behavior. Commonly, dynamic priority function governs most of

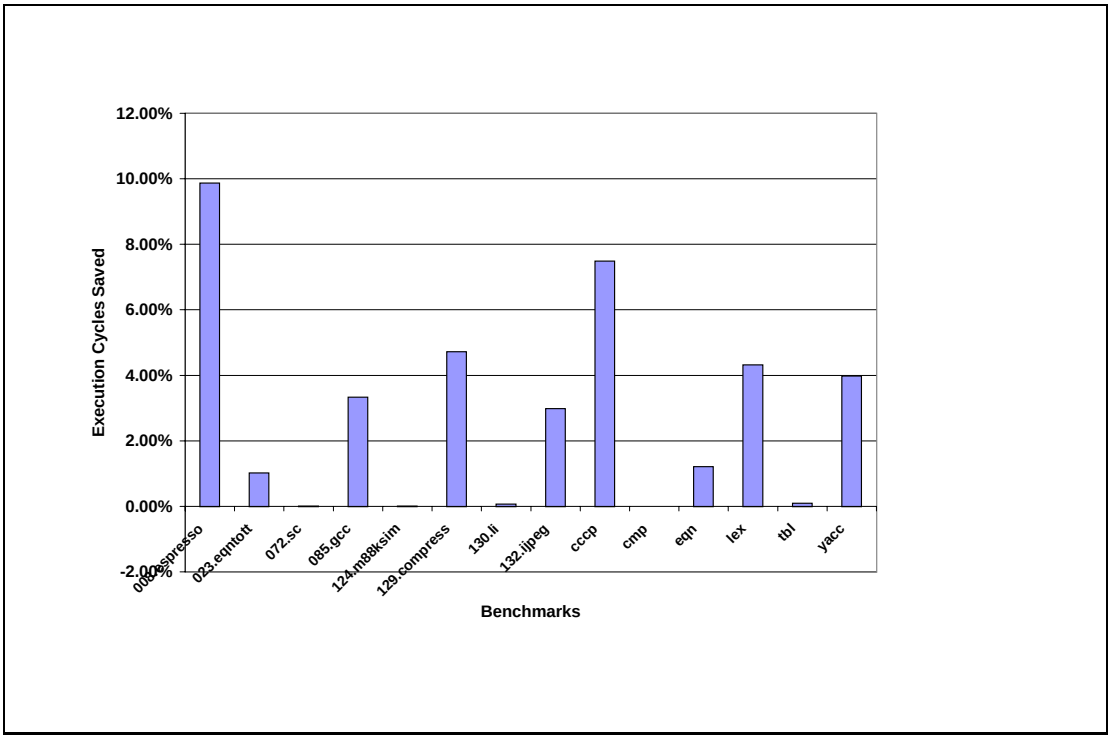


Figure 6.9: Performance Improvement by Predicate-aware Priority Function

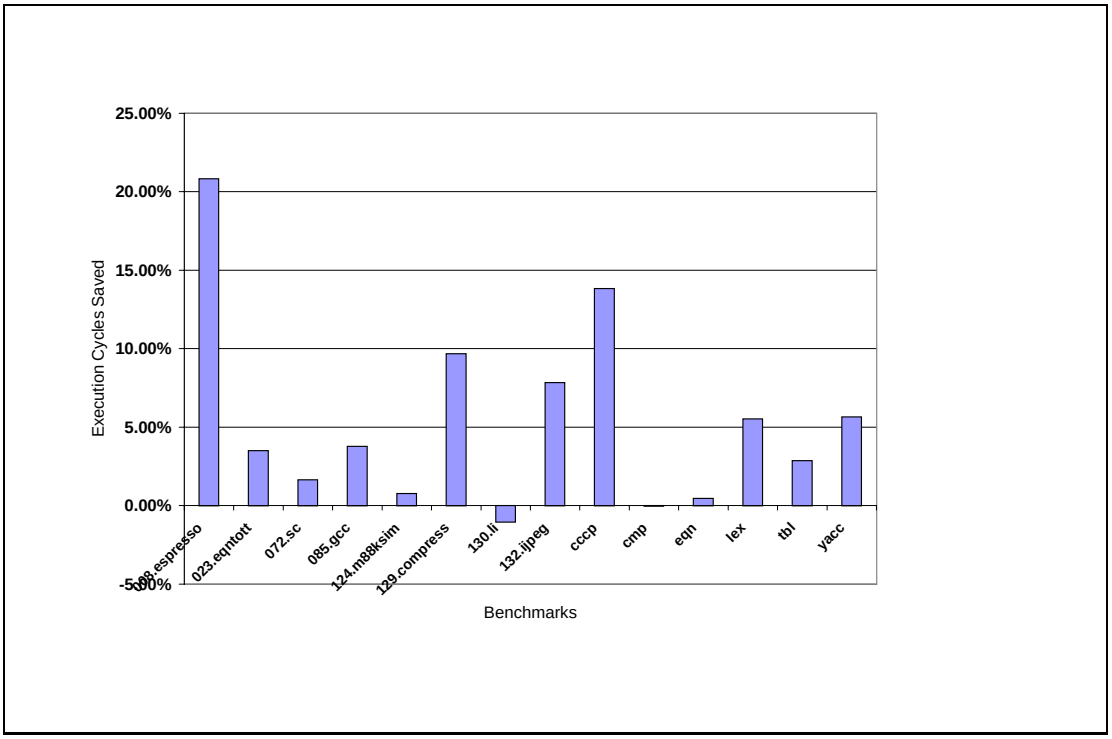


Figure 6.10: Performance Improvement by Dynamic Priority function

the performance improvement as in **008.espresso**, **023.eqntott**, **132,ijpeg** and **yacc** while priority function with propagation plays the important roll in **cccp** and **lex**. In **129.compress**, the performance improvement is equally distributed by dynamic priority function, priority function with propagation.

One of the important considerations in the region-based approach is savings in compile time. As we illustrate in the Table 6.1, it turned out that compilation time overhead for our priority function is minimal. In some cases, the complex priority function has smaller compile time than its counterpart because the time consumed in priority function can be compensated by smaller spill code insertion. More details about the compilation time comparison will be elaborated on a later chapter.

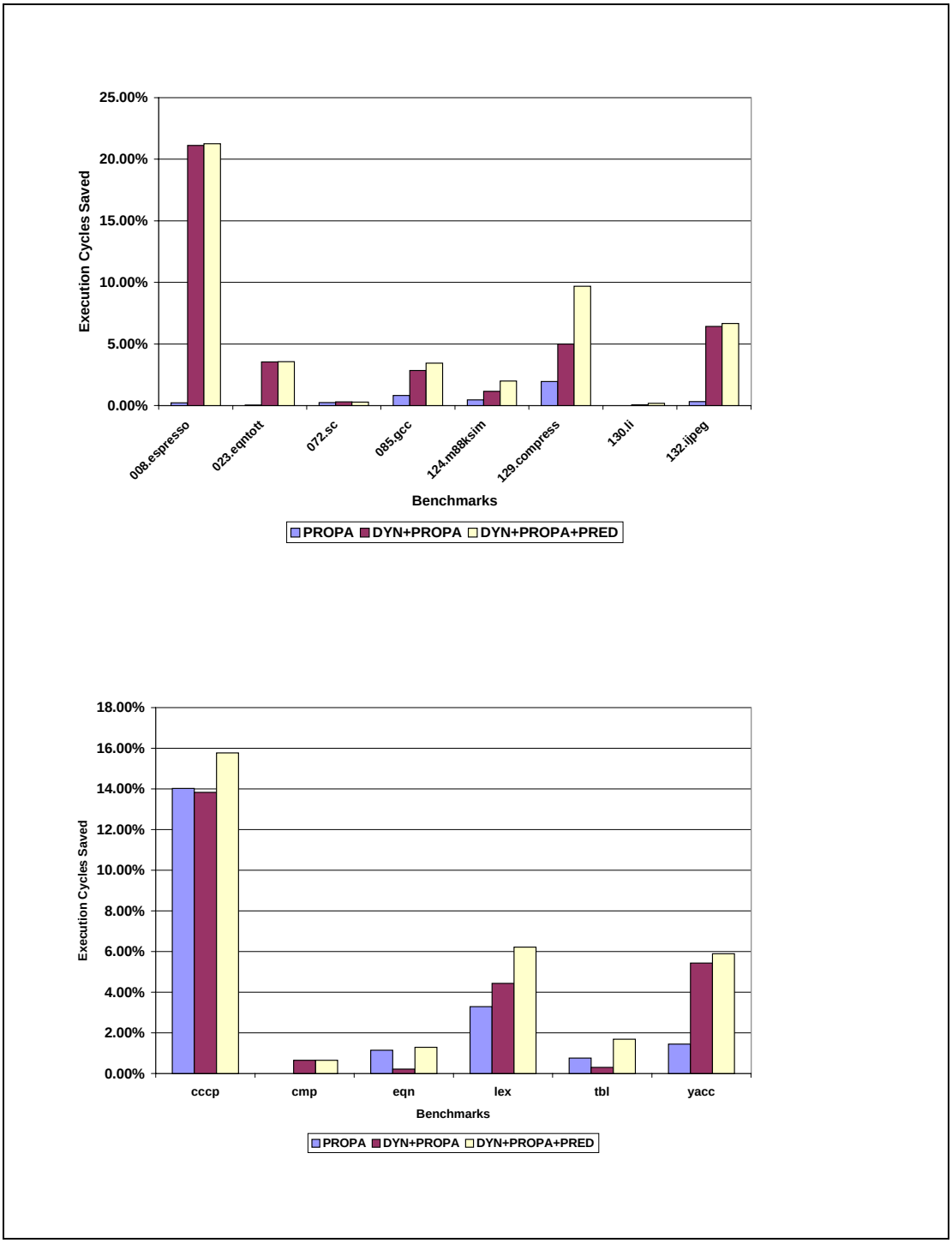


Figure 6.11: Performance Improvement by Dynamic Priority Function

Benchmark	Compile Time		Time Saved
	Chow	PROPA+PRED+DYN	
008.espresso	24439	23913	2.15%
023.eqntott	4325	4272	1.23%
072.sc	10125	10204	-0.78%
085.gcc	93111	93366	-0.27%
124.m88ksim	8430	8568	-1.64%
129.compress	2294	2305	-0.48%
130.li	1652	1689	-2.24%
132.ijpeg	26354	26497	-0.54%
cccp	19906	22038	-10.71%
cmp	86	88	-2.33%
eqn	2020	2089	-3.42%
lex	12699	12570	1.02%
tbl	6922	7192	-3.90%
yacc	14946	14466	3.21%

Table 6.1: The Compilation Time comparison of Chow's approach (Chow) and the combination of our priority function with changes

Chapter 7

Calling Convention

In register allocation without inter-procedure analysis, some registers (not all, but certain – namely callee-save registers) should be freed on procedure entry points and restored on procedure exit points. This cost should be considered in register allocation cost and it is called *call cost*. The call cost is influenced by the compiler's calling convention and many compilers divide the registers into two sets, *caller-save* and *callee-save* registers, respectively. When a live range is allocated with a caller-save register, we need a store and a load operation at every function call that is crossed by the live range. If a live range is allocated to a callee-save register in the function f, then this register must be saved and restored at the entry point and the exit point of a called function g to keep the semantics of the caller f as well as the function g. Figure 7.1 shows the example

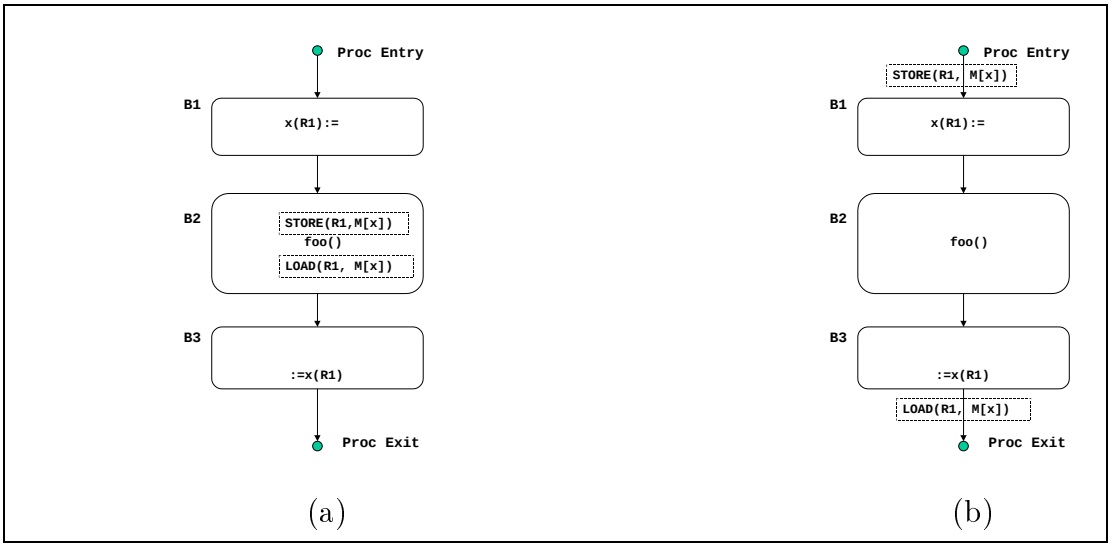


Figure 7.1: (a) Using caller-save register (b) Using callee-save register

of extra codes when variable x is allocated for caller-save and callee saved registers respectively. In Figure 7.1 (a), the live range of x is assumed to be allocated to a caller-save register. It is needed to insert STORE and LOAD operations right before and after the procedure calls as "foo()". If the live range of x is allocated to the callee-save register as in Figure 7.1 (b), STORE and LOAD operations are inserted at procedure entry and exit points.

The distinction between these registers provides the register allocation with more choices when we are minimizing the call overhead. In this chapter, we explore many issues related to distinguishing these two sets of registers, including priority function, live range splitting, and code insertion for the procedure calls. We also propose a code insertions scheme based on execution frequency.

