McGill University
School of Computer Science
ACAPS Laboratory

Advanced Compilers, Architectures and Parallel Systems

A Register Allocation Framework Based on Hierarchical Cyclic Interval Graphs

Laurie J. Hendren
Guang R. Gao
Erik R. Altman†
Chandrika Mukerji

ACAPS Technical Memo 33 (Revised)
February 26, 1993

(Original version November 25, 1991)

†Dept. of Electrical Enginering, McGill University, 3480 University Street, Montréal, Québec

Abstract

In this paper, 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. We demonstrate that cyclic interval graphs provide a feasible and effective representation that accurately captures the periodic nature of live ranges found in loops.

A new heuristic algorithm for minimum register allocation, the *fat cover* algorithm, has been developed and implemented to exploit such program structure. In addition, a new spilling algorithm is proposed that makes use of the extra information available in the interval graph representation. These two algorithms work together to provide a two-phase register allocation process that does not require iteration of the spilling or coloring phases.

We extend the notion of cyclic interval graphs to hierarchical cyclic interval graphs and we outline a framework for a compiler to use this representation when performing register allocation for programs with hierarchical control structure such as nested conditionals and loops. The effectiveness of our approach has been demonstrated by experimenting with a collection of challenging loops.

Contents

1 Introduction **1**

2 Cylic Interval Graphs **3**

2.1 Interference Graphs and Chaitin's Heuristics 3

2.2 Cyclic Interval Graphs 3

2.3 Coloring Cyclic Interval Graphs 6

2.4 Further Observations about Cyclic Interval Graphs 7

3 Finding a Minimal Coloring of Cyclic Interval Graphs **8**

3.1 The Fat Cover Algorithm 8

3.1.1 An Introductory Example of the Fat Cover Algorithm 9

3.1.2 A Formal Description of the Fat Cover Algorithm 9

3.2 Two Other Approaches 11

3.2.1 An Interference Graph Algorithm 11

3.2.2 A Greedy Algorithm 11

3.3 A Hybrid Algorithm 12

3.4 An Experimental Comparison 13

4 Finding a k-Coloring of Cyclic Interval Graphs **16**

4.1 Chameleon Intervals, Register Floats, and Register Spills 17

4.2 Reducing the Width of an Interval Graph 18

5 Hierarchical Cyclic Interval Graphs **20**

6 Interval Graph Performance on Benchmark Programs **25**

7 Related Work **29**

8 Conclusions and Future Work **31**

R References **32**

List of Figures

1 An Interference Graph and an Interval Graph 4

2 A Cyclic Interval Graph 5

3 An Example of the Fat Cover Algorithm 9

4 An Example of Applying the Hybrid Fat Cover Algorithm 14

5 Number of Extra Registers Used in Coloring 1000 Random Graphs 15

6 An Example of Register Spilling and Register Floating 17

7 An Example of Introducing Spill Code 21

8 Nested Loops as Hierarchical Structured Interval Graphs 22

9 Examples of Consistent Allocation Required for Conditionals 23

10 A Nested Conditional and its Hierarchical Structured Interval Graph 24

List of Tables

1 Number of Double Precision Loads and Stores with 16 registers for Tomcat. . . . 27

2 Number of Double Precision Loads and Stores with 16 registers for Loop 8. . . . 27

3 Number of Double Precision Loads and Stores with 32 registers. 27

4 Interval Graph Statistics (after Spilling). 28

1 Introduction

Register allocation 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 [1]. The technology advance in the past decade has 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 [2]. Therefore, the benefit of keeping variables in registers is increasing, and thus the impact of good register allocation strategies is also increasing.

Register allocators in many modern compilers employ the classical graph coloring method originally proposed by Chaitin and improved by others[3, 4, 5, 6]. In this method, an *interference graph* is built to direct register allocation. 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 ranges. Chaitin's heuristics color the graph with k colors such that adjacent nodes are assigned different colors. Thus, a k-coloring of the interference graph corresponds to a feasible register assignment with k registers. If the graph is not k-colorable, *spill code* is introduced.

