REGISTER ALLOCATION USING CYCLIC INTERVAL GRAPHS

by
Chandr.ka Mukerji

School of Computer Science
McGill University, Montreal

July 1994

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

Name CHANDRIKA MUKERJI

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

COMPUTER SCIENCE
SUBJECT TERM

0984
SUBJECT CODE

UMI

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS
Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION
General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523
Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS
Language
- General 0679
- Ancient 0289
- Linguistics 0290
- Modern 0291

Literature
- General 0401
- Classical 0294
- Comparative 0295
- Medieval 0297
- Modern 0298
- African 0316
- American 0591
- Asian 0305
- Canadian (English) 0352
- Canadian (French) 0355
- English 0593
- Germanic 0311
- Latin American 0312
- Middle Eastern 0315
- Romance 0313
- Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY
Philosophy 0422
Religion
- General 0318
- Biblical Studies 0321
- Clergy 0319
- History of 0320
- Philosophy of 0322

Theology 0469

SOCIAL SCIENCES
American Studies 0323
Anthropology
- Archaeology 0324
- Cultural 0326
- Physical 0327

Business Administration
- General 0310
- Accounting 0272
- Banking 0770
- Management 0454
- Marketing 0338

Canadian Studies 0385
Economics
- General 0501
- Agricultural 0503
- Commerce Business 0505
- Finance 0508
- History 0509
- Labor 0510
- Theory 0511

Folklore 0358
Geography 0366
Gerontology 0351
History
- General 0578
- Ancient 0579
- Medieval 0581
- Modern 0582
- Black 0328
- African 0331
- Asia, Australia and Oceania 0332
- Canadian 0334
- European 0335
- Latin American 0336
- Middle Eastern 0333
- United States 0337

History of Science 0585
Law 0398
Political Science
- General 0615
- International Law and Relations 0616
- Public Administration 0617

Recreation 0814
Social Work 0452
Sociology
- General 0626
- Criminology and Penology 0627
- Demography 0938
- Ethnic and Racial Studies 0631
- Individual and Family Studies 0628
- Industrial and Labor Relations 0629
- Public and Social Welfare 0630
- Social Structure and Development 0700
- Theory and Methods 0344

Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES
Agriculture
- General 0473
- Agronomy 0285
- Animal Culture and Nutrition 0475
- Animal Pathology 0476
- Food Science and Technology 0359
- Forestry and Wildlife 0478
- Plant Culture 0479
- Plant Pathology 0480
- Plant Physiology 0817
- Range Management 0777
- Wood Technology 0746

Biology
- General 0306
- Anatomy 0287
- Biostatistics 0308
- Botany 0309
- Cell 0379
- Ecology 0329
- Entomology 0353
- Genetics 0369
- Limnology 0793
- Microbiology 0410
- Molecular 0307
- Neuroscience 0317
- Oceanography 0416
- Physiology 0433
- Radiation 0821
- Veterinary Science 0778
- Zoology 0472

Biophysics
- General 0786
- Medical 0760

EARTH SCIENCES
Biogeochemistry 0425
Geochemistry 0996
Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES
Environmental Sciences 0768
Health Sciences
- General 0566
- Audiology 0300
- Chemotherapy 0992
- Dentistry 0567
- Education 0350
- Hospital Management 0769
- Human Development 0758
- Immunology 0982
- Medicine and Surgery 0564
- Mental Health 0347
- Nursing 0569
- Nutrition 0570
- Obstetrics and Gynecology 0380
- Occupational Health and Therapy 0354
- Ophthalmology 0381
- Pathology 0571
- Pharmacology 0419
- Pharmacy 0572
- Physical Therapy 0382
- Public Health 0573
- Radiology 0574
- Recreation 0575
- Speech Pathology 0460
- Toxicology 0383

Home Economics 0386

PHYSICAL SCIENCES
Pure Sciences
Chemistry
- General 0485
- Agricultural 0749
- Analytical 0486
- Biochemistry 0487
- Inorganic 0488
- Nuclear 0738
- Organic 0490
- Pharmaceutical 0491
- Physical 0494
- Polymer 0495
- Radiation 0754

Mathematics 0405
Physics
- General 0605
- Acoustics 0986
- Astronomy and Astrophysics 0606
- Atmospheric Science 0608
- Atomic 0748
- Electronics and Electricity 0607
- Elementary Particles and High Energy 0798
- Fluid and Plasma 0759
- Molecular 0609
- Nuclear 0610
- Optics 0752
- Radiation 0756
- Solid State 0611

Statistics 0463

Applied Sciences
Applied Mechanics 0346
Computer Science 0984
Engineering
- General 0537
- Aerospace 0538
- Agricultural 0539
- Automotive 0540
- Biomedical 0541
- Chemical 0542
- Civil 0543
- Electronics and Electrical 0544
- Heat and Thermodynamics 0348
- Hydraulic 0545
- Industrial 0546
- Marine 0547
- Materials Science 0794
- Mechanical 0548
- Metallurgy 0743
- Mining 0551
- Nuclear 0552
- Packaging 0549
- Petroleum 0765
- Sanitary and Municipal 0554
- System Science 0790

Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY
General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

Abstract

We propose the use of cyclic interval graphs as an alternative representation for register allocation. The "thickness" of the cyclic interval graph captures the notion of overlap between live ranges of variables relative to each particular point of time in the program execution. It is effective in capturing the regular periodic nature of loop-carried dependent live ranges found in loops.

Unlike Chaitin's approach where the spilling and coloring phase can alternate several times, we propose a two-step register allocation scheme based on the cyclic interval graph representation. The first step is the spilling phase. The spiller uses the exact times of overlap between intervals to discover opportunities to avoid spills by inserting register move instructions. The spiller is followed by a coloring phase. One of the coloring algorithms developed, the *fatcover algorithm*, makes use of the cyclic intervals to find a near-optimal register allocation for innermost loops which have no embedded flow of control.

As most scientific code spend a lot of time executing loop structures, it is most crucial to perform well when register allocating for it. A good spilling algorithm and a close to optimal coloring algorithm is invaluable in minimizing the cost that may be incurred while performing register allocation for loop structures. Minimization of spill code often greatly increases the performance of the code. Our proposed spilling and coloring scheme is very well suited to these loop structures, and could be used to reap maximum benefit when used in tandem with loop scheduling algorithms [NG93, Nin93].

A collection of real program loops are used to test the effectiveness of our approach. From our limited experimental results, we find that the spiller generates 37% to 56% fewer spill store instructions and about 90% fewer spill load operations in comparison to commercial compilers. We also observe that on the average the fatcover coloring algorithm requires 5.2% more registers than that required by an optimal coloring algorithm!

Even though we focus on loops which consist of single basic blocks in this thesis, we propose ways of extending our method to register allocate for loops having embedded control flow.

Résumé

Dans cette thèse nous proposons l'usage de graphes d'intervalle cyclique comme méthode de modélisation pour le problème de l'allocation des registres relié à la compilation. Un graphe d'intervalle cyclique se définit sur un espace temporel et l'"épaisseur" d'un tel graphe à un moment précis reflète le nombre de variables qui coexistent à ce moment exact. Ces graphes réussissent à modéliser la périodicité d'une boucle en plus de soutenir un méchanisme de représentation pour les variables dont la durée d'existence s'étend sur multiples itérations de la boucle.

Différente de la méthode de Chaitin où les phases de débordement et de coloration peuvent être répétées à plusieurs reprises, notre technique fondée sur les graphes d'intervalle cyclique ne requiert que deux étapes. L'étape de débordement s'effectue en utilisant les bornes précises de chevauchement entre intervalles. Ceci permet l'identification de transferts de registres qui limitent le nombre total de débordements. À la suite de l'étape de débordement vient l'étape de coloration. L'un des algorithmes présentés (l'algorithme *fatcover*) performe de façon presque optimale pour les boucles les plus emboîtées qui sont sans opérations de contrôle.

Étant donné que les programmes de nature scientifique écoulent une très grande partie de leur temps d'execution sur des boucles, il est primordial d'avoir un algorithme performant pour l'allocation des registres à l'intérieur des boucles. Un bon algorithme de débordement et un algorithme de coloration presque optimal sont indispensables pour minimiser l'implémentation des boucles. La minimisation du code de débordement représente un gain substantiel dans la performance des programmes. En particulier, notre méthodologie produit des gains quasi-optimaux lorque utilisée conjointement avec des algorithmes calculateur d'échéanciers, tel celui de [NG93, Nin93].

Une série de boucles soutirée de programmes existants est utilisée pour vérifier l'efficacité de notre approche. Pour le débordement, les résultats experimentaux indiquent que notre méthodologie génère de 37% à 56% d'instructions de storage en moins et environ 90%

d'instructions de chargement en moins par rapport aux compilateurs disponible sur le marché. De plus, en comparaison à un algorithme de coloration optimal, notre algorithme de coloration ne requiert que 5.2% de registres en surplus.

Cette thèse n'étudie que les boucles d'un seul bloc; cependant nous proposons des modifications qui permettraient des structures de contrôle plus complexes à l'intérieur des boucles.

Acknowledgments

I first met Prof. Guang Gao in the fall of 1990. He was teaching computer architecture – a subject in which I had no background. Amongst all the courses that I took that semester, Prof. Gao's lectures were undoubtedly the most captivating and dynamic. His enthusiasm for research and knowledge is very contagious and has been the key factor in motivating me to work in the area of high performance computing. Due to his enormous patience, frequent discussions and sound advice, I have gradually become more confident of my understanding of compiler issues. I am indebted to Prof. Gao for his unfailing support for the past three years.

This thesis could not have materialized without the help of Prof. Laurie Hendren. Ideas and algorithms mentioned in this thesis evolved during meetings with Prof. Hendren, Prof. Gao, Erik R. Altman – a fellow-student, and myself. Interactions with Erik has made work on this thesis enjoyable. Despite his busy schedule and an impending comprehensive exam, Erik has always found time to discuss concerns and questions about the thesis material. He also provided me with a super standalone system with his coloring algorithm that complemented mine. I am also very grateful to Dr. Qi Ning for answering my innumerable questions about his papers. He helped me select and process benchmarks for the experimental section of the research. Informal talks with Dr. Govindrajan Ramaswamy has provided different and new perspectives on various problems. Interactions with Roch Archambault, Dave Gillies, Bob Blainey and Andrew McLeod of the Language Technology Lab at IBM Toronto helped me to gain a more practical understanding of the register allocation problem. The technical and computer support provided by Jacek Sla! szewicz and Charles Arsenault have been invaluable.

I would also like to take this opportunity to thank all my friends. Shashank Nemawarkar has been a perennial source of encouragement and strength. Bhama Sridharan, Dr. Govindrajan, Eric Masson, Sanjay Gupta, my office-mate Larry Baer, and all the members of the Advanced Computer Architecture and Program Structure (ACAPS) lab as well as the VLSI lab have been more than helpful and have made the atmosphere at school very enjoyable for me.

Inspite of the distance, the support provided by my parents and brother has been most precious. Phone calls from my parents and visits from my brother helped to put life in perspective during trying times.

Contents

List of Figures

Chapter 1

Introduction to Register Allocation

1.1 General Introduction

With the advent of new architectures, more emphasis is being laid on the development of portable optimizing compilers. Such compilers comprise of two broad stages.

- The "front-end", which is machine independent, translates a high level source program into an intermediate form. Syntax checking is performed at this level.
- Next, the "back-end" performs various kinds of optimizations like dead code removal, jump optimization, loop transformations, and register allocation on the intermediate code. Finally, the intermediate language is translated into machine code. Of course, this stage requires knowledge of machine specific information.

Register allocation is thus a phase of the back-end of the compiler. This phase plays an important role in compiler optimization. In fact, for modern high-performance processor architectures, register allocation has been viewed as a technique which "adds the largest single improvement" among various compiler optimizations [HΓ90]. Technology advances in the past decade have widened the gap between the speed of the CPU and memory (DRAMs), and this gap (a form of the Von Neumann bottleneck) is expected to continue to grow [HJ91]. Therefore, the benefit of keeping variables in registers, which are high speed

on-chip memory, is increasing. Thus the impact of good register allocation strategies is also increasing.

In this thesis we focus on register allocation techniques. We shall assume the presence of RISC architectures as the target platform in all of our examples unless it is mentioned otherwise.

The reader will first be introduced to the register allocation problem in Section 1.2. Some terminologies which are used in the thesis are mentioned in Section 1.3. Section 1.4 discusses register allocation from a historical perspective. In Section 1.5 we discuss Chaitin's well-established register allocation method along with some limitations and extensions. This lays the foundation for and motivates a presentation of our framework for register allocation which is based on cyclic interval graphs as discussed in Section 1.6. Finally, Section 1.7 outlines the general structure of the rest of this thesis.

1.2 What Is Register Allocation?

In many compilers, after the back-end is invoked, the intermediate form generated by the front-end undergoes a transformation in which variables or operands of instructions are assigned "*pseudo registers*". At this stage it is assumed that the architecture has an infinite supply of pseudo or symbolic registers. This transformed intermediate code is used by the register allocation phase of the optimizing back-end of the compiler. Since architectures cannot have an infinite supply of registers, the pseudo registers used have to be mapped onto the actual number of registers available on the target machine. This is the first basic problem of register allocation.

The register set available on most RISC-based machines tend to be small. For example, IBM's RS6000 workstation has 32, 32-bit general purpose registers and 32, 64-bit floating point registers. Due to it's faster speed of access, we are tempted to hold as many variables as possible in registers. However, as there are only a limited number of registers available, this critical resource must be used judiciously. In a RISC architecture, memory load and store operations as well as all other (ALU) instructions are performed using registers. Assume that at a certain point of time we need to read in a variable, x, from memory. x has to be read into a register, but registers already have values in them. As program execution on most traditional architectures can not continue without x being read, we have to devise a way of making an occupied register available for it. One option is to store a variable that is being held in a register, to memory. This process is known as "*spilling*". Spilling frees up a register which can then be reassigned to the new variable, like x. Making a good choice of variables to spill is an important problem of register allocation.

In summary, we look upon the register allocation problem as the following two sub-problems :

- (A) Given a program, we would like to assign *the minimum number* of registers to the program variables. Regardless of the actual number of registers available in the architecture, we wish to reduce the number of required registers to a minimum possible number. This has some important applications. For example, when allocating registers interprocedurally it is beneficial to allocate a minimal number of registers to each procedure. This reduces the amount of register saving required at procedure call time, and can also improve interprocedural register allocation [SH89b, Cho88].

- (B) Unlike in (A), in this case we assume that there are k registers available on the target machine. The minimum number of registers required to map each variable to a register may exceed k. Hence, spill code has to be introduced and our objective is to keep the *total cost incurred by the spills to a minimum.*

The above mentioned informal problem statements are stated precisely in Chapter 2.2.

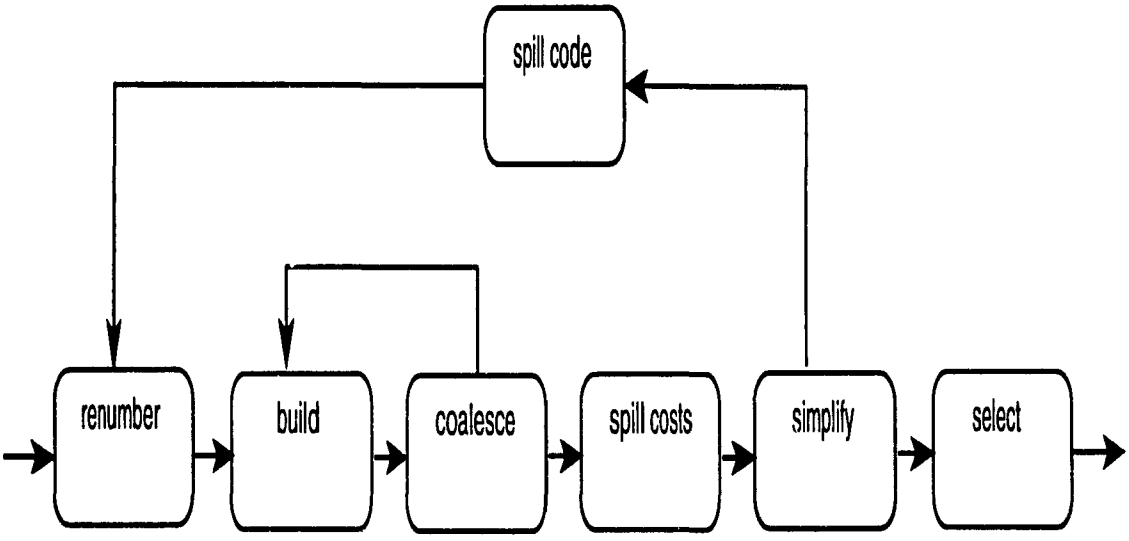


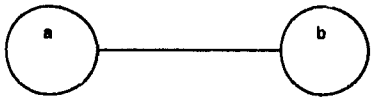
Figure 1.1: Chaitin's Global Register Allocator

1.3 Terminology

For the purposes of this thesis we define some terms. The piece of code shown below in Fig. 1.2(a) is used while defining some of the terms.

$a = \ldots\ (s_1)$
$b = \ldots\ (s_2)$
$\ldots = a\ (s_3)$
$\ldots = b\ (s_4)$
$\ldots = a\ (s_5)$

(a) Code Segment



(b) Interference Graph

Figure 1.2: Explaining Terminologies With An Example

- *Live* variable : A variable, a, is *born* at the point p where it is defined in a program and it dies at the point where it is last used. a is said to be live at points along those paths of the program starting at p where its value can be used. For instance, a is born in s_1 and dies in s_5. a comes alive at s_1 – that is at its' point of birth.

- *Live range* of a variable : is the duration of liveness of the variable. The live range extends from the point of birth of the variable till its point of death. In our example, a's live range extends between statements s_1 and s_5.

- *Def-Use* chain of a variable : The def-use chain of a variable is the *"set of uses, s, of a variable such that there is a path from p to s that does not redefine it"* [ASU88]. In our example, a is defined at s_1 and this definition is linked with its uses at statements s_3 and s_5 to create the def-use chain for a.

- *Interference* between variables : Two variables are said to interfere at a point p of the program if they are both live at p. In our example, variables a and b interfere.

- *(Register) interference graph* : is an undirected graph in *"which the nodes are symbolic registers and an edge connects two nodes if one is live at a point where*

the other is defined" [ASU88]. a and b of our example are two interfering nodes of the interference graph (Fig. 1.2(b)).

- *Degree* of a node : is the total number of neighbors of the node in an interference graph. Each neighbor of a node is connected to it with an edge.
- *Spill cost* of a node : is an attribute of the node. It is the cost incurred to store the variable to memory and reload it from memory.
- *Spill code* : consist of the group of instructions which store and load a spilled node to and from the memory respectively.
- *Basic block* : is the smallest sequence of statements which have no flow of control.
- *Global register allocator* : A global register allocator assigns registers to all the variables and compiler generated temporaries of an entire function or procedure. On the other hand, a local register allocator works at the basic block level.
- *Loop carried dependence* : is caused when a variable defined in one iteration of a loop is used by a following iteration of the loop.

1.4 History Of Global Register Allocation

A number of researchers have looked into the memory and register allocation problem [Ers90, Fre74, MB83]. We refer the reader to Chapter 7 for a detailed survey of work being done in this area. At this point we concentrate on providing a historical perspective of global register allocators.

John Cocke came up with the idea of transforming the register allocation problem to a graph coloring problem [Ken71, Bri92]. An interference graph is created such that the nodes (or vertices) of the graph are live ranges, and edges exist between those nodes which are live simultaneously. If there are k hardware registers available on the target architecture, the goal of the register allocator is to map the k registers to the nodes of the interference graph. This is accomplished by finding a k-coloring of the (interference) graph. Each node of the graph is colored with one of the k registers and no two interfering nodes can receive the same color.

While working on the PL.8 compiler, Gregory Chaitin at IBM, T.J. Watson Research Center at Yorktown Heights, developed the first complete global register allocator. Chaitin's global allocator also uses the graph coloring approach, and spilling and coloring is performed

in a coordinated fashion [CAC+81, ASU88]. This work laid the foundation for most future work done in the area of register allocation. Over time, Chaitin improved the heuristics used by his original algorithm [Cha82]. Others, notably Briggs, Bernstein and, Nickerson have improved and extended Chaitin's algorithm [BCKT89, BGG+89, BCT92, Nic90]. One of the more significant improvements included reducing the amount of spill code generated [Bri92]. Today, widely used optimizing compilers like IBM's XLC and the GCC avail of Chaitin's method of register allocation.

While Chaitin worked on the interference graph approach to register allocation, Fred Chow and John Hennessy developed the priority based global register allocator [Cho83, CH84, Cho90]. This allocator is also based on the graph coloring method, but differs significantly from Chaitin and his successors work in the spilling techniques used. Extensions and modifications of this work have been reported in [Cho88, GSS89, LH86].

1.5 Evolution Of Chaitin's Global Allocator

As the work presented in this thesis is based on some limitations of Chaitin's approach, we shall first review his method and then motivate our approach in Section 1.6.

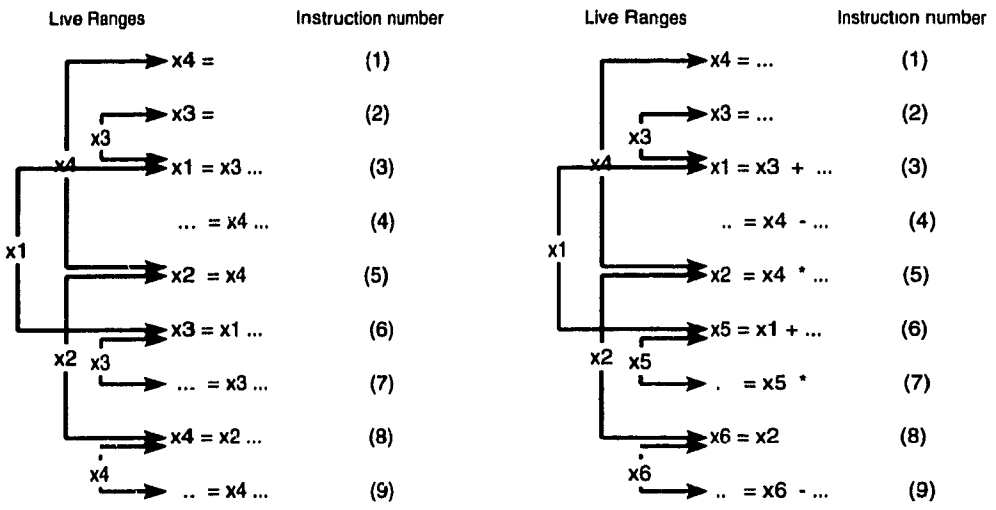
1.5.1 An Informal Description Of Chaitin's Algorithm

The essence of Chaitin's algorithm is probably best understood through an example. After presenting an intuitive description of the algorithm we will go through an example which will help illustrate the details of the algorithm.

Assume that the target architecture has k available registers. Given an input set of live ranges, the algorithm first builds an interference graph, and annotates the nodes with information that is used by the coloring phase. Next, the interference graph is colored. This phase is the core of the allocation algorithm. The interference graph is repeatedly reduced by removing nodes which have a degree **strictly less** than k. The reason is because if a node, n, has at most $(k-1)$ neighbors then it is always possible to color n with a color distinct from it's neighbors.

If at some point the graph can't be reduced as all the nodes in the partially reduced graph has $degree \geq k$, spill code has to be introduced. A node is chosen to be spilled and spill code is generated for it. A node having a lower spill cost is a better candidates for a spill than a node having a higher spill cost. The interference graph is then reconstructed to

reflect the new conflicts of the spilled live range. The coloring process is rerun on the new interference graph. This two step spilling and coloring process is repeated until a k coloring of the interference graph is found.



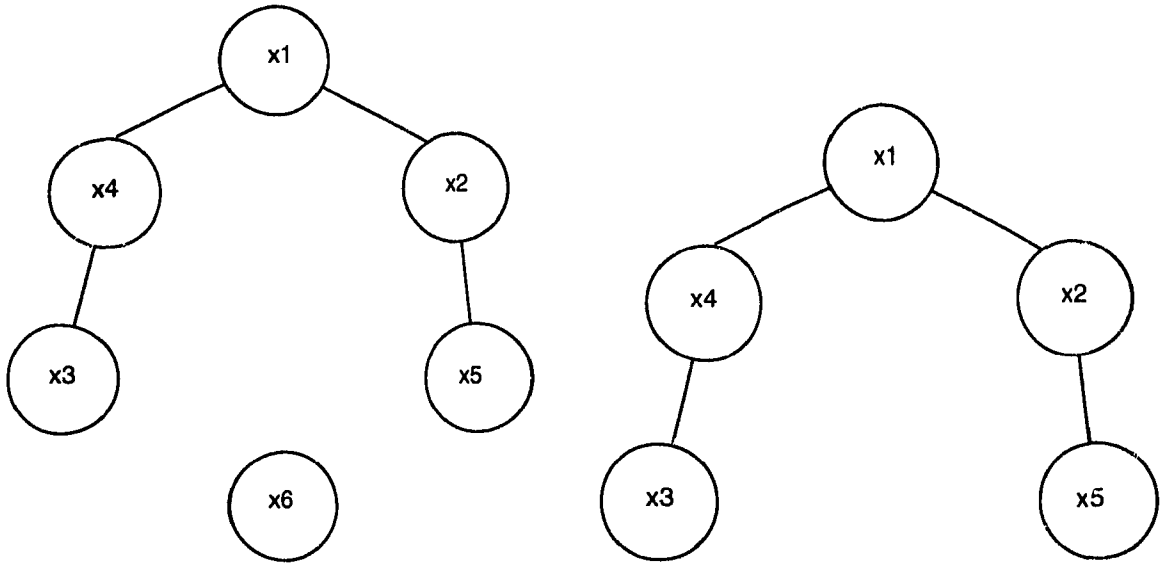
(a) Original Code With Live Ranges (b) Renumbering *x3* and *x4*

Figure 1.3: Using Chaitin's Algorithm On An Example

1.5.2 The Original Algorithm By Chaitin

Fig. 1.1 illustrates the flow diagram of Chaitin's register allocator. This diagram is similar to the one presented by Preston Briggs in his thesis [Bri92]. An interference graph is the expected input of the algorithm and, a colored interference graph that may have undergone transformation due to spill code insertion is the output. We use the example in Fig. 1.3 to understand the various steps of the algorithm illustrated in Fig. 1.1.

- Renumber : All the live ranges of a function are given unique names. This allows greater flexibility in assigning different registers to live ranges arising from a single



(c) Interference Graph (d) After Coalescing

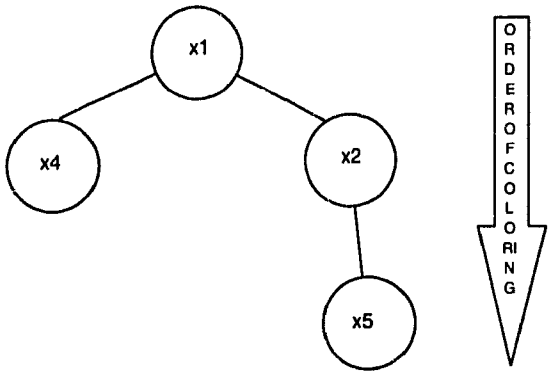
Figure 1.4: Using Chaitin's Algorithm On An Example

variable. If renumbering were not done, then all the live ranges of a single variable would have to be assigned the same register and this creates an unnecessary constraint.

Fig. 1.3(a) shows a code segment and the live ranges of the program variables. *x3* and *x4* have two live ranges each. Since the numbers of all the live ranges must be unique the live range of *x3* that extends between statements 5 and 7 is renumbered as *x5* in Fig. 1.3(b). Similarly the the live range of *x4* between statements 8 and 9 is renumbered as *x6*.

- Build : An *"interference graph"* graph is built based on the live range information. Recall that each node in the graph corresponds to a live range of a program variable. An edge between two nodes in the graph represents interference between the two live rages. This means that, the two interfering live ranges cannot share the same register and must be assigned different registers.

  The interference graph of our example is shown in Fig. 1.4(c).



Node	Color
x3	R1
x5	R1
x2	R2
x4	R2
x1	R1

(e) Reduced Graph when k=2 (f) Final 2-coloring Of Graph

Figure 1.5: Using Chaitin's Algorithm On An Example

- Coalesce : Unnecessary copy instructions are eliminated if the source and result live ranges do not interfere. This phase reduces the number of nodes of the graph.

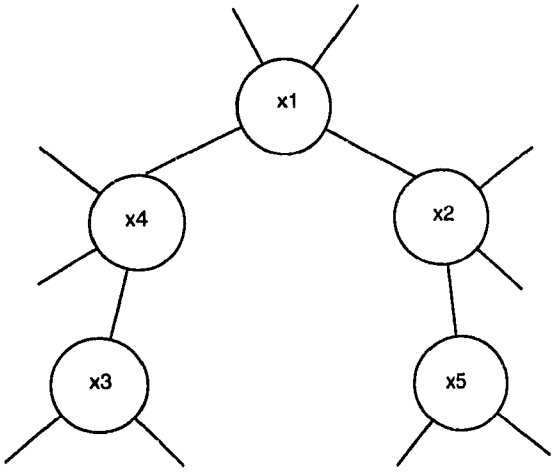
  In our example statement 8 is a copy instruction (Fig. 1.3(b)). *x6* can be coalesced with *x2* as the result and source live ranges don't interfere. The coalesced node *x6* is removed from the interference graph as shown in Fig. 1.4(d).

- Spill Costs : Each node is annotated with a spill cost, which is used to choose a node if one needs to be spilled. Spill costs are calculated on the basis of heuristics. According to one such heuristic, live ranges of variables which are used in deeply embedded loop nests are given a high spill cost as we would prefer to hold it within a register rather than load and store it during each iteration of the loop [Bri92].

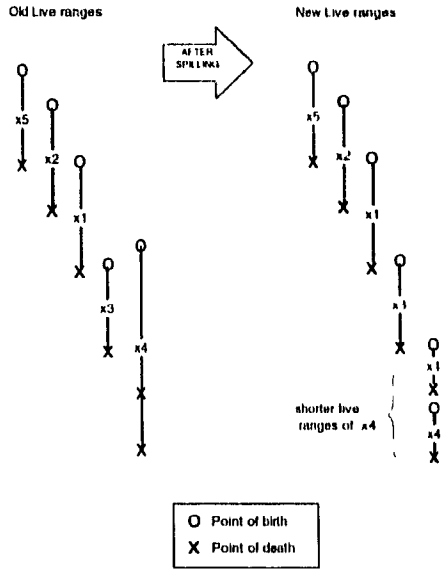
  For the sake of simplicity, let us assume that all the nodes of our example have the same spill cost.

- Simplify : This phase reduces the interference graph and is the key to the coloring process.

  If there are k registers available in the architecture, then nodes having a degree $< k$ are repeatedly removed from the graph. The degree of a node is given by its number of neighbors. The rationale behind removing nodes having degree less than k is if the node that is being removed has (at most) *k-1* neighbors then each of the neighbors



(g) Modified Interference Graph



(h) Spilling When k=3 (Only Some Live Ranges Shown)

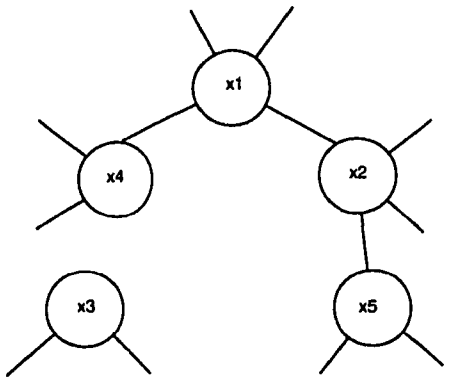
Figure 1.6: Example For Chaitin's Algorithm Continued

can receive a different color. This leaves the last color, of the k colors, for the node being removed.

If we encounter the situation where there are no nodes in the graph having a degree less than k, then a node has to be chosen as a spill node. Heuristics involving spill costs of nodes guide the choice of a spill node. The chosen node is marked to be spilled and is removed from the graph. The graph reduction process then continues on the resultant reduced subgraph.

At the end of this phase, the graph is completely reduced so that it becomes empty. Generally, three to five iterations of the algorithm may be needed to k-color graphs of real world benchmarks. Only a few benchmarks require more than ten to twelve iterations before a k-coloring is found [Cha82, ABGM93].

For the purpose of our example, assume that there are two available hardware registers, *r1* and *r2*. We need to simplify the interference graph of Fig. 1.4(d). The nodes *x3*, or *x5* can be chosen to be removed from the graph as they have the highest degree < 2.



(i) Interference Graph Updated After Spilling

Figure 1.7: Example For Chaitin's Algorithm Continued

Let's say that we remove *x3*. This leaves us with the partially reduced interference graph shown in Fig. 1.5(e). Now, we must remove *x5* as it is the only node whose degree < 2. Then, we have an option of removing either *x2* or *x4* and we first choose *x2* and then *x4*. Finally *x1* is removed and we are left with an empty graph.

- Select : This phase assigns colors to the nodes in the reverse order in which they have been reduced. Each node gets a color distinct from its' neighbors. Assigning a color is equivalent to assigning a register and henceforth we shall use colors and registers interchangeably.

  The nodes of our example are colored in the reverse order in which they are removed from the graph. Fig. 1.5(f) shows the final 2-coloring of the graph.

- Spill Code : If at any point, the interference graph fails to reduce as the degrees of all the nodes of the subgraph are greater than k, we have to introduce spill code. For each variable that is chosen to be spilled, a store instruction is added after the point of definition of the variable and, load instructions are inserted before every point of use of the variable in the live range. This splits one live range into two or more shorter live ranges.

  For instance, let us increase the number of interferences of the nodes of the graph of Fig. 1.4(d). Fig. 1.6(g) shows the modified graph. If only two hardware registers are available to color the graph, then the simplify phase can not remove any of the

nodes as their degrees are all > 2. At this point, we have to choose a node to spill. Assuming that the spill cost of all the nodes are the same, let's say we choose to spill *x4*. Spill code may be inserted for *x4* so that the single long live range for *x4* is replaced by several several shorter live ranges as shown in Fig. 1.6(h).

The change in the live range information changes the structure of the interference graph as shown in Fig. 1.7(i). So, the interference graph is modified and the algorithm is executed on the new interference graph.

Spill code is not inserted after each spill node is chosen in the simplify stage. Instead, chosen spill nodes are marked and spill code is inserted all the spill decisions have been made in this phase. This of course, avoids repeated reconstruction of the interference graph. Instead of splitting a spilled node into several smaller live ranges immediately, we choose to depict these smaller live ranges as one large conglomerate node till the end of the phase. This makes the graph imprecise and it doesn't necessarily reflect the true state of the program at all times. In the context of our example, two shorter lie ranges replace the live range for *x4* (Fig. 1.6(h)), however this change is not reflected in the interference graph until the end of the spilling phase. At the end of this phase, the interference graph is modified to accurately reflect the changed live range information due to the spilled live ranges.

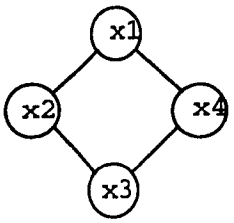
1.5.3 Brigg's Improvement To Chaitin's Algorithm

Essentially, Chaitin reduced the register allocation problem to a graph coloring problem. However, the k-coloring of a graph, where k is assumed to be the number of registers available on the target machine, is known to be a classical NP-complete problem [GJ79]. Hence, the coloring algorithms employ heuristics and may not provide optimal results in all cases and, Chaitin's algorithm is no exception to this rule.

As an example, Ken Kennedy, of Rice University [Bri92], constructed a very simple diamond shaped interference graph which causes Chaitin's algorithm to perform sub-optimally in two respects :

1. Failure to find a k coloring in some cases even when one exists.
2. As a side-effect of being unable to find a k-coloring, unnecessary spill code is generated.

We provide such a similar diamond shaped graph in Fig. 1.8. Let us assume that we have only two colors, red and green, available for this graph. By inspection, we give *X1* the



An Interference Graph

Figure 1.8: Sub-optimal Performance Of Chaitin's Algorithm On A Small Example

color red. *X2* and *X4* can be assigned the same color as they do not interfere with each other, but they must receive a color different from the one assigned to *X1*. So, the only color that can be given to *X2* and *X4* is green. Finally, *X3* is colored red.