7.1 Background

7.1.1 Priority Function with Calling Convention

Many of the aspects of register allocation in a coloring approach is affected by the calling convention. As we showed in Figure 7.1, allocating different type of registers to live ranges requires different store and reload costs, and this needs to be reflected in some phases of register allocation like priority function. The two register classes require different priority functions. For each live range, Chow and Hennessy use the caller-save priority function $P_r(lr)$ and the callee-save priority function $P_e(lr)$ for the live range of x. So the basic priority function we explained in equation 3.3 can be redefined as

$$Priority(lr) = \frac{max(P_r(lr), P_e(lr))}{size(lr)} \tag{7.1}$$

$$P_r(lr) = \sum_{lr\ in\ B_1..B_i} (D_i * ST\_COST + U_i * LD\_COST - C_i * (ST\_COST + LD\_COST)) * W_i \tag{7.2}$$

$$P_e(lr) = \sum_{lr\ in\ B_1..B_i} (D_i * ST\_COST + U_i * LD\_COST - (ST\_COST + LD\_COST)) * W_i \tag{7.3}$$

where D_i is the number of definitions, U_i is the number of uses, C_i is the number of procedure calls in block B_i and W_i is the weight of B_i.

In the case of caller-save priority function, the cost of store and reload of the registers around the function calls weighted by frequency is considered in the priority function. For the callee-save register, the extra cost occurs only once for each callee-save register at each procedure entry point. Thus only the first live range that uses a given callee-save register needs to account for these savings and restoring costs. Once these costs have been considered, the same callee-save register can be used to contain other live ranges for free. So the value of $P_e(lr)$ for a given live range has two alternatives. When a used callee-save register is available to the live range, $P_e(lr)$ does not include the extra costs in procedure entry and exit points.

7.1.1.1 Implications of region-based compilation for Callee/Caller Cost Models

Many standard assumptions need revision with region-based register allocation when addressing machines with caller-save registers and callee-save registers. We will first consider caller-save/callee-save cost models. caller-save registers do not pose major difficulties: the cost of store and load operations for a caller-save registers around function calls are absorbed by the live range in the region.

There are many ways to assign the cost of callee-save and restore to the live ranges that use a callee-save register. Let us first consider a region that is a

function. The first user of a callee-save register can bear both costs of save and reload, while subsequent users incur the same costs if they were allocated to a caller-save register[11]. Or, the first user of a callee-save register bears the save cost and the last user the reload cost, though this has not been reported in the literature. Lueh [32] reports another model that all users of a callee-save register bear the costs. When the color assignment phase finishes, his approach spills all live ranges that were bound to callee-save register e if

$$\sum_{lr \in \delta(e)} spill\_cost(lr) < callee\_cost(e)$$

where $\delta(e)$ is the set of live ranges that was bound to e. It has been reported that this model gives incremental performance benefits on some benchmarks compared to the first model, but has no perceptible difference in some benchmarks [31].

Now consider a region that is completely within a function. Since we process regions based on frequency information, one possibility is that the first live range that uses a callee-save register in a region with the highest weight in the procedure bears all the costs. If $spill\_cost(lr) > callee\_cost(e)$, then we can definitely allocate a callee-save register to the live range in the region (R). Another model is the

following: if

$$\sum_{lr \in \delta(e)_R} spill\_cost(lr) > callee\_cost(e)$$

$$\delta(e)_R = \{lr_i | lr_i \text{is the live range that were bound to } e \text{ in region } R\}$$

then we can allocate a callee-save register to the live range in the region R. Otherwise, we are in a difficult situation concerning with our current region-based vertical model of compilation, as we are not in a position to know about the live ranges in other regions (even adjoining ones in the same function) as they have not yet been processed. However, we use the heuristic that live ranges in regions with larger frequencies can absorb all the callee-save costs due to the larger number of accesses in them.

Next, consider a region across more than one procedure. The live ranges can now possibly be across functions. Allocating a caller-save register on one side of the function implicitly results in splitting the live range on a function call whereas allocating a callee-save register may or may not result in splitting the live range. It will not result in splitting the live range if the callee-save register is not used in the called function. It will result in splitting only if it is used in the called function, but the location of the splitting depends on whether *shrink wrapping* has been used. If shrink wrapping is used, the original live range is kept as long

as is feasible. Again, assigning callee-save costs to various live ranges in this case is even more difficult, due to the vertical model of compilation. However, it seems best to allocate callee-registers to live ranges that straddle functions when shrink wrapping is used.

One way out of these problems is to deviate slightly from the vertical model of compilation and first process all regions only to construct live ranges. In that case, we can compute the spill costs of all variables beforehand and use that to decide when to allocate callee registers by using the same formula as before, keeping in mind the function boundaries. Currently, our system can only handle basic blocks, super-blocks, and hyper-blocks; so the complex issues of live ranges across functions discussed above are not that important and we have ignored them.

Another model that needs revision is the allocation of caller-save registers to live ranges in leaf procedures. In function-based register allocation, caller-save registers are preferred for leaf procedures. As a heuristic, we have used that same concept for the region-based approach also: if there are no function calls in the region, it is a "leaf". Strictly, all of the regions have to be checked to see if a function is a leaf but because of the frequency-based order for processing the regions, live ranges in regions of high frequency with no function calls can be allocated caller registers and this preference is propagated in the reconcile part of the algorithm to other less frequently executed regions.

7.1.1.2 A Optimization Formulation

The caller-save/callee-save cost functions should be accurate enough that these can be negative also (*i.e.*, it is cheaper to spill than to bind to a register even if registers are available). The model can be local (considering the region alone), considering only immediate neighbors that have been processed or all connected regions that have been processed. In the cases of pass-through live ranges, the benefit is negative by the difference of weighted cost of spill code (=0) and caller-save/callee-save cost. For a caller register bound to a pass-through live range, if we assume local cost model, the benefit can be 0 when there are no function calls, or negative when function calls need STORE/LOAD bracketing operations. If immediate neighbors are considered, the reconciliation cost by shuffle code comes into the picture on the edges incident on the region. But they cannot be known in all cases due to the vertical model of compilation. Similarly, for a callee-save register, the benefit is store and load costs weighted by frequency in a local model for the first live range using the register. Later live ranges use it at no cost (similar to Chow and Hennessy's strategy as explained earlier) or all the live ranges in the region that use the register can share this cost.

Let the number of variables be n. Let $b(k, i, j), k \in \{ER, EE\}$, be the benefit of allocating a caller-save register (ER) or callee-save register (EE) to the i^{th} variable for its j^{th} segment of the live range. This is equal to the difference of

spill cost and caller-save/callee-save cost. Let $c(k,i,j)$ be the cost of shuffle code (weighted by frequency) for the j^{th} live range segment of the i^{th} variable in all the adjacent live regions. As each variable can be split during register allocation, let the number of split segments of a live range for a variable i be $L(i)$.

The callee-save cost model can be complex and a register allocation with weighted regions has to use heuristics. Let $Ncaller$, $Ncallee$ be the number of available caller-save and callee-save registers in the architecture. Let $x(k,i,j) = 1, k \in \{ER, EE\}$ if i^{th} variable's j^{th} segment is allocated to a caller-save/callee-save register. Otherwise, set it to 0. Then at each minimum unit of live range:

$$
\begin{aligned}
x(ER,i,j) + x(EE,i,j) <= 1 \qquad & \forall i \in \{1..n\}, j \in \{1..L(i)\} \\
\sum_{i:1..n} x(ER,i,j) <= Ncaller \qquad & \forall\ j \in \{1..L(i)\} \\
\sum_{i:1..n} x(EE,i,j) <= Ncallee \qquad & \forall\ j \in \{1..L(i)\} \\
x(k,i,j)\ \in \{0,1\} \qquad & \forall\ i \in \{1..n\}, j \in \{1..L(i)\}, k \in \{ER..EE\}
\end{aligned}
$$

The optimization problem is to maximize

$$\sum_{i:1..n, j:1..L(i), k \in ER,EE} x(k,i,j) * (b(k,i,j) - c(k,i,j))$$

while minimizing $L(i)$ for each i. First, $L(i)$ and $b(k,i,j)$ cannot be computed

straightforwardly as the priority depends on the spill costs, caller(callee) costs and the current set of shuffle costs but the act of splitting changes the costs and introduces new live ranges. A combinatorial or iterative method is needed for an approximate solution. Luckily, the priority function orders the live ranges and ensures that any costs are borne by less important live ranges processed later: this happens as the splitting comes into play only after allocation of all available registers to higher priority live ranges and some or all of the remaining interfering live ranges are split. Please remember than all pass-through live ranges have already been set aside as candidates for coloring by the design of the algorithm in core register allocation. Similarly, $L(i)$ is minimized for the important live ranges.

Second, as $c(i, k, ENDS)$, where $ENDS$ refers to segments of live ranges that are adjacent to nearby regions, cannot be determined unless other regions have also been processed by this time, we need again to use some combinatorial or iterative method for an exact or at least approximate solution. To simplify the problem, one can ignore $c(k, i, ENDS)$ by setting it to zero where it is not known, and then solve it. This has the effect of incorporating costs of reconcile code only in later regions; this means that live ranges in regions with bigger weights are given more flexibility in selecting the type of register needed and later live ranges in other regions are "requested" to use the same type of register.

With this heuristic, we can solve the above problem by ordering all $b(k, i, j) -$

$c(k,i,j)$ in descending order (by using the value of c where it is known or 0 otherwise) and pick first $Ncaller + Ncallee$ at each minimum unit of live range while taking care to include only one from caller-save/callee-save registers for each variable segment and picking only $Ncaller$ and $Ncallee$ registers. Other heuristics are possible: for example, compute the difference of caller-save/caller-save costs and priorities[33].

The actual priority function with predicate analysis we used in Chapter 1.2.3 is defined as,

$$Priority(lr) = max(P_r(lr), P_e(lr)) \tag{7.4}$$

$$P_r(lr) = \sum_{lr\ in\ B_1..B_i} (D_i * ST\_COST * PR(D_i) + U_i * LD\_COST * PR(U_i) - B_i * (ST\_COST + LD\_COST) * PR(B_i)) * w_i \tag{7.5}$$

$$P_e(lr) = \sum_{lr\ in\ B_1..B_i} (D_i * ST\_COST * PR(D_i) + U_i * LD\_COST * PR(U_i) - (ST\_COST + LD\_COST)) * w_i \tag{7.6}$$

7.1.2 Live Range Split by Calling Convention

For the live range which does not includes any function calls, Chow's algorithm uses up caller-save registers before it starts to use the callee-save ones. The live ranges with function calls have bigger callee-benefit than caller-benefit, and the

callee-save registers are much more in demand. By the time that all callee-save registers are used up, the remaining live-ranges can be bound to caller-save registers. But in many cases, caller-benefit of each of the remaining live-range is negative because there are too many calls within the live ranges. Even though the live ranges have positive caller-benefit, it is more useful to split the live range in some cases. In Figure 7.2 (a) shows an example of using caller-save register for the variable x when callee-save registers are used up but caller-save registers are available. We need a store operation (STORE) and a load operation (LOAD) before and after the function call respectively. If the live ranges of x are split as in Figure 7.2 (b), the upper segment can be bound to callee-save register $R2$ while the lower segment is spilled. We need only one STORE and one LOAD operations in this case.

It is not clear from Chow's literature what the condition of splitting is if no callee-save register are available. In our implementation, we use callee-save registers when caller-save registers are all used. But we aggressively split the live range when callee-save registers are all used, even if caller-save registers are available and its caller-benefit is positive.