Given a set of live ranges, the interference graph encodes the overlapping or interference of all live ranges for an entire code segment, most often an entire procedure or function body. Thus, this representation provides a concise summary of the constraints that must be met for a correct register allocation. Furthermore, interference graph approaches are well-suited to general purpose register allocation that works even for programs with irregular and complex flow of control.

However, we believe that this concise and general strategy is sometimes also a weakness of the interference graph approach. From our point of view, interference graphs sometimes provide a representation of the problem that is too abstract, particularly in the case of well-structured inner-loop nests. More specifically, an interference graph does not encode any notion of the relative time of overlaps between live ranges. This information 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. 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.

Although there have been successful efforts so improve Chaitin's original interference graph approach (these are presented in Section 7), we propose a new approach that is based on a different representation - cyclic interval graphs. Intuitively, the "thickness" of each point in the cyclic interval graph captures information about overlapping live ranges of variables at a particular location in the program. We will see that this information — the "fat spot" — is crucial in developing our new heuristic algorithms.

We argue that an approach based on such interval graphs can address some of the shortcomings of the interference graph approaches. Since our approach takes advantage of program structure and the relative times of live ranges, we have focused on methods for structured programs, and in particular large inner-loops and loop nests. The main contributions of this paper include:

- We demonstrate that cyclic interval graphs provide a feasible and effective representation to characterize sequences of live ranges of variables in successive iterations of a loop. (Section 2)
- A new heuristic algorithm for minimum register allocation, the *fat cover* algorithm, has been developed, implemented, and studied. (Section 3)
- A new spilling algorithm is proposed that makes use of the extra information available in the interval graph representation. Whenever possible, it favors *register floats* (moving values from one register to another) over the traditional *register spills* (storing a spilled variable into memory). Furthermore, the spilling phase is only invoked once, and there is no need for iterating the allocation and spilling phases.(Section 4)

The possibility of using interval graphs as a model for register allocation was noted in [7, 8]. However, to our knowledge, previous research was theoretical in nature and mainly focussed on the algorithmic aspects of the interval graph model. Furthermore, the issues of applying such models to real programs with hierarchical control structures (such as nested conditionals and loops) have not been addressed. In this paper, our objective is quite different: we are primarily interested in the feasibility of using interval graphs in register allocators of real life compilers.

It is important to note that we do not claim that our interval graph methods can solve all the problems in register allocation. In general, we do not accept the view that there is an ultimate "best" method for register allocation. On the contrary, we believe that different approaches are suited to different sorts of programs. For example, for program segments with very irregular control flow, there is little program structure to exploit, and the interference graph representation is probably the best abstraction. However, for program segments such as nested loops and structured conditionals, there is often program structure that can be exploited by encoding the problem as cyclic interval graphs. Thus, we view the interference graph approach and the cyclic interval graph approach as complementary.

It should be noted that in most of the paper we concentrate on examples that are innermost loops. However, in Section 5, we discuss how our scheme naturally extends to a hierarchical method that can handle nested loops and conditionals. In Section 6, we present experimental results to demonstrate the effectiveness of our approach for two large loops that are difficult to color. We compare the results of our prototype spilling and coloring implementations with the results observed from compiling these loops with state-of-the-art optimizing compilers (SUN

Sparc C compiler, the MIPS C compiler, and the IBM RS6000 compiler). Finally, we present related work in Section 7 and our conclusions in Section 8.

2 Cylic Interval Graphs

In this section, we first review the traditional interference graph representation, and then introduce our cyclic interval graph representation.

2.1 Interference Graphs and Chaitin's Heuristics

As outlined in Section 1, the traditional approach uses an interference graph representation where nodes represent live ranges of variables, and edges represent interference between two live ranges. To be more precise, according to [4], two live ranges interfere, "if one of them is live at a definition point of the other." A node has *degree* k if it has k neighbors. Chaitin's method colors the graph with k colors such that two adjacent nodes are assigned different colors. Thus, a k-coloring of the interference graph corresponds to a feasible register assignment with k registers.