Unfortunately, Chaitin's algorithm fails to find a 2-coloring for this graph. As there are no nodes in the graph whose degree < 2, we will be forced to spill a node. Assuming the same spill cost for all the nodes, we choose to spill *X1*. Once *X1* has been removed from the graph, the other nodes can be reduced very easily. This example does not claim to show that a 2-coloring of this graph can never be found. It could, using some heuristic other than the one used by Chaitin [MB83].

If we were to ask ourselves why Chaitin's heuristic fails to color this graph with two colors, we notice that the heuristic used in reducing the interference graph is the root cause of this problem. Since nodes with degree strictly $<$ than k can be removed from the graph, it is implicitly assumed that every node that interferes with the node being removed, say n_{remove}, is going to be assigned an *unique color*. Different colors don't necessarily have to be assigned to neighbors in all cases, they could share colors. If the neighbors of n_{remove} do not interfere with each other, then there is no reason not to assign them the same color. Instead, Chaitin chose to be pessimistic and assumed that all neighbors of n_{remove} conflict.

Preston Briggs suggested a very simple and elegant solution to that overcomes this suboptimal behavior. Fig. 1.9 is taken from his thesis and shows his suggested improvement to Chaitin's algorithm. Notice that the only difference between Fig. 1.1 and Fig. 1.9 is that the spill code decision is made after the select phase instead of after the simplify phase. With this improvement, the simplify phase doesn't make any spill decisions at all. Nodes are removed from the graph even if their degree are $\geq k$. Spill decisions are delayed till the

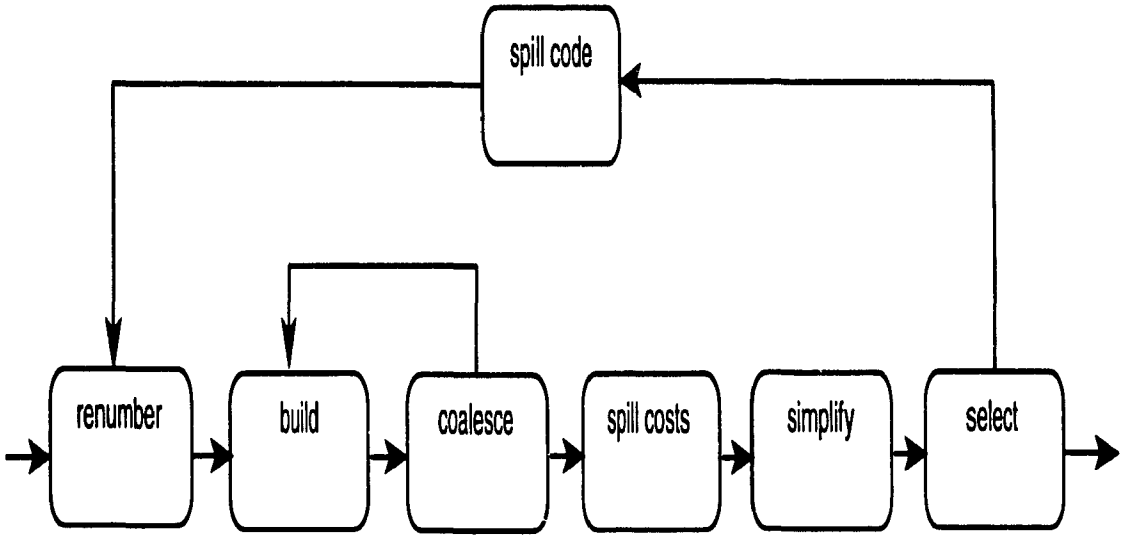


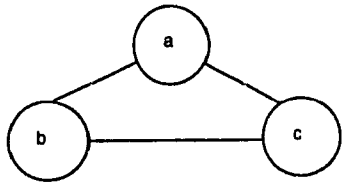
Figure 1.9: Briggs Improvement to Chaitin's Algorithm

select phase. When the select phase is invoked, colors are assigned to nodes in the reverse order in which they have been reduced and, a node, say n_{color}, is assigned a color different from all its neighbors. This is the crux of the solution - we are no longer bothered if n_{color}'s neighbors have been assigned distinct colors, all that we are concerned with is that n_{color} receive a color distinct from all its' neighbors (as n_{color} interferes with its neighbors). This leaves us the opportunity to color neighbors of a node with the same color, and of course, this leads to the reduction of unnecessary spill code.

Keeping Brigg's modification in mind, we take another look at the example in Fig. 1.8. Instead of spilling *X1* when there are only two colors available, Briggs removes it from the graph and continues the simplification process. Assume that we remove node *X2*, then X4 and finally *X3*. Then the nodes are colored. *X3* receives the color red. *X4* is colored green as it interferes with the red node *X3*. Next, *X2* is also colored green as it interferes with *X3* but not with *X2*. When we encounter the node *X1* we notice that it can be assigned the color red as it is different from the colors assigned to its' neighbors. This saves us from spilling a node.

```
for () {
   b = ...   (1)
     = c     (2)
   a =       (3)
     = b     (4)
   c =       (5)
     = a     (6)
}
```

(a) Loop1



(b) Interference Graph

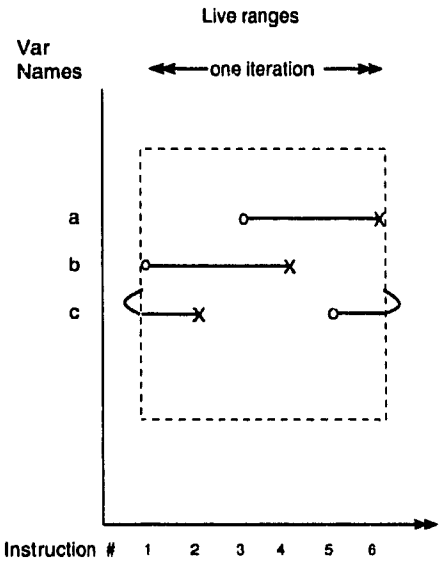
Figure 1.10: Impreciseness Of Interference Graphs

1.6 Motivating Our Approach

Chapters 1 through 5 focus on register allocation for loop structures, especially *"perfect"* loops. In this thesis we use the word "perfect" to mean basic blocks which do not have any embedded flow of control.

Numerically intensive real world benchmarks tend to have a very high number of perfect innermost loops [Huf93] and hence we zero in on the allocation problem for these structures. Huff studied DO loops in the Lawrence Livermore, SPEC89 Fortran and the Perfect Club benchmarks. The DO loops that he focused on had acyclic loop bodies with no procedure calls, and assigned or computed goto statements. Surprisingly enough, out of the 1525 loops that he isolated, 23% of them had loop carried dependences and no conditionals, while 70% of them had neither loop carried dependences nor any conditionals. Both these kinds of loops fall under our category of perfect loops. Due to the high percentage of occurrence of perfect loops they merit special consideration. These programs spend a lot of time executing the loop structures, therefore it is very important to obtain a good register allocation for this structure so that the best possible speedup can be achieved.

While studying Chaitin's interference graph coloring approach to register allocation, we investigated a representation different from the interference graph. The representation chosen by us works well for perfect loop structures. Here are our motivations :

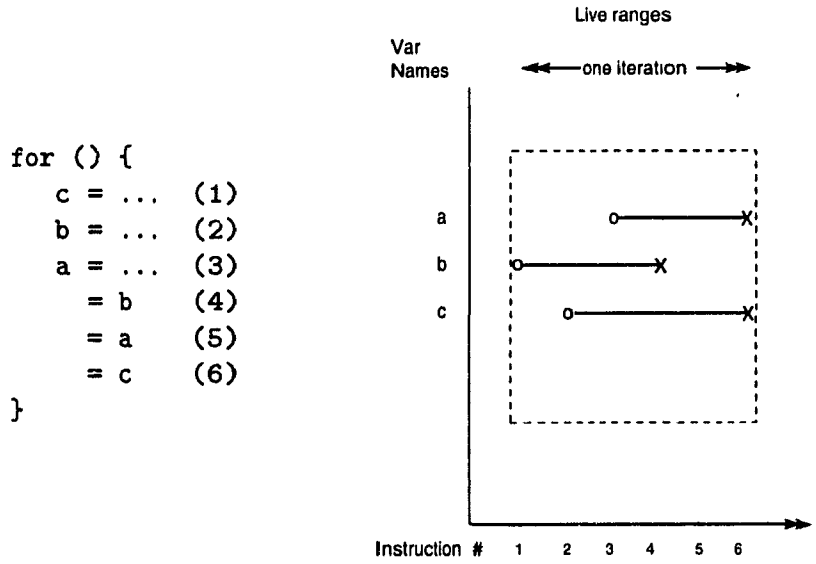


(c) Interval Graph For Loop1

Figure 1.11: Impreciseness Of Interference Graphs Contd

- From our point of view, interference graphs do not encode any notion of the relative time of overlaps between live ranges. As an example, we refer to the two loop bodies shown in Figs. 1.10(a) and 1.12(d). The live range of c in loop2 (of Fig. 1.12(d)) extends from instructions 2 through 6 of each iteration of the loop, while c in loop1 (of Fig. 1.10(a)) is defined at instruction 5 of one iteration and used by instruction 2 of the following iteration. The live range of c of loop1 is split into two segments per iteration as is seen in Fig. 1.11(c). The interference graph created for both the loops **are exactly the same** (Fig. 1.10(b)). However, when we see the live ranges of the loops in Figs. 1.11(c) and 1.12(e) we get a precise picture of the times of overlap of the various intervals.

- The exact times of overlap of live ranges is very useful in developing effective coloring and spilling heuristics. This is particularly true when one considers how to effectively model the live range of a loop variable: its lifetime may cross the boundary of iterations, and it may be defined and used repetitively at regular intervals. Let



(d) Loop2 (e) Interval Graph For Loop2

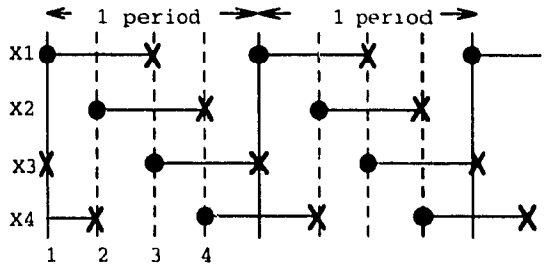
Figure 1.12: Impreciseness Of Interference Graphs Contd

us consider another example to illustrate this case. Fig. 1.13(a) shows a loop with n iterations. Four scalar variables are defined and used in the loop: $X1$ - $X4$. Note that in the case of loops each variable has a sequence of live ranges that correspond to different iterations of the loop. For example, the live range of the variable $X4$, can be split into several segments. For the first iteration, $X4$ is defined outside the loop and dies at instruction 2 within the loop. This is one section of $X4$'s live range. In addition, for each iteration i of the loop, $X4$ is defined in instruction 4 of iteration i, and is live between this definition and the last use in instruction 2 of the following iteration $i+1$. There is a similar situation for $X3$. In order to accurately capture this information in our approach, we would like to find a representation that incorporates the regular periodic nature of variables that are defined in some iteration i and last used in some later iteration i'.

Fig. 1.13(b), shows the *interval graph* for the program segment. The X axis represents the instruction numbers of the code, while the Y axis represents the variables of the program. The solid circles of the diagram illustrate the points of definition while the crosses illustrate the points of last use. For example, $X1$ is defined in instruction 1 and last used in instruction 3. Note that the lifetimes of each variable are represented

```
for i = 1 to n {
  X1 = X3 * 10;  (1)
  X2 = X4 * 20;  (2)
  X3 = X1 + 5;   (3)
  X4 = X2 + X3;  (4)
}
```

(a) A Loop Program



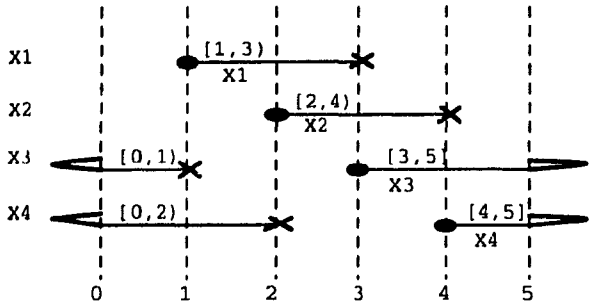
(b) An Interval Graph

Figure 1.13: Introducing An Interval Graph

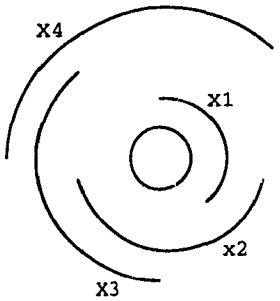
by a sequence of intervals, one interval for each iteration.

As illustrated in Fig. 1.13(b), the live range of a loop variable can be represented as a *periodic interval*: a sequence of lifetime intervals that are equally spaced in time by some *period*. Such a periodic interval can be characterized by the interval corresponding to one period. For example, the live ranges of variables $X1$ $X4$ in Fig. 1.13 (b) have a period of one iteration. The live ranges of variables $X1$ and $X2$ do not extend across the boundary between iterations, therefore, they each can be expressed as one interval, i.e. $X1$: $[1:3)$, $X2$: $[2:4)$. The variables $X3$ and $X4$, however, are defined in one iteration and used in the next. Therefore, for convenience, we represent its live range as a pair of two intervals, i.e. $X3$: $([0:1),[3:5])$ and $X4$: $([0,2),[4,5])$, where the interval $[0:1)$, for $X3$ as an instance, can be considered an extension of the interval $[3:5]$ that is wrapped around to fit in one period. We call such a "wrapped" interval a *cyclic interval*. In Fig. 1.14(c), we show the cyclic interval graph representation for Fig. 1.13(b). The numbers 0 and 5 do not correspond to any instructions but merely provide a joining point for two successive iterations.

- Another weakness of the interference graph approach is the potential expense required to to rebuild and recolor the interference graph after spill code has been introduced. Based on the cyclic graph representation we have devised a two step method of register allocation, the first step involves the spilling process and in the second step the interval graph is colored. Intuitively, the "thickness" of each point in the cyclic interval graph captures information about overlapping live ranges of variables at a



(c) A Cyclic Interval Graph



(d) A Circular Arc Graph

Figure 1.14: Cyclic Interval Graphs

particular location in the program. The points of the graph which are the thickest — the "fat spot" — is crucial in developing our new heuristic algorithms. A detailed description of the algorithm developed on the basis of the fatspots can be found in Chapter 3.2 and, Chapter 2 details our two-step methodology and explores the advantages offered by cyclic intervals graphs in detail.

The possibility of using interval graphs as a model for register allocation was noted in [Tuc75, Tuc84]. However, to our knowledge, previous research was theoretical in nature and mainly focused on the algorithmic aspects of the interval graph model, while we are primarily interested in the feasibility of using interval graphs in register allocators of real life compilers.

1.7 Summary Of Chapter And Structure Of Thesis

In this chapter we provide the reader with a background on Chaitin's allocation strategy based on the interference graph coloring method. We also discussed some limitations of Chaitin's method and proposed some advantages offered by the cyclic interval graph representation for register allocation for perfect loop blocks.

In Chapter 2, cyclic and non cyclic intervals are formally defined and the properties of specific cyclic interval graphs are explored. A formal problem statement is also provided.

Chapter 3 discusses heuristic algorithms developed for minimum register allocation for perfect loop blocks. Chapter 4 describes a new spilling algorithm, the *sweep and split* algorithm, that is used to transform an interval graph to one whose thickness is k as this is the form in which the colorer expects to receive the interval graph. Chapter 5 reports experimental results of our two step approach that uses heuristic graph coloring algorithms mentioned in Chapter 3.

Since our allocation strategy works well for perfect loop structures that is what we have concentrated on in this thesis. However, we have given thought to ways of extending our method to other structures like loops having embedded flow of control or loops having dependences which extend beyond one iteration. We describe difficulties we encountered in creating such an extension as well as some plausible solutions in Chapter 6.

Finally, Chapter 7 concludes the thesis with a summary of related work done in the area of register allocation. This helps to put our work in a broader perspective.

Chapter 2

Introduction And Background

This chapter formally introduces the reader to the cyclic interval graph representation that forms the basis of our register allocation approach. It is crucial in that it lays the foundation for all the following chapters of the thesis.

Section 2.1 describes cyclic interval graphs and their properties. Then Section 2.2 formulates the register allocation problem in terms of the cyclic interval graph representation. Finally, in Section 2.3 we argue that our chosen representation is beneficial and absolutely crucial to the development of the algorithms used in our spilling and coloring phases.

2.1 Cyclic Interval Graphs

In Chapter 1.6, we introduced the concept of cyclic interval graphs with the help of an example shown in Figs. 1.13 and 1.14. Now, we shall generalize the description of the interval graph and, make explicit any assumptions which we make while interpreting the graph.

In general, we use the following conventions in our cyclic interval graph representations. Let $t_0, t_1, \ldots$ be the starting *time points* of a sequence of machine operations. Without loss of generality, we use non-negative integers for the time points. We use $[t : t']$ to denote the interval between t and t' including both end points. The notation $[t, t')$ denotes the same interval but with the end point t' left out.

We assume that each machine operation is in the form of a quadruple, e.g. `x = y + z`, which begins at some time point t. To be precise, we say that variable `x` is defined at time

point t. The live range of x will continue to the time point t', $(t' > t)$, where it is last used in a statement, e.g. `u = x + v`. After time t', the value in x is no longer live. In this paper, we define the *lifetime interval* of x to be $[t, t')$. When no confusion may occur, we use the terms *interval* and *lifetime interval* interchangeably. The relation between the live ranges of a set of variables is completely defined by the corresponding set of lifetime intervals.

Our problems are related to the class of *circular-arc graph coloring* problems [Kle69, GJMP80]. A graph G is called a circular-arc graph if its vertices can be placed in a one-to-one correspondence with a set of circular arcs of a circle in such a way that two vertices of G are joined by an edge if and only if the corresponding two arcs overlap one another. In Fig.1.14 (d), we show the circular-arc graph representation of our example in Fig. 1.13(b). Intuitively, one can think of "bending" each of the interval into an arc. Since the intervals are periodic, we can fit them into one circle. Theoretically, the problem of determining a k-coloring for a circular arc graph with n arcs has a complexity of $O(nk!klogk)$ [GJMP80].

As in any general graph coloring problem, finding the minimum coloring of a cyclic interval graph is NP-hard [GJMP80]. For our purpose of register allocation, it is most important to use the information provided in the interval graph as guiding heuristics for our algorithmic solutions.

Now, we take the opportunity to introduce some definitions which will be used at some points of the thesis.

Definition 2.1.1 *A time t is* covered *by an interval $I1 : [t1, t1')$, if $(t1 \leq t < t1')$, or by an interval $I1' : [t1, t1']$ if $(t1 \leq t \leq t1')$, or by a cyclic interval $I2 : ([t1, t1'), [t2, t2'])$, if t is covered by either $([t1, t1')$ or $[t2, t2'])$.*

Definition 2.1.2 *Two intervals $I1, I2$* overlap *if there exists a time t that is covered by both $I1$ and $I2$.*

2.2 A Formal Problem Statement

Having been introduced to the representation that forms the basis of our allocator, we formulate the register allocation problem in context of cyclic interval graphs.

- Problem 1 (Finding a Minimum Coloring of a Cyclic Interval Graph):
  Given a set of live ranges represented as a cyclic interval graph G, find a minimum

register (color) assignment for the intervals in G such that overlapping intervals are assigned different registers.

Chapter 3 deals entirely with this problem.

- Problem 2 (Finding a k-coloring of a Cyclic Interval Graph with Minimum Spilling Cost):
  Given a set of live ranges represented by a cyclic interval graph G and a set of k registers, find an assignment of the k registers for the intervals in G. Introduce spill code when necessary, and keep the spill cost to a minimum.

  This problem is given full attention in Chapter 4.

These problem statements are a refinement of the ones stated in Chapter 1.2.

2.3 Features Of Cyclic Interval Graphs

Having understood how to interpret and read an interval and a cyclic interval graph, we elaborate on the advantages that this representation offers over the traditional interference graph. While pointing out the differences between Chaitin's approach and our own, we will frequently refer to phases of Chaitin's algorithm (Fig. 1.1 in Chapter 1.2).

1. A two-step allocation approach :
   Now, that we have an interval graph, we are interested in fast heuristic methods which find a k-coloring quickly, and generates efficient code for spilling when necessary.

   We have observed that the number of minimum registers needed for a cyclic interval graph is related to the thickness of the graph, which we will formally define below[1].

   Definition 2.3.1 *The* width *of a cyclic interval graph G at time t, written as width(G,t), is the number of intervals covering t.*

   Definition 2.3.2 *The* maximum width *of a cyclic interval graph G, written* $W_{max}(G)$, *is the maximum width*(G,t), *for all t which is covered by some interval in G. The* minimum width *of a cyclic interval graph G, written* $W_{min}(G)$, *is the minimum width*(G,t), *for all t which is covered by some interval in G.*

[1] Parts of this section has been excerpted from [HGAM92].

Now, we state the following theorems about the number of colors required to minimally color acyclic and cyclic interval graphs.

The following theorem addresses the problem of optimal coloring of acyclic interval graphs.

Theorem 2.3.1 *Let G be an interval graph containing no cyclic intervals. Then G is optimally colorable with $W_{max}(G)$ colors.*

Proof: First, it is obvious that G cannot be colored with less than k colors. Now let us complete the proof by sketching an algorithm (called *left edge* algorithm[2] [HS71]) which will guarantee to find the optimal coloring of G. Assume G spans from time 0 to time n. Starting from the left end (at time, $t = 0$), move from left to right along the time line. For each interval, I, which ends at t, release its color back to the pool of free colors. For each interval I beginning at t, give t a free color which is not being used by any interval covering t. Initially, the pool contains $k = W_{max}(G)$ free colors. Since there will never be more than $W_{max}(G)$ intervals covering any time t, the algorithm will successfully find a k-coloring for G. ■

For a cyclic interval graph G, $k = W_{max}(G)$ may not be enough to color G. This is due to the constraints caused by the cyclic intervals. However, we can establish the following upper bound:

Theorem 2.3.2 *Let G be an interval graph containing cyclic intervals. Then G is optimally colorable with $W_{max}(G) \leq k \leq W_{max}(G) + W_{min}(G)$ colors.*

Proof: First, it is obvious that G cannot be colored with less than $W_{max}(G)$ colors. Now let us complete the proof by sketching an algorithm which will guarantee to find the coloring of G in $W_{max}(G) + W_{min}(G)$ colors. Cut G at the point where it has the minimum width $W_{min}(G)$. Take the intervals covering the cutting point out of G and call the remaining part G'. Obviously, we can now treat G' as a non-cyclic interval graph. Coloring G' with the left-to-right algorithm guarantees it to be colored with no more than $k' = W_{max}(G)$ colors (Theorem 2.3.1). Then, it is trivial to see that we can use $W_{min}(G)$ more colors to color the removed intervals. ■

Since we are able to predict the upper and lower bound of the colors that will be needed to color an interval graph, we use this information to device a two step allocation policy as shown in Fig. 2.1. Assume that the number of available registers is k. According to Theorems 2.3.1 and 2.3.2 the thickness of the interval graph provides us with the number of colors which will be required to color it. If the

[2]A description of the algorithm can be found in Appendix A.

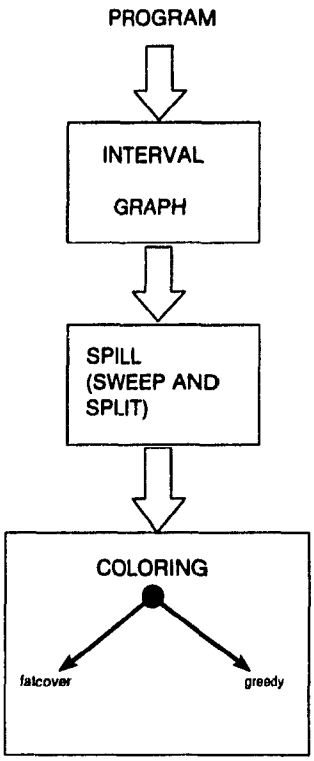


Figure 2.1: Flow Diagram For Register Allocation Based On Cyclic Interval Graphs

number of colors needed is greater than k, then we have to transform the interval graph so that it becomes k-colorable. As we have a priori knowledge of the number of intervals to spill, we perform the spilling phase first and only once to transform the interval graph so that its' thickness is $\leq k$ as expected by the colorer. Then the coloring phase uses k or fewer registers to color the transformed graph. This dual step process avoids the potential expense that may be required to rebuild and recolor the interference graph after the introduction of spill code in Chaitin's approach.

Notice that it is not easy to determine a lower bound on the number of colors that will be needed to color an interference graph. In order to determine the lower bound, we would have to find the largest clique in the interference graph and once again, we encounter an NP problem [GJ79]. Or we would have to use a backtracking algorithm to actually color the interference graph before obtaining an answer. This makes

it difficult to implement a two step approach like ours on the interference graph representation.

2. Use of explicit timing information provided by interval graphs :
   The interval graph representation encodes the exact times of overlap amongst the various intervals. This makes it easy to assign colors to intervals using the left edge algorithm (Appendix A) for instance. Since the interval graph can be swept in any direction along the time line and fatspots of the graph are easily identifiable, heuristics can very naturally use this information to choose and prioritize intervals which are to be assigned colors. We can also keep track of the times when registers (or colors) are free and busy and, this information can be used as heuristics by the colorer. Interval graphs provide a very convenient and natural framework to identify intervals which can share colors.

   Unfortunately, the interference graph doesn't include time in its' representation. Hence, timing information can not be availed of by heuristics used to choose nodes to color or spill. For instance, it is not possible to directly know which nodes are live simultaneously at a point of time. This make it difficult to detect nodes which could conceivably share the same colors. Chapter 4.2 illustrates a case where our spilling algorithm makes use of the timing information offered by interval graphs.

3. Natural way of capturing loop carried dependence information :
   As we saw in Fig. 1.14(c), loop carried dependence information can be very easily captured by cyclic interval graphs. As we shall see in Chapter 4.2 our two step allocation method takes advantage of the explicit presence of cyclic intervals to discover opportunities to avoid spill code by inserting register move instructions instead. We expect this to improve the performance of the loop.

4. Renumbering phase becomes redundant :
   Each interval of the interval graph is akin to a renumbered live range or a node of the interference graph. The mere presence of the intervals in the interval graph makes the renumbering phase redundant.

   The intervals are essentially def-use chains [ASU88] of program variables. Each interval captures specific information about the point of definition and the points of use of that live range. Since we are focusing on perfect loop blocks, every variable of a perfect loop block has a single def-use chain and an interval of this block is the same as a def-use chain of that variable.

2.4 Summary

This chapter provides formal definitions of cyclic interval graphs as well as a description of their properties. We present a formulation of the register allocation problem in context of cyclic interval graphs and finally, points justifying the use of cyclic intervals graphs as the basis of our register allocation method is brought forth.

Chapter 3

Register Allocation

In this chapter we examine heuristic algorithms for coloring cyclic interval graphs using a minimal number of colors. More specifically, given a cyclic interval graph G, we would like to find a fast algorithm that can color G with as few colors as possible. All the algorithms of this chapter address Problem 1 of Chapter 2.2. Recall that Problem 1 was stated as the following :

- Problem 1 (Finding a Minimum Coloring of a Cyclic Interval Graph):
  Given a set of live ranges represented as a cyclic interval graph G, find a minimum register (color) assignment for the intervals in G such that overlapping intervals are assigned different registers.

This has important applications in situations when the smallest number of registers is required. For instance, when allocating registers interprocedurally it is beneficial to allocate a minimal number of registers to each procedure. This reduces the amount of register saving required at procedure call time, and may improve interprocedural register allocation [SH89b].

Our algorithms are based on the important assumption that the number of registers, k, required to color the graph, G, is available on the target architecture. If the graph required more than k registers then we assume that it has been transformed by the spilling phase (Chapter 4) to G, which is k-colorable.

As the graph coloring problem is known to be NP-complete, the coloring algorithms use some sort of heuristic, and therefore are not guaranteed to find the optimal solution.

However, our goal is to come up with heuristic based algorithms that find near-optimal solutions for the majority of the graphs being colored.

Section 3.1 outlines some terminology that will be used with regard to the cyclic intervals of the graphs. In Sections 3.2 and, 3.3 we develop new algorithms which address the issue of the minimal coloring problem and describe some implementation details. Lastly, in Section 3.4 we mention the assumptions which the coloring algorithms are based on as well as some of the caveats.

3.1 Terminology

Before embarking on a discussion of coloring algorithms, we explain terminologies which will be used to describe cyclic intervals in this and the following chapters.

Cyclic intervals wrap around from one iteration to the next, it is one long interval. However, in the interval graphs, we choose to depict it as two intervals instead. Variables *x3* and *x4* are cyclic intervals in Fig 1.14(c) of Chapter 1.

- **Front-end** of cyclic intervals : While sweeping the graph from left to right, the first section of the interval that is encountered is called the front-end of the cyclic interval. For instance, the front-end of *x3* extends from instruction 0 to 1 (Fig. 1.14(c)).
- **Tail-end** of cyclic intervals : The second section of the cyclic interval that is encountered while sweeping the graph is the tail-end of the interval. The tail-end of *x3* extends from instructions 3 to 5.

3.2 The Fat-Cover Coloring Algorithm

Given the fact that the optimal number of colors required to color a cyclic interval graph G is bounded by $W_{max}(G)$ and $W_{min}(G) + W_{max}(G)$ (Theorem 2.3.2), and our experimental observations which indicate that a large majority of graphs that could represent programs can be colored in W_{max} colors, we have developed an algorithm, called *the fat cover algorithm* [HGAM92], that is specifically designed to work well for graphs that can be colored in W_{max} colors.

The key to this algorithm is the observation that the *fat spots* which are the thickest parts of the interval graph are the locations that are most important. We can iteratively reduce

the maximum width of the uncolored portion of the graph by finding a non-overlapping set of intervals that covers all of the fat spots and coloring all of these intervals with the same color. First, we introduce this idea informally with an example, and then give a more formal description of the algorithm[1].

3.2.1 An Example

Consider the example graph given in Fig. 3.1.

Fig. 3.1(a) : In this picture we give the input interval graph. Notice that this graph has a maximum width, $W_{max}(G)$, of 3, and two cyclic intervals *a* and *b*. The fat cover algorithm will try to color it with $k = W_{max}(G) = 3$ colors.

Fig. 3.1(b) : The fat spots, or the points of maximum width, are indicated by arrows. The objective of the fat cover algorithm is to find for each cyclic interval, a set of non-overlapping intervals that includes the cyclic interval. The intervals are chosen such that one of the intervals is live at each fatspot of the graph. We call this set of intervals the *fatcover* relative to the cyclic interval.

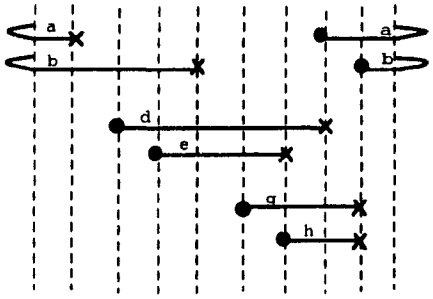
In this example, there are four fat spots to be covered and the fatcover for two cyclic intervals, a and *b*, have to be found. We find the fatcover for *a* in this step. By traversing left from interval a, the cover {a,*d*} is found. We can now color *a* and *d* with a new color *red*, and proceed to the next phase.

Fig. 3.1(c) : The intervals of the fatcover of *a* are removed and the remaining uncolored intervals are shown in this figure. Note that $W_{max}(G)$ is now 2, and there are three fat spots as indicated by the arrows. We find that {*b*,*g*} forms a fat cover for *b*, the only remaining cyclic interval, and we color *b* and *g* with a new color *blue*.

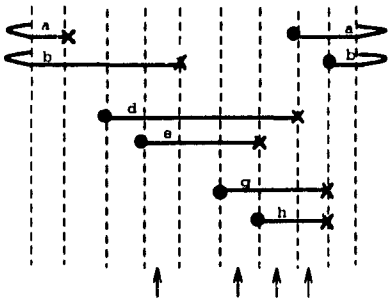
Fig. 3.1(d) : This figure shows the only remaining intervals to consider. Note that there are no cyclic intervals, and we can easily color these intervals with the third color *green*.

Fig. 3.1(e) : The final coloring of all the intervals is shown in this figure. Note that **a** and **d** are *red*, **b** and **g** are *blue*, **e** and **h** are *green*.

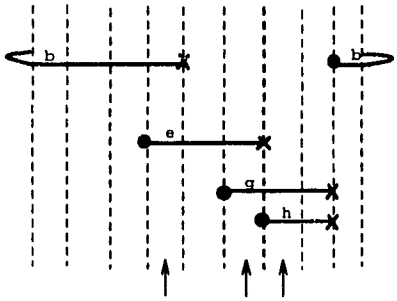
[1]Parts of this section has been excerpted from [HGAM92]



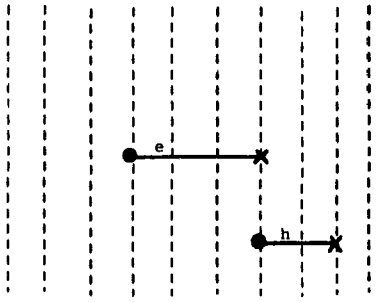
(a) The Original Interval Graph



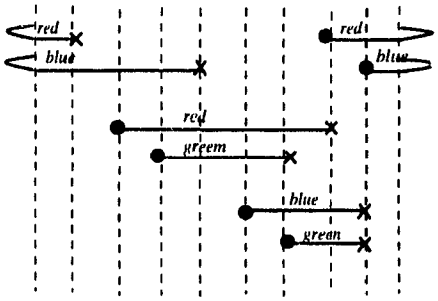
(b) Finding A Fat Cover For a - {a,*d*}



(c) Finding A Fat Cover For *b* - {*b*,*g*}



(d) Coloring The Rest



(e) The Final Coloring

Figure 3.1: An Example Of Applying The Fat Cover Algorithm

3.2.2 Theoretical Background

Given the basic idea of the algorithm as presented in the previous section, we now provide some definitions which will be frequently used.

Definition 3.2.1 *The* fat spots *of a cyclic interval graph G, written fatspots(G), is the set of all times t_i where width(G,t_i) = $W_{max}(G)$.*

Definition 3.2.2 *A* fat cover *of a cyclic interval graph, G, relative to interval I is a subgraph F ($I \in F$) of G that obeys the following two properties: (1) all intervals in F are non-overlapping, and (2) $\forall t_i \in fatspots(G)$, there exists an interval in F that covers t_i.*

Theorem 3.2.1 *If a cyclic interval graph G is colorable in $k = W_{max}(G)$ colors, then for each cyclic interval I_c of G, there exists a fat cover for G relative to I_c, call it F, such that $G - F$ is $k-1$ colorable.*

Proof: Given a $k = W_{max}(G)$ coloring of G, pick the color associated with any cyclic interval I_c, call it C. Now form a set F of all the intervals from G that were colored with C. First, let us show that F is a fat cover of G relative to I_c. By definition of a valid coloring, all intervals in F must be non-overlapping, and thus F satisfies the first property of Definition 3.2.2. Furthermore, since G is colorable in exactly $W_{max}(G)$ colors, then exactly one interval at each fat spot must be colored with C. Thus, F clearly satisfies property 2 of Definition 3.2.2. Secondly, it is clear that by removing F from G we are left with a graph that is colored with $k-1$ colors. ■

3.2.3 A Description Of The Algorithm

The development of the fat cover algorithm was inspired by Theorems 2.3.1 and 3.2.1.

Given a graph G with m cyclic intervals $C_{i_1}, C_{i_2}, \ldots, C_{i_m}$, the algorithm proceeds in two phases.

1. The first phase :
   attempts to use m colors to find a fat cover for each of the m cyclic intervals. At the ith step, a traversal from left to right is performed to find a fat cover for interval C_i (call this fat cover F_i). If such a cover is found, a traversal from right to left is performed which assigns the same new color C_i to all of the intervals in F_i.