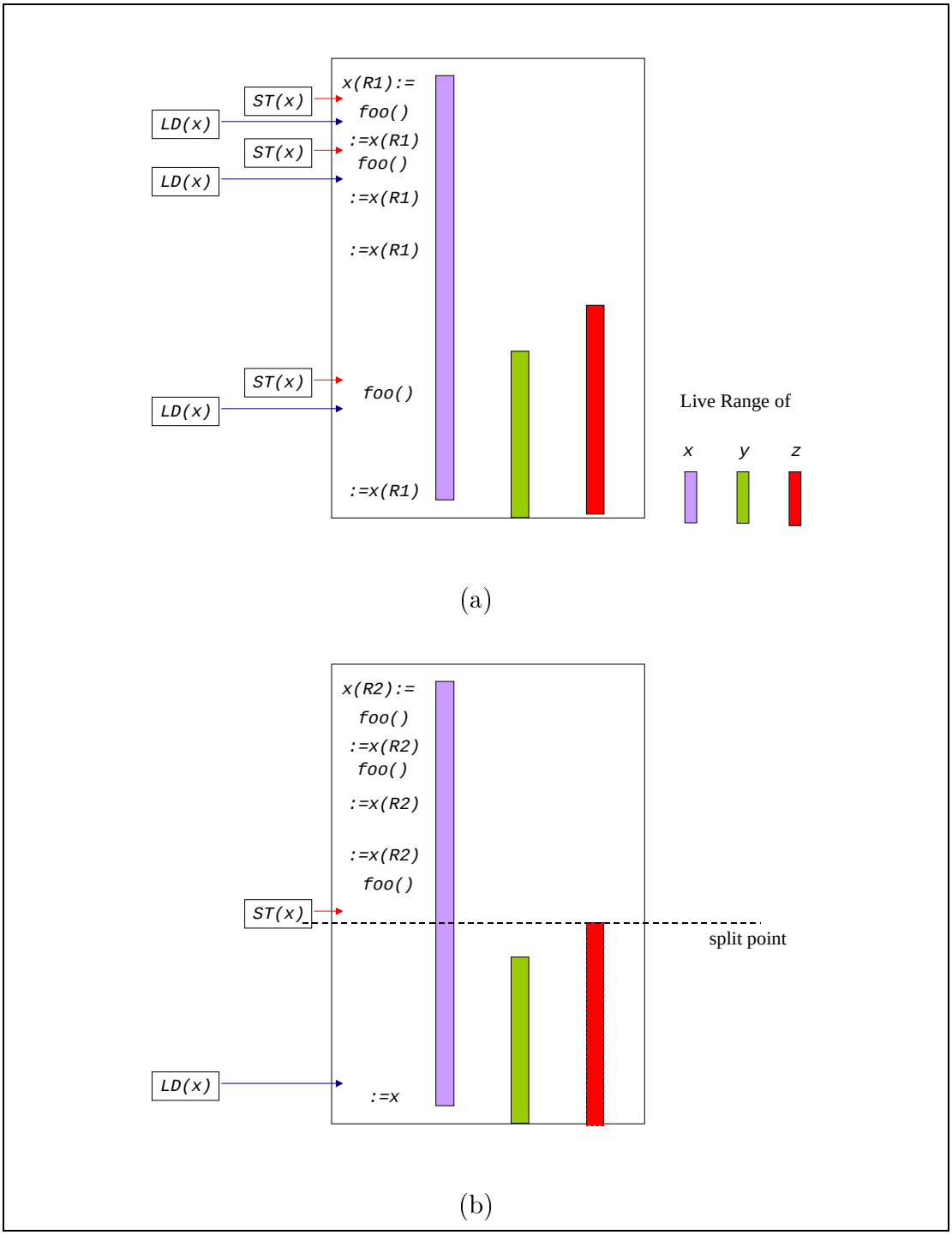


Figure 7.2: Aggressive Live Range Split by Calling Convention

7.2 Shrink Wrapping

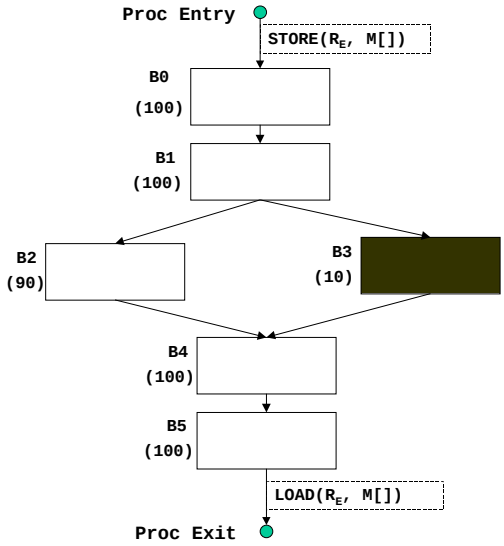
As we covered in the last section, the store and load operations for callee-save registers are done at the entry and exit points of the function. But the saved register may not be used for the execution path in some invocations of that function unlike a caller-save register where store and load always happens before and after the function call. Chow [34] tries to optimize the placement of the store and load for callee-save registers so that they occur only over regions where the registers are used. This optimization is called *shrink wrapping* and Figure 7.3 shows an example of how shrink wrapping can help to reduce the callee-save register overhead, where shaded block($B3$) denotes the use of a callee-save register R_E.

Chow's approach has two simple rules;

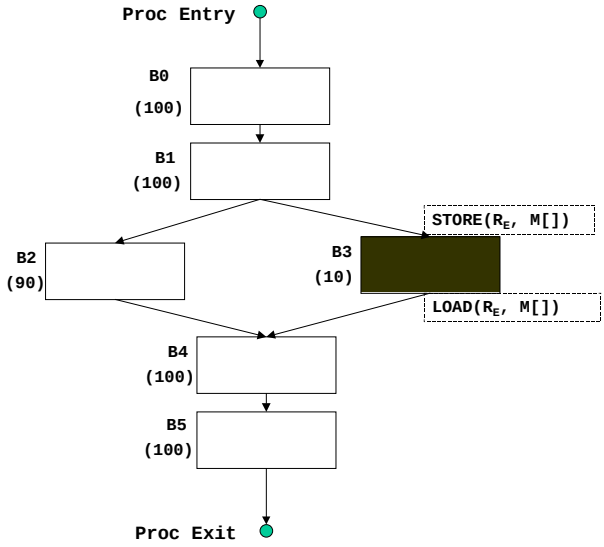
1. They always insert register save code at basic block entries, which will not limit the effects of their optimization.

2. The insertion should be at the earliest points in the program leading to one or more "contiguous" regions where the register is used.

In his experiment [34], his approach shows limited performance improvement with negative result in some cases. The negative impact in some cases may occur when each individual use of R_E is in the block B_i which has less frequency than the entry block, but the sum of every B_i is larger than the frequency of the procedure



(a) STORE and LOAD of the callee-save register R_E is inserted at the entry and the exit points of a function



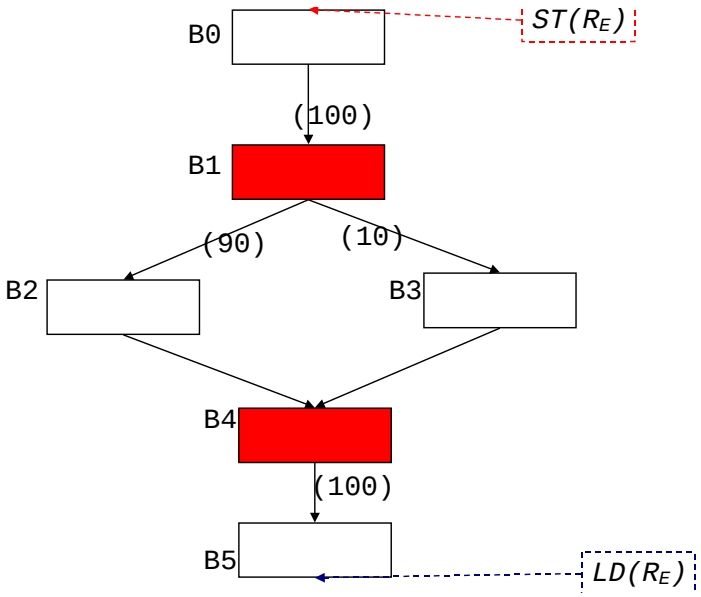
(b)STORE and LOAD of the callee-save register R_E is inserted at the entry and the exit points of a live range

Figure 7.3: The Benefit of Shrink Wrapping

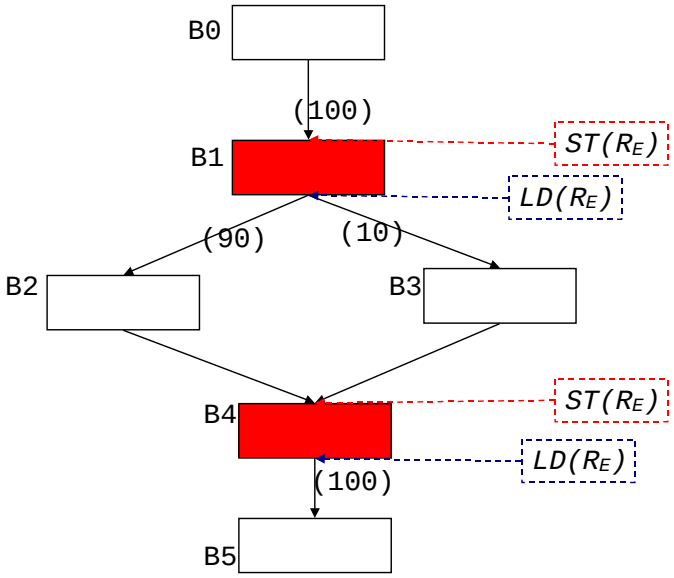
as illustrated in Figure 7.4. The variable x is live-in in the block $B1$ and the variable y is live-in the block $B4$, and both of these live ranges are bound to the same callee-save register. Without shrink wrapping we need store and reload at the entry and the exit of the procedure ($B0$ and $B5$ respectively) as in Figure 7.4 (a) and it is more efficient than the shrink wrapping by Chow's approach as we showed in Figure 7.4 (b).

In our approach, we extend Chow's *shrink wrapping* in our register allocation by selectively using shrink wrapping when the frequency of the live range assigned to callee-save register is less than the frequency of the procedure. Dominator trees and post-dominator trees are used for finding the best place to insert the callee-save register wrapper code. The algorithm consists of the following steps:

1. For each callee-save register, identify all the blocks B_i sharing the register.
2. Create the dominator tree for finding the store point (or post-dominator tree for finding the load point) from all B_i's with the weight of the block.
3. For each B_i in the dominator (or post-dominator) tree repeat
   (a) Select minimum weight node n along the path from the root s to B_i as point.
   (b) Ignore every child nodes of n.
   (c) Deduct the weight of n from the all nodes from s to n.



(a) The Effect of Shrink Wrapping by Chow's approach(a) store(ST) and load(LD) of the callee-save register R_E is inserted at the entry and the exit points of the function



(b) ST and LD of the callee-save register R_E is inserted at the entry and the exit point of the live range

Figure 7.4: Effect of Shrink Wrapping

Figure 7.5 and 7.6 show that how our algorithm works. The program control flow graph is shown in Figure 7.5 (a) and a callee-save register is assumed to be used in region $B4$, $B5$, $B6$, $B8$, and $B9$. Our approach eventually chooses region $B4$, $B5$, and $B3$ as insertion points for shrink wrapping store operations. A symmetric approach is performed to find the points for load operations and $B4$, $B5$, and Bb will be used in this example.

7.2.1 Shrink Wrapping on the Edges

The previous algorithm based on dominator and post-dominator trees works well in most cases and it always produces better quality code compared to register allocation without shrink wrapping. But more efficient code of shrink wrapping can be generated by inserting wrapping code on control edges. In Figure 7.7, a callee-save register is used inside of $B2$ only. By using our shrink wrapping algorithm based on the dominator and post-dominator trees, the wrapping point can be moved to $B1$ and $B4$ but if we insert wrapping code on the edge of $e1$ and $e3$, the frequency of wrapping code can be reduced to 100 to 10 and the execution performance can be improved further. The problem of finding optimal edges for inserting register saved code can be reduced to the *max-flow min-cut* problem [17] [30] [37] [47].

Given directed graph $G = (V, E)$, a special node s called the *source*, a node

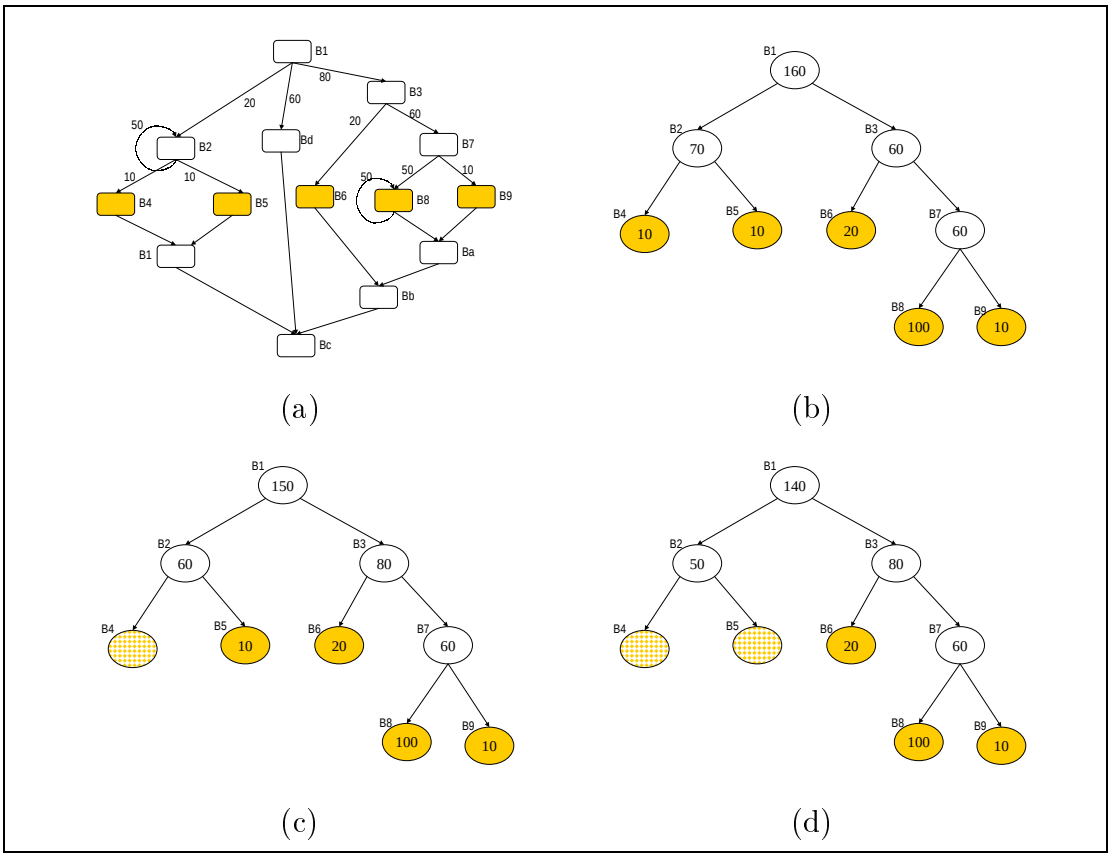


Figure 7.5: The execution of the shrink wrapping based on a dominator tree. (a) Control flow graph of example program. Edges are annotated with frequency. (b) Constructed dominator tree with the callee-save register used block. (c) $B4$ is selected as the insertion point since it has least frequency along the path from $B4$ to $B1$. The frequency of $B4$ to $B1$ is decreased by 10. (d) $B5$ is selected as the insertion point. The frequency of $B4$ to $B1$ is decreased by 10.

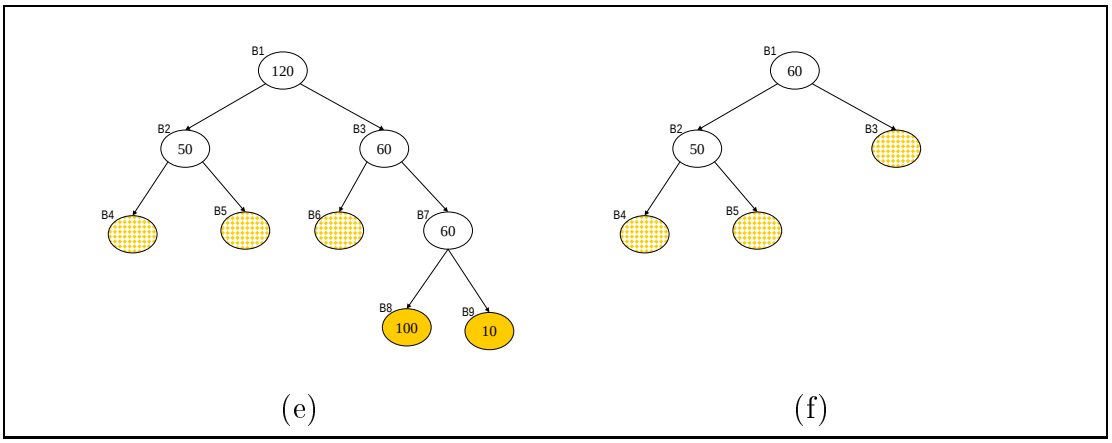


Figure 7.6: The execution of the shrink wrapping based on a dominator tree. (e) $B6$ is selected as insertion point. The frequency of $B3$ and $B1$ is decreased by 20. (f) From the path of $B8$ to $B1$, $B3$ is chosen. All descendants of B are ignored and the frequency of $B1$ is decreased by 60.