The basic heuristics of Chaitin's original method are based on a simple observation: a graph G having a node X with degree less than k is k-colorable if and only if the reduced graph G' formed by removing X with all its adjacent edges is k-colorable. Thus, Chaitin's algorithm tries to remove all nodes of degree less than k. If at some point there remains only nodes with degree greater or equal to k, then *spilling* is performed. This involves the introduction of some *spill code* (to store the definition of the spilled variable to memory and to load it for later uses) according to some heuristic. As the spill code replaces one long live-range with several short live-ranges, the interference graph is rebuilt and the coloring process must be repeated until a k-coloring succeeds without introducing any new spill code. Many improvements on Chaitin's original method have been proposed. For example, one such improvement is based on the observation that it is not always necessary to spill a node with k or more neighbors [6]. If some of the neighbors can be allocated the same color, then spilling is not necessary. This is one example where one can observe that a k-coloring can be achieved if the correct subset of nodes are colored the same color.

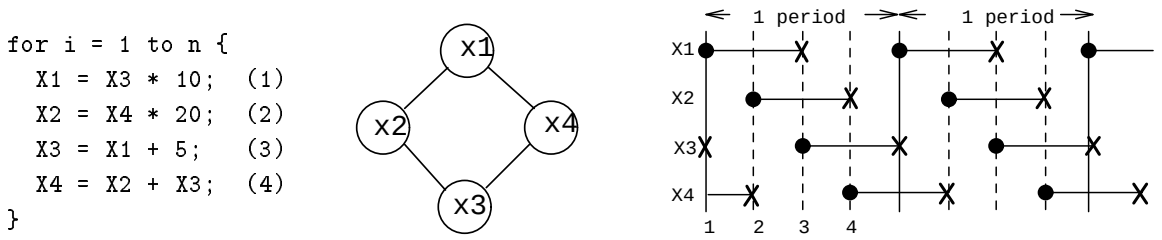
2.2 Cyclic Interval Graphs

The cyclic interval graph approach has been designed to expose program structure that is useful in choosing which intervals to color the same color, and which intervals are best to spill.

One challenge in designing this new representation is determining how to represent live ranges of loop variables. Figure 1 (a) shows a loop with n iterations. The numbers written

alongside the instructions are the instruction numbers. 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'.

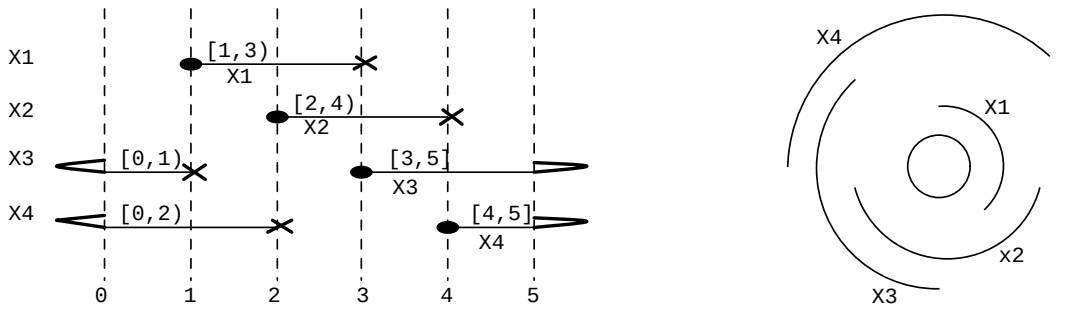
In Figure 1 (c), we show the interval graph for the program in Figure 1 (a). The X axis represents the instruction numbers of the code, while the Y axis represents the variables of the program. The solid circles of the diagram illustrate the point of definition while the crosses illustrate the points 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 by a sequence of intervals, one interval for each iteration.



(a) A Loop Program (b) An Interference Graph (c) An Interval Graph

Figure 1: An Interference Graph and an Interval Graph

As illustrated in Figure 1(c), 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 Figure 1 (c) 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. The numbers 0 and 5 do not correspond to any instructions but merely provide a joining point for two successive iterations. We call such a "wrapped" interval — a *cyclic interval*. In Figure 2 (a), we show the cyclic interval graph representation for Figure 1 (c).



(a) A Cyclic Interval Graph (b) A Circular Arc Graph

Figure 2: A Cyclic Interval 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.

We should note that cyclic intervals can also be used to represent live ranges of variables in loops which are first *unrolled*. Of course, each variable in the original loop will be split into several variables depending the unrolling factor. Then, the interval graph can be constructed where each new variable in the unrolled loop is associated with a cyclic interval.