2. The second phase :

   If the first phase *succeeds in coloring m cyclic intervals* with m colors, then the second phase need only consider a reduced graph G' that contains no cyclic intervals. G' has a maximum width of $w = W_{max}(G) - m$. The coloring of G' is guaranteed to use only w new colors (see proof of Theorem 2.3.1). A straightforward left-to-right algorithm is used to color the remaining non-cyclic intervals. We will not dwell on the left edge algorithm [HS71] as it is a very simple linear algorithm[2]. Thus, we can find an optimal coloring in $k = W_{max}(G)$ colors for graph G.

   However, if the first phase *fails to find a fat cover at some stage*, the second phase simply colors the remaining cyclic intervals with new colors, and applies the simple left to right algorithm to color the remaining intervals. In this case, the resultant coloring may or may not be optimal.

Our fat cover algorithm can be thought of as a smart way of deciding which subset of intervals should be colored with the same color. In some of the more traditional approaches using interference graphs, a simplification phase is applied to the interference graph in which pairs of nodes are coalesced into one node, thus forcing them to be colored the same color [CAC+81]. In our case we are searching for sets of nodes that have a very specific property, that is they all belong to a fat cover of some cyclic interval. Finding such a set of intervals requires information regarding the location of all the fat spots in the interval graph. This information is explicit in our cyclic interval graph representation, and is not available in the interference graph representation.

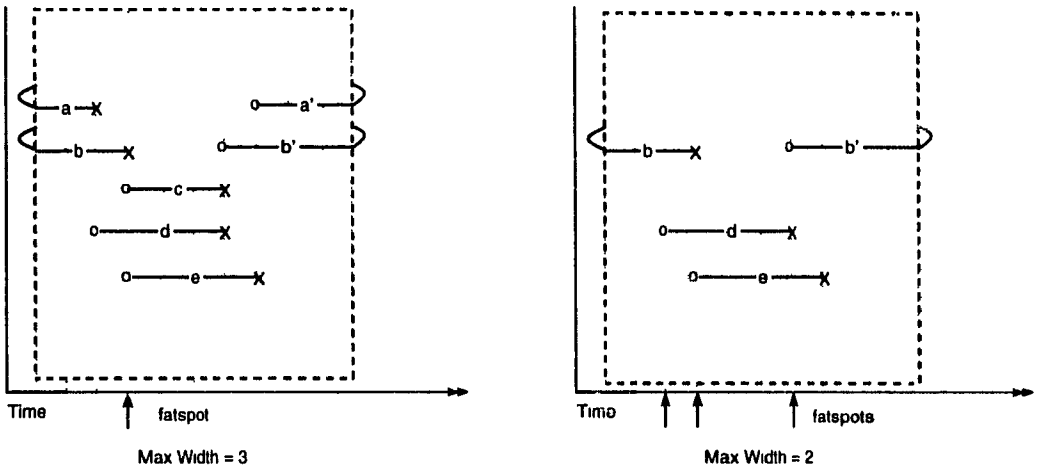
3.2.4 Sub-Optimal Performance Of The Algorithm

Although the algorithm finds *a* fat cover at each step, it may not find the fat cover that leads to an optimal solution. That is, for a k-colorable graph G, it may select a fatcover FC_i such that $G - FC_i$ is not $k-1$ colorable, even though there exists another fatcover FC'_i such that $G - FC'_i$ is k -- 1-colorable. In parts of the chapter FC_i is also defined as FC_i or F_i.

Fig. 3.2 shows an example illustrating this case. The interval graph (Fig. 3.2(a)) has a maximum width of 3 and the fatspot is marked by an arrow.

Assume that we are trying to find the fatcover of the cyclic interval a first. It has three potential fatcovers :

[2]Appendix A (Algorithm A.0.1) gives a brief description of the left edge algorithm.



(a) The Original Interval Graph, G (b) Reduced Graph, G' -- Without a, c, a'

Figure 3.2: Sub-Optimality Of The Fatcover Coloring Algorithm

1. {a, c, a'}
2. {a, d, a'}
3. {a, e, a'}

If one of the last two fatcovers are chosen then a fatcover for the cyclic interval b can be found. The fatcover for b consists of the intervals $\{b, c, b'\}$.

However, if the { a, c, a' } is chosen as the fatcover for a then we will be unable to find a fatcover for b. Fig. 3.2(b) shows the reduced graph after the fatcover for a is removed. From this graph, a fatcover for b can not be found. From our example we see that a specific choice of a fatcover paves the way for finding fatcovers of other cyclic intervals. 3 colors (which is $> (W_{max}(G') = 2)$) will be required to color the reduced graph, G'. The cyclic interval b as well as intervals d and e will have to be assigned new colors each. *Therefore, it is important to note that G is uncolorable with $W_{max}(G)$ colors, it requires more colors.*

Theorem 3.2.2 *For a graph, G, which is colorable with $W_{max}(G)$ colors and has only one cyclic interval, the fatcover algorithm always finds the optimal ($W_{max}(G)$) coloring of G.*

Proof: Given a graph, G, which is $W_{max}(G)$-colorable and has only one cyclic interval, C_i. The fatcover algorithm is guaranteed to find a fat cover, F_{C_i}, if one exists. Furthermore, after a fat cover is found, we know that the graph $G - F_{C_i}$ will have maximum thickness of $W_{max}(G) - 1$, and $G - F_{C_i}$ will contain no cyclic intervals. Therefore, by theorem 2.3.1, we can guarantee that $G - F_{C_i}$ can be colored with $W_{max}(G) - 1$ colors, and G can therefore be colored in $W_{max}(G)$ colors. ∎

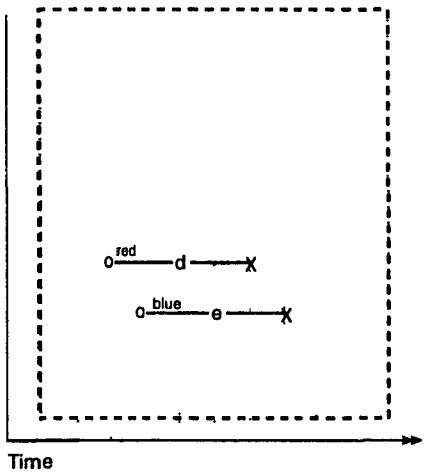
3.2.5 An Improved Fatcover Algorithm

As we saw in the section above, when there are more than one cyclic intervals in the graph, we may make a bad choice of a fatcover from amongst all the possible fatcovers of a cyclic interval. Our choice may prevent us from finding the fatcover for a subsequent cyclic interval and this renders the graph uncolorable in k colors! Finding the right fatcover is an NP-hard problem and we could attempt to use a backtracking algorithm to find the right solution. After making a choice of a F_{C_i} if $G - F_{C_i}$ is uncolorable with $W_{max}(G - F_{C_i})$ colors then it would have to backtrack and choose another F_{C_i} for the cyclic interval i. If there are several cyclic intervals then the algorithm may spend most of its time backtracking. Hence, this is not a very practical solution.

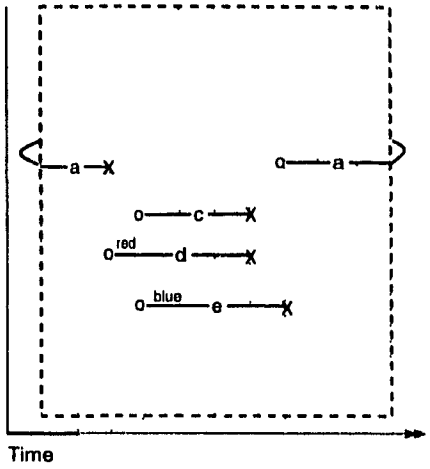
Perhaps we need to see the problem in a different perspective. A $W_{max}(G)(= k)$ coloring of the graph is not possible when the right fatcover has not been found for one of the cyclic intervals. Instead of focusing on obtaining the right fatcover for all the cyclic intervals, we shift our attention to ensuring that we always obtain a k coloring of the graph.

First, we ask why making a wrong choice of a fatcover renders G' uncolorable in $k - 1$ colors. According to the fatcover algorithm, assume that we color the wrong fatcover {a, c, a'} the color red in Fig. 3.2(a), then this color is unavailable to color {b, b'}, d and e and all of them require to be assigned distinct colors. *However, if we were to assume that all the intervals of the fatcover do not have to receive the same color then we can find a 3-coloring of the graph.* Of course, the front and tail ends of the cyclic intervals must be assigned the same colors, but if there are acyclic intervals in the fatcover then they do not necessarily have to share the color assigned to the cyclic interval. This is the key intuition behind the improvement of the fatcover algorithm [Gao93]. The fatcover of a cyclic interval is useful in reducing the maximum thickness of the graph and, we exploit this property of fatcovers. But choosing to assign the same color to all the intervals of the fatcover imposes an unnecessary constraint.

For instance, how do we find this 3-coloring of G of Fig. 3.2(a)?



(c) Coloring The Non Cyclic Interval Graph



(d) Coloring Cyclic Interval a

Figure 3.3: Illustrating A Modified Fatcover Algorithm

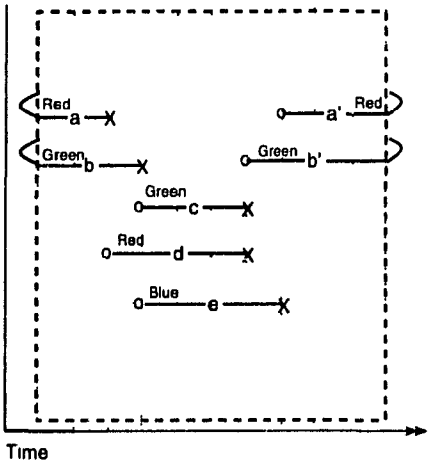
1. STEP 1 : Find fatcovers of cyclic interval, but do not assign colors
   After choosing the fatcover {a, c, a'} we remove it from G *but unlike the fatcover algorithm, do not assign it any color.*

   This is the first deviation from the original fatcover algorithm.

2. We proceed to find a fatcover for the cyclic interval b of Fig. 3.2(b) and, discover that a fatcover can't be found for it. So, we remove b from G' and refrain from coloring the cyclic interval.

3. STEP 2 : Color acyclic interval graph
   Next, we are left with the non-cyclic interval graph of Fig. 3.3(c). Like in the fatcover algorithm, the left-edge algorithm is used to color these intervals. Interval d is colored red, while e is colored blue.

4. STEP 3 : Color fatcovers, (FC_i), containing acyclic intervals
   Now, we put back the fatcover of a into the graph and concentrate on coloring the acyclic intervals of this fatcover (Fig. 3.3(d)). As c is a part of the fatspot of the graph and, the graph can be colored in $W_{max}(G)$ colors, a free color must be available for



(e) The Colored Cyclic Interval Graph

Figure 3.4: Illustrating A Modified Fatcover Algorithm

it. We assign it a color other than the ones which have been assigned to the intervals that it conflicts with. In our example, b has to be assigned the color green, as the colors red and blue have been assigned to d and e which are a part of the fatspot.

Having the flexibility to assign the acyclic intervals of FC_i a color different from the cyclic interval of FC_i is the most important modification made to the fatcover algorithm. This does not tie up a single color for all the intervals of FC_i and allows colors to be shared between FC_i and the other acyclic intervals of $G - FC_i$.

This is the second distinguishing trait of the modified algorithm.

5. STEP 4 : Color cyclic intervals having no fatcovers
   Lastly, we put back { b, b' }. In general, cyclic intervals for which no fatcover have been found are put back in the last step. This cyclic interval is assigned a color that is free during its lifetime. From Fig. 3.4(e) b is assigned the color green, it could not be assigned red as it has been assigned to the cyclic interval a and blue is assigned to interfering interval e.

   And, this is the third and last factor that separates the modified algorithm from the

original one.

3.2.6 The Implemented Algorithm

In this section we discuss the original fatcover algorithm as it has been implemented and the next section outlines some data structures it uses.

The procedures that we will refer to have been described in detail within in boxes in the following pages. The names of procedures have been italicized.

1. *Main Fatcover*
   Expected inputs and the output of the algorithm are mentioned in the boxes labelled *Input* and *Output* respectively. The procedure *Main Fatcover* is the driving routine for the fatcover algorithm.

   Main Fatcover chooses and colors fatcovers for each cyclic interval of the graph. If a fatcover is not found for a cyclic interval, then it is assigned a previously unused color. The colored intervals are removed from the graph and the resulting graph contains no cyclic intervals. The non cyclic intervals are assigned colors using the left-edge algorithm. The final colorings of all the intervals are output. Below, we give a high level algorithm for this procedure. The terms reg_{class} denote the register classes available on the target architecture, while $reg_{pref}(i)$ is the register class preference of a register i.

High Level Algorithmic Description Of Main Fatcover :

FOR each reg_{class}
  Subgraphs,(G'), are created
  where *all* the intervals, i, of G' are such that :
    $reg_{pref}(i) = reg_{class}$
  FOR every G'
    FOR every cyclic interval, C_i, in G'
      Sweep G' from left to right once, incrementally gathering
      all the possible fatcovers of C_i, $FC_{C_i}[]$ (in Function
      Establish All Covers Of C_i)
      Sweep G' from right to left again and choose *one fatcover*
      from the set $FC_{C_i}[]$ (in Function
      Choose A Cover For C_i)
      IF $FC_{C_i}[] == \emptyset$ **THEN**
        Mark C_i a chameleon interval
      ELSE
        Assign ($reg_{number}(C_i)$) new colors to $FC_{C_i}[]$.
    Run Left-Edge-Algorithm on the subgraph
    ($G' - FC_{C_1 \ldots m}$) which has only non-cyclic intervals
DONE

2. *Establish All Covers Of* C_i
   This routine (described in the box labelled *Establish All Covers Of* C_i) is invoked by *Main Fatcover*. It is responsible for finding all the fatcovers of a cyclic interval, C_i, of G.

   We give a high level algorithm for this procedure.

parents[] is initialized to NULL before we begin the sweep of G'.

A left to right sweep of G' is performed and each fatspot is visited.

At each fatspot the following steps are performed :
(1) We obtain candidate intervals for each of the possible fatcovers. Candidate intervals of a fatspot are chosen such that they do not conflict with the previously chosen parent intervals, *parents[]*.
(2) As each candidate interval is chosen it is stored in *children[]*.
(3) Once all the *children[]* intervals have been found at a fatspot they're moved into *parent[]*.
(4) We continue our sweep of the graph and move onto the next fatspot

C_i is always a part of its fatcover, so the front and tail ends of C_i are always inserted as intervals of it's fatcover.

We should note that in this function all the possible fatcovers for each cyclic interval is *established* and maintained as a graph structure *without their explicit enumeration.* A record is made of all the acyclic intervals of a fatspot that do not conflict with the parent intervals. In this way enough information is captured about all the feasible fatcovers of a cyclic interval without explicitly enumerating them. This information is later used in function *Choose A Cover For* C_i to create and choose one fatcover from amongst all the possible ones.

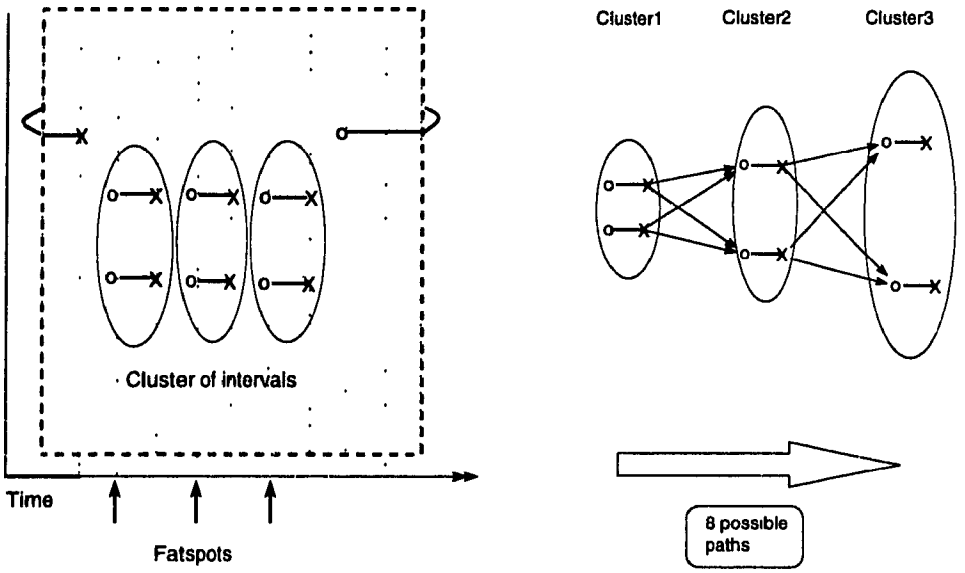
The time complexity of this procedure dominates the total complexity of the fatcover algorithm. Hence, we analyze a couple of aspects of its complexity.

(A) The worst case time complexity of Establish All Covers Of C_i *(where* C_i *is a cyclic interval) is* $O(num\_fs * n^2)$*, where num_fs is the number of fatspots in* G' *and,* n = *number of intervals(G').*

From the described algorithm, we see that in the worst case this procedure has a complexity of $O(num\_fs * (W_{max}(G'))^2)$. At every fatspot, we need to look at every interval live at that time instant and it's parents, to see check if it can be a candidate for the fatcover for C_i. And we know that there are at most $W_{max}(G')$ intervals live at any fatspot. If all $W_{max}(G')$ intervals are live at every

fatspot and the intervals at a fatspot do not conflict with those of other fatspots. Fig. 3.5(a) shows the structure of the interval graph in the worst case. Hence, we check $(W_{max}(G'))^2$ intervals at each fatspot. We hope that $W_{max}(G') < n$, where $n = number\ of\ intervals(G')$.

In the absolute worst case, $W_{max}(G) = n$ and the complexity becomes $O(num\_fs * n^2)$.



(a) The *Worst Case Interval Graph* (b) The *Number Of Fatcovers*

Figure 3.5: Number Of Fatcovers Per Cyclic Interval (Worst Case Situation)

(B) If we were interested in enumerating all the fatcovers of a cyclic interval, then in the worst case the total number of possible fatcovers of a cyclic interval, C_i is $W_{max}(G')^{num\_fs}$.

In the worst case scenario, we assume that the set containing $W_{max}(G')$ intervals at each of the *num_fs* fatspots are unique. Furthermore, none of the intervals of the fatspots conflict with the cyclic interval whose fatcover is being found. Fig. 3.5(a) shows the structure of the interval graph. From the figure we see

3 (= *num_fs*) clusters of 2 (= $W_{max}(G')$) intervals each. In order to find all the fatcovers we need to find all the paths from *cluster_1* to *cluster_(num_fs)*. Fig. 3.5(b) shows all the paths from from *cluster_1* to *cluster_3*. In general, we observe that there will be $W_{max}(G')^{num\_fs}$ paths for every cyclic interval in the graph. Paths are equivalent to the fatcovers of the cyclic interval.

3. *Choose A Cover For* C_i

This is a subsidiary routine that is invoked by *Main Fatcover* as well. A high level description of the algorithm is provided below.

We now have *parent[]* (from *Establish All Covers Of* C_i) containing the last interval of all the fatcovers. The last interval is always the tail end of C_i.

We traverse the graph *backwards* visiting each fatspot.

At each fatspot the following steps are performed :
intervals chosen at each fatspot of G'.
(1) The list of candidate intervals are reviewed. None of the candidate intervals at that fatspot conflict with the *parent[]* interval.
(2) From the list of candidates one is chosen. The choice is based on heuristics. Our heuristics favor :

- Intervals which have earlier start times.
- When two intervals have the same start time then the interval which is live longer is chosen.

These criteria tend to choose intervals of longer duration. This ensures color assigned to fatcovers are kept busy for long stretches of time instead of being live for several short durations. all the candidate intervals at that fatspot.
(3) Once an interval is it is entered in *parent[]*.
(4) The previous fatspot of the graph is visited.

Please refer to the box labelled *Choose A Cover For* C_i for a detailed description of the algorithm.

We also provide a time complexity analysis for this procedure.

(A) In the worst case, the time required to choose a cover for a cyclic interval is $O(num\_fs * W_{max}(G'))$.

From the described algorithm we see that in the worst case this procedure has a complexity of $O(num\_fs * W_{max}(G'))$. Given that we have all the possible

fatcovers, we traverse the graph backwards selecting acyclic intervals at each fatspot. By the end of the traversal, a fatcover is created.

Assume that the intervals at each fatspot do not conflict with intervals at the next fatspot (Fig. 3.5(a)). Hence in our backward traversal when we encounter the first fatspot we choose one interval from amongst the possible $W_{max}(G')$ intervals. Since none of the intervals of one fatspot conflict with those of the next fatspot, the interval chosen for the fatcover has $(W_{max}(G'))$ possible parents. Therefore, at each fatspot we have to look through $(W_{max}(G'))$ intervals in order to choose one. Since there are *num_fs* fatspots, the total complexity is $O(num\_fs * W_{max}(G'))$.

Algorithm 3.2.1 *Given a* `cyclic interval graph`, *G, use the fatcover algorithm to color G.*

Input: A `cyclic interval graph, G,` constructed using the def-use information of the variables. The cyclic and the non-cyclic intervals of the graph are identified,

The `register classes`, reg_{class}, present on the target architecture,

The `register preference`, reg_{pref}, of each interval of the cyclic interval graph. This determines the class of register that needs to be assigned to each interval. And,

The `number of registers`, reg_{num}, needed by each interval of the cyclic interval graph. Recall that we use registers and colors interchangeably.

Output: The cyclic interval graph, G, such that
every interval, i, of the graph has been
assigned a color, c_1, and,
$reg_{class}(c_1) = reg_{pref}(i)$ assigned colors, $c_{1\,..reg_{num}(i)}$, and,
$reg_{class}(c_{1\,.\,reg_{num}(1)}) = reg_{pref}(i)$

Procedure Main Fatcover :

FOR each subgraph, G',
  Identify all the fatspots ($fs[]$) of G'
  Record the *number* of fatspots in *num_fs*
  FOR every fatspot, $fs[t]$),
    Find all non cyclic intervals, $NC_{i_1}, \ldots, NC_{i_k}$,
    live at time t
  FOR every cyclic interval, C_i,
    Establish All Covers Of C_i, $FC_{C_i}[]$, by a left to right traversal of G'
    IF $FC_{C_i}[] \geq 1$ **THEN**
      successful = *Choose A Cover For* C_i from
      $FC_{C_i}[]$ by a right to left traversal of G'
      IF *successful* **THEN**
        Assign $FC_{C_i}[chosen]$ ($reg_{number}(C_i)$) new colors to $FC_{C_i}[chosen]$
      ELSE
        Mark C_i to be a *chameleon* interval (These are
        special intervals and will be discussed in Chapter 4).
    ELSIF $FC_{C_i}[] = \emptyset$ **THEN**
      Assign C_i ($reg_{number}(i)$) new colors, c_i
  Use Left-Edge-Algorithm to color the subgraph ($G' - FC_{i_1 \ldots i_m}$),
  which has only non cyclic intervals
DONE

Procedure Establish All Covers Of C_i

cur_fspot = 1
parent[]=*children[]*= *roots* = NULL
front_end_live = FALSE
tail_end_live = FALSE
WHILE *cur_fspot* < *num_fs*
(If the front end of the cyclic interval is live)
(then it is the root interval of all possible fatcovers)
IF front-end of $C_{front-i}$ is live **THEN**
Choose C_i to be a part of the fatcover
Insert C_i into *children[]*
parents[] = *children[]* (in preparation for *cur_fspot* +1)
Increment *cur_fspot*
IF *cur_fspot* == 1 **THEN**
Set parent of C_i = NULL (as this has to be the first interval in all possible fatcovers)
roots[] = *children[]* (to keep track of the first interval of the fatcovers)
front_end_live = TRUE
ELSE
Set parent of C_i
CONTINUE with next iteration of loop
(If the front end of the cyclic interval is not live at the first)
(fatspot, include it in $FC_{C_i}[]$ anyhow, as it must be a part)
(of the fatcover)
IF (*cur_fspot* == 1 && !*front_end_live*) **THEN**
Set parent of C_i = NULL (as this has to be the first interval in all possible fatcovers)
roots[] = *children[]* (to keep track of the first interval of the fatcovers)
parents[] = *children[]* (in preparation for the next fatspot)
front_end_live = TRUE
Increment *cur_fspot*
CONTINUE with next iteration of loop
FOR each non-cyclic interval, NC_i, live at cur_fspot
FOR each *int* in *parent[]*
IF NC_i and *int* do not conflict
Insert NC_i into *children[]*
Record parent of NC_i to be *int*
Record child of *int* to be NC_i
IF tail-end of C_{tail-i} is live at *cur_fspot*
tail_end_live = TRUE
Insert tail-end of C_{tail-i} in *children[]* so that it
is chosen to be a part of the fatcover
Eliminate all *parent[]* intervals which conflict with C_{tail-i}
IF *parent[]* = $\emptyset$ **THEN**
RETURN FALSE (No fatcover can be found)
ELSE
RETURN TRUE (Fatcover can be found)
parent[] = *children[]*
Increment *cur_fspot*
(If C_{tail-i} was not live at any of the fat spots then insert it)
(in the fatcover array to make it the last interval to be seen by all)
(fatcovers)
IF !*tail_end_live* **THEN**
Insert tail-end of C_{tail-i} in *children[]* so that it
is chosen to be a part of the fatcover
Eliminate all *parent[]* intervals which conflict with C_{tail-i}
IF *parent[]* = $\emptyset$ **THEN**
RETURN FALSE (No fatcover can be found)
RETURN TRUE (Fatcover can be found)
DONE

```
Choose A Cover For C_t :

chosen_fc[] = NULL
earliest_start_time = LARGE_NUM
WHILE parent[0] != NULL
    FOR each int in parent[]
        IF int was before and s in chosen_fc[] THEN
            int_num = int
        ELSIF start_time(int) < earliest_start_time THEN
            int_num = int
            earliest_start_time = start_time(int)
    chosen_fc[] = int_num
    parent[] = parents of int_num
DONE
```

3.2.7 Main Data Structures Used

In order to implement the fatcover algorithm, we have used the data structures described below.

- **Interval Array :**
  As shown in Fig. 3.6(a), this is an array that holds all the intervals of the graph, cyclic as well as the non-cyclic ones. Cyclic intervals occupy the first portion of the *interval array* and, the front and tail ends of each cyclic interval are kept adjacent to one another. The rest of the structure is occupied by non-cyclic intervals. Basic information about all the intervals is recorded here and is used by all the coloring algorithms. The index of the array gives each interval an unique *interval number*. The following attributes of intervals are recorded :

  1. Var number : This tells us which variable in the source code gives birth to this interval. The *var number* is mainly used for debugging purposes.
  2. Reg preference : The $reg_{class}(i)$ is needed so that we can create subgraphs, G', such that all the intervals of G' have the same register preference.
  3. (Number) # of regs required : Intervals may require more than one register. Some data modes require two or even four registers. When coloring, we need to know the number of colors required by each interval in G'.
  4. Interval Property : This flag tells us whether an interval is cyclic, non-cyclic or chameleon.

5. Start time, End time : The start time and end times of every interval provides us with information about the life length of the interval. The exact timings are used by the coloring algorithms.
6. Register assigned : The color assigned to the interval is recorded in this field.
7. Start operation, End operation : These are pointers to the first and the last operations of the interval.

- **Fatspot Array :**
  Fig. 3.6(b) illustrates an array where we keep record of all the live non-cyclic intervals at each fatspot of every G'. In essence, this array contains a part of the interval graph. The array is first indexed by the reg_{class} and then for each reg_{class} the following information is kept :

  - **FOR** every fatspot (each of which is a cell of an array)
    1. Time : of G' where the fatspot occurs.
    2. Maximum intervals live : the maximum number of intervals live at this fatspot.
    3. Interval Vector : which is an array containing the interval number of all the live intervals of this fatspot. The interval numbers are used to access the *Interval array*.

  Since live intervals at any fatspot can be available in constant time from this structure, the complexity of the fatcover algorithm is reduced.

- **Roots and Fatcover array :**
  Fig. 3.6(c) shows the *roots* and the *fatcover* arrays. Together these two structures hold all the possible fatcovers for a C_i.
  The roots array points to the root interval for all the fatcovers of C_i. The root interval is the front-end of C_i. The intervals pointed to by the root are the first level intervals The first level intervals maintain pointers to candidate children and are candidate intervals for the fatcover of C_i for the first fatspot. The first level intervals maintain pointers to candidate children intervals. The structure is recursive and ends with the last interval of all the fatcovers which has to be the tail-end of C_i. This is the leaf interval of the graph.
  We see that *roots* holds a directed graph structure and this is built by the procedure *Establish All Covers For* C_i. The function *Choose A Cover For* C_i chooses a path starting with the leaf interval and ending with the root interval. Intermediate members of the path (which may be acyclic intervals) are chosen by heuristics.

  Each element of the fatcover array has information about the following fields :

1. Number of Interval : The candidate intervals' number. This is used to access the *interval array*.
2. Number of parents : The number of parent intervals of this interval.
3. Number of children : The number of children candidate intervals of this interval.

3.3 The Greedy Coloring Algorithm

Another approach to coloring a cyclic interval graph is to first color the cyclical intervals with unique colors, and then use a greedy algorithm to color the remaining intervals [HGAM92]. We describe this approach in Section 3.3.1 and then provide implementation details in Sections 3.3.2, 3.3.3 and 3.3.4. Like the fatcover algorithm, the greedy algorithm also makes use of heuristics to address the issue of minimum coloring of a cyclic interval graph which was stated as Problem 1 in the beginning of this chapter.

3.3.1 A Description

Given a graph G with m cyclic intervals $C_{i_1}, C_{i_2}, \ldots, C_{i_m}$, the following steps are performed :

1. Assign $(reg_{number}(C_i))$ unique colors to each cyclic interval C_i.
2. After coloring the cyclic intervals, we are left with a graph that contains only non-cyclic uncolored intervals, $NC_1 \ldots NC_i$. So, we color these intervals of G using the three step process mentioned below :
   (a) From among the uncolored intervals, choose the "best" one to color next, call it NC_{next}. NC_{next} is chosen on the basis of heuristics.
   (b) From among the colors previously used, choose the "best" available color, call it c_{next}. If no color is available for this interval, then allocate a new color. Like NC_{next}, c_{next} is heuristically chosen.
   (c) Assign color c_{next} to interval NC_{next}.

3.3.2 The Implemented Algorithm

We outline the algorithm as it has been implemented. While giving an overall description, we will refer to routines which are described in detail in the following pages. The routines are enclosed in labeled boxes.

1. *Main Greedy*
   The expected input and output of the algorithm are mentioned in *Input* and *Output* respectively. The main driving routine is *Main Greedy*. In this algorithm, first each of the cyclic intervals are assigned new colors and removed from the graph. Once all the cyclic intervals are removed, we are left with non cyclic intervals only. These non cyclic intervals are colored by choosing one interval from amongst all the uncolored ones and assigning it a color from amongst all the colors available for it. These choices are heuristically determined.

2. Section 3.3.3 defines two functions which are invoked by *Main Greedy*. The function *Choose A Non-Cyclic Interval*, chooses a non cyclic interval, NC_i, to color from amongst all the uncolored non cyclic intervals and, *Choose A Color For Non-Cyclic Interval* chooses a color to assign to NC_i.

An analysis of the time complexity of the greedy algorithm is presented in Section 3.3.4.

Algorithm 3.3.1 *Given a* `cyclic interval graph`, *G, that is colorable with $W_{max} = k$ colors, use the greedy algorithm to color G.*

Input: A `cyclic interval graph`, G, constructed using the def-use information of the variables. The cyclic and the non-cyclic intervals as well as the fatspots of the graph are identified,

The `register classes`, reg_{class}, present on the target architecture,

The `register preference`, reg_{pref}, of each interval of the cyclic interval graph. This determines the class of register that needs to be assigned to each interval. And,

The `number of registers`, reg_{num}, needed by each interval of the cyclic interval graph. This is the same as the number of colors required by an interval.

Output: The cyclic interval graph, G, such that every interval, i, of the graph has been assigned colors, $c_{1...reg_{num}(i)}$, and, $reg_{class}(c_{1...reg_{num}(i)}) = reg_{pref}(i)$

Procedure Main Greedy :

(Assign colors to all the cyclic intervals of G)
FOR each cyclic interval C_i
    Assign $(reg_{number}(C_i))$ new colors to C_i
    Update status of assigned colors to mark their times of use
(Now, we are left with a reduced graph, G', where all the uncolored)
(intervals are non-cyclic. We assign colors to them.)
Sort the uncolored non-cyclic intervals in order of increasing start time
WHILE there exists uncolored non-cyclic interval, NC_i, in G'
    Choose an interval, NC_{chosen} from amongst all the available uncolored intervals
    Choose $(reg_{num}(NC_{chosen}))$ colors to assign to NC_{chosen}
    Assign the chosen colors, $(c_{1...(reg_{num}(NC_{chosen}))})$, to NC_{chosen}
DONE

3.3.3 Two Important Steps Used By The Greedy Algorithm

The two functions described here are used by the greedy algorithm. Heuristics used by these functions have been mentioned in [HGAM92].

Choose A Non-Cyclic Interval, NC_i :

Given a set of uncolored intervals, we have to choose one to color. Intervals are chosen according to heuristics. Some possible criteria that are used in making this choice are :

- the leftmost uncolored interval (the interval with the lowest starting time),
- the longest uncolored interval,
- the interval which overlaps with the most uncolored intervals, or

- the interval which has the fewest number of available colors (where a color c is *available* for interval NC_i, only if c has not been used for any interval that overlaps with NC_i).

Choose A Color For Non-Cyclic Interval, NC_i :

Algorithm : All the colors assigned to the acyclic intervals that conflict with NC_i are made unavailable for NC_i Choose ($reg_{num}(NC_{color})$) colors from amongst the available colors for NC_i.

Some possible criteria for choosing the "best" color for a given NC_i include:

- (a) best-fit (each color is available for some time intervals, a color that best-fits is one where the starting and ending times for the color best match the starting and ending times for the interval NC_{next}),
- (b) worst-fit, and
- (c) the color which can be used for the fewest number of unallocated intervals.

3.3.4 Main Data Structures Used

We use the following heuristics in the greedy algorithm,

1. When choosing an interval :
   Higher priority is given to those intervals which have fewer colors available to be assigned to it.

2. When choosing colors to assign to an interval :
   Colors which can color fewer number of uncolored intervals are given a higher priority.

Together these two heuristics aim at coloring constrained intervals first as there are fewer available colors for it. The color that is assigned to the chosen interval is one which is available to color the fewest intervals. This heuristic used to choose a color ensures that colors are maximally reused.

Given these heuristics, we use the following data structures in the implementation of the greedy algorithm.

- **Interval Array :**
  This is the same structure as the one used by the greedy algorithm (Fig. 3.6(a)). It stores basic information about all the intervals of the graph and the index of the array is used to number intervals. The *interval array* is used in building the structure *greedy intervals.*

- **Greedy Intervals :**
  This is a sorted linked list of intervals where the first interval of the list has highest priority for coloring (Fig. 3.7(a)). For our heuristic, each element of the list has the following information :

  1. (Number) # of interval : like NC_i.
  2. (Number) # of overlap, Overlap List : These fields are crucial to our heuristic. *Number of overlaps* records the number of all the uncolored intervals which have the same preference as NC_i and which overlaps with it, while *overlap list* maintains a list of all the overlapping intervals, $NC_{overlap_1} \ldots NC_{overlap_{num\text{-}overlap}}$. The *number of overlaps* of NC_i is used to position the interval in the sorted *greedy intervals* list.
  3. (Number) # of colors, Register list : The *number of color* gives us the number of colors available to color NC_i, while the *register list* is an array that holds the colors that could be used to color this interval. When choosing a color, these available colors are considered by the heuristic algorithm.
  4. Prev(ious), Next : are pointers to the previous and next intervals of the list.

- **Greedy Registers :**
  This structure provides the heuristic algorithm information about the list of available registers (Fig. 3.7(b)). The information is used when choosing a color to be assigned to an interval.

  Each cell of the *greedy registers* array represents a register class that is supported by the target architecture and, it points to a list of all the registers of that class. Each register in the list has the fields :

  1. Number of the register : The register number is recorded. For example, *reg 1* is known as *1*.
  2. Number of intervals : that this register can color. This is the key factor involved in our heuristic choice of a color.