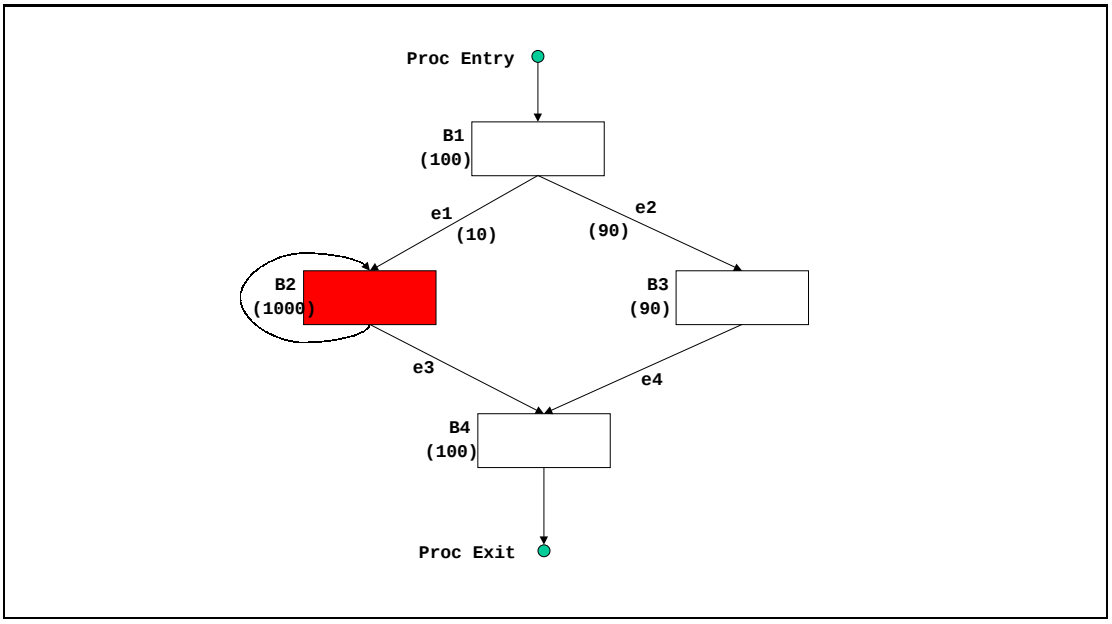


Figure 7.7: The Better Shrink Wrapping by inserting on the control flow edge

t called the *target*, and positive numbers $c(i,j)$ representing the capacity of the edge $(i,j) \in E$, the *max-flow problem* of G is to maximize the total flow from s to t. A *min-cut* of graph $G = (V,E)$ is a partition of V into S and $T = V - S$ such than $s \in S$ and $t \in T$ and the sum of edge frequencies from source partition to destination partition is minimized where s is the single source and t is the single destination of G.

Theorem 1 *The maximum weight among all (i,j)-flows in G equals the minimum capacity among all sets of edges in E whose deletion destroys all directed paths from i to v.*

In other words, the sum of the capacity of all the links in the *cut* is equal to the maximum flow through the graph when all the edges in the cut are fully utilized without more capacity.

To apply the max-flow min-cut algorithm to the control flow graph, we need some modifications to the control flow graph. We need to have all blocks using same callee saved register into one side of the partition. For each callee saved register EE considered, we need to create a pseudo destination node for the store point decision or a pseudo source for the load point decision. Every block which uses EE is connected to a pseudo destination node with infinite frequency edge capacity and the edge frequencies of the control flow remain as the capacity. This

ensures that all blocks use EE is considered as target of the problem so the source block will be located in at least one side of the partition and all blocks sharing callee-save register will be located in the other side of the partition. Likewise, Every block which uses EE is connected to a pseudo source node with infinite frequency for the load point decision.

7.2.2 Experiment

Figure 7.8 shows the results of our experiment with shrink wrapping. Our experimental machine has 32 GPR and 32 FPR's as before, and all these registers are allocated within each region-based on hyper-block. The chart shows the saved execution cycles compare to the total execution cycles when the caller-save registers are saved and loaded upon the entry and exit points of the function.

Shrink wrapping based on the dominator and post-dominator trees (DOM) shows performance improvement in every case of our experiment. The second approach by *min-cut* algorithm (CUT) performs better than DOM in some cases, but it may have negative effects in other cases, or may perform worse than the base case. We account for this as the overhead resulting from creating a new block when we need code in control flow edges. As we covered in Section 5.1, there are many cases that the code inserted between two blocks cannot be moved to either source or destination region and a new region is needed. Creation of new

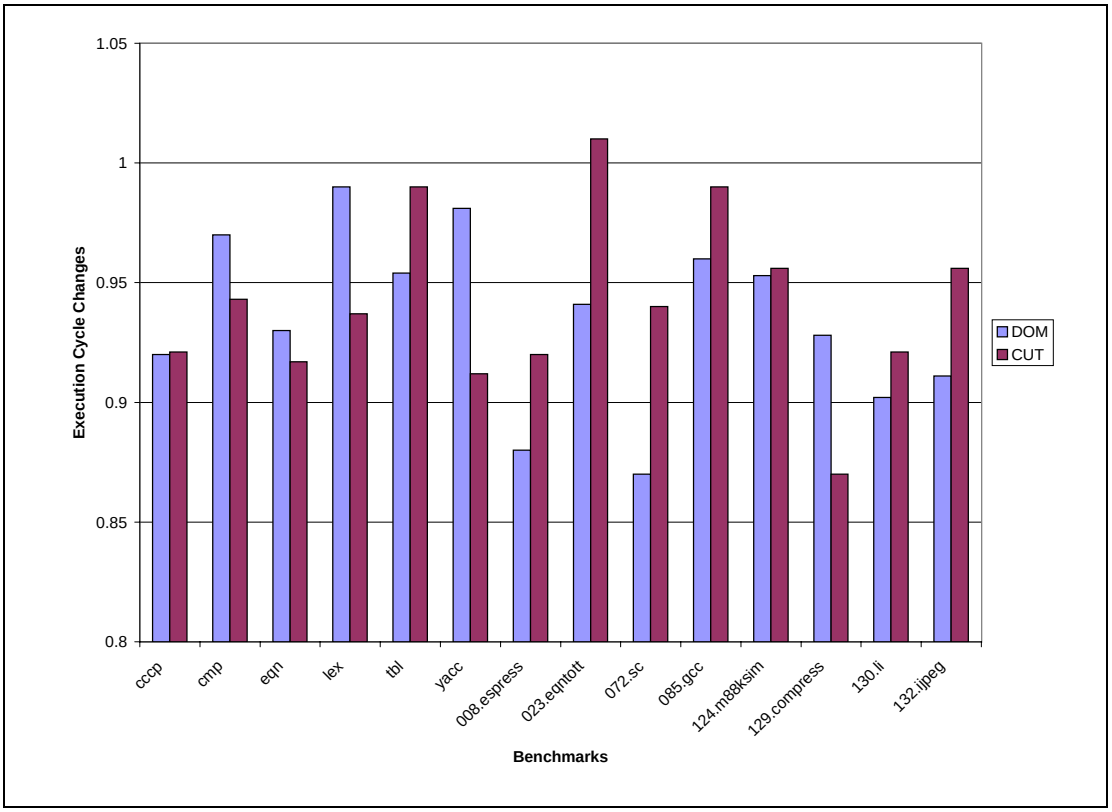


Figure 7.8: Performance comparison of shrink-wrap algorithm

blocks requires the introduction of extra branches into the program code, which lengthens the execution time. To avoid this case, Chow's approach always inserts register save code to the basic block entries. The number of cycles increased by creating a new block in compile time to ILP compiler is measured about 4% of the total execution cycles.

Chapter 8

Register Allocation with Region Restructuring

In previous chapters, we explored several techniques to improve the execution performance of the region-based register allocation by generating code that performs as well as or even better than the global register allocation. Our interest was how we can get at least the same quality of code as that derived by global register allocation while paying the lower compilation cost. We have conducted several experiments to validate the above.

In this chapter, we first compare the execution performance and compilation time of two types of region-based register allocation; one where the region is whole function (*i.e*, function-based register allocation) and the other where the region

is based on blocks commonly constructed in EPIC compilation like basic blocks, super-blocks and hyper-blocks. Then, we present the comparison of our global register allocation to IMPACT global register allocation showing how our global register allocation compares to other state-of-art global register allocation.

Next, we explore the relationship between region size and performance. One of the parameters in a region-based register allocation is the size of the base regions where we perform the intra-region register allocation. The size of the base region affects spill and shuffle costs. Our experiment shows that there is an interesting relationship between them, and there is a region size where both compile time and quality of code generated can be improved than function-based regions or hyper-block-based regions.

Finally, based on our observation, we propose fast region restructuring with register pressure thereby improving execution performance and compilation time over a given compilation unit. We introduce two approaches of estimating register pressure for restricting regions, and compare them to lead us to formulate it.

8.1 Comparison of Global-based and Region-based Register Allocation

8.1.1 Parameters of Comparison

The performance comparison of optimizing compiler tends to show dramatically different output according to all sorts of different parameters. We will describe the parameters we use in our experiment in this section. We used the combination of best possible techniques from our heuristic for each granularity of the region. For each region granularity, the effectiveness of each technique varies and they are summarized as follows.

8.1.1.1 Live Range and Interference Graph Construction

Live ranges are constructed as multiple hierarchies of sub-regions based on each operation, for both region-based and global register allocation. A live range may include one or more sub-region like loop, basic block, super-block or hyper-block and each sub-region consists of smaller sub-regions recursively all the way down to the operation level. Each operation is augmented with predicate expression and predicate analysis is used for interference graph construction.

8.1.1.2 Live Range Splitting

Frequency-based live range splitting is used for the experiment. Live range splitting becomes more effective when there is high register pressure, and this register pressure tends to increase along with the size of live range. The region hierarchy of a live range is used to decide live range splitting so a live range is split at region boundaries first. Live range spitting algorithm works exactly same in region-based and global register allocation, once the region considered reaches at the level of blocks like basic blocks or hyper-blocks.

8.1.1.3 Register Binding

Frequency-based propagation, propagation of unavailable registers, delayed bindings and clock hand register selection as described in Chapter 5.4 are used for the experiment in this chapter. These techniques shows more effectiveness for a smaller region where more propagation is required. In global register allocation when the register allocation is one function itself, the techniques for register binding is not required usually except the cases where the live-ranges are split.

8.1.1.4 Priority Function

We used the combinations of dynamic priority function and priority function with propagation presented in Chapter 6 for the comparison. Priority function with

predicate analysis is not used, since it shows very minimal performance improvement with much bigger compilation time. The compilation time overhead is pretty constant regardless of region granularity. Dynamic priority function and priority function with predicate analysis are effective regardless of the size of the region, but the priority function with propagation is more effective for a smaller region where more register propagation is required.

8.1.1.5 Machine Model

The Trimaran has five different types of register files: General purpose register(GPR), floating-point register file(FPR), predicate register file(PR), branch-target register file(BTR) and control register file(PR). Because each variable is assigned to a specific type of register file and they do not overlap, the interference of variables can be defined for each register file type separately.

There are many different machine factors that effect the performance of register allocation. One major factor is the size of various register files, and the other is the the number of functional units. In our experiment, we measure the register allocation performance primarily for GPR and FPR files. We varies the size of GPR and FPR files to 32, 64 and 96. We also vary the number of integer functional units as 4 ALU, 6 ALU and 8 ALU's.

Unlike Chow's original implementation, but more like the implementation of

Briggs and the Yorktown allocator [2], our register allocation is performed on lower level intermediate language after all optimizations, address mode selection and instruction scheduling have been completed. We do post-pass instruction scheduling again for register allocated code. The advantage to using the low-level form is a greater accuracy in allocation; the advantage to using the high level form is allocation speed and greater machine-independence. It seems to be more suitable to perform register allocation on a lower-level form for the EPIC. One of the most important optimization, the instruction scheduler, needs to be done by the EPIC compiler, and it cannot be done on high level intermediate language.