We should also note that the period of a cyclic interval graph may be greater than 1 iteration. This may happen when a loop body contains array references that have a loop-carried dependency with dependence distance greater than 1. A straightforward extension can represent certain array references in a loop using cyclic intervals. A full discussion is beyond the scope of this paper, and readers are referred to [9].

Cyclic interval graphs can also be used to represent programs with hierarchical control structures such as nested loops and conditionals. In the case of nested loops, we naturally get nested cyclic interval graphs. In the case of conditionals, we create a structure similar to nested loops by introducing the proper constraints between the two branches of the conditionals. In Section 5, we discuss this process further.

Before presenting the main problems to be solved on cyclic interval graphs, we first introduce the following definitions.

Definition 2.1 *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 2.2 *Two intervals $I1, I2$* overlap *if there exists a time t that is covered by both $I1$ and $I2$.*

2.3 Coloring Cyclic Interval Graphs

Our two main coloring problems can be formulated as:

- 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.
- 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.

The importance of Problem 2 is obvious and is probably familiar to most compiler writers. We therefore focus on the importance of Problem 1, the problem of finding a minimum coloring of a cyclic interval graph. We argue that this is an important problem for the following two reasons:

1. It has important applications in situations when the smallest number of registers is required. For example, when allocating registers interprocedurally it is beneficial to allocate a minimal number of registers to each procedure using such a solution. This reduces the amount of register saving required at procedure call time, and can also improve interprocedural register allocation [10].
2. Using the information captured by interval graphs, we have developed a two-step approach for solving Problem 2. This approach makes effective use of the optimal solution of Problem 1 to minimize the spilling cost. As we show in Section 4, this is particularly important for programs in which the register pressure is close to k.

The above two problems are treated in Section 3 and Section 4 respectively.

2.4 Further Observations about Cyclic Interval Graphs

Our problems are related to the class of *circular-arc graph coloring* problems [11, 12]. 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 intersect one another. In Figure 2 (b), we show the circular-arc graph representation of our example. 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 $O(nk!klogk)$ [12]. In this paper, we are interested in fast heuristic methods which find a k-coloring quickly, and generates efficient code for spilling when necessary.

As in any general graph coloring problem, finding the minimum coloring of a cyclic interval graph is NP-hard [12]. 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. From our examples, we can observe 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.

Definition 2.3 *The* width *of a cyclic interval graph G at time t, written as width(G,t), is the number of intervals covering t.*

Definition 2.4 *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.*

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 2.1 *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-to-right* algorithm) 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 2.2 *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 k 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 an acyclic graph. Coloring G' with the left-to-right algorithm guarantees it to be colored with no more than $k' = W_{max}$ colors (Theorem 2.1). Then, it is trivial to see that we can use $W_{min}(G)$ more colors to color the removed intervals. ∎

3 Finding a Minimal Coloring of Cyclic Interval Graphs

In this section 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. Such algorithms will all use some sort of heuristic, and therefore they are not guaranteed to find the optimal solution. However, our goal is to find algorithms that always find the optimal or close to optimal solutions. In subsection 3.1 we develop a new algorithm, the *fat cover* algorithm. In subsection 3.2 we give a short summary of two other approaches to the minimal coloring problem and, in subsection 3.3 we describe another important method, the *hybrid* algorithm. Finally, in subsection 3.4 we present experimental results to compare the effectiveness of the four algorithms.