Once NC_i is chosen, it is assigned a color, c_{NC_i}. Next, the data structures are updated in the following fashion :

- the overlapping intervals, $NC_{overlap_1} \ldots NC_{overlap_{num-overlap}}$ of NC_i can no longer be assigned the color c_{NC_i}. The *register list* field of all the conflicting intervals have to be updated to make c_{NC} unavailable for it.

- As NC_i is being colored, it can no longer belong to the overlap lists of $NC_{overlap_1} \ldots NC_{overlap_{num-overlap}}$ (- recall that the overlap list contains a list of only the uncolored overlapping intervals). And finally,

- The *number of intervals* that the color c_{NC_i} can color decreases by the number of overlapping intervals of NC_i. This change is reflected as well.

These above steps must be performed and the *greedy intervals* list has to be re-sorted to keep it in a state where it can be easily used by the colorer. Updating the *greedy intervals* list contributes to the time complexity of the greedy algorithm. We outline the process of updating the list briefly to understand the time complexity.

Updating Greedy List :

FOR each conflicting interval, $NC_{overlap}$, of NC_i
  (Temporary pointer used to traverse greedy intervals list)
  $traverse_{greedy}$ = greedy intervals list
  WHILE $traverse_{greedy}$! = NULL
    IF $traverse_{greedy} \rightarrow number\_of\_interval == NC_{overlap}$ **THEN**
      Make c_{NC} unavailable for $NC_{overlap}$ by
      changing the register list
      Remove NC_i from the *overlap list* of $NC_{overlap}$
      Decrement the *number of intervals* for c_{NC}
      in *greedy registers* array
Re-sort *greedy intervals* list in increasing order of *number of color*
DONE

(A) In the worst case the time complexity of the Greedy Algorithm to update the greedy intervals list after having chosen one interval is $O(W_{max}(G) * n)$ where, n= total number of intervals in the graph.

In the worst case scenario, NC_i can have a maximum of $W_{max}(G) - 1$ conflicting intervals, and each of these conflicting intervals have to be searched for in the *greedy intervals* list which is as long as n = *total number of intervals in the graph.* Thus, the complexity is $O(W_{max}(G) * n)$, assuming $W_{max}(G) < n$.

3.4 Assumptions And Caveats

Currently our coloring algorithms have been implemented on the basis of several assumptions.

Subsets of register classes :

We take various register classes (like floating point registers and, general purpose registers) into account. Every symbolic register is assumed to have a preference for *one* specific class of register only. At this point of time, we do not allow one register class to be a subset of another class. This makes the allocation problem simpler for us, as it eliminates the possibility of an interval being members of several subgraphs of G. The subgraphs of G are created such that all the intervals of G' have the same register preference.

Adjacency and alignment constraints of multi-registers :

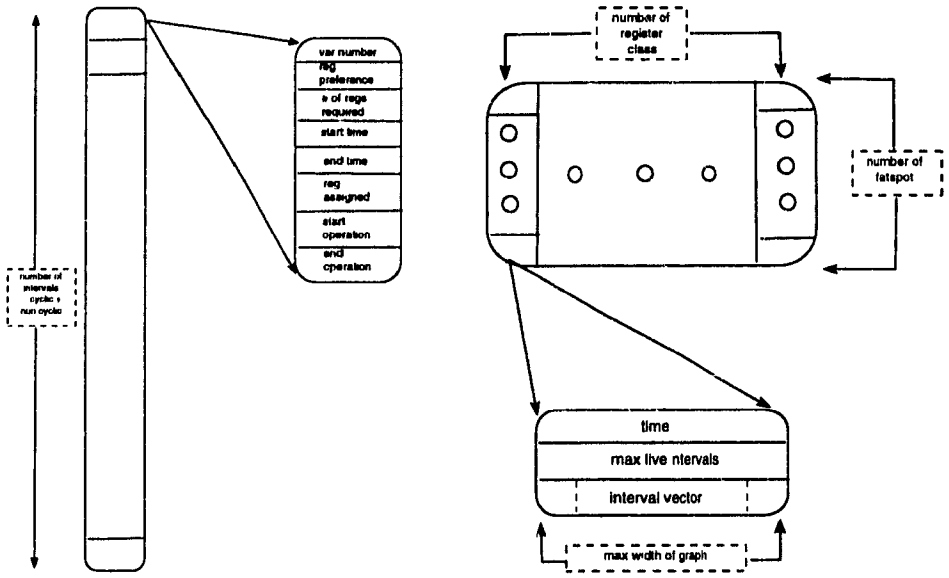
Different kinds of architectures impose different adjacency and alignment constraints of registers.

Typically, intervals which require multi-registers to be assigned to them must be aligned on certain word boundaries and may require the allocated registers to adhere to some adjacency requirement as well. These constraints decrease the size of the register set available to the graph and also affect the way intervals conflict with it's neighbors [Nic90, BCT92].

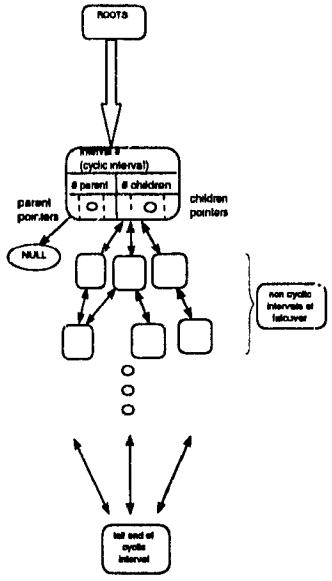
Currently, our colorer takes very simple constraints into account. Intervals may be assigned one or two registers. If two or more registers are to be assigned then the first register assigned must be an even numbered one. More over, the assigned registers should all be contiguous. We chose to impose this restriction as current architectures like the SPARC and the MIPS assign even/odd register pairs [SPA92].

3.5 Summary

Three coloring algorithms, the fatcover, a modified fatcover and a greedy algorithm are presented in this chapter. All the algorithms are based on the interval graph representation. The fatcover and the greedy algorithms have been implemented and, we discuss the data structures used as well as the time complexity of these algorithms.

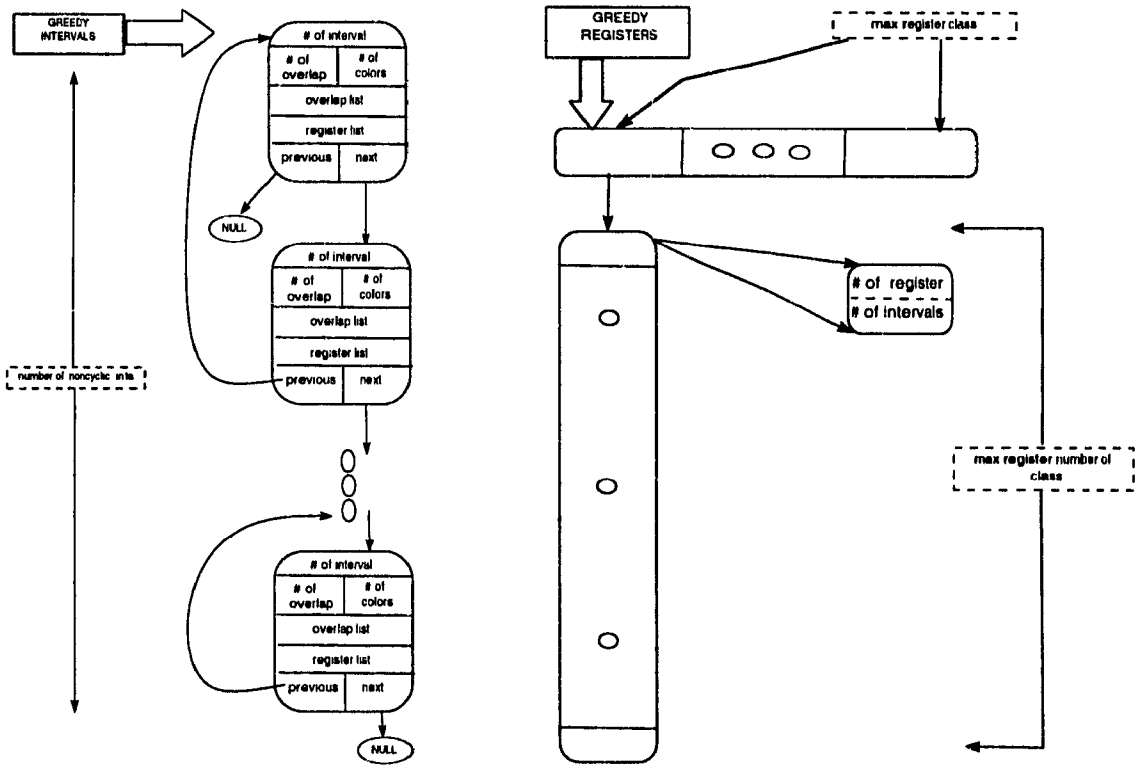


(a) The *Interval Array* (b) The *Fatspot Array*



(c) The *Roots And Fatcover Arrays*

Figure 3.6: Data Structures Used By Fatcover Colorer



(a) The *Greedy Intervals List* (b) The *Greedy Registers Array*

Figure 3.7: Data Structures Used By Greedy Colorer

Chapter 4

The Spilling Phase

4.1 An Introduction To The Spilling Phase

In Chapter 3, we assumed that the target architecture always provides the number of registers needed to color the interval graph without causing any spilling to occur. The aim was to color the interval graphs using the minimum number of colors.

However, in many instances, we may have fewer than the minimum number of registers required by the coloring process. As it was outlined in Chapter 1, whenever the interference graph becomes uncolorable, Chaitin's algorithm introduces spill code, rebuilds the interference graph and reinvokes the coloring algorithm. This multi-pass process may be iterated several times. In contrast, our algorithm avoids this repetition. Since the width of the interval graph gives us the number of registers required to color it, we introduce spill code first and, appropriately transform the interval graph which can then be colored in one simple pass using one of the algorithms described in the previous chapter.

The one step spilling phase of our allocation method (Fig. 2.1) constitutes the subject matter of this chapter. We need to understand

- the algorithm that is used in the spilling process as well as
- the heuristics used by the algorithm to choose intervals to spill.

In essence we are dealing with Problem 2 of Chapter 2. The problem was defined as follows :

- Problem 2 (Finding a k-coloring of a Cyclic Interval Graph with Minimum Spilling Cost):
  Given a set of live ranges represented by a cyclic interval graph G and a set of k registers, find an assignment of the k registers for the intervals in G. Introduce spill code when necessary, and keep the spill cost to a minimum.

Given that we have a limited number k of registers, the objective is to use them as best as we can with minimal sacrifice of the performance of the program. We introduce an algorithm that spills intervals when necessary to ensure that the graph is colorable with k-colors.

The rest of this chapter is structured such that Section 4.2 describes the different cases where spilling can occur in the interval graphs. Section 4.3 outlines a new spilling algorithm which has been developed, the *sweep and split* algorithm. Implementation details are provided in Section 4.4. Lastly, Section 4.5 points to some shortcomings and optimizations made by our algorithm[1].

4.2 Chameleon Intervals, and Register Spills

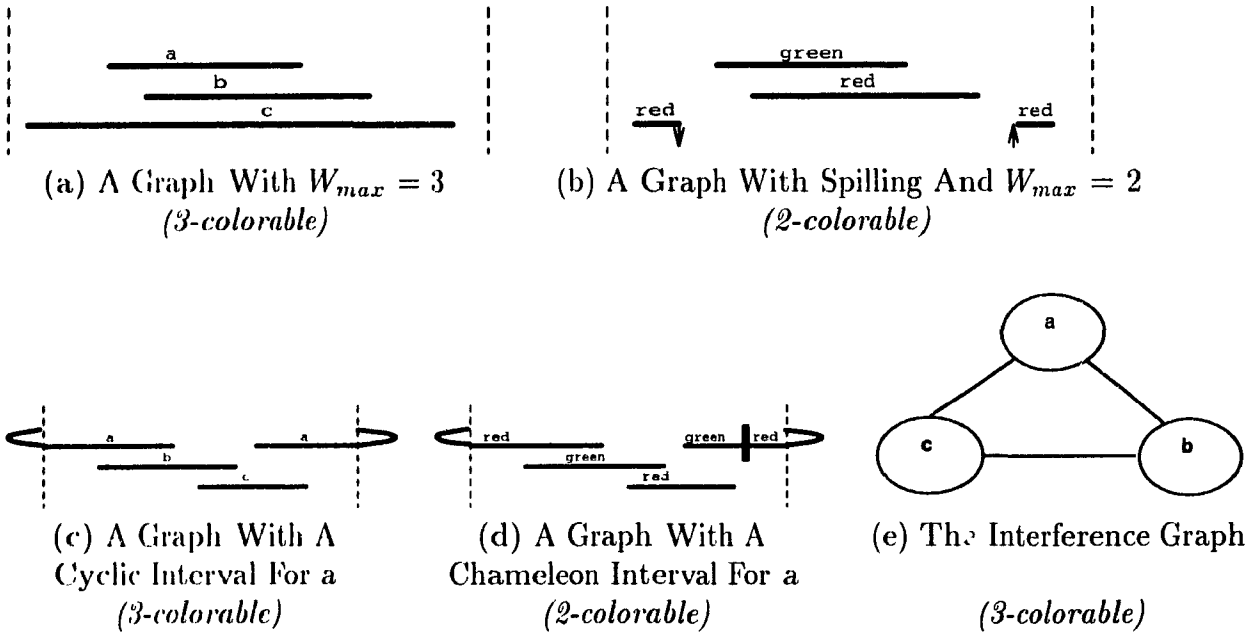
In this section, we identify two cases which could lead to the introduction of spill code.

- $W_{max}(G) > k$:
  This case is most evident. If a graph G has some time, t_i, where there are more than k intervals covering t_i, then it is impossible to allocate a different color to each interval at t_i. For example, consider the graph given in Fig. 4.1(a). Here there are three intervals, a, b, and c that overlap. The only way in which this graph can be colored with 2 colors is to spill one of the intervals to memory. We illustrate this process in Fig. 4.1(b), where the interval for c has been spilled leaving two short intervals representing the definition of c followed by a store to memory ($\downarrow$) and a load from memory ($\uparrow$) followed by a use.

  It was important to choose c to spill. Spilling either a or b would not result in reducing $W_{max}(G)$ to 2. Both would have to be spilled to reduce the thickness of the graph to 2. For instance, had a been chosen to to be spilled at the point where the thickness of the graph becomes 3, then it would have to be reloaded from memory at its' next point of use. At the point where it is reloaded $W_{max}(G) = 3$ and, b or

[1]Some parts of this chapter has been excerpted from [HGAM92].

c would have had to be spilled in order to reduce $W_{max}(G)$ to 2. This would entail spilling two variables instead of just one. Hence two load and two store instructions would have to be introduced instead of just one load and one store instruction. It is quite apparent that timing information about the next use of a variable can be very helpful in making a good choice of an interval to spill.



(a) A Graph With $W_{max} = 3$ *(3-colorable)*

(b) A Graph With Spilling And $W_{max} = 2$ *(2-colorable)*

(c) A Graph With A Cyclic Interval For a *(3-colorable)*

(d) A Graph With A Chameleon Interval For a *(2-colorable)*

(e) The Interference Graph *(3-colorable)*

Figure 4.1: An Example of Register Spilling and Register Floating

- $W_{max}(G) = k$:
  The second situation is more subtle. Consider the graph given in Fig. 4.1(c). This graph has a maximum width of 2, but is not 2-colorable. In this situation we have not really run out of colors, and we need not resort to spilling in order to make this graph 2-colorable. Instead, we use the notion of a *chameleon interval*, an interval that can change color depending on its surroundings. Since the width of the graph is 2, the spiller mechanically assumes that the graph is 2-colorable and leaves this interval graph exactly as it is for the coloring phase.

In the coloring phase, if we allow the interval for variable a to change hue at the location indicated by the solid bar in Fig. 4.1(d), then we can easily color this graph with only two colors. Thus, instead of introducing the loads and stores required for a register spill, we need only introduce a register move that corresponds to the location that interval a changes from green to red. We call this register move operation a *register float* - a value floats from register to register, but is **not** spilled.

By using chameleon intervals to find register floats, we can color any cyclic interval graph G that has $W_{max}(G) = k$ with exactly k colors without introducing any spilling. This is because any graph with $W_{max}(G) = k$ that is not immediately k-colorable must belong to the class of graphs that can be colored if we allow chameleon intervals (as illustrated in Fig. 4.1(d)).

Thus, we can use our fat cover algorithm to color the graph, and for each cyclic interval that cannot be covered, we simply introduce a chameleon interval. No extra loads or stores need be introduced: we simply introduce a register float for each chameleon interval. Since we introduce chameleon intervals only for the cyclic intervals that do not have a fat cover, the number of chameleon intervals introduced is small (at most $W_{min}(G)$).

If more than one register float is introduced, it is possible that some of the register moves depend on each other. For example, perhaps a red interval needs to turn to green, and a green interval needs to turn to red. This can be accomplished either by rotating values through a temporary register, or by swapping the contents of registers using a trick such as *a := a xor b; b := b xor a; a := a xor b*. It is most straightforward to use a temporary, and since the number of cyclic intervals is likely to be less than the maximum width of the graph, a temporary register will be available.

The idea of register floats is not new [CCK90], however difficulty in efficiently identifying values to treat as register floats has prevented their widespread use. For instance, Fig. 4.1(e) shows the interference graph for the interval graph of Fig. 4.1(c). Since *a*, *b*, *c* are all seen to conflict with each other, both Chaitin and Briggs' algorithm will try to assign distinct colors to them. As we have only 2 colors available, spill code will be introduced. Unlike the interval graph, the interference graph does not encode the exact times of overlap between nodes (or intervals). This makes it very difficult to identify opportunities where nodes could share a color with a neighbor for a part of its' live range. This could lead the traditional approach to miss opportunities to prevent a spilling by inserting register moves instead. Our interval graph representation provides a natural mechanism—chameleon intervals - for recognizing when to use register floats and the quantities on which to use them.

4.3 The Sweep And Split Algorithm

4.3.1 The Problem Definition

Given that we have the coloring algorithm described in Chapter 3, the problem of k-coloring now reduces to the problem of transforming a graph G, with $W_{max}(G) = k'$, $k' > k$, to an equivalent graph G' with $W_{max}(G') = k$. Since we are trying to reduce the width of a graph (as shown in Fig. 4.1(b)), this transformation must introduce register spills. Therefore, we would like an approach which attempts to minimize the number of register spills.

On the other hand, if $W_{max}(G) = k'$, $k' = k$, then the spilling phase leaves the interval graph untouched. It is the coloring phase that detects chameleon intervals.

4.3.2 An Overview

We have developed an algorithm, the *sweep and split* algorithm that is based on the cyclic interval graph representation.[2] Like the fat cover coloring algorithm, the sweep and split algorithm takes advantage of the extra information available in the interval representation. Since this algorithm is straight-forward, we only give an overview.

The central idea of the algorithm is to sweep from left to right over the cyclic interval graph. The invariant is that at each time step i, any time to the left of time i is guaranteed to have a maximum width $W_{max}(G, i) \leq k$. To move to the next time step, $i+1$, there are two situations. The first is that $width(G, i) \leq k$, and the second is that $width(G, i) = k', k' > k$. In the first case, no action is required. In the second case, one must select $k' - k$ intervals to split by introducing spill code. Thus, the only difficulty is developing a good heuristic for selecting which intervals to split.

We have developed a heuristic that uses information about time which is readily available from our interval graphs. This heuristic favors intervals that will clear the longest time interval to the right of i. For non-cyclic intervals this is equivalent to choosing the one with the farthest next use from i. Note we do not split all the intervals of the variable, but only the segment that overlaps time i. The other segments will be split only if the sweep selects those intervals as the ones to split at some later step i'. The reasoning behind our heuristic is that according to the invariant, all times to the left of i have already had their widths reduced, and so we should favor intervals that will reduce widths to the right of i.

[2] A similar method was proposed for basic blocks in [CH84]

If multiple intervals clear the same longest distance, an interval that requires only a load, is preferred over an interval that requires both a load and a store, and if a store is required, then a store that is outside of the loop is preferred.

4.3.3 An Example

Let us now go through a concrete example of applying the sweep and split algorithm to a cyclic interval graph that corresponds to a small program. In Fig. 4.2(a) we give an illustrative program, and in Fig. 4.2(b) we give the 3-address code.[3] Assuming that the number of available registers (k) is 3, Fig. 4.2(c) gives the 3-address code program that results from applying the sweep and split algorithm. Fig. 4.2(d) shows the cyclic interval graph for this loop, and Figs. 4.2(d-f) illustrate the sweep and split process that transforms the original graph into one that has a maximum thickness of 3 (i.e. transforms it into a 3-colorable graph). These steps are as follows:

Fig. 4.2(d): (Time = 1) The sweeping process starts at time step 1. Note that the sweeping line is indicated by the vertical dotted line. There are 6 intervals covering time step 1. In order to reduce the width to 3, three intervals must be selected to be split. Interval c cannot be split because its use is at time 1. Of the remaining intervals, the best intervals to select are a, b, and n. Splitting each of these intervals frees the longest time interval to the right of the sweeping line.

Fig. 4.2(e) (Time = 2): This figure illustrates the graph resulting from splitting the a, b, and n in the previous step. There are four intervals covering time step 2, and so interval c is chosen to split.

Fig. 4.2(f) (Time = 3): There are four intervals covering time step 3, and interval $t1$ is chosen because it has the rightmost next use. We can see the importance of our heuristic here. Note that in this situation, choosing $t1$ is very important, choosing $t2$ would not reduce the width at time step 4, and another spill would have to be introduced.

Fig. 4.2(g) (Time = 4): At time step 4, and all times greater than 4, the width is 3, and so no further spilling is required.

[3] Assuming right-associativity, and right to left order of evaluation.

```
while (i<n)
  { sum =
      b[i] * a[i] + c + sum;
    i = i + 1;
  }
```

(a) Original Loop

```
L1: t1 = c + sum
    t2 = a[i]
    t3 = b[i]
    t4 = t3 * t2
    sum = t4 + t1
    i = i + 4
    if i < n goto L1
```

(b) 3-address Code

```
L1: load c
    t1 = c + sum
    store t1
    load a
    t2 = a[i]
    load b
    t3 = b[i]
    t4 = t3 * t2
    load t1
    sum = t4 + t1
    i = i + 4
    load n
    if i < n goto L1
```

(c) 3-address Code After Spilling

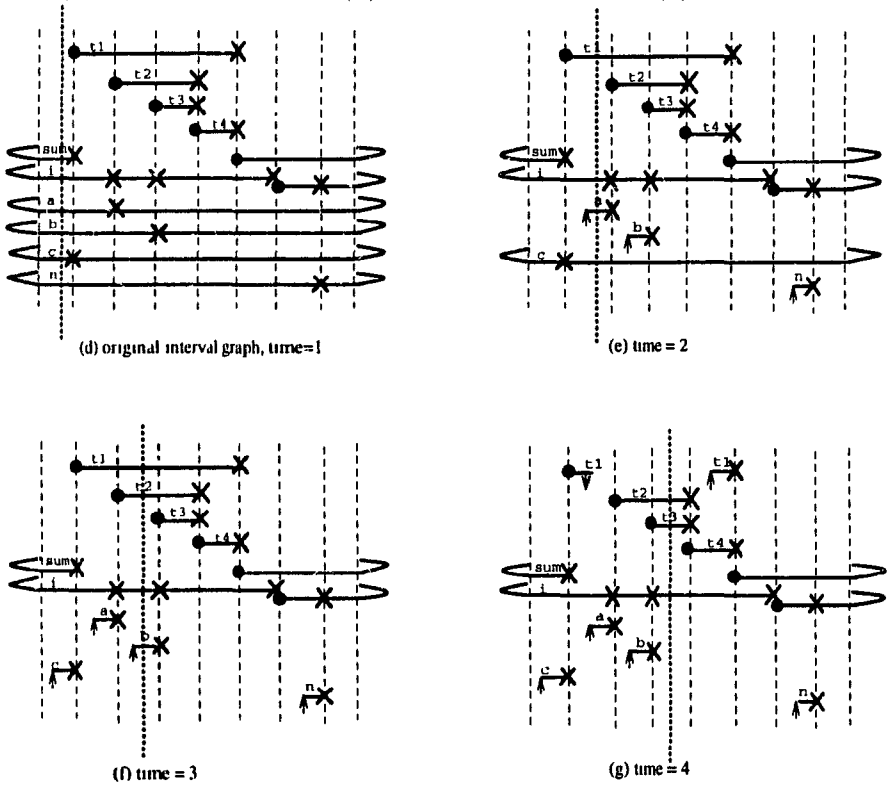


Figure 4.2: An example of introducing spill code

4.4 Implementation Details

In the last section we provided an intuitive description of the sweep and split algorithm through an example. In this section we formalize the method into an algorithm and provide a description of the data structures that have been used in its implementation.

4.4.1 The Implemented Sweep And Split Algorithm

We describe the implemented sweep and split algorithm referring to detailed descriptions of various routines which are used in its' implementation. The main procedure is enclosed by a box in this section while subsidiary procedures are in the next section.

1. *Main*
   The expected inputs and the output of the algorithm are given in *Input* and *Output*. The *Main* procedure sweeps the graph from left to right and at each time spot, t, where the $width(G, t) > W_{max}(G)$, $width(G, t) - W_{max}(G)$ intervals are chosen to be spilled so as to reduce the thickness of the graph at t to $W_{max}(G)$. $width(G, t)$ is also defined as W_t in the description of the algorithm. *Main* refers to the two subsidiary functions mentioned below.

2. *Heuristic Based Choice Of Intervals*
   This function is in Section 4.4.2 and outlines the method used in choosing an interval to spill.

3. *Finding The Width of Interval Graphs*
   As a preprocessing step to the spilling phase, the width of G is found for every time step of the graph. This process is described in *Finding the Width of Interval Graphs* of Section 4.4.2.

Algorithm 4.4.1 *Perform the sweep and split spilling algorithm described in Section 4.3 on a cyclic interval graph.*

Input: A `cyclic interval graph, G,` constructed using the def-use information of the variables,

The `register classes`, $class_{reg}$, present on the target architecture,

The `number of registers`, reg_{avail}, present in each class of registers on the target architecture,

The `register preference`, reg_{pref}, of each interval of the cyclic interval graph. This determines the class of register that needs to be assigned to each interval. And,

The `number of registers`,reg_{num}, needed by each interval of the cyclic interval graph.

Output: **FOR** each class of register, $class_{reg}$,
A transformed cyclic interval graph, G', such that $W_{max}(G') = k', k' \leq k$,
where $k = reg_{avail}$, and,
reg_{avail} = `number of available registers` for $class_{reg}$.

Procedure Main :

FOR each class of register, $class_{reg}$,
  FOR each time instant, t_i, in G
    Find the width, W_{t_i} of G using
    function *Finding The Width Of Interval Graphs*, $W_{t_i}(G)$
    and record $W_{max}(G)$

  IF $W_{max}(G) > k$ **THEN**
    FOR each time instant, t_i, in G
      IF $W_{t_i} > k$ **THEN**
        Using heuristics in function
        Heuristic Based Choice Of Spill Intervals,
        choose $(W_{t_i} - k)$ intervals to **spill**
        FOR each interval to be spilled, $spill_{int}$,
          IF $spill_{int}$ is `acyclic` **THEN**
            IF $spill_{int}$ has been defined at $time < t_i$ **THEN**
              Introduce a `store` instruction at $t_i - 1$
            IF $spill_{int}$ is used at $time > t_i$ **THEN**
              Introduce a `load` instruction before the next use
              of $spill_{int}$
          ELSIF $spill_{int}$ is the `front-end` of a cyclic
          interval **THEN**
            IF $spill_{int}$ has been defined at $time > t_i$,
            that is, at the tail-end of the cyclic interval, **THEN**
              Introduce a `store` instruction at $t_i - 1$
            IF $spill_{int}$ is used at $time > t_i$
              Introduce a `load` instruction before the next use
              of $spill_{int}$
          ELSIF $spill_{int}$ is the `tail-end` of a cyclic
          interval **THEN**
            IF $spill_{int}$ has been defined at $time < t_i$ **THEN**
              Introduce a `store` instruction at $t_i - 1$
            IF $spill_{int}$ is used at $time > t_i$
             || $time < t_i$ (at the front-end of the cyclic interval)
              Introduce a `load` instruction before the next use
              of $spill_{int}$
      ELSE
        Continue with outer **FOR** loop
DONE

4.4.2 Two Important Steps Used By The Spiller

The sweep and split algorithm uses two important steps – (1) to find the width of the interval graph and, (2) to choose intervals to be spilled. We look into these two steps in more detail.

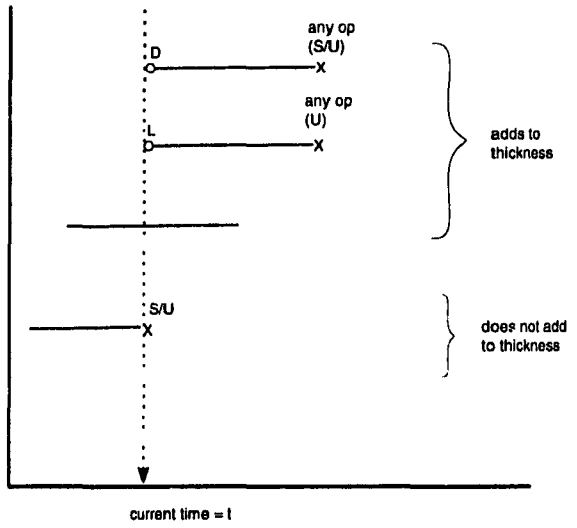


Figure 4.3: Finding The Thickness Of The Interval Graph At A Time, t

Finding The Width of Interval Graphs, $W_{t_i}(G)$:

Fig. 4.3 illustrates how the width of the interval graph is computed at a time instant t_i. The letters `D` mark a point of definition, `L` a point of load, `S` a point of store and finally, `U` a point of use for the interval. Time t_i is represented by the dotted line in the graph. At t_i, we see that the topmost interval is defined, while the second interval is loaded from memory. Even though the third interval is live at t_i, it is not referenced. All these three intervals are live at time t_i and contribute to the width, $W_{t_i}(G)$, of the graph. The fourth interval is last used at this time instant, and essentially dies. So, it doesn't add to the thickness of the graph. Assuming that each interval of the example graph requires one register to be assigned to it, $W_{t_i}(G) = 3$.

Heuristic Based Choice Of Spill Intervals : If at a time, t_i, $W_{t_i}(G) > k$ then we need to choose intervals to spill. All the intervals which are live at t_i, and are

not referenced (through load, store, def or use instructions) are considered to be candidates for spilling. The crux lies in choosing an interval from amongst all these candidates such that the spill cost (in terms of run time) is as small as possible.

In our current implementation, we choose to spill the interval that is referenced farthest away from the time at which the spill occurs. Information about the next time of reference of an interval is very easily available from an interval graph, but it is unavailable in the traditional interference graph.

Other criteria which are often used for spilling are :

- frequency of reference made to an interval [ASU88],
- intervals having a high degree of conflict with other intervals [CAC+81],
- intervals having a high ratio of degree of conflict with cost to reload the variable [Bri92].

The sweep and split algorithm could these above criteria when choosing an interval to spill as well.

4.4.3 The Data Structures Used

In order to understand the complexity of the implemented spilling algorithm, we need to consider the data structures used.

Fig. 4.4 shows the main data structure used by the sweep and split algorithm. Even though we conceptually think of spilling *interval graphs*, at this point, we do not create intervals. We maintain lists of variables (or symbolic registers) and the operations (like load, store, def and use) that they are involved in. Lists are maintained for easy insertion of spill code and operations are exposed so that some optimizations (which are explained in Section 4.5) can be easily performed. It also allows the spiller to exploit information about the next reference made to the symbolic register.

- **Variable access list (var list) :**
  The main part of this structure is named `var list`. The names of the structures appear in solid boxes while the size of the structures are enclosed by dashed boxes in the diagrams. Var list is essentially an array of records. *There are as many records in var list as there are variables (and not intervals) in the basic block* being considered. Each variable that is encountered in the basic block is given an unique number. Recall from Chapter 2.1 that a variable gives rise to several intervals. An interval is akin to a node of the traditional renumbered interference graph. Each record in var list stores information about one variable. The fields of each record are :

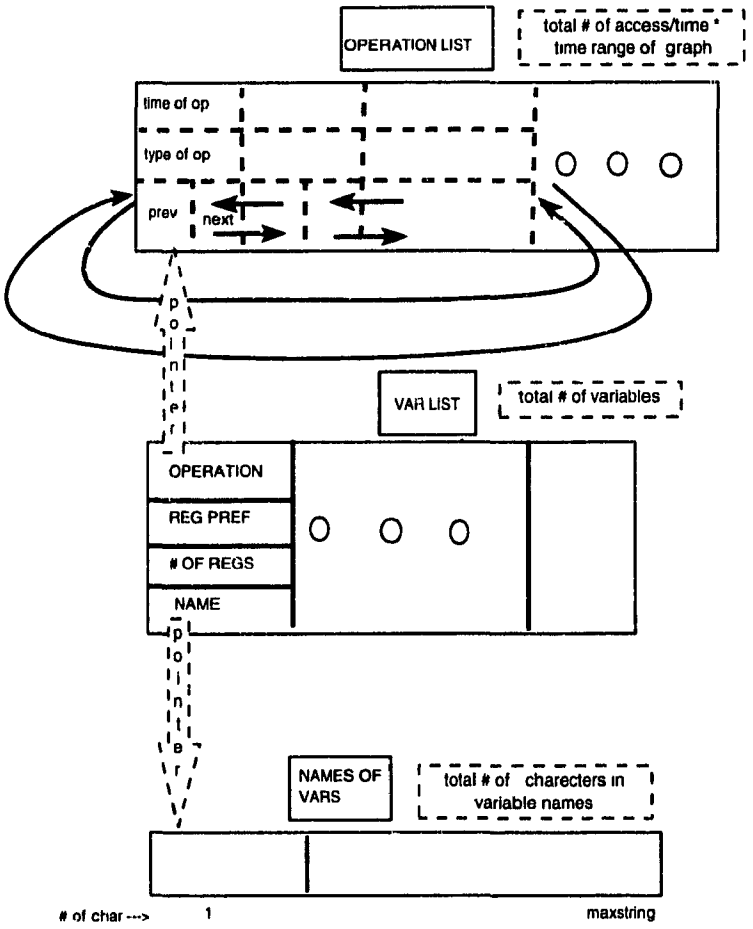


Figure 4.4: Data Structure Used By The Sweep And Split Spiller

1. Name : This points to an array of characters that store the name of the variable. Names of variables are used for debugging purposes.
2. Register pref(erence) : This records the class of register, $class_{reg}$, that is to be assigned to all the intervals of this variable.
3. (Number) # of regs (registers) : This stores the data mode of the variable. Short variables require single registers, while long or, double require two registers to be assigned to them. The number of registers required by each variable affects the width of the interval graph for that class of register.
4. Pointer to **operation list** : The operation list is a doubly linked circular list of all the operations that each variable is involved in. Each cell of the list contains the following information :
   (a) Time of op(eration) : The time at which an operation takes place. This time corresponds to a time in the cyclic interval graph.
   (b) Type of op(eration) : The possible operations which we take into account are loads, stores, defs and uses.
   (c) Next pointer : Points to the next cell of the list.
   (d) Prev(ious) pointer : Points to the previous operation of the operations list.

   At any time, t_i, the pointer to operations list is updated to point to an operation that executes at t_i if one exists, or to an operation that executes at a time step $> t_i$ when the variable is not accessed at t_i.

The main field of var list that is used by the spiller is the "pointer to the operation list". There is a three fold use for this field of the structure :

- To calculate W_{t_i} :
  The average complexity to find the width of G at time t_i is of the order $O(n)$, where n =number of variables in the block.

  Finding the width of the graph at time t_i could be potentially expensive and the complexity depends on the kind of data structure used to implement the algorithm. Since at time, t_i, the pointer to the operations list from var list always points to the next operation to be executed, we need to check the time of the operation of every variable to find out whether or not it is alive at t_i. So, the complexity of this step is of the order $O(n)$, where n =number of variables in the block. Recall that we keep track of all the operations of of every variable in the code. Therefore, n(*the number of variables*) $<$ *number of intervals*. Worst case performance is obtained when static single assignment form is followed so that each variable gives rise to exactly one interval. In this case, n = the number of intervals of the graph.