8.1.2 Region-based register allocation and function based register allocation

We perform experiments to compare two different types of region. In one case, a region is a basic block, a super-block or a hyper-block; in the other case, a function. When we perform the register allocation on the whole function, it becomes global register allocation. Our experiment shows our region-based register allocation generates comparable quality code to that generated by function based register allocation with smaller compilation time. Figure 8.1 shows the relative dynamic execution cycle (Execution) and compilation time (Compile) of the region-based approach compared to global register allocation. In this experiment, we use 64

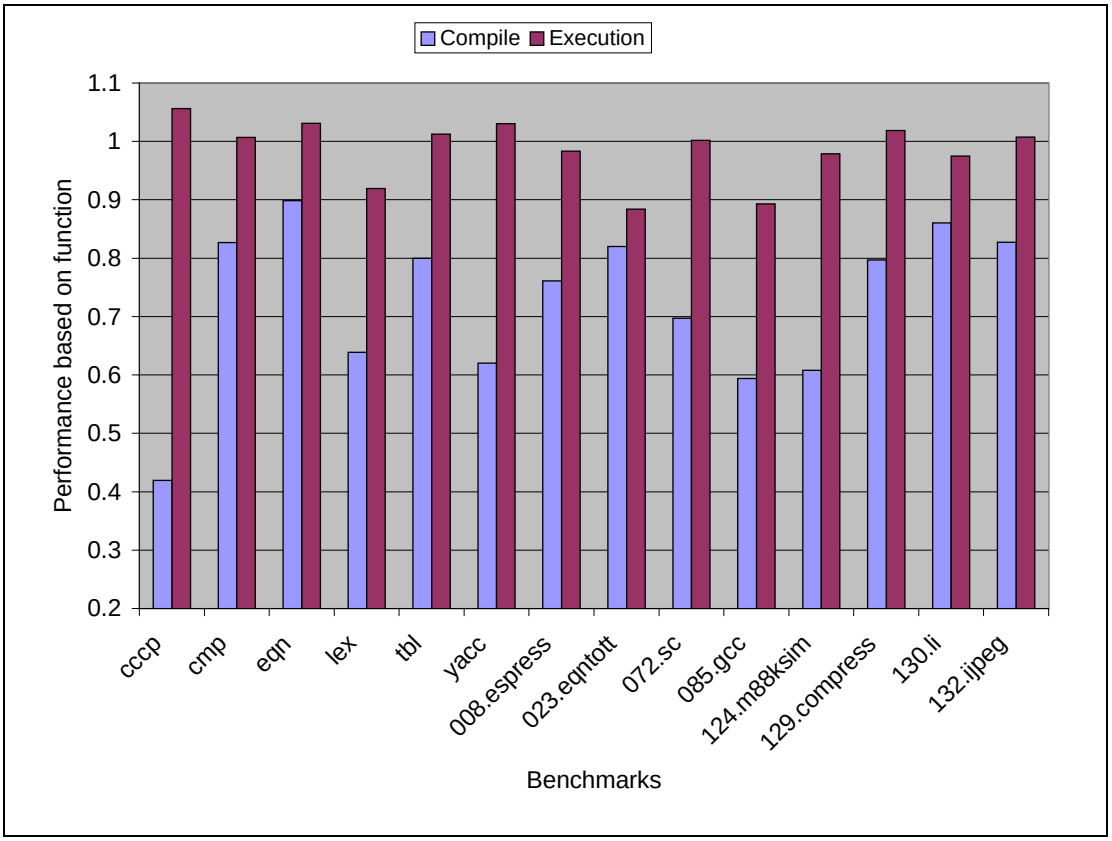


Figure 8.1: Dynamic execution cycles and compilation time performance compared to global register allocation

GPRs and 64 FPRs with 4 integer ALUs.

In most cases, block-based register allocation gives execution performance comparable to function-based register allocation. In some benchmarks like **lex**, **023.eqntott** or **085.gcc**, the execution performance of the register allocation based on smaller region shows better execution performance, up to 10%, than the global approach. This improvement of execution time is mostly due to the smaller

size of the interference graph comes from the smaller number of live ranges in the region-based approach. Figure 8.2 shows an example of how the register allocation of smaller region can be beneficial. The left column of Figure 8.2 shows the CFG assume that we have only one register available for either variable x or y. The middle column shows what happens in the global approach. One of the whole live range needs to be spilled, and y is chosen to be spilled in our example. On the other hand, register allocation is done at a region level, and both x and y can be spilled selectively as shown in the right most column of Figure 8.2. Rather than spill the whole live range of variable y in high frequency region $B3$, the variable x was spilled in region $B3$ and a physical register can be freed for y.

Figure 8.1 also shows the register allocation time for region-based register allocation. The amount of time required for smaller regions is the sum of the times required to allocate each region. We show relative compilation time of the region-based register allocation to the compilation time of function based register allocation. Register allocation based on regions show significantly faster compilation time in every case. Our experiment shows that we could save almost half the compilation time on average, by using a region-based register allocation and this savings may be as much as 60%. The biggest portion of compile time savings is the interference graph construction which requires $O(n^2)$ time for the n variables. Overall, our region-based register allocation generates code of comparable quality

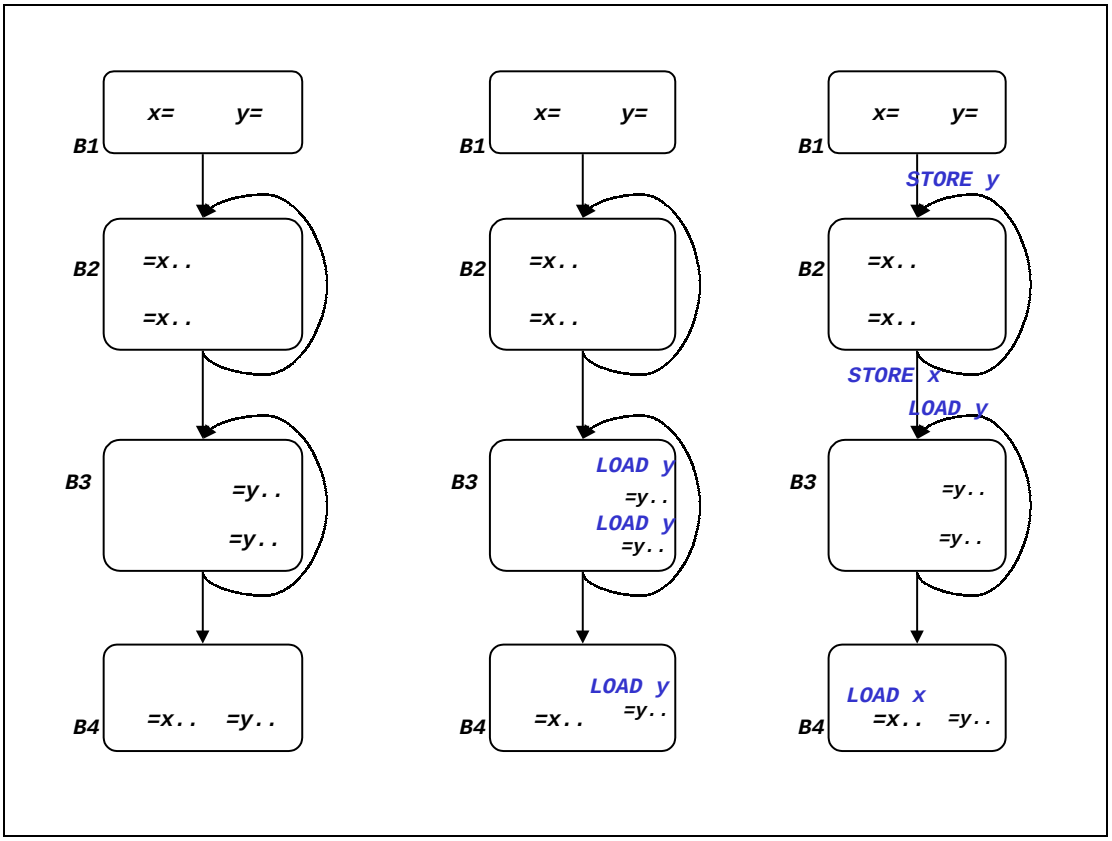


Figure 8.2: The benefit of smaller region

in a much shorter time.

8.1.3 Comparison to IMPACT register allocation

To validate the efficiency of our experiment, we tried to conduct a comparison of our global register allocators to other global register allocators. An important alternative approach to global register allocation is developed by Briggs *el al.*. Unfortunately, it is difficult to perform useful comparisons to their work, since their approach is based on the Chaitin style of allocation while our register allocation is based on Chow and Hennessy priority-based approach. It requires a large amount of time to implement Briggs style register allocation in our framework, and it is difficult to achieve the best possible performance without extensive testing and tuning.

Another register allocation based on the priority-based coloring approach is the IMPACT register allocation [21]. Since, the IMPACT-I compiler [10] has intermediate language which is compatible to Trimaran, we could fairly easily test the performance of IMPACT register allocation by converting our intermediate language. IMPACT register allocation made many improvements to Chow's approach like Briggs' style loop-based splitting and many performance tunings.

Figure 8.3 shows the performance improvement data of our global register allocation compared to IMPACT global register allocation. We have conducted

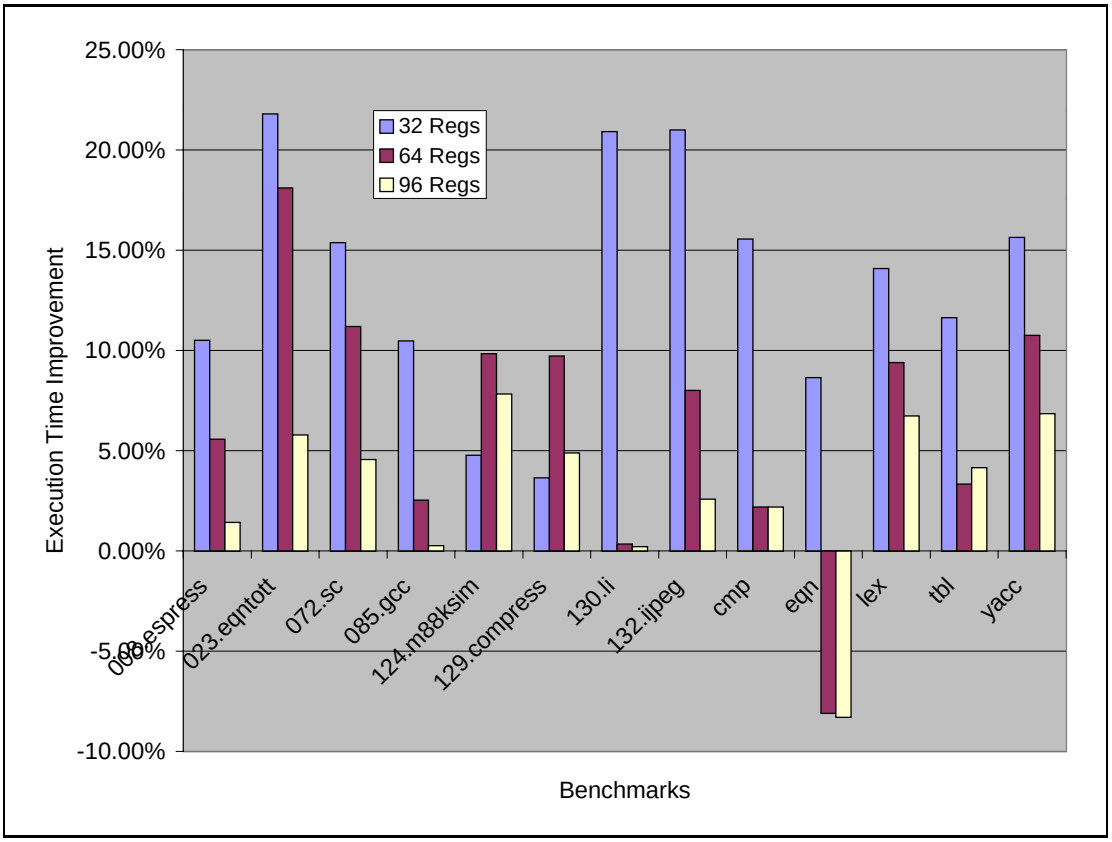


Figure 8.3: Dynamic execution cycles compared to IMPACT global register allocation

three different series of experiments with different register file sizes; 32 GPR + 32 FPR, 64 GPR + 64 FPR and 96 GPR + 96 FPR. Our register allocation shows an average execution time improvement of 12%. As the register file sizes increase, the benefit of our register allocation decreases, since both register allocations have close to the maximum required registers. When there is high register pressure, our approach produces better quality of code, and this benefit results mostly from our live range splitting. In some cases like **eqn**, IMPACT register allocation shows better performance than our global register allocation. Our analysis shows that the different approach of using callee-save register and generating the spill codes may have minor benefits in some cases.

8.1.4 Effects of register pressure

In an ILP compiler, register pressure is one of the most important factors in deciding the quality of the code generated by register allocation. Two of the major components that determine register pressure in a compilation unit are register file size and the number of functional units of the machine. Today's modern architectures, like EPIC processors, are designed with much larger register files, so that each compilation unit has freedom to use many registers. At the same time, many functional units sharing the same register file in EPIC lead to higher level of ILP, thus introducing higher register pressure by allowing many live vari-

ables in the program. Our interests is in how region-based register allocation and function-based register allocation perform when we change register pressure. Our research in Chapter 4 shows that the register pressure increased by larger number of functional units is relatively small, and the number of functional units is not the bottleneck in obtaining the higher level of ILP. In this section, we study the effect of register file sizes on our register allocation.

With more registers, it is not hard to see execution performance will improve up to a certain point and stop once there are enough registers for all live ranges regardless of the size of the region. The compile time can be divided into several components in our framework: live range and interference graph construction time, live range coloring time, live range splitting time, and spill code and shuffle code generation time. The time required for live range construction including the time to compute priority depends on the total number of live ranges, N, and the total number of operations, P, before register allocation. These are not dependent on region size or the number of registers. However, it is showed that the interference graph construction time becomes larger as the region size grows in the order of $O(n^2)$, where n is the number of live ranges in each region considered, and this time is not dependent to the number of physical registers either.

Live range splitting time and coloring time with propagation is closely related to register pressure. As register file size decreases, the register pressure in com-

	Function-based	Region-based	Time saved
008.espresso	140970	83097	41.05%
023.eqntott	23299	12851	44.84%
072.sc	37602	22498	40.17%
124.m88ksim	41872	25618	38.82%
129.compress	14145	8274	41.51%
130.li	5747	4572	20.45%
132.ijpeg	110986	87414	21.24%
cmp	1257	1017	19.09%
cccp	48172	20803	56.82%
eqn	8138	6379	21.61%
lex	98467	45294	54.00%
tbl	52409	21021	59.89%
yacc	112109	57727	48.51%

Table 8.1: The summary of the relationship between register file size and compilation time with 32 register set.

pilation unit increases. In turn, the number of live range splitting increases as the register pressure increases. For the coloring phase of register allocation, the number of propagation remains the same regardless of the register pressure, since the number of propagation is related only to the number of live ranges across the regions. But, the register selection algorithm with propagation needs to scan preferred, forbidden or unavailable register sets, and this required time is correlated to the number of physical registers. Therefore, both techniques need more time as register file size grow.