- Use in choosing a spill interval, $spill_{int}$:
  A quick answer to a question like

"How far away is the next reference to this variable?"
can be provided by checking the time of execution of the operation pointed to by the "pointer to operations list" field of var list for that variable. Such a query is requested by the spill heuristic when choosing an interval to spill from amongst a pool of potentially qualified intervals.

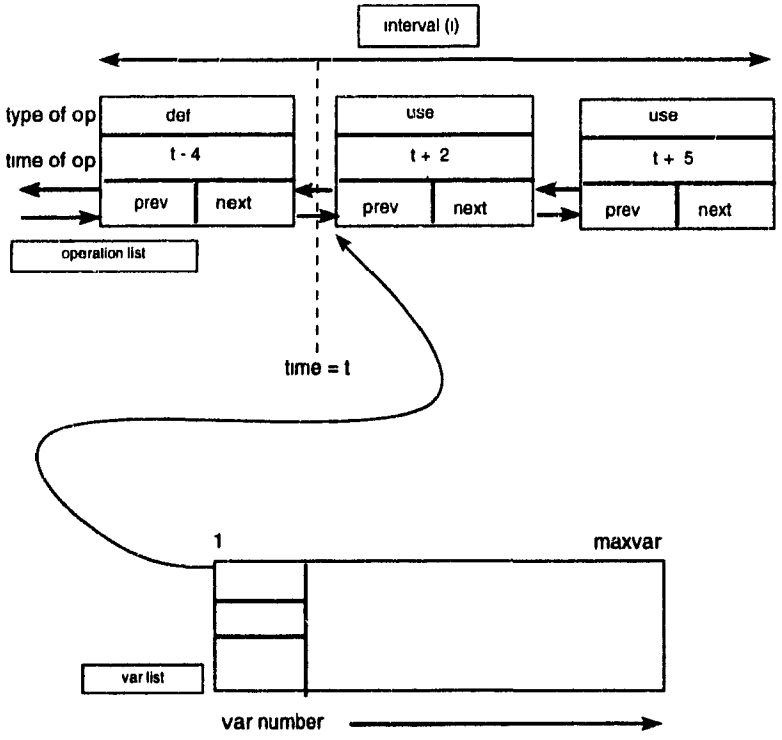


Figure 4.5: Use Of Var List In The Spilling Phase

- Inserting spill code for interval, i :
  When an interval, i, is chosen to be spilled, the linked list of operations provide an easy access to the previous and next operations of i. Inserting load and store instructions entails adding new operations to the "operations list" and this can be done in constant time. When we are at time t which is the dotted line in Fig. 4.5, interval i is chosen to be spilled. The "pointer to the operations list" field of the variable points to the next reference made to the variable at time $t + 2$. Since no operation involving i exists for time t it could be chosen to be spilled. As the operation at $t + 2$ is an use of the

variable, and it is defined at $t-4$. As this definition of the variable is not stored in memory before it is used at $t+2$, a **load** and a **store** instruction has to be inserted. The **store** operation is inserted at time $t-1$ while the **load** operation is inserted at time $t+1$ just before the next use of the variable.

It is essential to know the previous and forthcoming operations to assess whether or not spill and reload code have to be introduced. Blind insertion of load and store operations may create redundant load and store operations. We elaborate on this in Section 4.5.

4.5 Optimizations And Caveats

While spilling, certain optimizations like the ones listed below are performed :

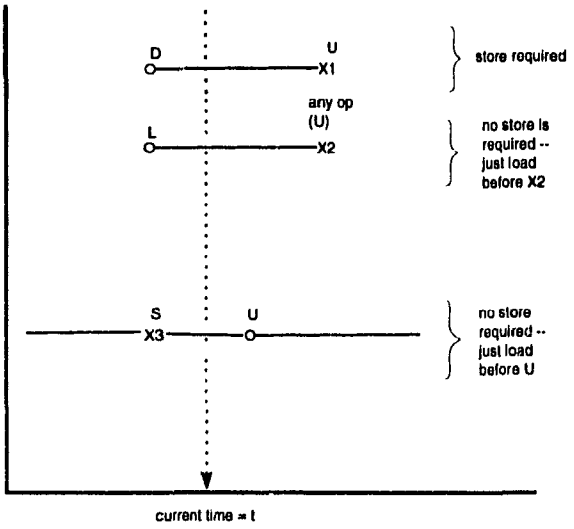


Figure 4.6: Finding Out If A Spilled Interval Requires To Be Stored And Loaded

Avoiding introduction of unnecessary store operations :

Assume that while scanning the interval graph from left to right, we have arrived at time instant t which is the dotted line in Fig. 4.6. Let us further assume that the top-most interval of this graph has been chosen to be spilled as its point of next use

is farthest away from t. As this interval is defined but not stored till its point of next use, it has to be stored in memory in case it is it is used by some later instructions. However, if the next intervals had been chosen to be spilled, it would not have to be stored. This is because the symbolic registers have not been redefined by any of the operation previous to time t.

Eliminating redundant load and store operations :

The spilling phase also looks for opportunities to pack code more tightly by removing one of the load instruction from a load followed by a load instruction sequence, or a store from a store followed by a store instruction sequence.

Before ending this chapter, we broach on a popular problem which appears to be endemic to register allocators.

Miscommunication between the spiller and the colorer :

At times the spiller may decide that a graph is colorable with k colors while the colorer may require more than k colors. How does this happen?

The spiller is oblivious to the adjacency and alignment constraints imposed on multi-register operands by the architecture. The spillers decision to k-color the graph could be respected by the colorer if register constraints were not taken into account. However, when coloring, these constraints are typically brought into the picture and this causes problems.

For example, imagine an architecture with three registers, $reg1 \ldots reg3$. We need to map two pseudo registers, $p1, p2$, to these hardware registers. $p1$ requires two contiguous hardware registers which are aligned on an even/odd boundary, while $p2$ can be assigned any single register. If $p2$ is assigned $reg2$ then, $p1$ can no longer be assigned any of the available registers, *reg1, reg3 or, reg4*, as they do not respect the alignment and adjacency requirements of $p1$. Notice that even though there are sufficient registers available, they can't be assigned because of architectural constraints. This problem is not unique to our allocation method. Other register allocators face the very same problem.

Since our allocation strategy involves a two step process where the spilling and the coloring phases are run only once we cannot insert spill code if the colorer discovers that the available registers can not be assigned as they violate the adjacency and alignment constraints of the interval. However, we could attempt to perform register moves so as to shift other intervals into different registers to accommodate the requirements of the multi-register interval. But this is a rather convoluted approach. Instead, we would like to have a spilling phase that takes architectural constraints into account. This is an area of future research.

4.6 Summary

In this chapter, we introduce the concept of chameleon registers and identify instances where spilling is required in cyclic interval graphs. We outline the sweep and split spilling algorithm illustrating it with an example. Implementation details are described as well. Some optimizations which are performed in the spilling phase are noted and the caveats are also mentioned.

Chapter 5

Results and Analysis

Chapters 4 and 3 described the sweep and split algorithm which is used for the purposes of spilling and, the fatcover and greedy algorithms which are used to color cyclic interval graphs. In this chapter we report quantitative results of the effectiveness of our cyclic graph based spiller and colorers. We compare

- the performance of the spilling and coloring algo .thms with that of commercial compilers in Sections 5.1 and 5.2.5 respectively.
- The coloring algorithms are also evaluated with a couple of other standalone coloring schemes and details of the results are described in Section 5.2.1.

Besides reporting results, each section describes the assumptions made and the process followed to obtain the results.

From our experiments we notice that, for the test programs we used, the sweep and split algorithm generates about 90% fewer spill load operations and 37% to 56% fewer store instructions than the Sparc CC and MIPS C compilers. A marked difference in the performance of the commercial compilers and our spiller is seen when loops are unrolled and have increased register pressure.

We also compared the performance of the fatcover and greedy algorithms to an optimal coloring algorithm. The fatcover algorithm requires only 5.2% more registers on the average than the optimal colorer!

5.1 Performance Of Sweep And Split Spiller

Standalone versions of our spilling and coloring algorithms were implemented[1]. In this section we compare the performance of our interval graph method of spilling to the performance of three advanced production *C* compilers for the *IBM RS6000* (Version 1.01.0003.0013), the *Sun Sparc* (version bundled with SunOS 4.1.1), and the *MIPS* (Version 2.11.2). Our comparisons use the highest level of optimization offered by these compilers. Each of these three architectures has 32, 32-bit integer registers. The RS6000 also has 32, 64-bit floating point registers, while the Sparc and the MIPS have 16, 64-bit floating point registers.

Before providing results in Section 5.1.3, we outline the process that was followed to get to the point where results could be obtained.

5.1.1 Obtaining Interval Graphs

We focus on two inner loop bodies, one taken from Livermore Loop 8, the other from the Tomcat[2] SPEC89 benchmark (Release 1.2).[3] Both of these benchmarks were selected because of the relatively large size of their loop bodies and the large number of variables referenced. This large size is necessary in order to evaluate the efficiency of register allocation and spilling on these three architectures with their large register sets. Both benchmarks are also floating point intensive and use double precision (64-bit) arithmetic. Hence we concentrate on the allocation and spilling of floating point registers. Although this restriction does limit the scope of our results, it provides a test of our approach on problems of real use.

1. Input for commercial compilers :
   In order to give all compilers approximately equal input,

   (a) we first transformed the input source code so that all of the complex transformations had been made explicit.

   (b) Both loops were manually unrolled and software pipelined.

   (c) Common subexpression elimination was performed and reused data values were explicitly assigned to local scalar variables.

[1] Most of this section has been excerpted from [HGAM92].

[2] To be precise, the Tomcat loop is the "I-LOOP" beginning with "DO 250 I= 1IP,I2M".

[3] The Standards Performance Evaluation Corporation (SPEC) benchmark suite can be obtained from 39510 Paseo Padre Parkway, Suite 350, Fremont, CA 94538 These benchmarks have been derived from publicly-available CPU intensive application programs.

This transformed code is isomorphic to the interval graphs which were fed through our spiller and allocator. These aggressive optimizations performed tend to increase the lifetime of variables thus increasing the importance of the register allocator. Thus no sophisticated analysis of array indices nor any unusual optimization was required by the commercial compilers to match the performance of our standalone implementation.

2. Input for our standalone system :
   Interval graphs are created using the hand transformed source code obtained from (1) above. In order to be as fair as possible we generated two interval graphs for each benchmark tested.

   (a) In the first interval graph we assumed that instructions are executed in their source code order.

   (b) However, most modern compilers do not generate code in source code order—sophisticated instruction scheduling is performed to minimize the pipeline stalls and the effect of hazards. In particular LOAD's are often moved long before their first use. Such scheduling tends to increase register pressure. Hence in our second interval graph we moved LOAD's as far as possible before their first use. Since we are dealing with cyclic interval graphs, this means moving LOAD's immediately after the last use of the previous iteration's value. In most cases, this is far earlier than the compiler would need to schedule the LOAD to avoid stalls, however it allows us to make an honest comparison with production compilers.

As will be seen shortly, our approach generates few spills on the scheduled interval graphs that are not present in the unscheduled graphs.

5.1.2 Assumptions Made

We make several assumptions when generating the interval graphs. We assume that

1. in an instruction like `x1 = x2 + 4` the same register could be used for both `x1` and `x2` if `x2` were dead after this point.

2. For simplicity we also assume in constructing our interval graphs, that all instructions execute in unit time. This assumption biases our results towards needing more registers. For example, if in executing a floating point divide, the destination register is not filled for 20 cycles after the initiation of the divide, that register could be used for some other purpose during those 20 cycles.

5.1.3 Results

In generating spills we scan the interval graph from left to right. Whenever the thickness exceeds the number of registers, the excess is spilled. The registers chosen for spilling are those whose next use is most distant. Since these are cyclic interval graphs, the distance measure is cyclic, i.e. distance is measured from the current time to the next use, wrapping around at the end of the iteration.

- When 16 registers are available :

  - Tomcat (Table 5.1) :
    When only 16 registers are available, our method requires the introduction of spills for these benchmarks. However as can be seen in Table 5.1, the number of spills required is substantially less than that required by either the Sparc or the MIPS compilers. The reduction in *load* spills ranges from 6 loads *per iteration* (or 37% for the rolled scheduled graph in comparison to the MIPS code) to 73 loads *per iteration* (or 56% for the scheduled unrolled graph in comparison to the SPARC code)! These are proportional to the dynamic loads and stores performed. 37% of the loads are reduced in the scheduled graph in comparison to the code generated by the MIPS C compiler, while 56% of the loads are reduced in the unscheduled graph in comparison to the code generated by the SPARC C compiler. The reduction in *store* spills ranges from 0 to 71 stores *per iteration*. 71 stores of the 79 stores generated by the MIPS compiler is eliminated in the scheduled unrolled graph. That is 90% of the store instructions are removed! Please note that the total number of loads and stores include the loads and stores introduced by spilling as well as the intrinsic loads and stores which are the first reference or final store of array elements respectively.
  - Livermore loop 8 (Table 5.2:
    It is also clear from Tables 5.1 and 5.2 that there is little difference in the number of spills of the scheduled and unscheduled interval graphs. Unrolled Livermore Loop 8 goes from 59 load spills in the unscheduled graph to 65 in the scheduled graph, while rolled Loop 8 has a more dramatic increase from 1 to 6.

  Note that the result in all cases is better than the performance achieved by either of the commercial compilers. These results also show the robustness of our approach: scheduling substantially increases the register pressure - from $W_{max} = 17$ to $W_{max} = 30$ for rolled Loop 8, from $W_{max} = 40$ to $W_{max} = 57$ for unrolled Loop 8, and from $W_{max} = 24$ to $W_{max} = 38$ for rolled Tomcat, yet relatively few additional spills result.

- When 32 registers are available :
  Tables 5.3 and 5.4 give analogous results when 32 registers are available on the

	Rolled				Unrolled - 3×			
	Interv Graph		SPARC	MIPS	Interv Graph		SPARC	MIPS
	Unsch	Sched			Unsch	Sched		
LOADS(total)	27	28	41	34	71	73	144	131
LOADS(spill)	9	10	23	16	53	55	126	113
STORES(total)	4	4	8	17	20	20	66	91
STORES(spill)	0	0	4	13	8	8	54	79

Table 5.1: Number of Double Precision Loads and Stores with 16 registers for Tomcat.

	Rolled				Unrolled - 3×			
	Interv Graph		SPARC	MIPS	Interv Graph		SPARC	MIPS
	Unsch	Sched			Unsch	Sched		
LOADS(total)	16	21	23	22	107	113	158	147
LOADS(spill)	1	6	8	7	59	65	110	99
STORES(total)	6	6	6	6	24	24	34	40
STORES(spill)	0	0	0	0	6	6	16	22

Table 5.2: Number of Double Precision Loads and Stores with 16 registers for Loop 8.

RS6000. The increased number of registers alleviates the need for many of the spills. The interval graph method allows the rolled version of Loop 8 to execute with no load spills in either the scheduled or unscheduled interval graphs.

	Rolled			Unrolled 3×		
	Interv Graph		R6000	Interv Graph		R6000
	Unsch	Sched		Unsch	Sched	
LOADS(total)	18	22	24	18	29	66
LOADS(spill)	0	4	6	0	11	18
STORES(total)	4	4	4	12	12	31
STORES(spill)	0	0	0	0	0	22

Table 5.3: Number of Double Precision Loads and Stores with 32 registers for Tomcat.

	Rolled			Unrolled 6×		
	Interv Graph		R6000	Interv Graph		R6000
	Unsch	Sched		Unsch	Sched	
LOADS(total)	15	15	16	33	34	81
LOADS(spill)	0	0	1	8	9	56
STORES(total)	6	6	6	19	21	51
STORES(spill)	0	0	0	1	3	33

Table 5.4: Number of Double Precision Loads and Stores with 32 registers for Loop 8.

Most interesting however, is our performance with the unrolled versions of Tomcat and Loop 8. Even using the scheduled interval graph, our approach generates only 11 load spills for unrolled Tomcat versus 48 for the R6000 (Table 5.3). We generate no store spills versus their 22. On Loop 8 the numbers are equally impressive 9 load spills to the R6000's 56 and 3 store spills to their 33 (Table 5.4).

The main point to be made from these experiments is that the cyclic interval graph approach does a good job on complex loops that have high register pressure. This can be seen by the very low number of loads and stores required for all cases, and the use of register floats instead of spills on the unrolled `tomcat` case. This is an absolute observation, and does not depend on comparing the results to other register allocation strategies.

The other point, is that when given the same programs, three production-quality compilers produce substantially worse spill code for these challenging examples. Slight variations among these three compilers might be due to slightly different low-level optimizations or instruction scheduling. However, the main point is that all of them performed significantly worse than the interval graph approach.

5.2 Performance Of Various Coloring Algorithms

In this section, we compare the performance of the fatcover and the greedy coloring algorithms (described in Chapter 3) with two other coloring algorithms - an optimal coloring algorithm and, a greedy coloring algorithm. Section 5.2.1 provides an explanation of the latter two coloring algorithms used in this comparative study, Section 5.2.2 describes the method used to obtain interval graphs and, Section 5.2.3 points out the assumptions made when coloring the graphs. Finally, we report comparative results of the various coloring algorithms in Section 5.2.4.

5.2.1 The Coloring Algorithms

We compare the performance of the fatcover and the greedy coloring algorithms with two others – a backtracking sequential algorithm which provides optimal results and a greedy algorithm which is based on heuristics. A brief description of both these algorithms follow.

Backtracking Sequential Algorithm :

This method colors graphs with the minimum number of colors using a backtracking sequential algorithm that expects an interference graph as its input. The optimal algorithm is presented here for the sake of completeness and, has been taken directly from [SDK83].

Assume that we are given a graph, G, with vertices $v_1 \ldots v_n$. The vertices may be ordered arbitrarily. As each vertex, v_i, is being colored, one color from a set of all feasible colors, U_i, is chosen to be assigned to v_i. Colors are represented as numbers, for example color 1 is the same as 1. Since the minimal coloring of the graph is to be found, all possible color assignments are taken into account except those that can lead to nonoptimal colorings or colorings equivalent to the ones which have already been found. Search paths in the backtracking phase are pruned using a branch and bound technique.

We provide a brief informal description of the algorithm. Assume that each new color that is needed to color G is assigned a number. The colors are numbered $1, 2, \ldots$.

1. Assign color 1 to v_1.

2. For each vertex v_i, such that $(v_2 \leq v_i \leq v_n)$

   (a) Find the set of feasible colors, U_i, for v_i.
   Assume that a maximum of l_{i-1} colors have been used to color vertices $v_1 \ldots v_{i-1}$. It may be possible to color v_i with any one of the l_{i-1} colors or, at worst a new color l may be needed. Colors which have been assigned to neighbors of v_i can't be members of the U_i set. Furthermore, if a complete q-coloring of G has already been found, then none of the numbers of the colors of the set U_i should exceed q. This restriction is imposed as we want to explore paths that use fewer than q colors and, it is used to bound the search of the backtracking algorithm.

   (b) If $(U_i = \emptyset)$ then it means that this path can not yield a better coloring of G than one that has already been found. So, we backtrack to vertex v_{i-1} and explore the color assignment of G had v_{i-1} been assigned an alternate color.

3. If a complete q-coloring of G has been found, check to see if q is the minimum number of colors required to color G by traversing the next path of the color tree in the depth first traversal. Essentially this entails backtracking and returning to step (2).

Another Heuristic Greedy Algorithm (Greedy2) :

This greedy algorithm, which we shall call greedy2, was devised by Erik Altman [ARG93] and is described briefly.

The algorithm expects a cyclic interval graph, G, as its input and it outputs a set of colors for each interval of G. G is first processed in preparation of the coloring phase and then it is colored. A modified left edge algorithm is used by the colorer.

1. The input graph G is rotated so that the first time step in the transformed graph G' is the point of minimum width, $W_{min}(G')$. Rotation of G reduces the number of cyclic intervals in G'. The cyclic intervals in the original graph, G, are shifted and some may become non cyclic intervals in G', while some non cyclic intervals of G become cyclic in G'. The number of cyclic intervals in G' is reduced to exactly $W_{min}(G)$. The left edge algorithm is able to color a graph containing no cyclic intervals in $O(n(logn))$ time, where n is the number of intervals in the graph. It is more complex to color cyclic intervals and reducing their number helps in improving the overall time complexity of this greedy algorithm, which in the worst case is $O(n^2)$ but is probably $O(n)$ on the average.

2. Next, each cyclic interval of G' is assigned an unique color and removed from the graph.

   The colors assigned to the cyclic intervals are recorded and marked. A set of all the colors assigned to cyclic intervals, $color\_set_{cyclic}$ is maintained. The set of colors which have not been assigned to cyclic intervals are maintained in another list, $color\_set_{non-cyclic}$.

3. The resulting graph, G'', contains only uncolored non cyclic intervals. A modified left edge algorithm is used to sweep G'' from left to right. As each uncolored non cyclic interval, i, is encountered, a color is assigned to it. The following heuristics guide the choice of a color for an interval :
   - An attempt is made to reuse the colors assigned to the cyclic intervals. Members of the set $color\_set_{cyclic}$ are checked and the first color that is free for the duration of the life time of i is assigned to it.
   - If a color couldn't be found from amongst the $color\_set_{cyclic}$ set, then a free color is chosen from the $color\_set_{non-cyclic}$ set and assigned to i.

5.2.2 Obtaining Interval Graphs

In order to obtain interval graphs for our colorers, we take advantage of the method used by Ning in his thesis [Nin93].

In his thesis, Ning proposes a method of generating time optimal schedules for loops having no embedded flow of control [Nin93, NG93]. The time optimal schedules are also allocated a minimum number of "buffers". Henceforth we shall call these schedules Ning's schedule.

First, time optimal parallel schedules are created for software pipelined loops. These schedules are obtained using integer linear programming methods and, their novelty lies in the presence of "buffers" which arise due to loop carried dependences. Buffers can be viewed as temporary storage places for values being generated by one iteration of the loop. Values in the buffers are consumed by a later instruction in the same or following iteration of the loop. The buffers together with live ranges of symbolic registers of the code are represented as intervals in a cyclic interval graph [NG93, Nin93].

Once a schedule is generated it is colored using the coloring algorithms of Section 5.2.1. We color Ning's schedules (which are cyclic interval graphs) with our fatcover and greedy algorithms. Our results can then be compared with the results obtained using the optimal and the greedy colorers.

As we shall see in the next section, selected loops from the Livermore Loops, SPEC89 and Whetstone were used as benchmarks. The following steps describe how the cyclic interval graphs are generated from the source code of the benchmark programs [Nin93].

1. Since all of the chosen loops are written in Fortran, they are hand coded into three address code which closely matches assembly code. Temporaries are generated when needed and, as in the previous experiments a schedule of only floating point operations are created.

2. Next, Parafrase-2, which was developed at University of Illinois Urbana-Champaign, is used to obtain data dependence information between pairs of instructions in the hand coded three address code.

3. Using the dependence information thus gathered, the optimal period of the loop is calculated and finally Ning's schedule is produced. An integer linear programming method is used to create the optimal schedule.

We color these generated schedules. Some of the interval graphs have also been unrolled before being colored.

5.2.3 Assumptions Made

We make the following assumptions about the target machine :

1. We assume the presence of a multiple instruction issue (VLIW) machine which has multiple adders, multipliers, dividers, copy units, load units and, store units.

2. The functional units are assumed to be pipelined.

3. The following operation latencies are used when generating a rate optimal schedule:

   (a) Add 1 cycle

   (b) Subtract 1 cycle

   (c) Multiply 2 cycles

   (d) Divide 17 cycles

   (e) Negate 1 cycle

   (f) Load 2 cycles

   (g) Store 1 cycle

5.2.4 Results Of Standalone Colorers

Table 5.5 lists the benchmarks which have been used to test the different coloring algorithms. We have carefully chosen perfect loop segments having loop-carried dependences. Our colorer is geared to handle cyclic intervals, and the presence of loop-carried dependences create such cyclic intervals. So, this is an important criteria that guided our choice of benchmark programs. All of the chosen loops were software pipelined.

Specifically, the loops of the benchmarks tested are :

1. Linpack : The ddot function from the linpack benchmark has been isolated. This function forms the dot product of two vectors. ddot1 is the rolled version of the loop, whereas ddot4 has been unrolled four times after software pipelining.

2. The next few benchmarks have been taken from the SPEC'89 benchmarks.

   (a) Doduc : Three loops have been taken from the debico.f. Loop1 is DO 300 loop, loop2 the DO 400 loop and loop3 is the DO 450 loop.

   (b) FPPPP : One loop has been chosen from the file fmtgen.f. The loop begins as DO 180.

   (c) Spice : Seven loops have been chosen from this benchmark program. Loop1 is from the file v01.f and is the DO 360 loop; loop2 is also from v01.f and is the DO 205 loop. Loop3 and 4 are from v06.f and they begin as DO 210, and DO 763 respectively. Loops 5, 8, and 10, are from v07.f and are the DO 30, DO 40 and, DO 50 loops respectively. None of these loops have been unrolled.

3. Livermore Loops : Three loops, loops five, eleven and, twenty three have been taken from the livermore loops. Loop 5 (liv5_61) has been unrolled three times, but rolled versions of the other two loops (liv11 and liv23) are used.

4. Rau : This loop was used as a working example in a paper by Rau [RLTS92]. A rolled version (rau_1) and, a four times unrolled version version (rau_4) of the loop is used.

5. Whetstone : All the loops are taken from the whetstone.f file. Loop1 begins as DO 18, while loop2 begins as DO 88 in the file.

6. Finally, the graphs cycle4_1 and cycle4_2 have been cooked up by Ning.

Coloring Ning's schedules :

The benchmarks in Table 5.5 are colored using the fatcover, greedy and the optimal coloring algorithms. Below we make a few observations about the results we obtain.

- We notice that the fatcover algorithm performs as well as the optimal colorer on all the loops except on the fifth livermore loop (liv5_61 in the table) and loop2 of the whetstone benchmark. It deviates from the optimal number of registers in only two cases, and on the average requires 10% more registers.
- The heuristics used in choosing a fatcover for a cyclic interval performs rather well, as we are able to color the graphs optimally in most cases even when there are more than one cyclic intervals in the graph. Recall that the heuristic used to choose a fatcover for a cyclic interval favor intervals with earlier start times and those which are live the longest.

	Fatcover Colorer	Optimal Colorer	Our Greedy Colorer	Max Width	# Of Cyc Ints
Linpack					
ddot1	7	7	7	7	7
ddot4	7	7	10	7	7
Doduc(SPEC89)					
loop1	7	7	7	7	3
loop2	4	4	4	4	2
loop3	4	4	4	4	2
FPPPP(SPEC89)					
loop1	3	3	3	2	2
Spice(SPEC89)					
loop1	3	3	4	3	3
loop2	15	15	15	15	15
loop3	2	2	3	2	2
loop4	9	9	9	9	9
loop5	2	2	3	2	2
loop8	8	8	8	8	8
loop10	3	3	3	3	3
Livermore					
liv11	5	5	5	5	4
liv23	12	12	12	12	5
liv5_61	4	3	4	3	3
Rau					
rau_1	19	19	19	19	19
rau_4	19	19	19	19	19
Whetstone					
loop1	6	6	6	5	5
loop2	6	5	6	5	5
Ning					
cycle1_1	1	1	1	1	1
cycle1_2	2	2	3	2	2

Table 5.5: Number Of Registers Required To Color Time Optimal Schedules

In Section 3.2.4 we were concerned about the suboptimal performance of the fatcover algorithm in the presence of two or more cyclic intervals. The fatcover algorithm may choose intervals for one fatcover which may block the algorithm from discovering a fatcover for another cyclic interval of the graph. The collection of real world loops that we have tested all have several cyclic intervals and, the fatcover colorer performs very well on them. There are few and, a lot of times just one, plausible fatcovers for the cyclic intervals of the graph.

- The greedy colorer is far less powerful than the fatcover algorithm. The greedy colorer requires more registers than the optimal colorer for 27% of all the benchmarks. In contrast, the fatcover performs worse than the optimal colorer in only 9% of the loops. This experiment certainly favors the fatcover algorithm to the greedy.
- The loops which we have tested tend to require a small number, typically 2 to 20 registers to be colored.

Coloring several schedules :

In this experiment some of the applications used above were taken and several (at most 100) schedules were created using an exhaustive search mechanism [ARC93]. All the schedules thus generated were colored using the fatcover, optimal, greedy2 and, our greedy algorithms.

Table 5.6 details the benchmarks studied and the number of schedules that were colored per benchmark. Abbreviated names of the benchmarks are written within parenthesis and are used in the plots of Figs. 5.1 and 5.2.

We present the average performance of each colorer over all the schedules of an application in Figs. 5.1 and 5.2 and note the following :

- Of the eleven applications, the fatcover performed worse than the optimal colorer in 18% of all the benchmarks whereas the our heuristic greedy colorer performed worse than the optimal colorer in 55% of the cases. Our greedy algorithm appears to be far weaker than the fatcover algorithm.
- On the other hand, it is interesting to note that the greedy2 technique is much more aggressive and performs optimally in *all* the cases! From Figs. 5.1 and 5.2 we see that our greedy algorithm performs worse than the greedy2 algorithm on ddot1 of Linpack, as well as loops 2, 3 and 5 of the Spice benchmark. These benchmarks have 7, 2, 2 and 15 cyclic intervals in the interval graphs. Shifting the cyclic interval graphs, as is done by the greedy2 algorithm, to reduce the number of cyclic intervals may contribute to its better performance. Having fewer cyclic intervals causes the algorithm to use fewer registers to color the graph.
- Now, we focus on the benchmarks where the fatcover and our greedy algorithms have performed worse than the optimal colorer. Table 5.8 shows that in the worst

	Number of Schedules Colored
Linpack	
ddot1	2
ddot4	2
Doduc(SPEC89)	
loop3(doduc3)	16
FPPPP	
loop1(fppp)	6
Rau	
rau1	2
Spice	
loop2(spice2)	4
loop3(spice3)	2
loop5(spice5)	3
loop8(spice8)	2
Whetstone	
loop2 (whtsto2)	12
Ning	
cycle4_1(cyc41)	2
Total	53

Table 5.6: Number Of Schedules Colored For Each Application

case both the fatcover and the greedy algorithms require 50% more registers on the average to color the schedules of loop5 and loop3 of the spice benchmark respectively. This difference is considerable. However, register allocators do not perform optimally in all cases, so these numbers ought not to dishearten us. Instead it motivates us to investigate the performance of our allocators in comparison to the ones used by commercial C compiler. This study is presented in section 5.2.5.

Coloring Tomcat and the Livermore Loop 8 With Fatcover Colorer :

In the final part of this section, we return to the benchmark programs of Section 5.1[4]. We had presented statistics on spilling the hand generated schedule for Tomcat and the Livermore Loop 8. The sweep and split algorithm had been compared against the spill code generated by the MIPS, SPARC and the RS6000 native C compilers. Now, we shall color these loops using the fatcover colorer.

Using the *fat-cover* algorithm all but the unrolled version of tomcat were successfully colored without using chameleon intervals as can be seen in Table 5.7. For the unrolled tomcat loop, the interval graph had a minimal coloring of 34, and our method introduced 4 chameleon intervals to make it 32 colorable. In all the state of-the-art C compilers that we have studied (GCC, SPARC, MIPS, and the RS6000 C compilers) costly spills to memory would have been used to make the graph 32 colorable. However, in our case we needed only 4 register moves because the interval graph again provided a natural representation which allowed us to avoid these spills.

	Cyclic Intervals	Min Width	Max Width	Colors	Chameleon
Rolled **Tomcat**, 16 regs	0	0	16	16	0
Rolled **Tomcat**, 32 regs	0	5	24	24	0
Unrolled **Tomcat**, 16 regs	5	1	16	16	0
Unrolled **Tomcat**, 32 regs	12	17	32	34	4
Rolled **Loop 8**, 16 regs	0	10	16	16	0
Rolled **Loop 8**, 32 regs	0	11	17	17	0
Unrolled **Loop 8**, 16 regs	0	8	16	16	0
Unrolled **Loop 8**, 32 regs	0	22	32	32	0

Table 5.7: Interval Graph Statistics (after Spilling).

[4]Most of this section has been excerpted from [HGAM92]

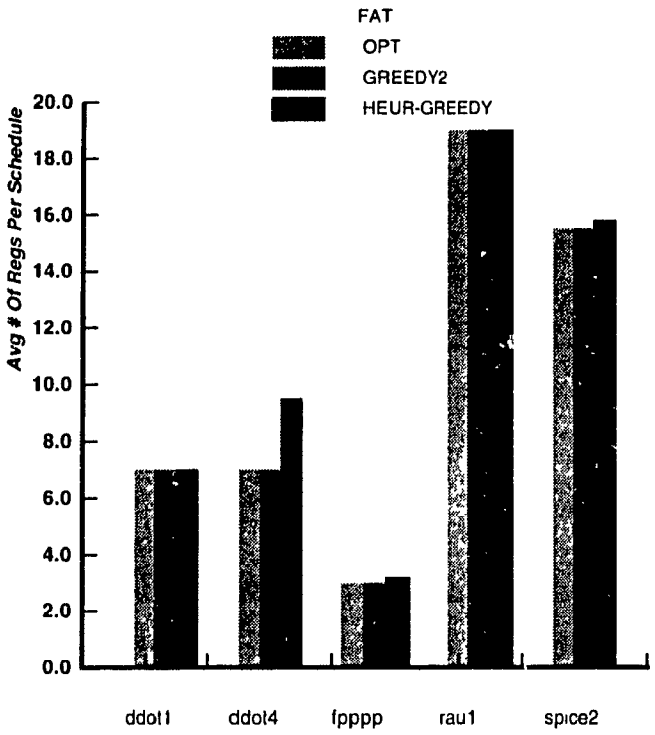


Figure 5.1: Performance Of Colorers On Several Schedules - I

5.2.5 Results : Comparison With Commercial Compilers

In the previous section we had noticed that in the worst case the fatcover algorithm was performing substantially worse (requiring 50% more registers) than the optimal coloring algorithm. In order to put these results in context of the world of commercially available compilers, we tested the benchmarks with the native C compiler on the SPARC10 and the XLC compiler on the RS6000 workstations.

The fortran loops were isolated from the benchmarks and transformed by hand to closely resemble three address code as was described in Section 5.2.2. F2c was then used to convert the fortran code into C code[5]. This code was compiled by the C compilers. The highest level of optimization were used on both machines (-O3 for SPARC10 and -O for RS6000). Assembly code was generated for each of the benchmarks which were then scanned to check

[5]F2c is a source to source translator. It converts fortran programs into semantically equivalent C programs

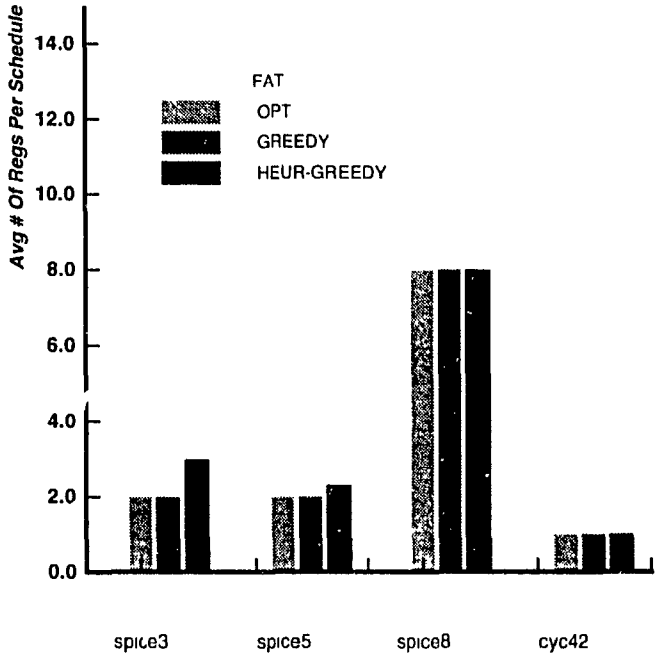


Figure 5.2: Performance Of Colorers On Several Schedules - II

for the number of registers used per iteration of the loop. The number of registers used by the commercial compilers to color the nodes of their code were then compared with the number of registers used by the fatcover and the greedy algorithms to color Ning's schedule (Table 5.5). We should point out that the schedule generated and colored by the commercial compilers may very likely differ from Ning's schedule which is colored by the fatcover, greedy and the optimal coloring algorithms. However, we still make a comparison between the performance of the commercial compilers and, and Ning's scheduling and our coloring algorithms to gauge if existing commercial compilers could potentially benefit by exploiting our methods on innermost loops.

We considered the number of floating point registers used by the loop body only. Integer operations like the ones used to set up the loop body were ignored. When using the native CC compiler on the SPARC, we noticed that the loops are automatically unrolled. To make the comparison fair, we report the number of registers used per iteration of the loop. Recall that the schedule colored by the optimal, fatcover as well as the greedy colorers have been software pipelined.

	Fatcover	Our Greedy
Linpack		
ddot2	0%	36%
FPPPP		
loop1(fppp)	6.7%	6.7%
Spice		
loop2(spice2)	0%	1.9%
loop3(spice3)	0%	50%
loop5(spice5)	50%	15%

Table 5.8: Average Percentage More Registers Required Than Optimal

From Fig. 5.3 summarizes the comparative results of the different commercial compilers with ours. From this figure we notice the following :

1. The SPARC-CC compiler requires more registers than the optimal number in four out of the five benchmarks. More registers were required in Linpack's doduc loop (ddot in Fig. 5.3), the livermore loop 5 (liv5), and the first and the fifth loops from spice (spice1 and spice5). As the loop bodies have been automatically unrolled by the compiler the size of the loop body increases, and this leads to an increased register pressure as well. This effect is aggravated in Livermore Loop5 (liv5) where it is most apparent. The code generated by SPARC CC requires 1.5 times as many registers as the optimal

2. On the other hand, code generated by the XLC compiler tends to require fewer registers than the optimal number. While studying the assembly code generated by XLC we realized that loop carried dependences are not being recognized so it is not taken advantage of. Instead of having cyclic intervals, shorter non cyclic intervals exist as live ranges are not kept live from one iteration to a successive one. This reduces the constraints and the register pressure in the graph. Hence the code generated requires fewer registers. Upon further investigation, we realized that in the process of converting the fortan benchmarks into C with f2c several temporaries variables are created. These variables contribute to the degraded output of the XLC compiler [Alt93]. If the fortran benchmarks were hand coded into C code then the comparison may have challenged the commercial compilers better. As f2c skews the results obtained for the XLC compiler, we can not make any conclusive remarks about it's performance.

3. In comparison to the number of registers required by the above two compilers, the

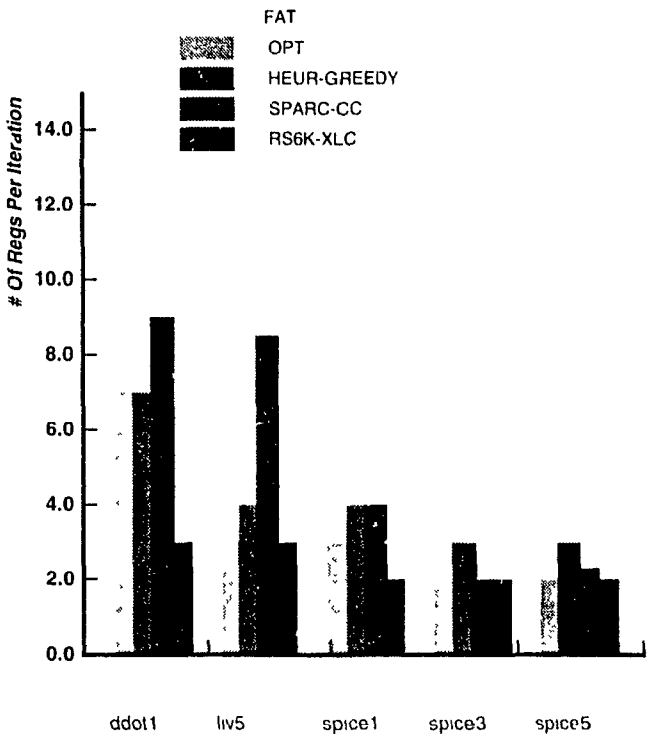


Figure 5.3: Comparing With XLC and CC Compilers

fatcover algorithm deviates the least from the optimal colorer. This algorithm is powerful for loop structures.

4. Our greedy algorithm still appears to perform the worst in comparison to the commercial compilers.

5. From the above observations, we conjecture that commercial compilers would most likely benefit by at least incorporating Ning's scheduling techniques for inner loop bodies. In addition, from Section 5.1, we notice that our two step allocation method can be useful in reducing spill code and, this could be of interest to commercial compiler writers as well.

5.3 Summary Of Main Points

Before concluding this chapter we highlight some of the main points of our experiments.

1. The sweep and split spilling algorithm works well on loops which have high register pressure like tomcat. It generates a small number of spill store and load operations in comparison to that generated by commercial compilers. Spill store instructions are lessened by approximately 90% while spill load instructions are lessened by 37% to 56% for the two test programs.

2. The fatcover coloring algorithm is able to color loops with loop carried dependences very well. It generates optimal to near-optimal results in most benchmarks which have been tested.

   Amongst the 53 schedules of the seven benchmarks tested (Tables 5.6 and 5.8) it required 50% more registers than the optimal in only one case (loop3 of spice). On the average the fatcover deviated by 5.2% from the optimal number of registers required.

3. Unfortunately, the greedy coloring algorithm performs worse than the fatcover algorithm.

   In the worst case, the deviance is 50% amongst all the 53 schedules of Table 5.6. This is no worse than the worst performance of the fatcover algorithm. However, the main difference lies in the average deviance from the optimal number of registers. The average deviance is 10% and this is about twice that observed by the fatcover algorithm.

4. Production compilers are sometimes unable to recognize loop carried dependences in some loops which have been converted by f2c. This greatly hinders the performance of the compiler and they are unable to exploit the fact that instruction scheduling coupled with loop unrolling can lead to values being retained over a period of time. This avoids the need to constantly load and store variables in between iterations.

   Although the version of XLC used by us generates code that requires fewer registers, it doesn't appear to perform dependence analysis on the f2c converted benchmarks that we have experimented with. In contrast, the SPARC CC performs automatic loop unrolling and this increases the size of the loop body and hence the register pressure. Our approach of using software pipelined and unrolled loops works well with the fatcover algorithm.

Chapter 6

Extensions And Enhancements To Our Allocation Scheme

In this chapter, we shall address two important issues.

- In Sections 6.1 and 6.2 we describe extensions of our register allocation scheme to handle a larger class of loops, those which include embedded structures, and a restricted set of other program constructs into consideration.
- Since our allocation scheme is best applicable for numerical benchmarks, which typically contain a lot of vectors within loop structures, we address the issue of handling these subscripted variables. This is discussed in Section 6.3.

6.1 A Hierarchical Coloring Approach For Other Constructs

So far in this thesis, innermost loops which have no embedded constructs have been considered for register allocation under our cyclic interval graph scheme. Numerically intensive real world benchmarks tend to have a very high number of perfect innermost loops [Huf93]. Huff studied DO loops in the Lawrence Livermore, SPEC89 Fortran and the Perfect Club benchmarks. Of the 1525 loops that he studied, 23% of them had loop carried dependences and no conditionals, while 70% of them had neither loop carried dependences nor any conditionals. Both these kinds of loops are classified as perfect loops by us and have been the focus of our study. However, always assuming the presence of perfect loops may limit

the practical applicability of our approach. Hence, it is imperative to develop a model that adequately handles a larger class of loops and eventually other program constructs.

The loops that we would like to concentrate on for the moment are those that have embedded loops, and multi-way branches like conditionals.

We employ a bottom-up approach when tackling complex (or hierarchical) program constructs. The basic assumption is that we are dealing with structured programs. This assumption, of course, disallows the presence of unstructured control flow, like goto statements. Unstructured programs may be transformed into structured code [EH93]. From now on, the term "program" is synonymously used with "structured programs".

Complex constructs are handled in the following manner :

- First, the structure is broken into smaller hierarchical blocks. This process is described in Sections 6.1.1 and 6.1.2.
- The hierarchical blocks are then colored in an inside-out fashion as described in Section 6.1.3.

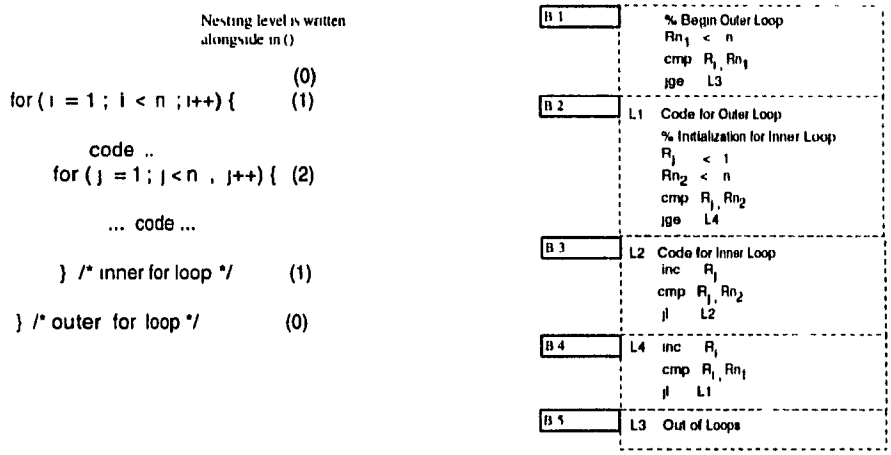
The hierarchical graphs could be represented as cyclic interval graphs or as circular arc graphs as outlined by Fig. 1.14 of Section 1.6. In this chapter, we demonstrate how to represent hierarchical graphs as cyclic interval graphs.

6.1.1 Creating Hierarchical Interval Graphs

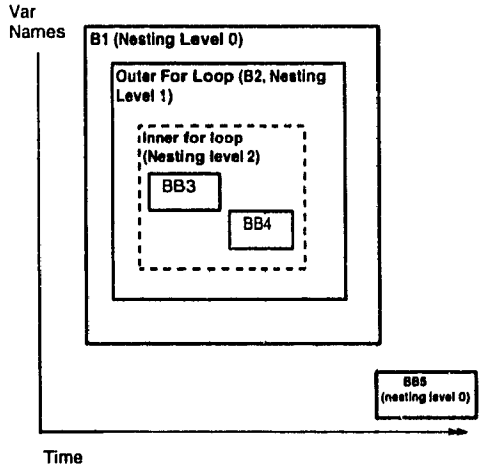
The first thing to do is to break complex program constructs, like multi-way branches, and loops which nest conditionals or other loops, into simpler more manageable blocks. These blocks are then tackled one at a time. Blocks may contain embedded blocks which are simpler in structure.

Through examples we show how such a hierarchy of interval graphs are formed. Appendix B outlines the process by which the intervals graphs are created from code segments. Here we shall merely show the final hierarchical interval graphs of some code segments. For the sake of simplicity, let us assume that each statement is executed in unit time cycle. We consider the following cases :

- Embedded Loops :
  Fig. 6.1(b) provides the assembly code for the piece of code with embedded loops of Fig. 6.1(a). Using the method given in Appendix B we construct the hierarchical interval graph shown in Fig. 6.1(c). The hierarchical graph representation is very

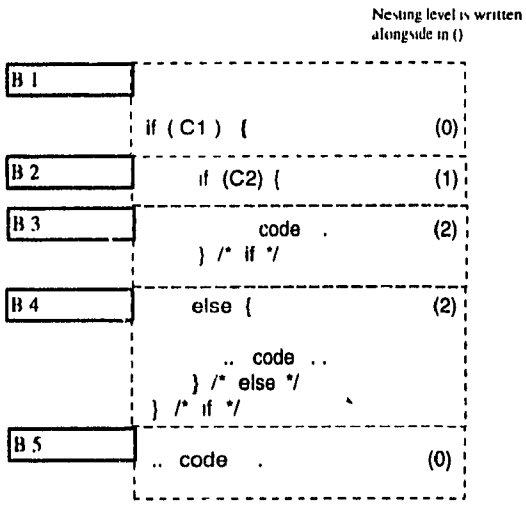


(a) Code Segment : Embedded Loops (b) Assembly Code

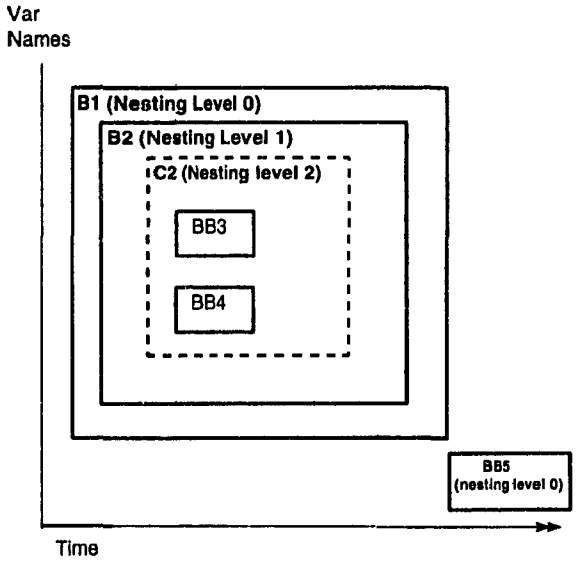


(c) Interval Graph For Nested Loops

Figure 6.1: Creating Hierarchical Graphs For Nested Loop Structures



(a) Code Segment : Multi-way Switch



(b) Interval Graph For Nested Conditionals

Figure 6.2: Creating Hierarchical Block Structures For Nested Branch Structures

intuitive and follows the same structure as the nesting level of the program. The two main blocks that we are concerned about are *B3* - the inner *j* loop block and, *B2* which is the outer *i* loop. Just as the *j* loop is embedded in the *i* loop in the code shown in Fig. 6.1(a), the basic block *B3* is enclosed by the block *B1*.

- Nested Branch Structures :
  Now, we consider the case of nested conditionals. Fig. 6.2(b) shows the interval graph of the code in Fig. 6.2(a). This time we do not bother with the assembly code as it introduces more blocks than the ones which interest us. For the sake of clarity, we omit these superfluous blocks. In this case, conditional *C2* is embedded in the conditional *C1*. *C2* is an if-then-else construct, and the if block *B2* is at the same nesting level as the else block *B4*. This nesting level is reflected in the interval graph as well.

6.1.2 Creating Hierarchical *Cyclic* Interval Graphs

Now, that we have seen how to create simple hierarchical structures, we will add a slight twist. When possible, we are interested in creating *cyclic* hierarchical interval graphs. Cyclic graphs take certain constraints into account. If a variable is shared by two sibling blocks and it is live at the point of entry and/or exit of these sibling blocks then the blocks can be manipulated so that the shared variable can be treated as a cyclic interval spanning the two blocks. This section describes exactly how and under what situations blocks can be manipulated to become cyclic interval graphs. Lastly, as described in Section 6.1.3, register allocation is performed from the innermost to the outermost block.

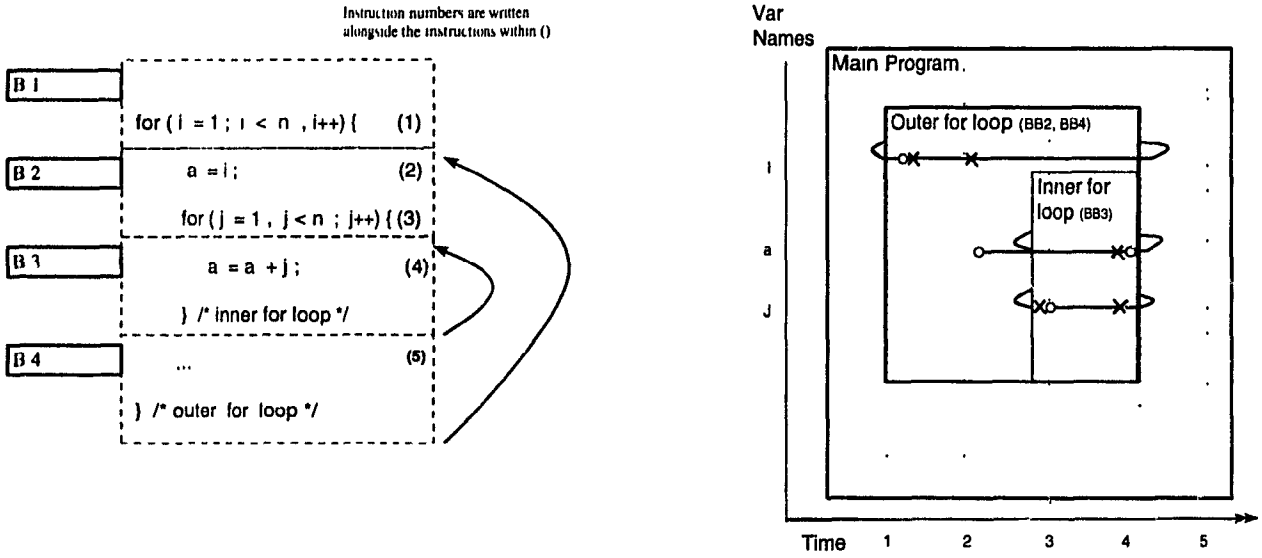
We illustrate this idea by building on the examples of nested loops and switch statements shown in Figs. 6.1 and 6.2 respectively. We fill in the bodies of the basic blocks to create situations which can lead to cyclic interval graphs in the hierarchical structure. In the previous section, we had converted the examples to assembly code to obtain the basic blocks. From here on, we may not show the assembly code for all examples and, may tend to ignore some of the standard blocks which check for the loop termination conditions.

Nested Loops :

Fig. 6.3 shows a *loop* structure where the variable a is first born in instruction 2 in the outer for loop. It is then used and redefined repeatedly in instruction 4 within the inner for loop. As this variables' *life extends between iterations*, we capture this information by wrapping the interval for a around the inner *for j* block. Similarly, the life of the *loop counter variables*, *i* and *j*, are live between successive iterations of the loop and are converted to cyclic intervals.

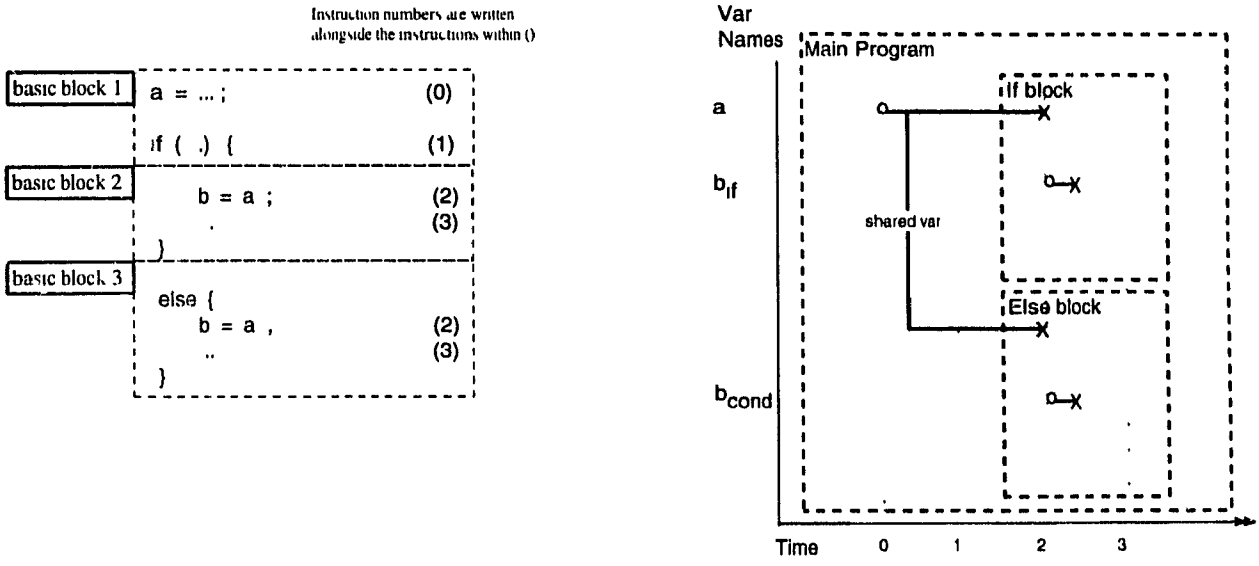
Switches :

Next, we look at a simple *branch* statement. In branch statements, constraints can be created in three ways.



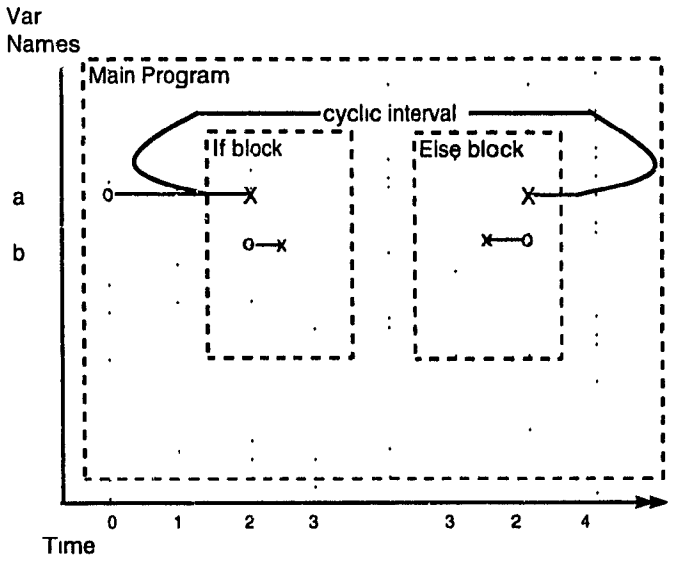
(a) Code Segment (b) Cyclic Interval Graph

Figure 6.3: Hierarchical Cyclic Interval Graph Of A Doubly Nested For Loop



(a) Code Segment

(b) Interval Graph



(c) Cyclic Interval Graph

Figure 6.4: Conditional - Case 1 : Constraint At Entry

Constraint from Entry Point :

From Fig. 6.4(a, b) we see variable a being born in instruction 0. Depending on the flow of control it is read in instruction 2, in either the if or the else part of the branch. As this variable is defined outside of the branch blocks and is shared by the if-block and the else-block from their *point of entry*, a has to be assigned to the same register in both the blocks. In order to maintain consistency and correct program semantics, the register containing a in instruction step 1 should be made visible to both the paths of execution (Fig. 6.4(b)). This constraint can be very naturally captured by rotating the else block by a $-180°$ so that the interval of a becomes a cyclic interval spanning the combined block formed by the if and the else blocks (Fig. 6.4(c)). Notice that the intervals in the rotated blocks have undergone reflection. The cyclic interval forces a to be in the same register in both blocks.

Constraint from Exit Point :

In Fig. 6.5, we see that the sibling blocks of the conditional statement share a variable at it's *exit point*. That is, a variable, like b, may be born within either the if or the else-block and then used outside the branch. b must be assigned to the same register in both paths of execution to maintain consistency in the program. This constraint can again be captured by a rotation of the else-block and then by joining the interval between the two sibling blocks (Fig. 6.5(c)). Notice that the interval formed by the variable b extends between the if-block and the else-block. Such an interval which extends between two blocks is named the *fusing interval*.

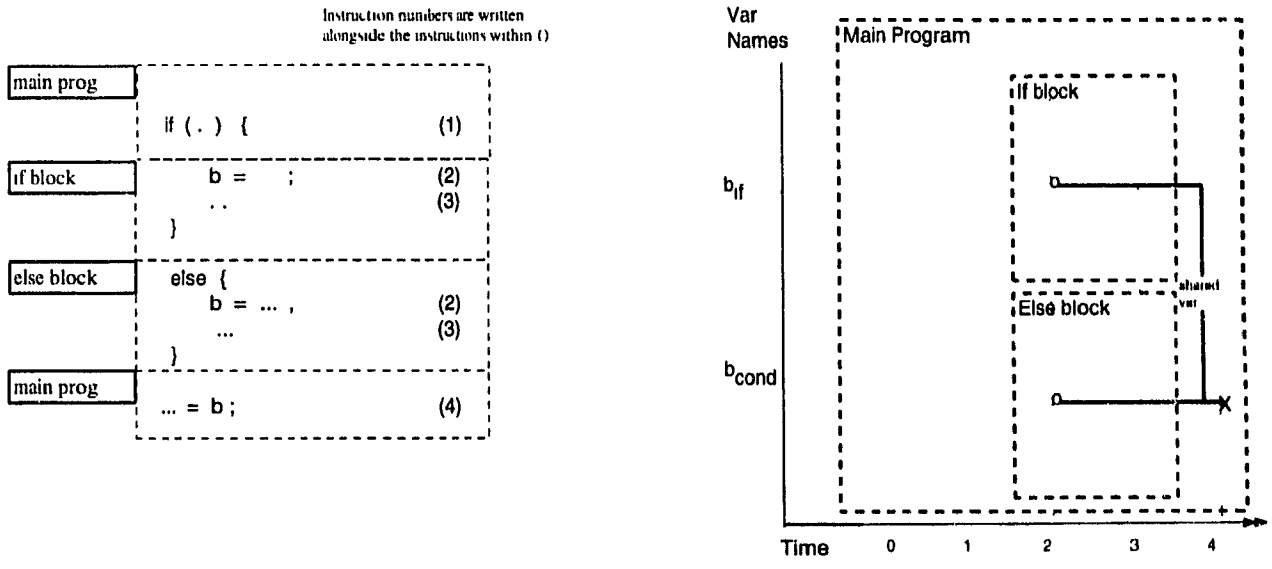
In the next step when register allocation is performed, one register is allocated to the interval for b, thus taking this constraint into account.

Constraint from both Entry and Exit Points :

Finally, in Fig. 6.6(a, b), we see a combination of the above two situations. a is born outside the conditional and is used in one of either sides of the branch, whereas b is born, in instruction 2, in both sides of the branch, and is used outside the conditional. In this situation, consistency has to be maintained between *both the entry and exit points* of the two sibling blocks and the parent block. Once again, these constraints are retained and taken care of by rotation of the else block as shown (Fig. 6.6(c)). The constraint at the entry point is preserved through the cyclic interval that wraps around, while the constraint at the exit point is preserved through the fusing interval that extends to both blocks.

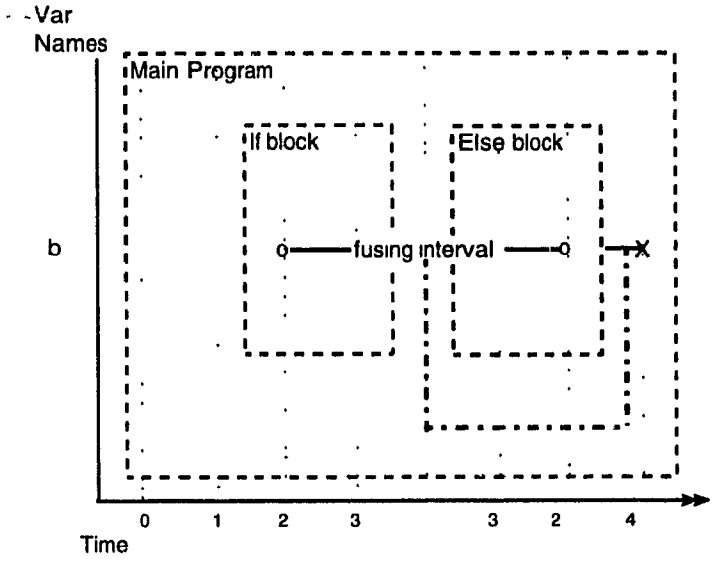
In essence, constraints are taken care of by rotating and merging sibling blocks to form one single unit. We shall call such units *fused blocks*.

Notice that in all of our examples we consider only intervals created by *scalar* variables as candidates for cyclicity. In Section 6.3, we shall deal with subscripted variables at some length. We observe that only those scalar variables



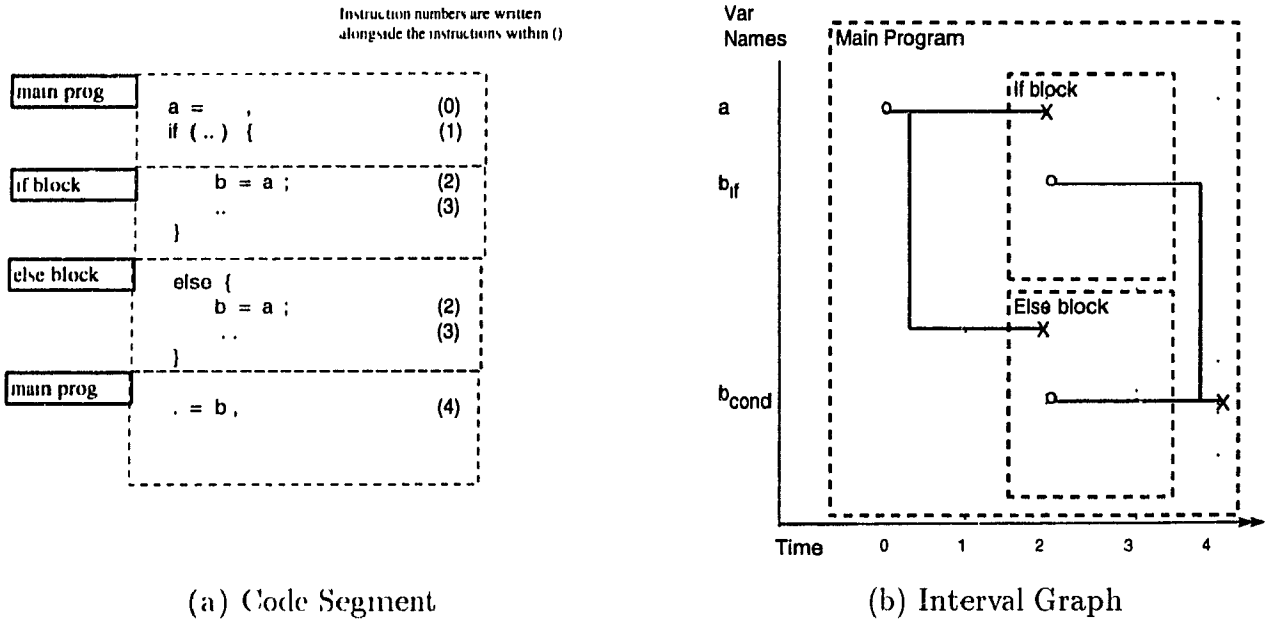
(a) Code Segment

(b) Interval Graph



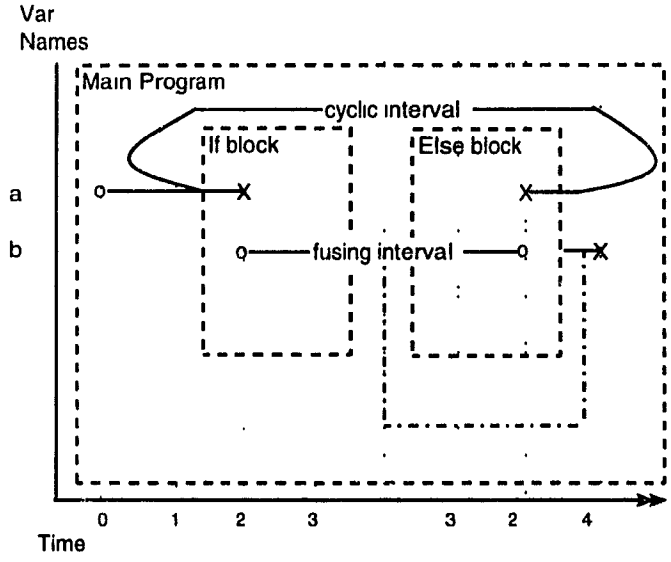
(c) Cyclic Interval Graph

Figure 6.5: Conditional - Case 2 : Constraint At Exit



(a) Code Segment

(b) Interval Graph



(c) Cyclic Interval Graph

Figure 6.6: Conditional - Case 3 : Constraint At Entry And Exit

- whose life extends between iterations of loops

can be candidates for cyclic intervals and can thus give rise to cyclic interval graphs.

6.1.3 Spilling And Coloring Hierarchical Cyclic Interval Graphs (HCIG)

Once the hierarchical cyclic interval graphs have been created,

- the spilling phase is invoked.

  The spiller works on each block as described before in Chapter 4. However, the question that arises is

  - how do we calculate the width (or the thickness) of hierarchical graphs.

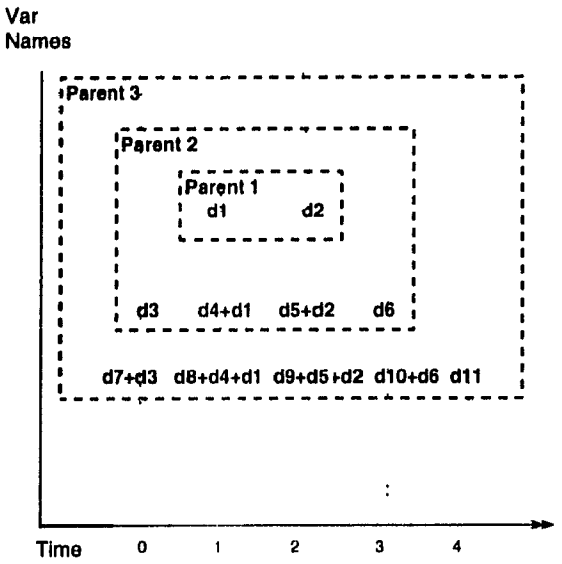
  The answer to this question can be obtained by considering two cases. The width is determined by working from the innermost block outwards.

  - Fig. 6.7(a): First, we shall deal with nested structures that do not embed sibling graphs. In these cases, we pay special attention to the times at which one block overlaps with an enclosing block. It is at these times that the width of the outer parent block is determined by adding the width of the parent block with that of the inner child block.

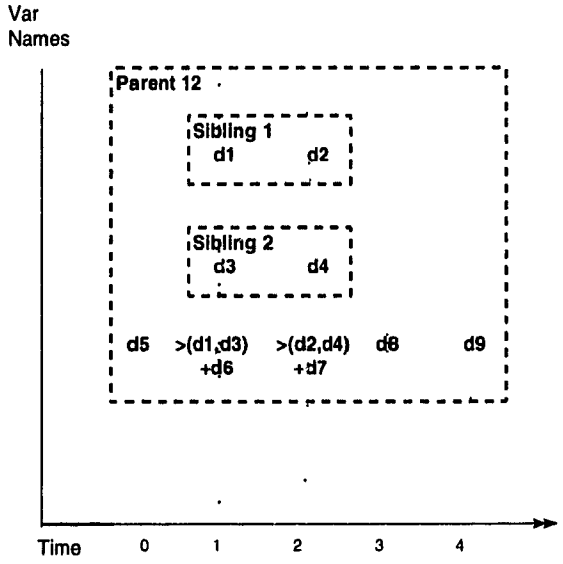
    In our three level deep nested hierarchical graph structure (Fig. 6.7(a)), we find that the width of the innermost (that is *parent1*) block, is *d1* and *d2* at times 1 and 2. We move onto the outer *parent2* block. At time 0 the width of the interval graph is *d3*. If, for just a moment we ignore the presence of the block *parent1* then the width of *parent2* at time 1 is *d4*. Since we must take the inner block into consideration the width at time 1 changes to $(d4 + d1)$. A similar situation is seen to occur again at time 2 and then at times $0 \ldots 3$ for the *parent3* block. We continue finding the width of the blocks incrementally while moving from the inner to the outer layer. Inner loops having a smaller width receive a high priority to be allocated and retained in registers while intervals in the outer blocks have a higher chance of getting spilled.

  - Fig. 6.7(b) : It is very likely that we will encounter parent blocks that nest children blocks. Note that the children blocks in this example overlap with each other, that is, either *sibling 1* or *sibling 2* is executed. This kind of an interval graph can arise from code shown in Fig. 6.2.