Table 8.1, Table 8.2 and Table 8.3 show the compilation time for our benchmarks for different register file size with the same machine we used in previous

	Function-based	Region-based	Time saved
008.espresso	108957	77823	28.57%
023.eqntott	15714	12212	22.29%
072.sc	31479	21452	31.85%
124.m88ksim	35641	24952	29.99%
129.compress	9118	7249	20.50%
130.li	5537	4680	15.48%
132.ijpeg	107798	87313	19.00%
cmp	1211	1023	15.52%
cccp	29796	25863	13.20%
eqn	7983	6207	22.25%
lex	63005	45666	27.52%
tbl	31692	19585	38.20%
yacc	78098	52274	33.07%

Table 8.2: The summary of the relationship between register file size and compilation time with 64 register set.

	Function-based	Region-based	Time saved
008.espresso	103350	78979	23.58%
023.eqntott	15227	12416	18.46%
072.sc	30383	19351	36.31%
124.m88ksim	35643	29965	15.93%
129.compress	9144	7311	20.05%
130.li	5385	4741	11.96%
132.ijpeg	105658	86819	17.83%
cmp	1167	1069	8.40%
cccp	27394	26438	3.49%
eqn	8034	6329	21.22%
lex	60233	47710	20.79%
tbl	28421	19481	31.46%
yacc	72140	50763	29.63%

Table 8.3: The summary of the relationship between register file size and compilation time with 96 register set.

section for both the function-based approach and the region-based approach. For function-based register allocation, compile time tends to decrease as register file sizes increase up to a certain point in most of our experiments. This change is closely related to the number of live range splitting. As register file sizes increase, register allocation requires less live range splitting, and thus the compilation time is decreased. On the contrary, compilation times for region-based register allocation does not show any specific patterns. In some benchmarks, the compilation time remains consistent. The compilation time may increase slightly in some case like **cccp**. This is caused when register allocation requires more possible choices for the register binding propagation.

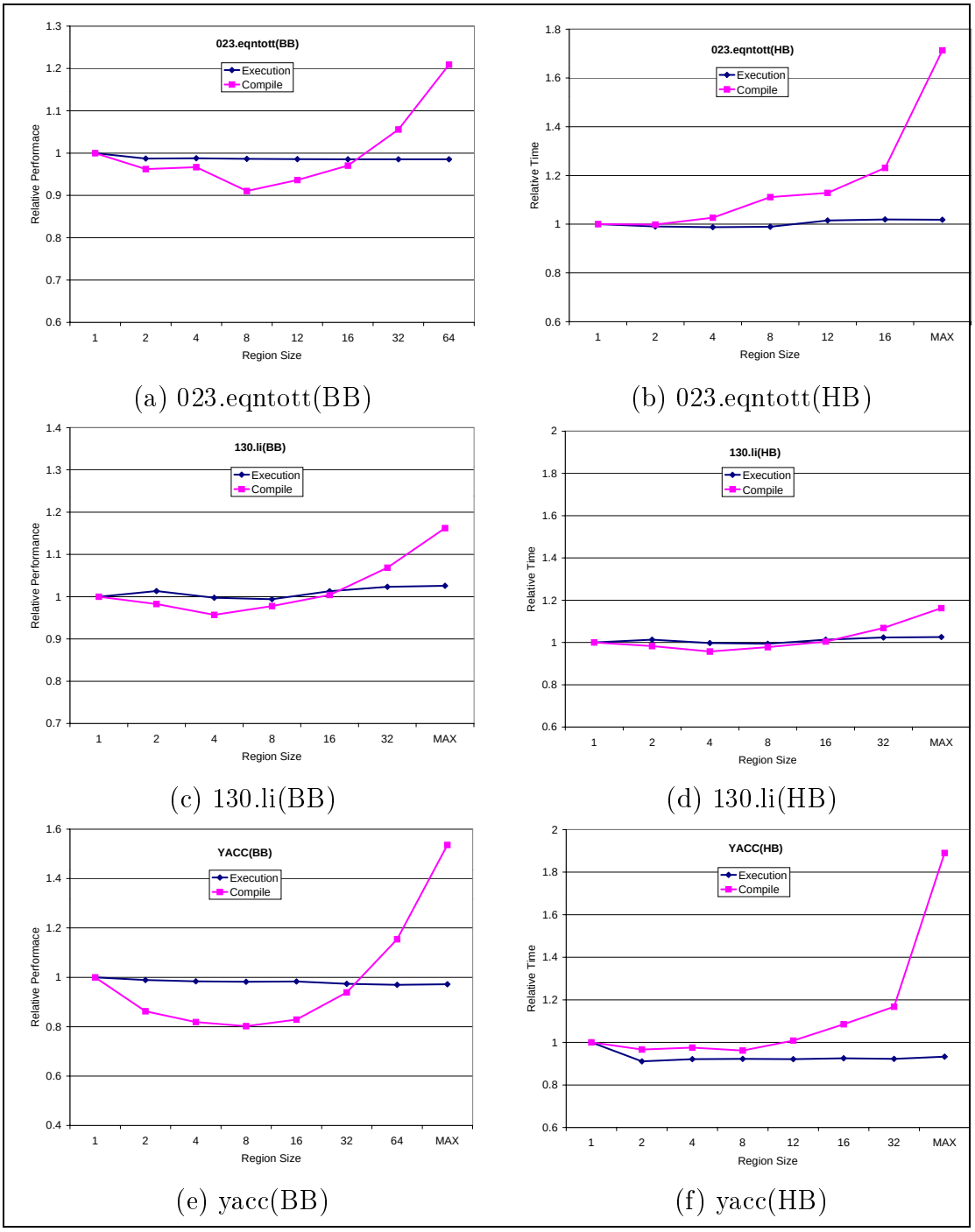
Overall, compile time benefit is larger under high register pressure. Even in the case of lower register pressure with large register file sizes, the overall compilation time of region-based register allocation is mostly smaller than that of the function-based register allocation, since the allocation time is dominated by interference graph construction. Based on these two observations, we explore the relationship between region size and register allocation in the following section.

8.2 Relations of the region sizes and register allocation

The size of compilation unit has a close relationship to the performance of the compiler in both the quality of the code it generates and the compilation time. As the size of the compilation unit increases, the register allocation sees a larger part of the program, and the quality of register allocation can be improved. But the super linear complexity of most of the analysis used in register allocation increases compilation time.

In the previous section, our experiment compared the performance of region-based register allocation and function-based register allocation where we use hyper-blocks and basic blocks as a region. Next, we are interested in how size of regions would effect the performance. To verify the actual trends of the compilation time and execution time related to the size of the region, we have conducted a series of experiments using each hyper-block as the base unit of our register allocation and increasing the size of the register allocation unit by merging several hyper-blocks.

Figure 8.4 and Figure 8.5 show the execution time and compilation time for various region sizes for selected benchmarks. We use a single basic block, super-block or hyper-block as a basic unit of compilation and increase the region size by merging them. The results show that execution time is nearly constant regardless

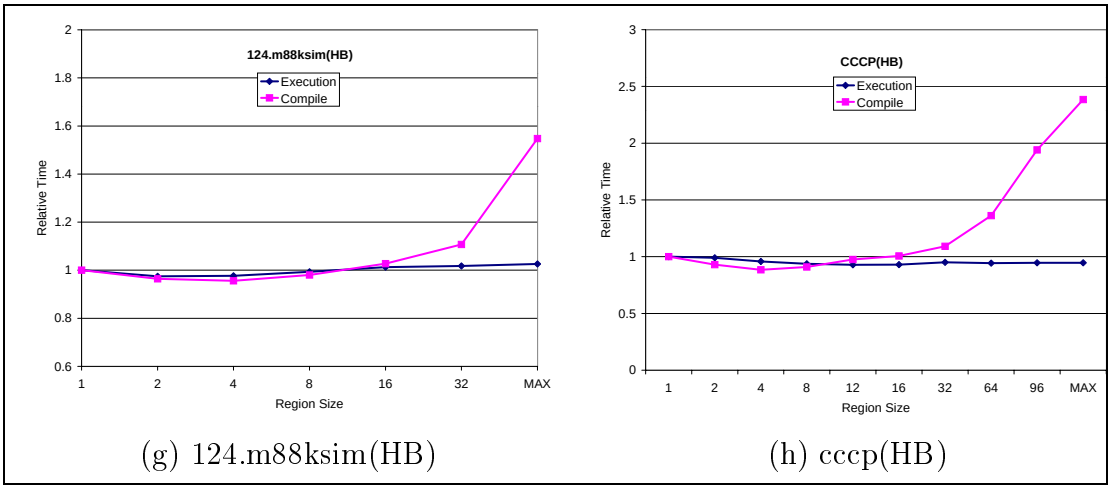


(a) 023.eqntott(BB) (b) 023.eqntott(HB)

(c) 130.li(BB) (d) 130.li(HB)

(e) yacc(BB) (f) yacc(HB)

Figure 8.4: Comparisons of compile time and execution time for different region size



(g) 124.m88ksim(HB) (h) cccp(HB)

Figure 8.5: Comparisons for compile time and execution time for different region size

of the compilation size, while compilation time increases in a super-linear manner. The constant execution time in our framework dues to the following. We have explored many techniques to overcome the possible coloring mismatches and performance drawbacks of the region-based approach in the previous chapter. By combining all these, we can obtain very comparable execution time performance in region-based register allocation regardless of region size or format. Usually, the execution time by the region-based register allocation still has less performance than the execution time by the global register allocation because of the extra shuffle code. However, region-based register allocation may have better register allocation quality in some cases as we explained in previous section. This is due to the cases that a large live range can be divided into several segment voluntar-

ily and register can be allocated to other live segment which has more savings, while the live range is split only when there is high register pressure when the compilation unit size is large.

On the contrary, the results show that the compilation time decreases slightly to a certain point. Then it increases consistently as the size of the region increases. Overall, higher compilation time in a large region can be explained by the greater complexity of live range construction and frequent live range splitting.

8.2.1 Analysis of the effect of the region size

Our experiments show that the compilation time increases when the compilation unit size is either too small or too large. The compilation time increase in large regions is mostly due to the following two reasons. The building of the virtual register interference graph consumes a great deal of time. Given the $O(n^2)$ nature of interference graph construction, the compilation time grows in a super-linear manner as the size of the region grows. The other major factor of compilation time in a large region is live range splitting. When a live range lr is split, we need to update information like interfering live ranges, preferred registers, forbidden registers, and the priorities of two split live ranges of lr, as well as all other live ranges interfering with lr.

Table 8.4 shows the average number of variables(nodes) in the interference

graph per region and the number of live range splitting for different compilation unit sizes. Column **RS** shows the size of compilation unit. We combine multiple adjacent hyper-blocks or basic blocks into a region in which register allocation is performed as in the previous section. A region constructed with all blocks in a procedure is designated as **MAX**. Column **GS** shows the average number of variables in each interference graph. As the size of compilation increases, we have more complex interference graph. The third column **SC** shows the total number of live range splitting required. We perform the live range splitting only to the level of block boundaries, since the live range splitting inside of each block does not show much performance difference for the different region sizes as we explored in Chapter 4. Therefore, when **RS** is one (each region is made of a single block), we do not need any live range splitting. As the size of live range increases, the number of live range splitting grows. But this growth slows down after a certain size. The last column **CT** shows the relative compilation time compared to the case when we use a hyper-block and basic block as a unit of compilation (i.e. **RS** is one). This tables shows that if we can construct a region in which the compilation time is minimum, we still need to perform considerable amount of live range splitting. For example, the compilation time is minimum for a region with 4 blocks for "124.m88ksim", and we need live range splitting of 393, which is about half of the live range splitting needed for region size of "MAX".

g	023.eqntott			124.m88ksim		
RS	GS	SC	CT	GS	SC	CT
1	108.6382637	0	1	44.2345397	0	1
2	178.4726688	242	0.998373022	75.40144728	242	0.96424614
4	304.6495177	444	1.026697234	131.426781	393	0.955713969
8	534.7717042	597	1.110930336	221.8033776	576	0.980350151
12	762.4630225	723	1.128679189			
16	971.9099678	701	1.230661145	352.9408163	681	1.027443304
32				494.5671642	797	1.107298515
MAX	1044.693548	791	1.713356012	618.6865079	829	1.547351703
	130.li			cccp		
RS	GS	SC	CT	GS	SC	CT
1	16.60858896	0	1	87.36078615	0	1
2	24.87345647	52	0.982722731	131.7222222	106	0.92931719
4	43.22495736	69	0.956910908	207.1353791	213	0.885054654
8	53.98479300	79	0.977726894	331.9324324	372	0.909367188
12				430.1232227	367	0.975197294
16	70.80958365	83	1.004163197	525.2060606	455	1.006176168
32	90.85275635	109	1.068692756	764.6	669	1.090926915
64				980.3896104	857	1.361845008
96				1026.513889	968	1.94054213
MAX	122.5664063	122	1.162156536	1090.238095	647	2.384441939
	tbl			yacc		
RS	GS	SC	CT	GS	SC	CT
1	108.0980392	0	1	238.6904025	0	1
2	160.6149194	348	0.973155908	381.0866935	919	0.966519092
4	250.2413127	489	0.998296344	622.8127413	1470	0.975568866
8	394.4370629	637	1.08200571	994.6083916	1849	0.96178261
16	592.0346821	757	1.199097523	1521.271676	2076	1.008642859
32	781.9344262	837	1.396813703	2109.877049	2145	1.085240405
64	902.8019802	934	1.760198913	2306.762376	2285	1.167827729
MAX	954.5168539	770	2.38419744	2308.280899	2374	1.890263355

Table 8.4: Effect of different region sizes on interference graph size and live range splitting. RS: the number of blocks in a region (compilation unit) GS: the average size of interference graph (number of nodes) SC: total number of live range splitting CT: compilation time

The compilation time related to register binding propagation tend to have an effect opposite to live range splitting. The smaller the compilation unit size, the more frequent propagation it requires. This leads to larger compilation time. For example, a live range spanning two separate regions needs one propagation pass for its register binding, while a live range spanning three regions needs more than two propagation passes. The compilation time is increased by the total number of propagations. Also, small segments of a live range are more likely to introduce coloring miss-matches, which is another cause of compilation time increase. Table 8.5 shows the experimental data about how many propagations and delayed bindings are performed. Our experiments indicate that the compilation time overhead is linear to the number of propagation and delayed bindings.