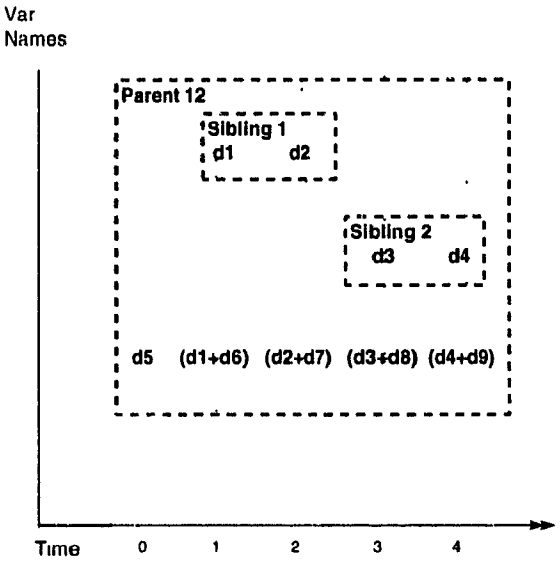
    What is the relationship between the width of the sibling blocks and the enclosing parent block? At each time of overlap between the parent and the children block,



(a) Width Of Nested Structures



(b) Width Of Nested,
Overlapping Sibling Structures



(c) Width Of Nested,
Non Overlapping Sibling Structures

Figure 6.7: Calculating Width Of Hierarchical Interval Graphs

the maximum width of all the children at that time is added to the width of the parent block to obtain the resultant width of the parent block.

In our example, we find the width of the *sibling1* and *sibling2* blocks first. Then we move onto the outer *parent12* block. At time 0 the width is $d5$ and if we ignore the presence of the embedded children blocks then the width at time 1 is $d6$. Once we take the children blocks into consideration we become aware of the overlap between the current block and its children block. For instance, at time 1 the parent block's width is $(d6 + maximum\{d3, d4\})$. We continue using this method for times 2...4.

- Fig. 6.7(c) : This figure is very much like Fig. 6.7(b). The only difference is that the nested sibling blocks, *sibling 1* and *sibling 2*, do not overlap, they are adjacent to each other. Both *sibling 1* and *sibling 2* are executed in this example. This interval graph structure can arise from the code shown below.

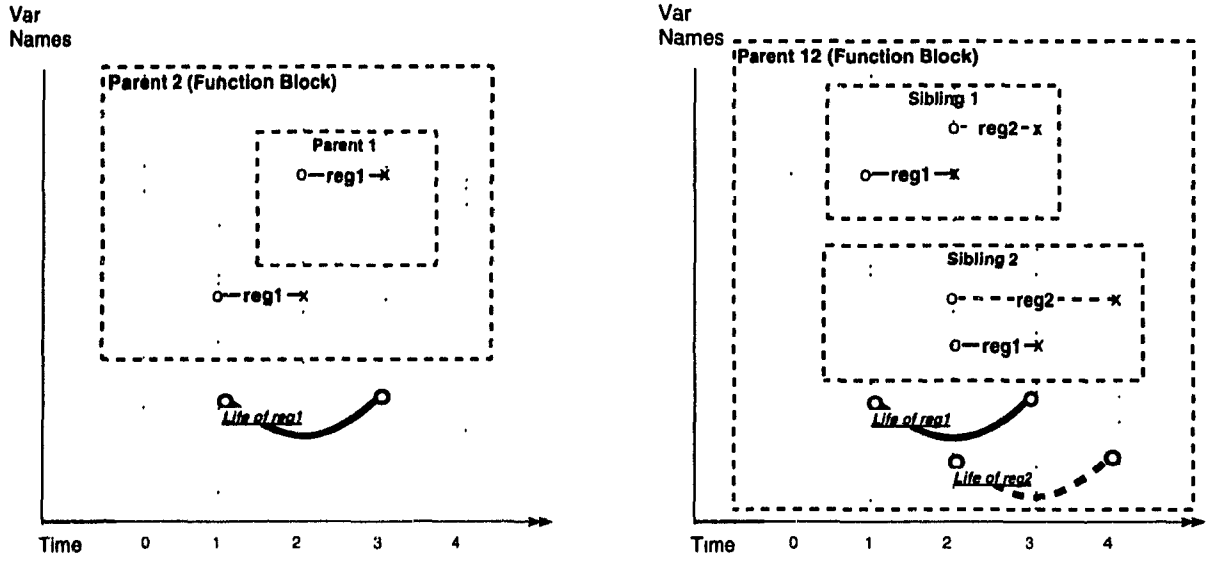
```
for (parent12) {
   if (sibling 1) {
   }
   for (sibling 2) {
   }
}
```

The width of *parent 12* is found by adding the widths of *sibling 1* with that of *parent 12* for times 0...1, and by adding the widths of *sibling 2* with that of of *parent 12* for times 3...4. Note that this calculation is an extension of the width calculation of *Parent 2* shown in Fig. 6.7(a).

- After completion of the spilling phase, register allocation is performed from the innermost block outward. The coloring algorithm (for instance the left edge algorithm of the fatcover colorer) has to be modified to take the hierarchical structure of the graphs into account. The modification made to the fatcover colorer is described in the next section. When using the fatcover algorithm, cyclic intervals of a block are assigned registers first.

- Next, registers are allocated to the local intervals of the inner blocks using one of the coloring algorithms mentioned in Chapter 3. A history is maintained for each register of the target architecture, and registers allocated to the children blocks are made unavailable for the duration in which they're used in the outer parent block. The duration of use of the registers is calculated in the manner described below.

  - Fig. 6.8(a) (*Nested Blocks*) :
    In this instance, we have a nested structure where register *reg1* is assigned to intervals in *parent1* and *parent2* blocks. When we are in the enclosing *parent2* block, register *reg1* is occupied from times 1...3. It's life is cumulative of it's life length in *parent1* and *parent2* blocks.



(a) Register Life In Embedded Blocks

(b) Register Life In Sibling Blocks

Figure 6.8: Life Of Registers Within A Function Block

– Fig. 6.8(b) (*Sibling Blocks*):
This figure illustrates the case where sibling blocks, *sibling1* and *sibling2*, are enclosed within a parent block, *parent12*. Sibling blocks receive a similar register history file and do not need to know about the other sibling blocks local register allocation information. Assume that two registers *reg1* and *reg2* are assigned to intervals in both the sibling blocks. Once we move out to the *parent12* block we notice that register *reg1* is live from times 1...3. The life (that is the duration in which it holds a value) of *reg1* from *sibling1* and *sibling2* are added to give the life of *reg1* in *parent12*. In contrast, the life of *reg2* is taken to be the greater of the lifetimes of the intervals it is allocated to in the sibling blocks. From the example, we see that *reg2* is live from times 2...4.

We do not show how the register lifetimes for an interval graph structure like the one in Fig. 6.7(c) is calculated as it is very similar to Fig. 6.8(a).

- Having knowledge of the lifetime of the registers, we allocate them to the outer parent block.

- When we have allocated registers to all the embedded children blocks within a parent block we refresh the register history file. That is, registers allocated to local intervals of the children block are freed.

  This process is recursively followed for the entire program on a function by function basis.

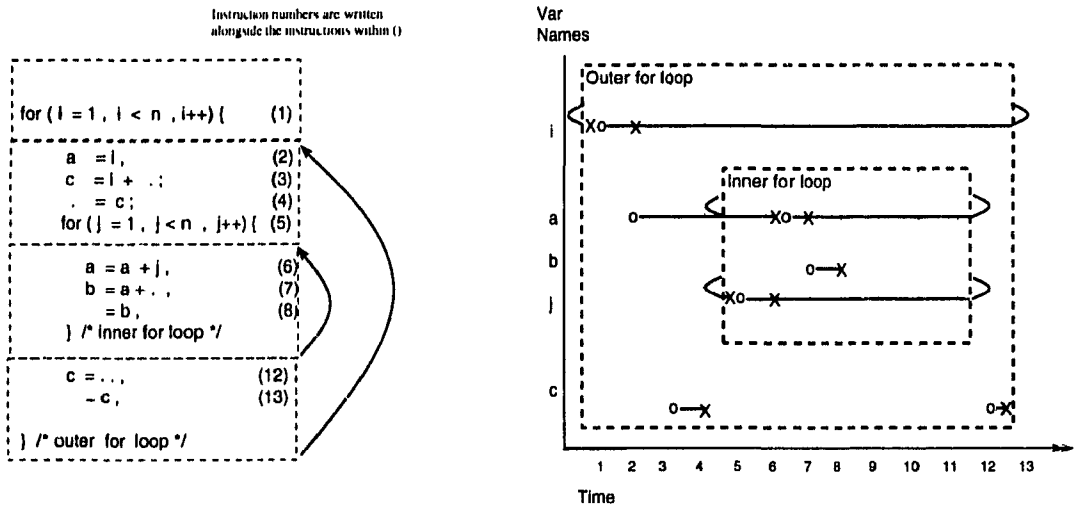
6.1.4 Examples : Coloring HCIG

Now, that we have a global picture of the coloring process, we will build on the previous examples to illustrate this idea. We shall consider

1. a simple nested for loop
2. a conditional statement and,
3. a for loop with an embedded conditional statement.

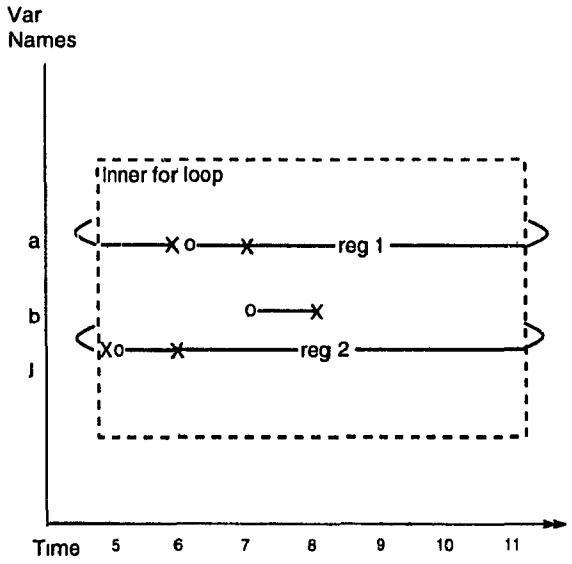
The previous examples of Section 6.1.2 is made more complex and interesting for this section.

Simple Loop :
To begin with, we consider the case shown in Fig. 6.9(a) which consists of an embedded loop. Like the example in Fig. 6.3, the inner "for loop" does not have any sibling

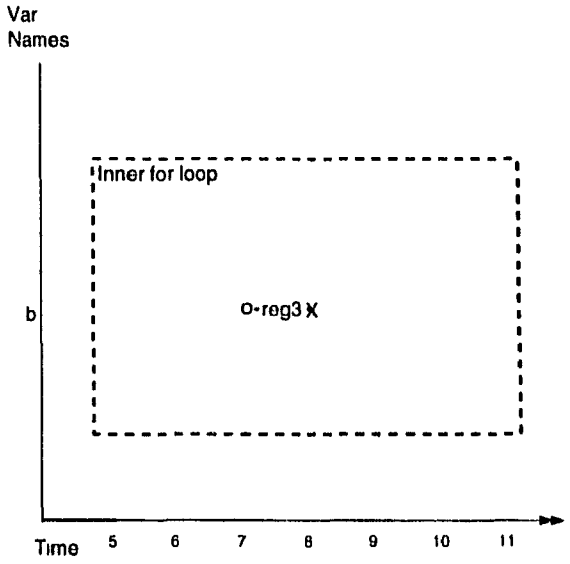


(a) Code Segment (b) Cyclic Interval Graph

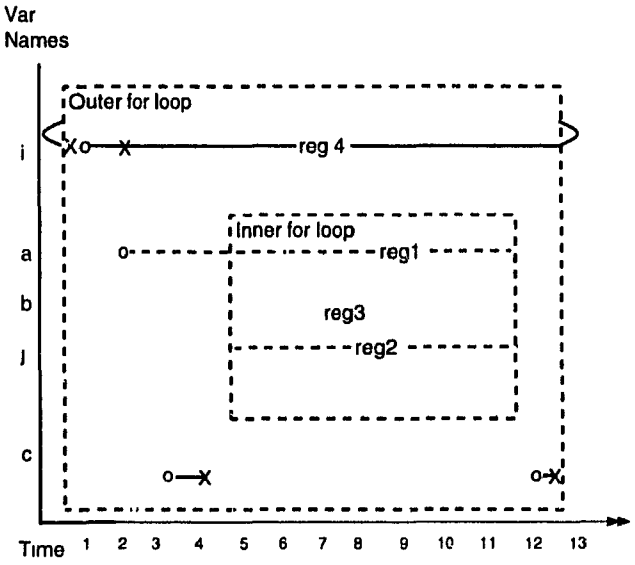
Figure 6.9: Example : Coloring A Hierarchical Loop Structure



(c) Cyclic Intervals Of Inner For Loop



(d) Non-Cyclic Intervals Of Inner For Loop



(e) Coloring The Outer For Loop

Figure 6.10: Example Continued

blocks. Inter-iteration dependences make i a cyclic interval for the outer for loop, while the variables a and j are cyclic intervals for the inner for loop (Fig. 6.9(b)).

We assume that our architecture has only four registers available. That is, the number of registers equal the maximum width of the interval graph, $W_{max}(G)$. Furthermore, we simplify our model by assuming that we have a single class of register only.

As far as the coloring algorithm goes, let's assume that we use the fatcover algorithm.

- Register allocation is first performed for the innermost block, which in this case is the inner for loop.
  - Fig. 6.10(c) : The **cyclic** intervals of the inner for loop are assigned registers. The fatcover of a consists of $\{a\}$ and it is assigned a new register, *R1*. Similarly, the fatcover of j is $\{j\}$ and it is assigned a new register, *R2*.
  - Fig. 6.10(d) : After the cyclic intervals have been taken care of, we are left with a reduced inner graph of **non-cyclic** intervals. A left to right sweep is performed and the non cyclic interval, b, is assigned a free register *R3*. This uses the left edge algorithm, but as we shall see below, the original left edge algorithm presented in Appendix A can not be directly used in the next step of this example. It needs to be extended and, we shall point out exactly where the modification is required in the next step.
- Once, we're done with the inner block, we move onto the outer parent block shown in Fig 6.10(e). The allocations made for the inner block are grafted into the parent block as colored intervals (depicted by the dotted lines of the inner block). Note that the grafted intervals are represented as non-cyclic intervals in the outer loop. Now, we shall color the uncolored intervals of the outer block.
  - According to the fatcover algorithm, we assign the fatcover of i (which is $\{i\}$) a register *R4*.
  - As there are no more cyclic intervals left, the graph is swept from left to right and colored using the extended left edge algorithm. We notice that three intervals, a, b and, j, are pre-colored. These intervals were assigned registers in the inner for loop block, they do not have to be recolored in this phase. The left edge algorithm ignores the colored intervals, however the register assignments made to a, b and, j are recorded and, the assigned registers are noted to be busy for the duration of the live ranges. After updating the status of the registers to reflect the colors of the precolored intervals, the colored intervals can be removed and the original left edge algorithm can be used.

    The uncolored intervals for the variable c is to be assigned a register. As we'd like to minimize the total number of registers used, we prefer to reuse a register that is free during the life-time of c rather than allocate a new register that has not been assigned previously. In our case the architecture has only four available registers, so we do not have the luxury of playing with a new register, so we must either reuse register *R2* or *R3*.

Conditional :

Our next example consists of a conditional statement shown in Fig. 6.11(a). This example is based on the one illustrated in Fig. 6.6.

- Fig. 6.11(b) : This diagram shows the hierarchical interval graph for the code segment shown in (a). The embedded conditional statement is constrained from both it's point of entry and exit due to interval *a*.
- Fig. 6.11(c) : Here the *else* block has been translated to merge with the *if* block and a cyclic interval and a fusing interval is created.

  This strategy of translating the blocks is merely a visual aid to illustrate the potential for a cyclic and fusing interval. In reality, we don't have to perform this rotation and translation stage and can work on the sibling blocks simultaneously as long as we are able to identify the cyclic and fusing intervals and mark them.

  Now, we attack the fused *if* and *else* block together. A fatcover has to be found for the cyclic interval *a*. The novelty of this cyclic interval is that half of it lies in one sibling (*if*) block while the other half lies in the other (*else*) sibling block. The *if* block is missing the tail end of the cyclic interval and the *else* block is missing the front end of the cyclic interval. Keeping this in mind, we find the fatcover of the cyclic interval by scanning the *if* block from left to right, and scanning the *else* block from right to left. The fatcover in the fused block is seen to consist of all the intervals (*b*, *c* and the fusing interval) of the blocks and is assigned a register *r1*.

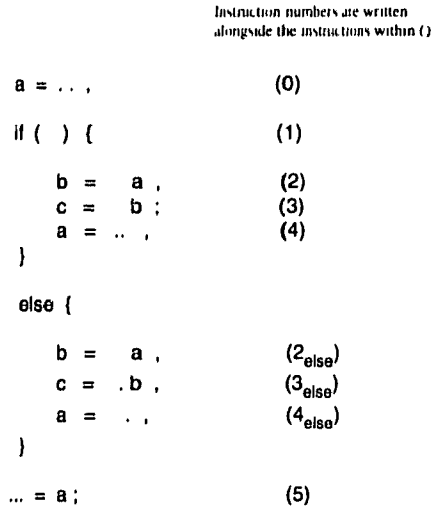
  If the fusing interval is a part of the fatcover of the cyclic interval in one sibling block, *sibling 1*, while it isn't in the other sibling block *sibling 2* then, it must be conflicting with one of the intervals of the fatcover of *sibling 2*. If this situation arises then the fusing interval can not be included in the fatcover.

Loop with Embedded Conditional :

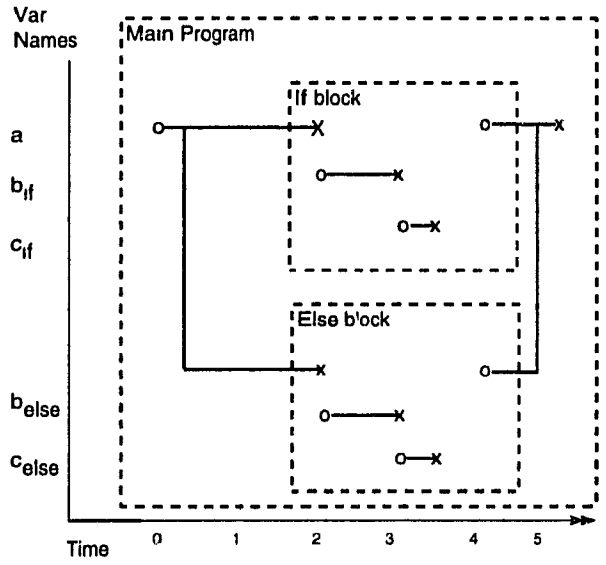
We move onto the last example of this section. Consider a conditional statement that is embedded within a *for loop* like the one shown in Fig. 6.12(a). Assume that the target architecture has two available registers, *r1* and, *r2*.

- Fig. 6.12(b) : The cyclic intervals have been identified and the hierarchical interval graph is ready for the coloring phase now.
- Fig. 6.12(c) : The *else block* undergoes rotation and translation in final preparation for the coloring phase. The embedded conditional statement is constrained from both it's point of entry and exit like in Fig. 6.6. Interval *a* is responsible for this constraint.

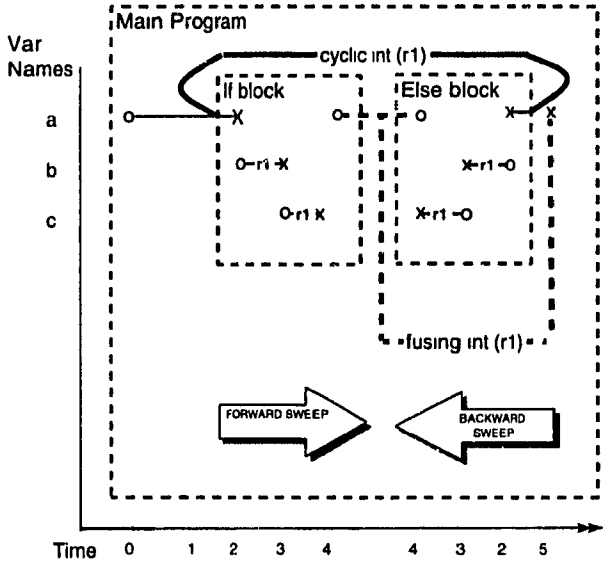
  There is a very subtle complication in this example. Both branches of the conditional statement read *a* which is defined outside the conditional blocks. This causes the constraint at the entry point. Later, *a* is defined in both sides of the conditional and is used in the outer parent block. This causes the constraint



(a) Code Segment



(b) Interval Graph For Conditional



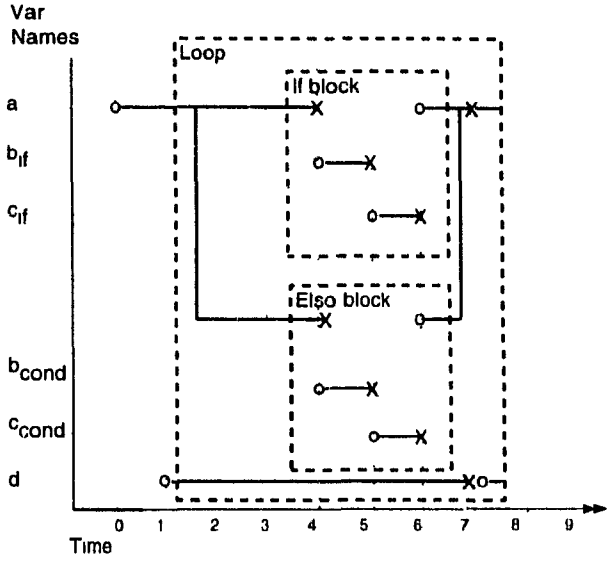
(c) Coloring The Graph

Figure 6.11: Coloring A Conditional Statement

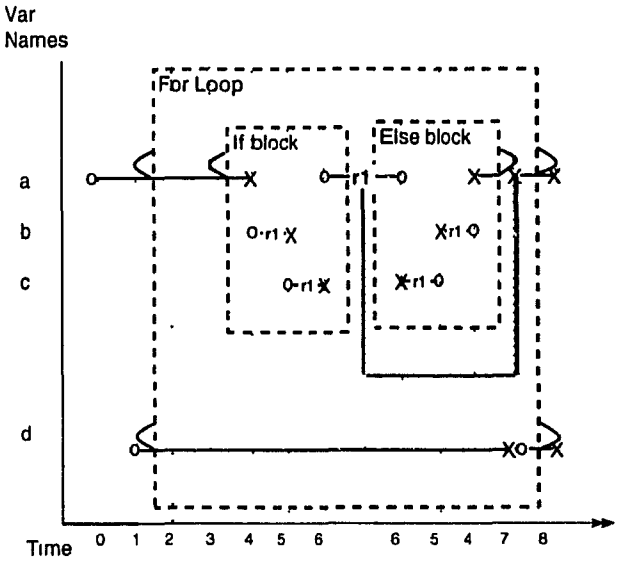
Instruction numbers are written alongside the instructions within ()

```
a =    ,                (0)
d =    ,                (1)

for(  ,  ,  ) {         (2)
  if (  ) {              (3)

    b =   a ,            (4)
    c =   b ,            (5)
    a =     ,            (6)

  } /* if */
  else {

    b =   a ,            (4else)
    c =   b ,            (5else)
    a =     ,            (6else)

  } /* else */
    d   =  a + d ,       (7)
} /* for */

  = a ,                  (8)
```

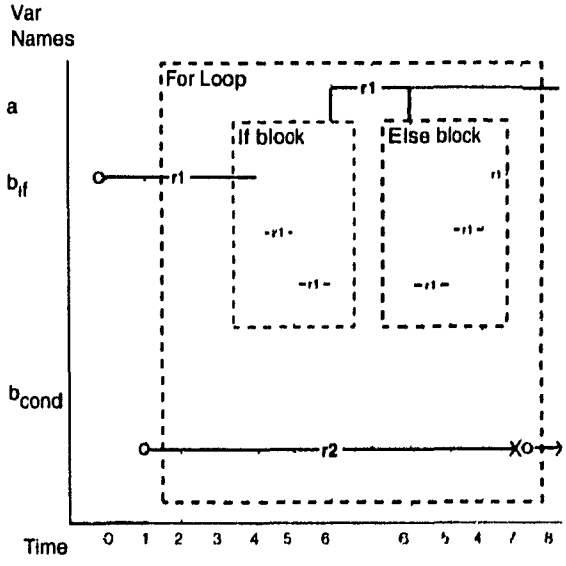
(a) Code Segment



(b) Interval Graph



(c) Colored Inner Blocks



(d) Colored Graph

Figure 6.12: Coloring A Conditional Embedded In A Loop

at the point of exit and gives rise to a fusing interval. Thus far, the situation is the same as that in Fig. 6.6.

If we work according to the inside-out method that we have followed so far, the cyclic interval and the fusing interval could be assigned two different registers like *r1* and *r2* respectively. This can happen if the fusing interval is not a part of the fatcover of the cyclic interval. Let us assume that we have used this allocation scheme. However, when we progress beyond the conditional blocks into the parent *for loop* block we are confronted with a conflict. The fusing interval transforms into a cyclic interval for the *for loop*. This cyclic interval carries the value of *a* across to the next iteration and into the *if* and *else* blocks.. Hence, it becomes imperative for the fusing interval to be assigned the same register as the cyclic interval for the conditional. If distinct registers are allocated to the fusing interval and the conditional cyclic interval, then the conflict could be resolved by performing move instructions at the end of the *if* and *else* blocks to transfer the value of the fusing interval into the register used by the cyclic interval. However our current inside-out approach fails to handle this conflict because we recognize this conflicting situation in the outer *for loop* and by then it is too late to go back into the inner block to change the allocation of the intervals. This clearly demonstrates the sub-optimality of our current approach. We shall explore this further in Section 6.2.

Fortunately in our example (Fig. 6.12(c)) we have avoided this conflict-resolution step as the fatcover of the cyclic interval is found to include all the non cyclic intervals (that is *b*, *c* and the fusing interval) of the *if* and the *else* blocks and it is assigned the register *r1*.

- Fig. 6.12(d) : After tackling the inner conditional, we move onto the intervals of the *outer loop block*. As the cyclic interval *d* is live all through the block, it forms it's own fatcover which is assigned a free register r2.

6.2 A Modified Approach For Coloring

In the previous section (Figs. 6.11 and 6.12), we may have encountered situations where our inside-out method would not have been able to color the interval graphs. In this section we shall illustrate a second problem that our method runs into with the help of a very simple example.

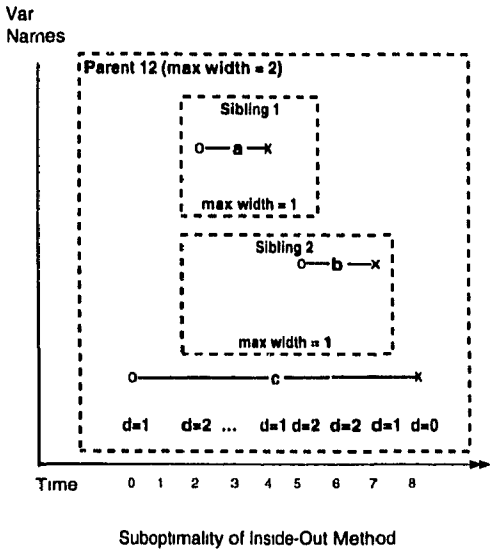


Figure 6.13: Suboptimality Of The Inside-Out Coloring Scheme

6.2.1 Sub-optimality Of The Inside-Out Approach : An Example

Assume that the target architecture has several register classes and there are only 2 available floating point registers, *r1* and *r2*. Now, take the case of two sibling blocks, *sibling1* and *sibling2*, embedded within a *parent12* block as shown in Fig. 6.13. The sibling blocks do not share any intervals and are thus not constrained at all. The sibling blocks and the parent block each have an interval. Such an interval graph may be a subgraph of a larger program graph and shows intervals that belong to a specific register class like the floating point registers.

We determine the width of the two sibling blocks, *sibling1* and *sibling2*. Then we find the width of the *parent12* block. The maximum width of the *parent12* block is 2 and it is clearly two-colorable. As the number of registers equal the maximum width of the graph the spilling phase does not change the interval graph.

Now, the graph is to be colored. According to our inside-out coloring scheme, we assign registers *r1* to interval *a* and *r2* to interval *b*. After this, we are ready to move to the *parent12* block. The interval *c* needs to be assigned a register. Interval *c* conflicts with intervals *a* and *b* and hence can not be assigned *r1* or *r2*. And unfortunately, at this point we do not have any new free registers! If *c* is to be colored, it has to be spilled. But in

our approach the spilling phase is invoked only once and that is before the coloring phase. Unlike Chaitin's approach we do not re-run the spiller. It is also not possible to introduce register move instructions to avoid spill code without changing the interval graph of the hierarchical blocks.

To handle such problems we need to modify our inside-out coloring approach. We propose a two pass, **inside-out-outside-in**, approach.

(1) Inside-Out Phase :

Temporary assignments of registers are made in the inside-out approach. If conflicts do not arise then the temporary assignment is made permanent and the outside-in phase is bypassed altogether.

(2) Outside-In Phase :

However, if conflicts do arise then the outside-in phase is invoked after the completion of the inside-out pass. In this phase we progressively move inwards changing some temporary register assignments in the appropriate blocks to remedy the conflict.

For instance, in our example above we assign *c* one of the registers like *r1* and then continue the outward coloring process. Once the outward coloring phase is complete, we initiate the outside-in phase. We move inwards to the *sibling1* block. Here we change the color of *a* from *r1* to *r2*. We must be able to change the color of the interval in one of the inner blocks (assuming that there are sibling blocks) as the spiller has determined that the outer parent graph is colorable with the number of registers available. If need arises, then register floats can be introduced in this pass.

However, this algorithm is probably not general enough to handle all cases optimally. We plan to state concrete algorithms for this modified approach in the future.

6.3 Handling Array Subscripts within loops

So far we have worked with only scalar variables only in the source code. However, as we focus mainly on scientific benchmarks which have a high proportion of loops with array accesses, we must be able to handle subscripted variables. When dealing with these cases we are faced with questions like :

- How do we create cyclic interval graphs for such loops?
- Do subscripted variables having inter-iteration dependences need special care?
- Would the inclusion of array dependence analysis information into our allocation scheme enhance performance of our allocation strategy? If so, how do we incorporate this information?

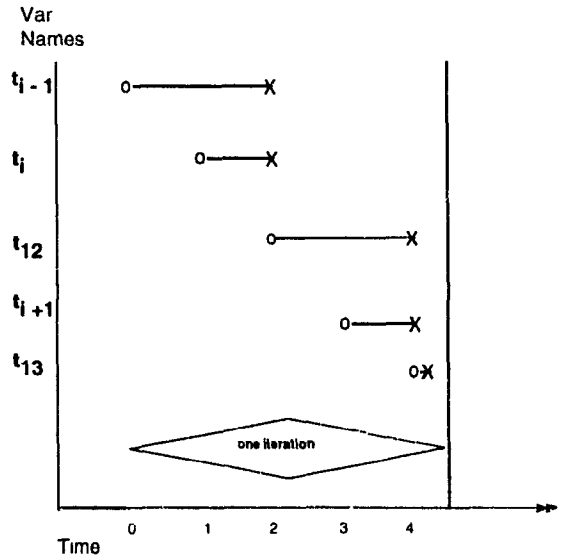
We shall answer these questions through the example of Fig. 6.14.

```
.
for (i = 1; i < n, i++) {
A[i] = A[i - 1] + A[i] + A[i + 1]
}
.
.
.
```

(a) Code Segment

```
                %Initialization

L1  t_i-1  <-- ld  a_i-1          (0)
    t_i    <-- ld  a_i           (1)
    t_12   <-- t_i-1  +  t_i     (2)
    t_i+1  <-- ld  a_i+1         (3)
    t_13   <-- t_12   +  t_i+1   (4)

    a_i    <-- st t_13

% Check for end of loop
bnez    L1
```

(b) Rolled Assembly Code With Temps



(c) Interval Graph For Rolled Loop

Figure 6.14: Handling Subscripted Variables - Naively

6.3.1 Introducing An Example

Fig. 6.14(a) shows a piece of code that we shall consider. It is a very simple loop that references three different elements of the vector a. The subscripted variables have a dependence distance of 1. Assuming that dependence analysis information is unavailable, a compiler may generate the three address code shown in Fig. 6.14(b). According to this code 3 load instructions are issued per iteration of the loop. This is rather inefficient. From Fig. 6.14(c) it is clearly seen that two registers are adequate for coloring the interval graph of this code segment. We set about bettering the code generated with our register allocation scheme.

6.3.2 Applying Our Method On The Example

In order to extract maximal efficiency from our register allocation scheme we need to find the period of the interval graph structure. A *periodic interval graph* captures all array dependences and this helps us to recognize all the cyclic intervals. This provides our allocation strategy with an enhanced graph and our method can then attempt to hold all the cyclic intervals in registers. Retaining as many cyclic intervals as possible in registers, of course, reduces spill code as we know that the cyclic intervals are going to be referred to at a later iteration of the loop. This certainly leads to better code generation.

The following describes a step by step method that is followed when handling subscripted variables. We use the code segment of Fig. 6.14(a) as our example again.

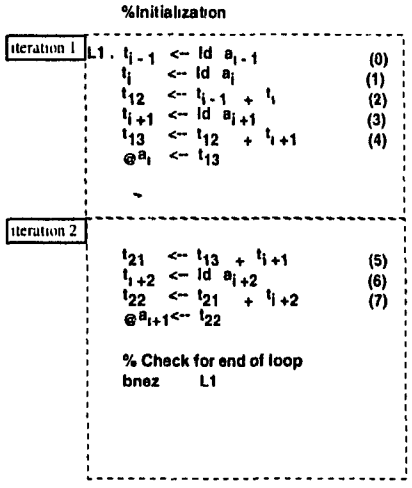
Step 1 : Finding The Period

(a) Dependence Distance = 0 : If the dependence distance is 0 then this step can be avoided as there are no inter-iteration dependencies. A single iteration of the loop body, in this case, constitutes a period.

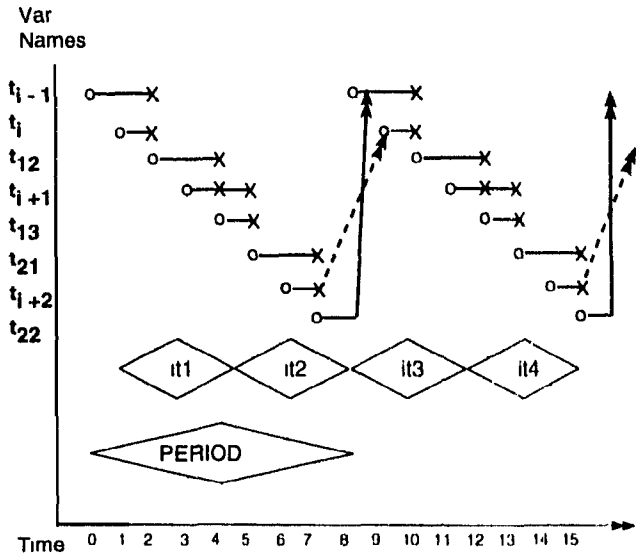
(b) Dependence Distance > 0 : When the dependence distance is greater than 0 then finding the period entails some work.

From Fig. 6.14(b) we see that each iteration references three vector elements $A[i-1]$ (through the temporary register t_{i-1}), $A[i]$ (through the temporary register t_i) and, $A[i+1]$ (through the temporary register t_{i+1}). $A[i]$ and $A[i+1]$ of one iteration becomes $A[i-1]$ and $A[i]$ of the next iteration respectively. The dependence distance is 1. We would like to embed this information in the cyclic interval graph that we create for this loop structure.