One of the techniques used in priority function is also closely related to the performance of our compilation. In the region-based register allocation, we adapt the register binding information from the neighboring regions and use it in our new priority function. Therefore, the number of propagations has a positive correlation to the compilation performance of the priority function with propagation, as above.

	023.eqntott		124.m88ksim		130.li	
RS	PR	DB	PR	DB	PR	DB
1	7094	769	33485	1705	32113	3870
2	6223	467	27179	866	23049	1959
4	5329	182	22095	442	15481	889
8	5112	75	17600	268	9814	287
12	4591	55				
16	3984	56	13185	150	6511	118
32			11936	97	5163	62
64			11188	51		
MAX	2724	1	8990	4	3845	18

	CCCP		TBL		YACC	
RS	PR	DB	PR	DB	PR	DB
1	193398	37405	16625	5202	82558	7067
2	165445	16041	15611	2849	70039	3643
4	142769	7738	14137	1028	60075	1664
8	132887	3148	12828	321	51922	1100
12	119036	1791				
16	125757	1809	12546	138	49414	1012
32	122822	686	11966	114	48092	624
64	108692	444	11153	79	44617	213
96	102308	906				
MAX	83000	368	10164	55	39524	10

Table 8.5: Effect of different region size on propagation. RS: the number of blocks in a region (compilation unit) PR: total number of propagation used in coloring DB: total number of delayed bindings performed.

	Region Size	
	Small	Large
Live Range Splitting	Less Effective	More Effective
Propagation	More Effective	Less Effective
Priority Function	More Effective	Less Effective

Table 8.6: The summary of the relationship between region size and the effectiveness of various technique in register allocation.

8.2.2 Summary of the effect of region size

Table 8.6 summarizes the relationship between region size and some of the techniques we studied in region-based register allocation. Live range splitting is more useful as the size of the region increases because of increasing register pressure. In smaller regions, propagation and priority function play bigger role, since there are more live ranges crossing regions. By performing more live range splitting, we increase the opportunity to produce better quality code but we also increase compilation time. Likewise, frequent use of propagation means larger compilation time as well as larger execution time caused by more shuffle code. Overall, the performance of register allocation, the execution time and the compilation time, is closely related to register pressure and the number of live ranges in each region. These observations motivate us to seek more optimal size regions, which might improve both compilation and execution time. We are interested in finding a region size where the compilation time overhead of propagation and live range splitting can be minimized. In the next section, we propose the concept of re-partitioning the regions based on register pressure, with the objective of minimizing such overheads.

8.3 Region Restructuring

8.3.1 Problems of other region construction methods

A region-based compiler begins by repartitioning the program into regions. The region selector may select one region at a time or it may select all regions a priori, before the compilation process begins on any region.

Following criteria can be used for region selection:

- Execution frequency information of the operations
- Memory access information
- Structure of region i.e. basic block, super-block hyper-block, or loop
- Size of the region such as the number of operations.

Frequency-based regions and structure-based regions provide very natural units of register allocation so that closely related regions are considered together. But the lack of register pressure information during region formation forces register allocation to have excessive live range splitting or propagation. If the region is too large and the register pressure is higher than the available number of registers, many live ranges are forced to spilt. If regions are too small, the compiler has to spend more time for propagation and thus may add unnecessary shuffle code due

to color miss-matches. Some other approaches use region selection based on register allocation [45] [40], but the overhead of determining the regions themselves are significantly larger and make these techniques impractical for the region-based approach.

8.3.2 Register Pressure Sensitive Region Restructuring

This thesis proposes linear time region restructuring based on the regions previously built for other compiler phases, for instance instruction scheduling. We estimate the register pressure in each region, which can be a basic block, super-block and hyper-block. These regions are grouped together according to their frequency information and the register pressure estimated , so that the number of live range splitting and the number of propagation for register allocation can be minimized. In this way, we can hope to reduce the compile time and improve the execution performance. We show two different techniques to estimate the register pressure; measuring the maximum bandwidth of the live variables and the counting the number of operations in each region. We describe these in the following sections in detail.

8.3.3 Region restructuring with register bandwidth

The maximum bandwidth of the live variables is the largest number of live ranges which are live at a certain point of the program. The maximum bandwidth can be obtained as a part of liveness analysis, so it does not cause large extra compilation time for register allocation. Since regions are formed in the previous stage, and the number of regions is much smaller than the number of operations used in region formation stage, the region restructuring with register bandwidth can be done both easily and quickly.

The region structuring for register allocation consists of the following steps. First, we select a start block s called *seed*. The seed block is the most frequently executed block which is not included in any other regions. Second, we expand the scope of the region to either the successor or predecessor block n of the current region-based on execution frequency. The block n is included as long as the register bandwidth of n is less then the number of available registers. The expansion stops if one of following conditions is satisfied:

- Every block belongs to some region.
- The next selected candidate block already belongs to another region.
- The next selected candidate block has higher register bandwidth than the available registers.

This region restructuring algorithm has many similarity to the frequency-based region formation algorithm, which is also used in the Trimaran compiler [49]. The principal difference is

- Register bandwidth is considered for region growing.
- The algorithm considers the control flow graph as an undirected graph and expansion can be to either successor blocks or predecessor block.
- If the next highest block is already selected for the other region, the expansion stops.

Unlike frequency-based region formation where multiple paths are expanded when one path is blocked, the region restructuring stops its expansion whenever expansion is blocked for the next selected region. This condition prevents shuffle code from being inserted at high frequency points. Figure 8.6 (a) shows an example of region structuring by the algorithm we have explained. The first region $R1$ consists of the block $B2$ alone since the bandwidth of $B1$ is larger than the number of physical registers. For the region $R2$, we choose $B3$ as a seed. The expansion may select $B2$ which already belongs to region $R1$. Assuming that the expansion continues to the next candidate block, $B4$, the region $R2$ will consist of two blocks, $B2$ and $B3$, as we show in Figure 8.6 (b). If variable x is spilled in $R2$, but bound to a physical register in $R1$, the shuffle code may be required at a more frequent

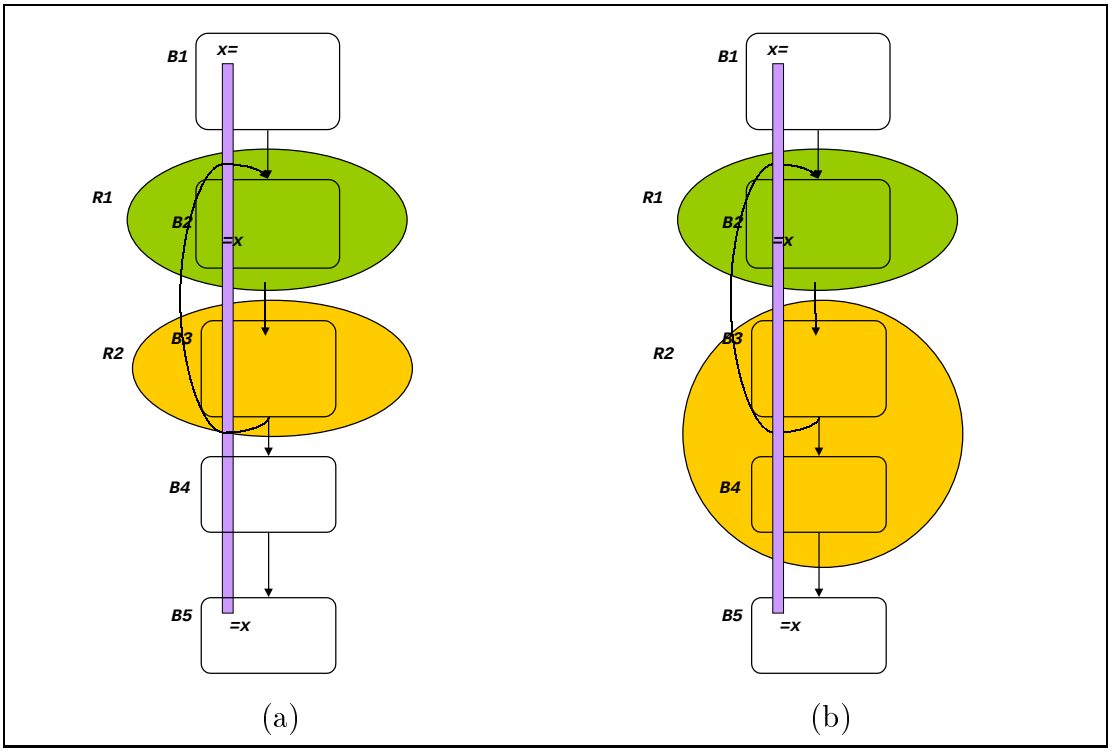


Figure 8.6: Two ways of region restructuring

execution point. The last condition in previous list prevents our algorithm from inserting expensive shuffle code.

Our experiment shows that this approach may not work and produces a large region in some cases. For example, if we have many basic blocks and none of them has the maximum bandwidth bigger than the number of registers, our technique includes very block into the region and it will have similar performance to function-based register allocation.

8.3.4 Region restructuring with the number of operations

Another approach of estimating register pressure for region restructuring is counting the number of operations in regions. We estimate the register pressure by assuming that every operation in a block creates a new live range and these variables are live to the end of the block. In very highly parallelized code, this approach models size the live range closely. Our research shows that this technique works well all over our benchmarks with even smaller compilation time overhead.

Figure 8.7 and Figure 8.8 shows the experimental result of our region restructuring (RR) compare to global register allocation (FB) and region-based register allocation (RR). Our approach works well in most cases by partitioning given regions into several groups where it can decrease the compilation time and execution time.