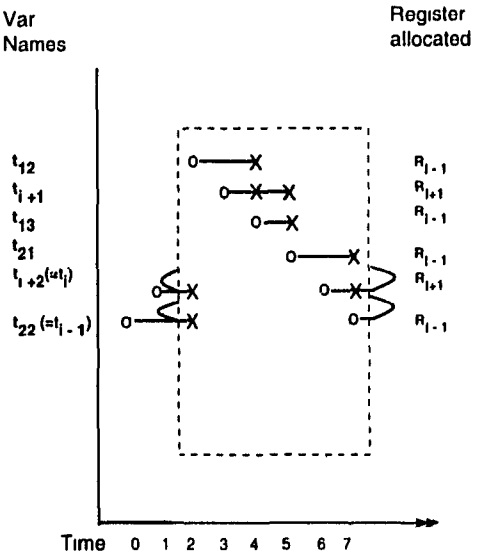
In order to do this we unroll the loop once (Fig. 6.15(a)) and create it's interval graph (Fig. 6.15(b)). A repeated pattern of access is seen after every two iterations and, this is what gives us the period. In this case the unrolling factor is the same as the



(a) Once Unrolled Code



(b) Interval Graph Of Unrolled Code



(c) Cyclic Interval Graph Of Loop

Figure 6.15: Period And Graph Of Loop

dependence distance. When several different dependence distances are present then we can get the unrolling factor by taking the least common multiple (lcm) of all the distance vectors of the loop. Consider the piece of code shown below.

```
FOR (i=..; i<..; i++) {
A[i] = A[i-2] + A[i-3] (1)
}
```

We see that instruction one has a dependence distance of 2 and 3. This means that $A[1]$ is defined in iteration 1 is read in iterations 3 as $A[i-2]$ and 4 as $A[i-3]$. Similarly, $A[2]$ is defined in iteration 2 and is read in iterations 4 as $A[i-2]$ and 5 as $A[i-3]$ and so on. Generally, $A[i]$ is defined at every iteration and this A[i] is read in the $i+2$nd iteration. So, if we unroll the loop twice (or any multiple of 2 like 2, 4, 6, ...) we will notice a regular pattern of access to $A[i]$ and $A[i-2]$. But $A[i]$ is also read in the $i+3$rd iteration and if the loop is unrolled three times (or any multiple of 3 like 3, 6, 9, ...) then we will see a regular pattern of access for $A[i]$ and $A[i-3]$. The first iteration at which the regular pattern of access for both $A[i-2]$ and $A[i-3]$ is seen together is 6, which is the lcm of the dependence distances, 2 and 3.

From Fig. 6.15(a) we notice that the dependence between the first and second iteration is folded into the body of the loop. t_{13} carries the value of $A[i]$ and t_{i+1} carries the value of $A[i+1]$ of the first iteration and they are used by the instruction 5 of the second iteration to calculate $A[i+1]$. Unrolling the loop has the effect of reducing the number of load requests. In this example the number of loads has decreased from 6 (for the rolled loop) to 3 for every two iterations.

The interval graph of the unrolled loop embed the dependence information between the intervals of the different iterations as is done in Fig. 6.15(b). The interval t_{i+2} of the first iteration carries the value of $A[i+2]$ and becomes $A[i]$ of the following iteration (that is iteration $i+3$). Similarly, the interval t_{22} carrying the value of $A[i]$ merges with $A[i-1]$ of the $(i+3)$rd iteration $(i+3)$. The solid and dashed arrows of Fig. 6.15(b) show these merges.

Step 2 : The Cyclic Interval Graph

After having unrolled the loop, we need to recognize the cyclic intervals of the graph. Once we know which intervals of the first period merge with which intervals of the second period we are able to create the cyclic interval graph as shown in step 1.

We see the cyclic interval graph in Fig. 6.15(c).

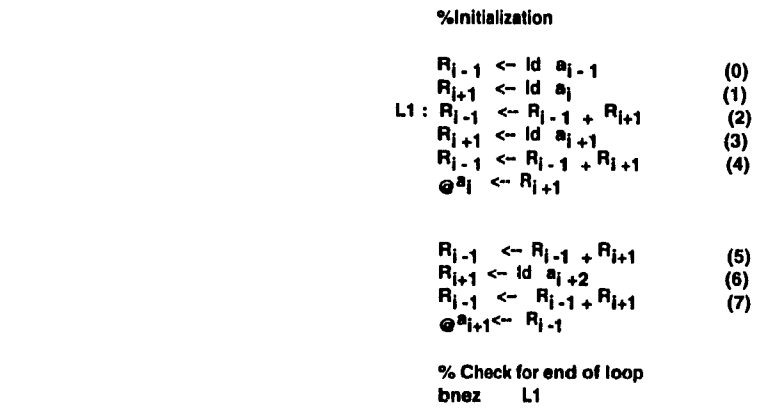


Figure 6.16: Final Register Allocation

Step 3 : The Coloring Phase

Now the colorer can be invoked. The graph is 2-colorable and is allocated the registers shown in Fig. 6.15(c). Finally, Fig. 6.16 shows the pseudo assembly code generated.

6.4 Summary

As only perfect loop structures have been taken into consideration in the previous chapters of the thesis, this chapter presents some thoughts on how the cyclic interval graph representation can be extended to take other program constructs, like loops having conditionals or nested conditionals, into account. The hierarchical cyclic interval graph is presented as a possible means of achieving this goal. We also touch on the topic of register allocation for subscripted array variables using the hierarchical cyclic interval graph approach.

Chapter 7

Related Works And Conclusions

In Section 7.1, we present a survey of work related to register allocation, graph coloring and, interval graphs. Finally, we summarize the main contents of the thesis and conclude in Section 7.2.

7.1 Related Work

In a number of recent publications, researchers have been trying to improve Chaitin's method for register allocation[1]. Briggs et al. recognized the fact that Chaitin's original heuristic is not guaranteed to find the minimum coloring [BCKT89]. They proposed a different heuristic method which simplifies the coloring phase by separating it from the spilling phase. That is, when the graph has been reduced to the stage where all remaining nodes have a degree greater or equal to k, it does not stop and spill. The algorithm continues the coloring process by selecting one remaining node to reduce the graph according to some heuristics. At the end of the reduction phase, the nodes are processed in the reverse order and are assigned colors. It is possible that during this process, a node with a degree greater or equal to k can still be colored, since more than one neighbor may have been allocated the same color. This method is based on interference graphs, and the coloring and spilling process may be iterated several times. Nonetheless, by avoiding some pointless spilling, improved code was generated for a number of test programs.

Bernstein et. al. have introduced a collection of heuristics which reduces the likelihood of excessive spill code generation [BGG+89]. The *width*, which is the number of live ranges at a certain point in the program, is used to compute the spill cost of a variable. The width

[1]Most of this section has been excerpted from [HGAM92]

coupled with the *depth* (of loop nesting) form the basis of their *area-based heuristics*. This method employs the interference graph as the basic representation of the program, and may require the graph to be rebuilt after spill code is introduced.

Callahan, Carr and Kennedy studied register allocation methods for subscripted variables, which poses a problem for many compilers [CCK90]. According to their method, array references which are live across several iterations are recognized and a source-to-source transformation called *scalar replacement* is performed so that they can be handled by coloring-based register allocators. Register moves are introduced to transfer values of subscripted variables across iterations, thus eliminating some load and store operations. However, the introduction of register moves, and the subsequent processing of register allocation seem to be orthogonal, and there exists no single unified framework for this optimization problem.

Another approach to the problem of register allocation for scalar and subscripted variables has been suggested by Duesterwald, Gupta, and Soffa [DGS92]. This method uses the *integrated register allocation graph*, which is an extension of the interference graph, to represent the coloring problem for both scalars and subscripted variables. The subscripted variables are allocated a set of registers that form a register pipeline.

Eisenbeis et. al. proposed a method based on cyclic scheduling for optimizing register usage on the Cray-2 [EJL90].

Interprocedural register allocation has been studied by a number of people [Cho88, SH89b, THL+86]. For example, Steenkiste and Hennessy have developed an algorithm for interprocedural register allocation where a *procedure interference graph* is constructed. Each node in the graph is a procedure of the program. Two procedures which are active at the same time are adjacent in the procedure interference graph. Each node of the graph is assigned a number of color that equals the number of registers needed by the local variables of the procedure. This number is determined by an intraprocedural-procedure allocation phase. A coloring algorithm assigns different colors to adjacent nodes of the procedure interference graph. Therefore, it is evident that a good solution for the minimum register allocation problem (as described in Problem 1 of Section 1.2) is important for the intraprocedural allocation phase.

Cytron and Ferrante have proposed a method of storage allocation where the amount of storage needed is equal to the maximum number of simultaneously live variables in the original program [CF87]. The objective of their work is to allocate storage for temporary variables by *renaming*, which is a compiler technique that transforms imperative programs to dataflow graphs [KKP+81, Den80]. They have pointed out that the formulation of the register allocation problem as a graph coloring problem based on the traditional interference graph may abstract away vital information present in the original program (like the width of interval graph), which their method uses to guide the register allocator to achieve an optimal solution efficiently. One difference between their work and the work proposed in

this paper lies in the treatment of loop variables. For example, a scalar variable defined in a loop, is either changed into an array by *scalar expansion* when the loop bound is known *a priori*, or it is transformed to a dynamically allocated variable when the loop bound is not known statically [CF87]. In our work, we treat such variables using cyclic intervals, thus the overhead of extra arrays or dynamic allocation is avoided.

Callahan and Koblenz have presented a register allocation method via hierarchical graph coloring [CK91]. The main idea is to represent the hierarchical program structure as a tree of *tiles*. Tiles are processed first in a bottom up fashion and the local interference graph is created and colored (perhaps with pseudo registers) on a tile by tile basis to capture the local usage pattern. Then a top down walk binds the pseudo registers to physical registers. Spill code is finally introduced in the less frequently executed portions of the program. Knobe and Zadeck have also proposed a hierarchical register allocation scheme based on *control trees* [KZ92]. A *prune* procedure is executed before coloring to reduce the register pressure to a desired threshold value by storing some values in memory on entry to a program region and then reloading them on exit. The authors claim that after pruning, the coloring process will terminate if the threshold value is set properly. A live range may need to be *split* during the coloring process. The coloring algorithms of both the hierarchical methods described above accept Chaitin's interference graph as their input.

Gupta et. al. reported their work in the area of global register allocation using *clique separators* [GSS89]. A clique separator is a completely connected subgraph. When it is removed from the graph, it disconnects the graph into at least two subgraphs. Their algorithm first partitions the code into code segments using clique separators. Each code segment is colored separately using the interference graph coloring method. Then, the colored subgraphs are combined by the global register allocator. In the presence of branching, the combining process may introduce register copying at the point where different control flow paths merge.

In their work, Proebsting and Fischer use a two step probabilistic approach to register allocation [PF92]. Local allocation is followed by global allocation. During the local allocation phase, the probability of a value residing in a register is calculated for every statement in the program and candidates whose next use is the farthest are chosen to be spilled. Global register probabilities are computed by combining local register probabilities with live variable analysis information. Probability information together with the profit obtained by keeping a value in a register guides the global allocation phase. Once this allocation process is complete, a graph coloring method is used to assign registers. Notice that in this approach, the problems of register allocation and assignment are separated. Register allocation involves making a choice of candidates to spill.

Kolte et. al perform the spilling on load/store ranges instead of on live ranges [KH93]. An interference graph of load/store ranges is created instead of live ranges and register allocation is performed using Chaitin's graph coloring scheme. The load range of a definition is the sequence of statements over which a variable must be allocated a register so as to

avoid a load before a use of the definition. Similarly, the store range of a definition is the sequence of statements over which a variable must be allocated a register so as to avoid a store at the definition. The authors claim that load/store ranges are advantageous as they provide information about the access patterns of variables.

As we pointed out before, our problems are related to the class of *circular-arc graph coloring* problems [Kle69, GJMP80]. The idea of using interval graphs for register allocation goes back over 15 years. Tucker was one of the first to note the advantages of the representation [Tuc75, Tuc84]. He also noted that the related concept of circular arc graphs could be applied to program loops. Interval graphs have also been used to overlay arrays and thereby minimize program memory requirements [Fab82], and to perform channel routing in VLSI layouts [Bur86, DGP88]. However, the practical use of interval graphs in register allocation appears to have been largely ignored because of perceived difficulties in dealing both with circular arc graphs and *hierarchical* interval graphs, both of which arise when dealing with real programs [BGG+89]. A great deal of theoretical work has been done, a good summary of which is found in [Gol85, SH89a].

Using circular arc graphs for register allocation has recently been proposed in high-level dataflow synthesis for digital systems [TS86, KP87, SvdB88]. In this application domain, the computation is represented by *data flow graphs*. Data flow graphs with loops can be modeled by *cyclic dataflow graphs* [PK87, Sto92], and the corresponding register allocation problem can be modeled by circular arc graphs [Sto92]. Unlike in compiler optimization, the hardware-oriented synthesis work traditionally does not address the issue of code spilling.

A recent application of the work described in this paper is the use of cyclic interval graph representation in a unified framework of loop scheduling and register allocation [NG93]. In fact, lifetime intervals can naturally be derived from an instruction schedule, and the register allocation scheme developed in this paper can be utilized effectively in the scheduling framework.

7.2 Conclusions

In this section we briefly summarize the most significant achievements of our research.

1. We presented the cyclic interval graph representation as an alternative to the interference graph that is used by traditional register allocators like Chaitin's global allocator. Our approach to register allocation based on cyclic interval graphs appears to be well suited to structured programs, and in particular large inner-loops having loop-carried dependences.

2. Based on the cyclic interval graph representation, we have devised a *two step* approach to register allocation.

From the interval graph we can establish the lower and upper bounds of the number of colors which will be required to color it. The number of colors required is related to the maximum and minimum thickness of the interval graph. We have noticed that a lot of interval graphs can be colored using as many colors as the maximum thickness of the graph. Based on this observation, the first step of our algorithm is the *sweep and split spilling algorithm* which transforms an input graph whose maximum thickness exceeds k to one whose maximum thickness equals k. We assume that the number of registers available on the target machine is k. Next, new heuristic algorithms, like the *fatcover and the greedy algorithms* color the transformed graph exploiting information that is readily available from the cyclic interval graphs.

3. We notice that the fatcover coloring algorithm performs sub-optimally in some situations and propose an important modification to the fatcover algorithm that overcomes this limitation.

4. The effectiveness of our approach has been tested on a suite of selected perfect loops from commonly used benchmarks like Livermore loops, SPEC89 and the whetstone benchmarks.

   From our limited experimental results, we find that in comparison to commercial compilers, like the native C compilers of SPARC and MIPS, the sweep and split spiller works well on loops which have high register pressure.

   We observe that the fatcover coloring algorithm is able to color loops with loop carried dependences very well. It generates optimal to near-optimal results in most benchmarks which have been tested. On the average the fatcover deviated by 5.2% from the optimal number of registers required. The fatcover algorithm also outperforms the greedy coloring algorithm in most cases.

5. Finally, we explore and describe plausible ways of extending our local loop allocator to handle broader and more complex classes of program structures.

   Eventhough the cyclic interval graph representation is well-suited to innermost loop structures, the the interference graph is the perhaps the best representation for less structured programs. We expect that these two representations can complement each other in a compiler, and we would like to combine them in an effective fashion.

Appendix A

Left Edge Algorithm

For the sake of completeness we provide a brief description of the Left Edge Algorithm [HS71].

Algorithm A.0.1 *Use the Left Edge Algorithm to color an* `interval graph`, *G such that G has no cyclic intervals.*

Input: An `interval graph`, G having no cyclic intervals.

The `register classes`, reg_{class}, present on the target architecture,

The `register preference`, reg_{pref}, of each interval of the cyclic interval graph. This determines the class of register that needs to be assigned to each interval. And,

The `number of registers`, reg_{num}, needed by each interval of the interval graph. The number of registers needed is the same as the number of colors that have to be assigned to an interval.

Output : The interval graph, G, such that
every interval, i, of the graph has been
assigned colors, $c_1 \ldots c_{reg_{num_i}}$, and,
$reg_{class}(c_i) = reg_{pref}(i)$ assigned colors, $c_{1 \ldots reg_{num}(i)}$, and,
$reg_{class}(c_{1 \ldots reg_{num}(i)}) = reg_{pref}(i)$

Main Left Edge Algorithm:

FOR each reg_{class}
Subgraphs,(G'), are created
where *all* the intervals, i, of G' are such that :
$reg_{pref}(i) = reg_{class}$
Initialize state of all colors of reg_{class} to be free
at all times, $t_1 \ldots t_n$
FOR every G'
FOR every time, t, in $t_1 \ldots t_n$ of G'
FOR each interval, i defined at t
Obtain colors, $c_i \ldots c_{regnum_i}$ that are free for the lifetime of i
Assign $c_i \ldots c_{regnum_i}$ to i
Mark $c_i \ldots c_{regnum_i}$ to i to be busy for the lifetime of i
DONE

Appendix B

Creating Hierarchical Interval Graphs From Code

In this section we describe how hierarchical interval graphs can be created from programs –

- by using the nesting levels of programs constructs,
- and by using control flow graphs created from the code.

Examples from Chapter 6.1 are reused to illustrate the process of creating hierarchical graphs.

For the sake of simplicity, let us assume that each statement is executed in unit time cycle.

The problem of creating hierarchical interval graphs can be approached in the following two ways :

In terms of the nesting level of the program :

Basic blocks are first created.
Next, the following two rules are used to establish the relationship between the various blocks :

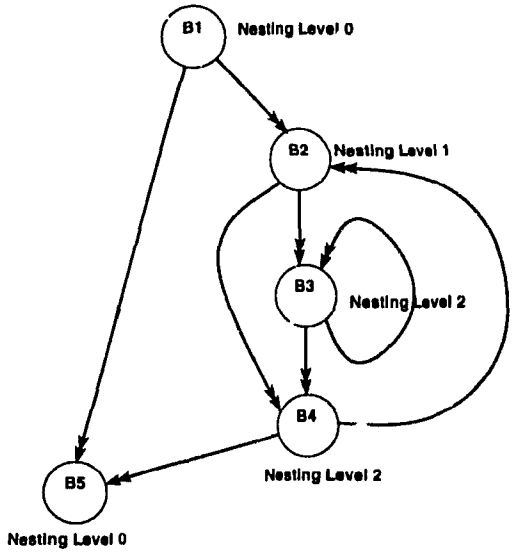
- Basic blocks at the same nesting level are considered to be sibling blocks.
- Blocks at a higher nesting level are contained within it's parent block which has a lower nesting level.

```
                                Nesting level is written
                                alongside in ()

                                                    (0)
for ( i = 1 ; i < n ; i++) {                        (1)

    ... code ...
    for ( j = 1 ; j < n ; j++) {                    (2)

        ... code ...

    } /* inner for loop */                          (1)

} /* outer for loop */                              (0)
```

(a) Code Segment : Loops

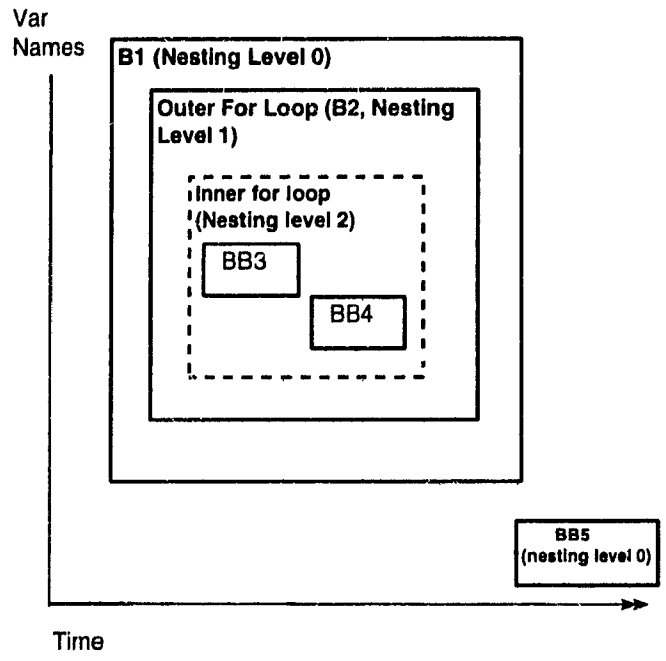
```
B 1     % Begin Outer Loop
        Rn1 <- n
        cmp Ri , Rn1
        jge L3
B 2     L1  Code for Outer Loop
        % Initialization for Inner Loop
        Rj  <-- 1
        Rn2 <-- n
        cmp Rj , Rn2
        jge L4
B 3     L2  Code for Inner Loop
        inc Rj
        cmp Rj , Rn2
        jl  L2
B 4     L4  inc Ri
        cmp Ri , Rn1
        jl  L1
B 5     L3  Out of Loops
```

(b) Assembly Code



(c) Control Flow Graph : Nested Loops

Figure B.1: (1) Creating Hierarchical Block Structures For Nested Loop Structures



(d) Interval Graph For Nested Loops

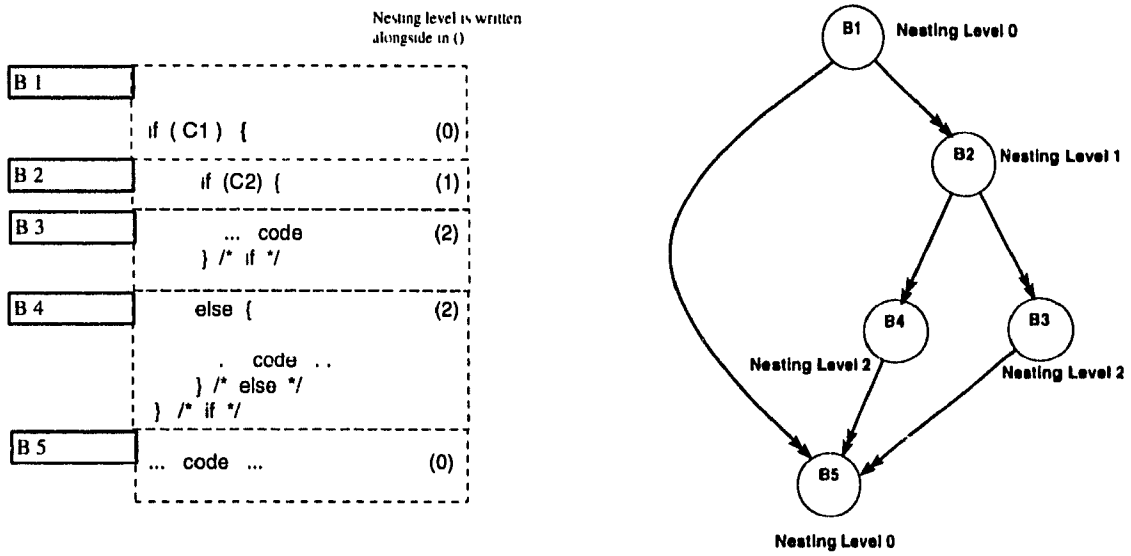
Figure B.2: (2) Creating Hierarchical Block Structures For Nested Loop Structures

The hierarchy of blocks created in this process follows the structure of the program. We take the small code segment of Fig. B.1(a) as a first example. The various basic blocks of the program segment is shown in Fig. B.1(b). Using the nesting level information from Fig. B.1(a), a hierarchical basic block structure is created as in Fig. B.2. Similarly, Fig. B.3(a, c) gives another example with embedded conditionals

The intervals within each block is created using def-use information gathered at an earlier stage of the compiler. This is one way of creating nested interval graph structures.

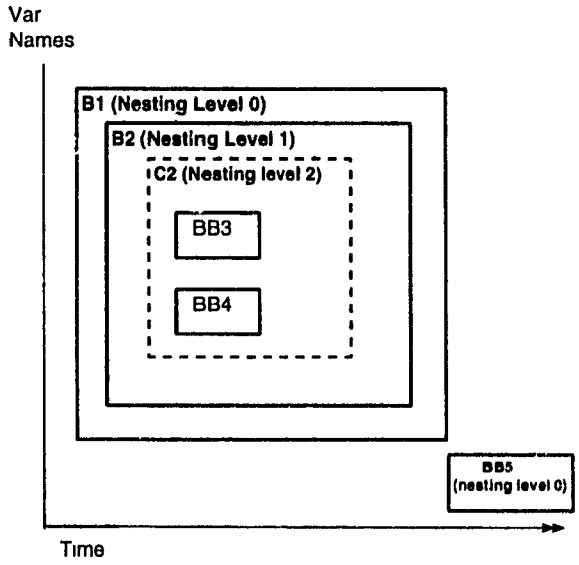
In terms of control flow graph :

It is easy to decipher the nesting level when working with a tree like intermediate structure. However, if we choose to have a flat intermediate structure, like a list, then the information about the nesting level is no longer explicitly available. At this level, we can create the hierarchical blocks from the control flow graph instead. Two algorithms are provided, one that is to be used with control flow graphs with back-edges and, the other that is to be used with control flow graphs without back-edges.



(a) Code Segment : Multi-way Switch

(b) Control Flow Graph



(c) Interval Graph For Nested Conditionals

Figure B.3: (1) Creating Hierarchical Block Structures For Nested Branch Structures

Non Looping Structures :

Assuming that the control flow graph is available and, it does not have any looping structures, that is there are no back-edges in the graph, we go back to our previous example of Fig. B.3. Part (b) shows the control flow graph for the assembly code shown in (a). The following steps are used to obtain a hierarchical interval graph structure while traversing the control flow graph in a top-down fashion.

1. Everytime a branch occurs, a counter is incremented. This counter essentially keeps track of the depth of the basic blocks in the program. Each basic block that can be branched into is assigned the value of this counter.
2. If two or more forward paths of execution reach a basic block then that basic block becomes a *converging* or *sync* block. For instance *B5* in Fig. B.3(b)) is a sync block.
   At a point when paths of execution merge, the counter (or the nesting depth) previously assigned to the sync block by a previous node is decremented and reassigned to it. In our example, *B1* branches into *B2* and *B5* both of which receive a nesting level of 1. Later, when either one of *B3* or *B4* branch into *B5*, we recognize *B5* as a sync node as two paths of execution are reaching it. The nesting depth of *B5* is decremented to 0. All subsequent paths of execution that lead to *B5* are not allowed to affect the nesting depth of *B5*.
3. Finally, each block, b, is embedded in a predecessor block, p, which has a nesting level that equals or is less than the nesting level of b.

These three steps find the nesting level of non-looping structures, and can be accomplished in one top-down pass of the control flow graph.

Looping Structures :

However, if loops or back-edges are present (Fig. B.2(c)), then these additional steps must be taken into consideration when assigning the nesting level of a block :

1. If the block, *B-succ* (or *B4* in Fig. B.2(c)), that immediately follows the block, *B-pred* (or *B3* in the diagram), is also a block from where back-edges emanate, then *B-succ* receives the same nesting level as *B-pred*. In general, two adjacent blocks which have back-edges emanating from them are assigned the same nesting level.
2. However, if the above condition does not hold true, then the block, *B-succ* (or *B5* in Fig. B.2(c)), that immediately follows a block from where back-edges emanate (*B4* in diagram) then, it is given a nesting level that is one less than the lowest amongst the blocks where the back-edge terminates. Back-edges from *B4* terminate at *B2* (which has a level of 1). Hence, *B5* receives a nesting level of (lower of (1) - 1) = 0.

Bibliography

[ABGM93] R. Archambault, R. Blainey, D. Gillies, and A. McLeod. Private Communication. June 1993.

[Alt93] E.R. Altman. Private Communication. December 1993.

[ARG93] E. R. Altman, G. Ramaswamy, and G. R. Gao. Software pipelining with resource and register constraints. Technical Report ACAPS-MEMO 79 (In Progress), McGill University, 1993.

[ASU88] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers—Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1988.

[BCKT89] P. Briggs, K. D. Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. *Proceedings of the 1989 SIGPLAN Conference on Programming Language Design and Implementation*, 24(7):275 284, July 1989.

[BCT92] P. Briggs, K. Cooper, and L. Torczon. Coloring register pairs. *ACM Letters on Programming Languages and Systems.* 1992.

[BGG+89] D. Bernstein, D.Q. Goldin, M.C. Golumbic, H. Krawczyk, Y. Mansour, I. Nahshon, and R.Y. Pinter. Spill Code Minimization Techniques for Optimizing Compilers. *Proceedings of the 1989 SIGPLAN Conference on Programming Language Design and Implementation*, 24(7):258 263, July 1989.

[Bri92] P. Briggs. *Register Allocation via Graph Coloring.* PhD thesis, Rice University, Houston, Texas, April 1992.

[Bur86] M. Burstein. Channel Routing. In T. Ohtsuki, editor, *Layout Design and Verification*, pages 132–167. North-Holland, 1986.

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

[CCK90] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Languages Design and Implementation*, pages 53–65, June 1990.

[CF87] R. Cytron and J. Ferrante. What's in a name? or the value of renaming for parallelism detection and storage allocation. *Proceedings of the 1987 International Conference on Parallel Processing*, pages 19–27, August 1987.

[CH84] F. Chow and J. Hennessey. Register allocation by priority-based coloring. *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 222 232, 1984.

[Cha82] G.J. Chaitin. Register Allocation and Spilling via Graph Coloring. *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*, pages 98–105, June 1982.

[Cho83] F.C. Chow. *A Portable Machine-Independent Global Optimizer - Design and Measurements*. PhD thesis, Stanford University, December 1983.

[Cho88] F. Chow. Minimizing register use penalty at procedure call. *SIGPLAN Notices, Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation.*, 23(7):85–94, 1988.

[Cho90] Hennessy J. Chow, F. and. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501 -536, 1990.

[CK91] D. Callahan and B. Koblenz. Allocation via Hierarchical Graph Coloring. *Proceedings of the 1991 SIGPLAN Conference on Programming Language Design and Implementation*, pages 192–203, July 1991.

[Den80] J. B. Dennis. Data flow supercomputers. *IEEE Computer*, 13(11):48–56, November 1980.

[DGP88] I. Dagan, M.C. Golumbic, and R.Y. Pinter. Trapezoid Graphs and their Coloring. *Discrete Applied Mathematics*, 21:35–46, 1988.

[DGS92] E. Duesterwald, R. Gupta, and M. L. Soffa. Register pipelining: An integrated approach to register allocation for scalar and subscripted variables. In *Proceedings of the CC '92 (Available in LNCS)*, October 1992.

[EH93] A. Erosa and L. J. Hendren. Taming control flow: A structured approach to eliminating goto statements. Technical Report ACAPS-MEMO 76, McGill University, September 1993.

[EJL90] C. Eisenbeis, W. Jalby, and A. Lichnewsky. Compiler techniques for optimizing memory and register usage on the CRAY2. *International Journal on High Speed*

Computing, 2(2):193–222. 1990. (Appeared also as INRIA Research Report no 1302), October 1990.

[Ers90] A. Ershov. Origins of programming: Discourses on methodology. page 133. Springer Verlag, 1990.

[Fab82] J. Fabri. *Automatic Storage Optimization.* PhD thesis, University of Michigan. 1982.

[Fre74] R.A. Freiburghouse. Register allocation via usage counts. *Communications of the ACM*, 17(11):638–642, 1974.

[Gao93] G.R. Gao. Private Communication. April 1993.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, New York, 1979.

[GJMP80] M.R. Garey, D.S. Johnson, G.L. Miller, and C.H. Papadimitriou. The Complexity of Coloring Circular Arcs and Chords. *SIAM Journal on Algebraic and Discrete Methods*, 1(2):216–227, June 1980.

[Gol85] M.C. Golumbic. Interval Graphs and Related Topics. *Discrete Mathematics*, 55(2):113–121, 1985.

[GSS89] R. Gupta, M. L. Soffa, and T. Steele. Register Allocation Via Clique Separators. *Proceedings of the 1989 SIGPLAN Conference on Programming Language Design and Implementation*, 24(7):264–274, July 1989.

[HGAM92] L. J. Hendren, G. R. Gao, E. R. Altman, and C. Mukerji. A register allocation framework based on hierarchical cyclic interval graphs. *Lecture Notes in Computer Science*, February 1992.

[HJ91] J. L. Hennessy and N. Jouppi. Computer technology and architecture: An evolving interaction. *Communications of the ACM*, 24(9):18 29, September 1991.

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

[HS71] A. Hashimoto and J. Stevens. Wire routing by optimizing channel assignment with large apertures. In *Proceedings of the 8th Design Automation Conference*, pages 155–169, 1971.

[Huf93] R.A. Huff. Lifetime-sensitive modulo scheduling. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Languages Design and Implementation*, pages 258–267, June 1993.

[Ken71] K. Kennedy. *Global Flow Analysis and Register Allocation for Simple Code Structures.* PhD thesis, Courant Institute, New York University, October 1971.

[KH93] P. Kolte and M. J. Harrold. Load/Store Range Analysis for Global Register Allocation. *Proceedings of the 1993 SIGPLAN Conference on Programming Language Design and Implementation*, 28(6):268–276, June 1993.

[KKP+81] D. J. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, pages 207–218, Williamsburg, Virginia, January 1981.

[Kle69] V. Klee. What Are the Intersection Graphs of Arcs in a Circle? *American Mathematics Monthly*, 76:810–813, 1969.

[KP87] F. J. Kurdahi and A. C. Parker. REAL: A program for register allocation. *Proceedings of the 24th Design Automation Conference*, pages 210–215, 1987.

[KZ92] K. Knobe and K. Zadeck. Register allocation using control trees. Technical Report CS-92-13, Department of Computer Science, Brown University, March 1992.

[LH86] R. Larus and P. Hilfinger. Register allocation in the SPUR Lisp compiler. *Proceedings of the 1986 SIGPLAN Conference on Programming Language Design and Implementation*, 21(7):255–263, July 1986.

[MB83] D. W. Matula and L. L Beck. Smallest-last ordering and clustering and graph coloring algorithms. *Journal of the ACM*, 30(3):417–427, 1983.

[NG93] Q. Ning and G. R. Gao. A novel framework of register allocation for software pipelining. *Proceedings of 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '93)*, pages 29–42, January 10 13 1993.

[Nic90] B. Nickerson. Graph coloring register allocation for processors with multi-register operands. *Proceedings of the 1990 SIGPLAN Conference on Programming Language Design and Implementation*, 25(6):40–52, June 1990.

[Nin93] Q. Ning. *Register Allocation for Optimal Loop Scheduling.* PhD thesis, School of Computer Science, McGill University, Montreal, Canada, May 1993.

[PF92] T. A Proebsting and C. N Fischer. Probabilistic Register Allocation. *Proceedings of the 1992 SIGPLAN Conference on Programming Language Design and Implementation*, 27(7):300–310, July 1992.

[PK87] P. G. Paulin and J. P Knight. Scheduling and Binding Algorithms for High-Level Synthesis. In *Proceedings of the 26th ACM/IEEE Design Automation Conference*, pages 1 6, Las Vegas, June 1987.

[RLTS92] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register allocation for modulo scheduled loops: Strategies, algorithms and heuristics. *Proceedings of the 1992 SIGPLAN Conference on Programming Language Design and Implementation*, 1992.

[SDK83] M. M. Syslo, N Deo, and J. S. Kowalik. *Discrete Optimization Algorithms with Pascal Programs.* Prentice Hall Inc., Englewood Cliffs, NJ, 1983.

[SH89a] W.-K. Shih and W.-L. Hsu. An $O(n^{1.5})$ algorithm to color proper circular arcs. *Discrete Applied Mathematics*, 25:321-323, 1989.

[SH89b] P.A. Steenkiste and J. Hennessy. A Simple Interprocedural Register Allocation Algorithm and Its Effectiveness for LISP. *ACM Transactions on Programming Languages and Systems*, 11(1):1-32, January 1989.

[SPA92] SPARC International, Inc. *The SPARC Architecture Manual*, version 8 edition, 1992.

[Sto92] L. Stok. Transfer Free Register Allocation in Cyclic Data Flow Graphs. In *Proceedings of the European Conference on Design Automation*, pages 181-186, Brussels, March 1992.

[SvdB88] L. Stok and R. van den Born. Synthesis of Concurrent Hardware Structures. In *Proceedings of the 24th Design Automation Conference*, pages 2756-2760, June 1988.

[THL+86] G. S. Taylor, P. Hilfinger, J. Larus, D. Patterson, and B. Zorn. Evaluation of the SPUR Lisp architecture. *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 444-452, June 1986.

[TS86] C. J. Tseng and D. P. Siewiorek. Automated synthesis of data paths in digital systems. *IEEE Transactions on Computer Aided Design*, 5(3), July 1986.

[Tuc75] A. Tucker. Coloring a Family of Circular Arcs. *SIAM Journal of Applied Mathematics*, 29(3):493-502, November 1975.

[Tuc84] Alan Tucker. *Applied Combinatorics.* John Wiley and Sons, Inc., 2nd edition, 1984.