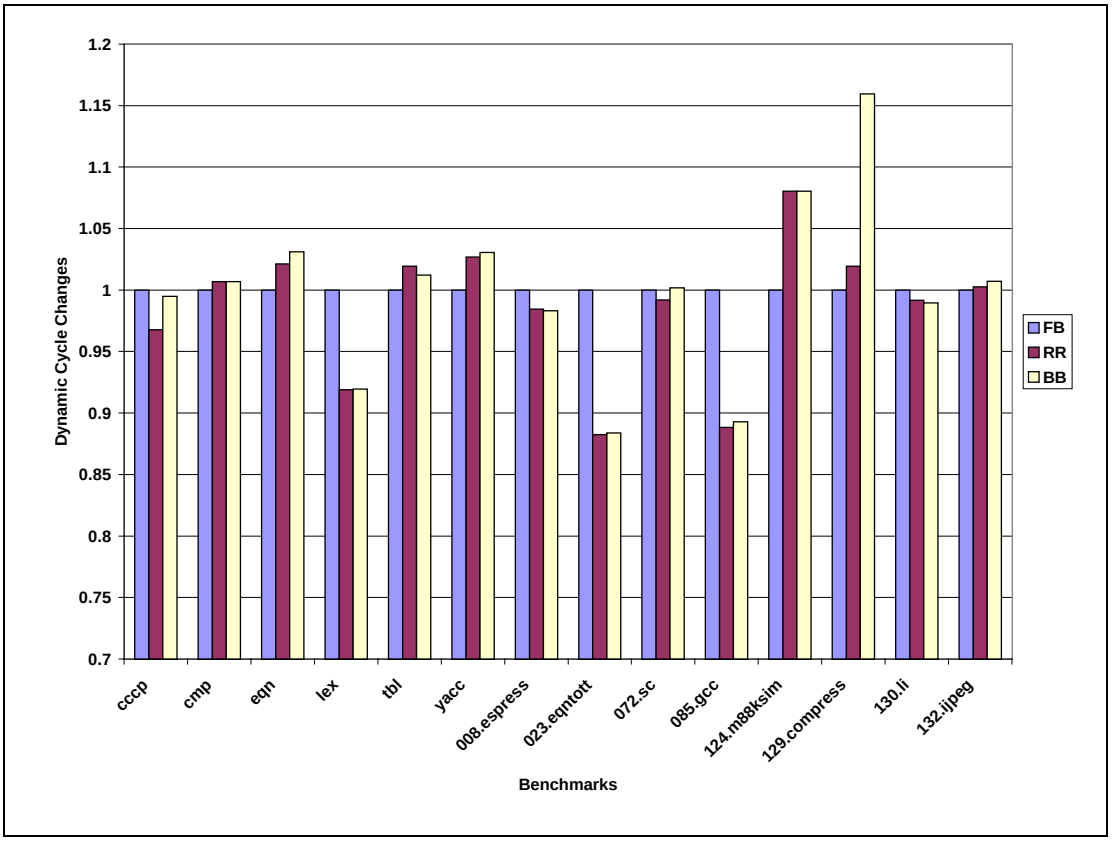


Figure 8.7: Total compilation time comparison with 64 GPR and 64 FPR

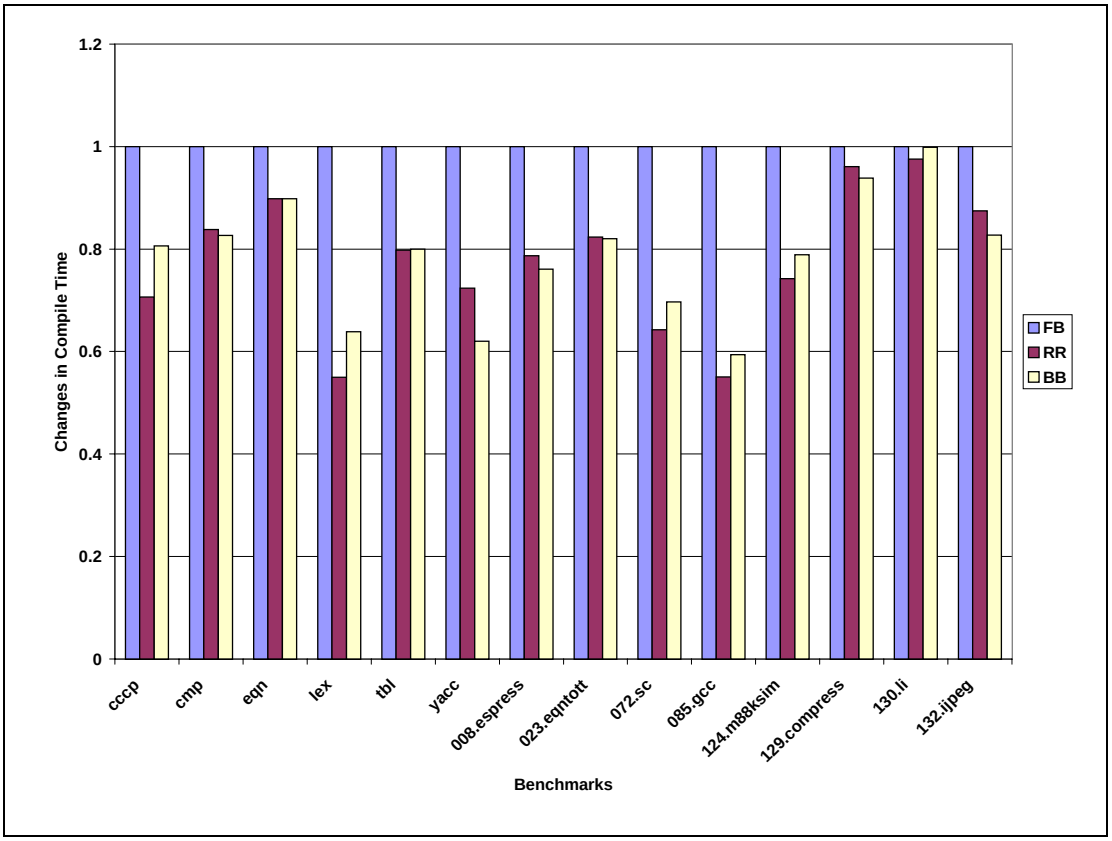


Figure 8.8: Total execution performance comparison with 64 GPR and 64 FPR

Chapter 9

Conclusions

In ILP processors, the compiler needs to use aggressive ILP techniques to achieve a high level of parallelism. However, the performance improvement by an aggressive ILP compiler comes with increased compilation time. Many of the analyses used in optimizing compiler have super linear complexity. As the size of the compilation unit grows, compilation time increase in a super-linear manner.

As the size of the compilation unit is limited, the compilation time can be reduced. But the limited scope of compilation may restrict the scope of optimization. As a result, the compiler may generate less efficient quality of code. Ideally, we want to get smaller compilation time and the same or better execution time as that obtained using the global approach.

This thesis achieved this goal in the context of one of the important phases of

compilation, namely register allocation. In this dissertation, we proposed several innovative techniques for frequency sensitive region-based register allocation to improve the code quality generated by the region-based approach. Through our region-based register allocation framework, we can reduce the compilation time by almost half of its global counterpart, while maintaining the execution time within 5% of the global approach.

In region-based register allocation, we used a local interference graph and local register allocation. But the limited scope of register binding caused many coloring miss-matches at each region boundary. By using inter-region optimizations such as frequency-based propagation or delayed-binding, we are able to overcome the problem of shuffle code introduced at region boundaries. Frequency-based propagation works efficiently since it uses register binding information from one region to the next. It also moves shuffle code to lower frequency points by maintaining frequency information. The order of register binding during coloring phase is also enhanced by using the register binding information from neighboring regions in the priority function. Unlike previous approaches, our priority function considers the effect of shuffle code required in the region-based approach. Thus, it represents the benefit of register binding more precisely.

Even though the region partition provides the natural live range splitting capability into the region-based register allocation framework, intra-region live range

splitting gives other performance improvements in some large regions. We propose another novel technique, frequency based live-range splitting and frequency based rematerialization. Live-range splitting points are searched in the order of control-flow frequency from the most frequent point of the program which ensures that we select the least frequent point. In frequency-based rematerialization, by generating rematerialization codes only on the splitting point, we are able to reduce the number of rematerialization instructions dramatically. Moreover, frequency-based splitting guides rematerialization to generate the rematerialization code at the least frequent points.

This thesis also presented and evaluated several enhancements to register allocation. We explored the issues related to predicated instructions and the two categories of registers related to calling convention: caller-save registers and callee-save registers. We studied the limitation of live-range splitting related to predicate registers and proposed a heuristics based on PQS. We also try to optimize the placement of the store and load operations for callee-save registers. We apply execution frequency to shrink wrapping by constructing dominator tree. We weight the node of the tree with execution frequency and find optimal points to add spill codes for callee-saved registers by weight reduction.

We showed considerable compilation time savings with comparable execution time performance by synthesizing our techniques in a region-based register al-

location. We also showed that the performance of the register allocation varies according to the region size. We proposed the concept of restructuring the regions based on register pressure and discussed how we can estimate the register pressure in order to improve compilation time while maintaining the execution time.

9.1 Contributions

Using the techniques developed in this thesis, we accomplished our goal of reducing compilation time in our region-based register allocation while maintaining the code quality (*i.e*, execution time) comparable to the global approach. We proposed new techniques in the various phases of region-based register allocation which improved the compilation time while having comparable execution time. Furthermore, we quantified the relationship between the region size and the effectiveness of our techniques. Based on our observations and a syntheses of our techniques, we developed the concept of region restructuring that leads to even lower compilation cost without sacrificing performance. Our register allocation is not limited to any specific type of program structures and we can maintain comparable quality of code in regions of every different size through our techniques. By combining hyper-blocks, super-blocks and basic blocks into restructured regions based on operation size and register file size, we were able to successfully reconstruct regions where the compilation time is close to the minimal compilation time which can be obtained

from optimally sized region.

9.2 Future Work

This thesis has demonstrated that region-based register allocation shows superior compilation time performance with compatible execution time performance compared to other global register allocation. However, there are still some important areas which require further research and investigation.

We proposed region restructuring by using register pressure. To estimate the register pressure, we used two different heuristics of selecting the operation count based on the register file size and using register bandwidth in each region. Even though, this gives us better performance compared to the region-based approach using hyper-blocks or super-blocks, the performance of our approach is not proven to be optimal. Ideally, the region itself should be constructed in a way to accommodate both instruction scheduling as well as register allocation.

Another challenging problem is to integrate region-based register allocation with instruction scheduling. We have observed that the codes given to register allocation after instruction scheduling have register pressure to the level that some of the live ranges assigned to registers need to be spilled. Less aggressive instruction scheduling will lead to much less spill code and an improvement to the overall execution time.

Finally, it is important to consider a register allocation process which use the available slots in ILP processors. In an ILP processor, adding the spill code into unused instruction slots does not increase the overall execution time. However, the traditional model of register allocation does not reflect this behavior well. For example, the priority function we used in our framework is only based on the spill counts. If we can add the spill code in empty slots in ILP processor schedule, the actual register allocation benefit becomes smaller. The register allocator can be improved by considering no-cost spilling.

Bibliography

[1] R. Allen and S. Johnson. Compiling C for vectorization, parallelization, and inline expansion. In *Proceedings of the SIGPLAN'88 conference on Programming Language design and Implementation*, page 241, 1988.

[2] Mark A. Auslander and Matrin E. Hopkins. An overview of the pl.8 compiler. In *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*, 1982.

[3] Preston Briggs. *Register Allocation via Graph Coloring.* PhD thesis, Rice University, April 1992.

[4] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practial improvements to the construction and destructon of static single assignment form. *Software–Practice and Experience*, 1(1):1–28, January 1988.

[5] Prston Briggs, Keith Cooper, and Linda Torczon. Rematerialization. In *ACM SIGPLAN Conference on Programming Language and Design*, pages

311–321, June 1992.

[6] Prston Briggs, Keith Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Program Languages and Systems*, 16(3):428–455, 1994.

[7] David Callahan and Brian Koblenz. Register allocation via hierarchical graph coloring. In *PLDI '91. Proceedings of the conference on Programming language design and implementation*, pages 192–203, 1991.

[8] G. J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 98–105. ACM, ACM, 1982.

[9] 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, 1981.

[10] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hsu. IMPACT: An architectural framework for multiple-instruction-issue processors. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 266–275, May 1991.

[11] 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, 1990.

[12] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman. A VLIW architecture for a trace scheduling compiler. In *Proceedings of th 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 180–192, April 1987.

[13] Keith Cooper and Taylor Simpson. SCC-based value numbering. Technical Report CRPC-TR95636-S, Center for Research on Parallel Computation, Rice University, October 1995.

[14] Keith Cooper, Taylor Simpson, and Christopher A. Vick. Operator strength reduction. Technical Report CRPC-TR95635-S, Center for Research on Parallel Computation, Rice University, October 1995.

[15] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method for computing static single assignment form. In *ACM SIGPLAN POPL*, pages 25–35, 1989.

[16] Alexandre E. Eichenberger and Edward S. Davidson. Register allocation for predicated code. In *Proceedings of the 28th International Workshop on Microprogramming and Microarchitecture*, pages 180–191, 1995.

[17] Shimon Even. *Graph Algorithms*. Computer Science Press, 1979.

[18] J. Fisher. Trace scheduling: A general technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, 1981.

[19] F. Gavril. Algorithms on clique separable graph. *Discrete Mathmatics*, 19:159–165, 1977.

[20] Johnson Gillies, Roy Ju and Schlansker. Global predicate analysis and its application to register allocation. In *Proceedings of the 29th International Workshop on Microprogramming and Microarchitecture*, 1996.

[21] R.E. Hank. *Machine independent register allocation for the IMPACT-I C compiler*. PhD thesis, Department of Electrical and Computer Engineering, Unviersity of Illinois at Urbana-Champaign, May 1995.

[22] R.E. Hank. *Region-Based Compilation*. PhD thesis, Department of Electrical and Computer Engineering, Unviersity of Illinois at Urbana-Champaign, May 1996.

[23] Richard E. Hank, Wen mei W. Hwu, and B. Ramakrishna Rau. Region-based compilation: An introduction and motivation. In *Proceedings of the 28th International Workshop on Microprogramming and Microarchitecture*, pages 158–168, 1995.

[24] W.W. Hwu, S.A. Mahlke, W.Y. Chen, P.P. Chang, N.J.Warter, R.A. Bringman, R.G. Ouellette, R.E. Hank, T. Kiyohara, G.E. Haab, J.G. Holm, and D.M. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. *Journal of Supercomputing*, January 1993.

[25] Anne M. Holler Jack W. Davidson. Subprogram inlining: A study of its effects on program execution time. *IEEE Transactions on Software Engineering*, 18:89–101, feb 1992.

[26] R. Johnson and M. Schlansker. Analysis of predicated code. In *Micro-29, International Workshop on Microprogramming and Microarchitecture*, 1996.

[27] J.R.Allen, K.Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pages 177–189, 1983.

[28] V. Kathail, M. Schlansker, and B. R. Rau. Hpl playdoh architecture specification version 1.0. Technical Report HPL-93-80, HP Labs, Palo Alto, CA, February 1994.

[29] Philip Klein, Ajit Agrawal, R. Ravi, and Satish Rao. Approximation through multicommodity flows. In *31st Annual Symposium on Foundations of Computer Science*, volume II, pages 726–737, St. Louis, Missouri, 22–24 October 1990. IEEE.

[30] Eugene L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Halt and Rinehart and Winston, 1976.

[31] Adl-Tabatabai Lueh, Gross. Global ra based on graph fusion. In *LCPC'96, Workshop on Languages and Compilers for Parallel Computing*, August 1996.

[32] Guei-Yuan Lueh. Issues in ra by graph coloring. Technical Report CMU-CS-96-171, CMU, 1996.

[33] Guei-Yuan Lueh. *Fusion Based Register Allocation.* PhD thesis, CMU, 1997.

[34] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Minimizing register usage penalty at procedure calls. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 85–94, 1988.

[35] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Micro-25, Proceedings of the 25th International Workshop on Microprogramming and Microarchitecture*, pages 45–54, 1992.

[36] Norris and Pollock. Register allocation over the pdg. In *PLDI '94. Proceedings of the conference on Programming language design and implementation*, 1994.

[37] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithm and Complexity.* Prentice Hall, 1982.

[38] J. Park and M. Schlansker. On predicated execution. Technical Report HPL-91-58, HP Labs, Palo Alto, CA, may 1991.

[39] P. P.Chang and W.-W. Hwu. Inline function expansion for compiling C programs. In *Proceedings of the SIGPLAN '89 Conference on Programming language design and implementation*, pages 246–257, 1989.

[40] Todd A. Proebsting and Charles N. Fischer. Probalistic register allocation. In *PLDI '92. Proceedings of the conference on Programming language design and implementation*, pages 300–310, 1992.

[41] B. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. *Proceedings of the 27th Annual Symposium on Microarchitecture*, December 1994.

[42] B. Rau and J. Fisher. Instruction level parallel processing: History, overview and perspective. *J. Supercomputing*, 7:9–50, 1993.

[43] Intel Press Release. *Merced Processor and IA-64 Architecture*, 1998. http://developer.intel.com/design/processor/future/ia64.htm.

[44] Schlansker, Rau, Mahlke, Kathail, Johnson, Anik, and Abraham. Achieving high levels of ilp with reduced hardware complexity. Technical Report HPL-96-120, HP Labs, Palo Alto, CA, February 1996.

[45] Soffa, Gupta, and Ombres. Efficient register allocation via coloring using clique separators. *ACM TOPLAS*, May 1979.

[46] R. E. Tarjan. Decomposition by clique separators. *Discrete Mathmatics*, 55(2):221–231, 1985.

[47] Robert E. Tarjan. *Data Structure and Network Algoithms.* Society for Industrial and Applied Mathematics, 1983.

[48] R.A. Towel. *Control and Data Dependence for Program Transformations.* PhD thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1976.

[49] Trimaran Consortium. *Trimaran Web Site*, 1998. http://www.trimaran.org.

[50] N. J. Warter. *Modulo Scheduling with Isomorphic Control Transformations.* PhD thesis, University of Illinois at Urbana-Champaign, 1994.

[51] M. Wegman and K. Zadeck. Constant propagation with conditional branches. *ACM Trans. on Programming Languages and Systems*, 13(2):181–210, April 1991.

[52] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *TOPLAS*, 13(2):181–210, April 1991.