3.1 The Fat Cover Algorithm

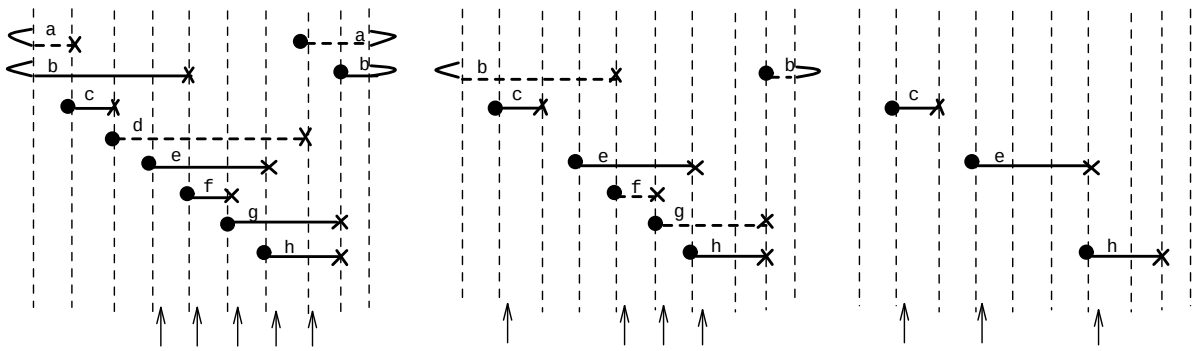
Given the fact that the optimal k for a cyclic interval graph G is bounded by $W_{max}(G)$ and $W_{min}(G) + W_{max}(G)$ (Theorem 2.2), and our experimental observations which indicate that a large majority of graphs that could represent programs can be colored in W_{max} colors, we have developed an algorithm, called *the fat cover algorithm*, 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* of the interval graph are the locations that are most important, and that 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. We first introduce this idea informally with an example, and then give a more formal development of the algorithm.

3.1.1 An Introductory Example of the Fat Cover Algorithm

Consider the graph given in Figure 3(a) which has a maximum width of 3, and two cyclic intervals `a` and `b`. The fat spots, or the points of maximum width, are indicated by arrows. The objective of the fat cover algorithm is to find a set of non-overlapping intervals that includes a cyclic interval. In Figure 3(a) we have indicated such a set with dashed lines (intervals `a` and `d`) – we call this a *fat cover* relative to `a`. If we color both `a` and `d` with the same color, then we reduce the original problem to that of finding a 2-coloring for the graph given in Figure 3(b). Here we find that a fat cover for `b` is {`b`, `f`, `g`}. We can then reduce the problem to a 1-coloring of the acyclic interval graph given in Figure 3(c), which is clearly 1-colorable.



(a) finding a fat cover for `a` (b) finding a fat cover for `b` (c) the remaining graph

Figure 3: An Example of the Fat Cover Algorithm

3.1.2 A Formal Description of the Fat Cover Algorithm

Given the basic idea of the algorithm as presented in the previous section, we now give a more rigorous description of the algorithm.

Definition 3.1 *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 *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 3.1 *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. 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. Secondly, it is clear that by removing F from G we are left with a graph that is colored with $k - 1$ colors. ■

The development of our *fat cover* algorithm was inspired by Theorems 2.1 and 3.1. Given a graph G with m cyclic intervals $I_{c_1}, I_{c_2}, \ldots, I_{c_m}$, the algorithm proceeds in two phases. 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 I_{c_i} (call this fat cover F_i). Our implementation ensures that a fat cover will be found in a left to right traversal, if one exists. 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.

After all of the m cyclic intervals are dealt with in this first phase, then the second phase uses a straightforward left-to-right algorithm to color the remaining intervals. If the first phase succeeds, then the second phase need only consider a reduced graph G' that contains no cyclic intervals, and 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 2.1). Thus, if the first phase succeeds, we can find an optimal coloring in $k = W_{max}(G)$ colors for graph G. 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 resulting coloring may or may not be optimal. However Theorem 4.1 ensures that, for a graph G which is $k = W_{max}(G)$ colorable, it is possible for our algorithm to succeed in handling the cyclic intervals by finding the appropriate fat covers.

It should be noted that 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 F' such that $G - F'$ is not $k - l$ colorable, even though there exists another fatcover F such that $G - F$ is k-colorable.

For graphs with one cyclic interval, any fat cover will do, and we do find the optimal solution. This is because we are guaranteed to find a fat cover, F, if one exists. Furthermore, if a fat cover is found, we know that the graph $G - F$ will have maximum thickness of $k - 1$, and $G - F$ will contain no cyclic intervals. Therefore, by theorem 2.1, we can guarantee that $G - F$ can be colored with $k - 1$ colors, and G can therefore be colored in k colors.

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 [3]. 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.

It should be noted that the fat cover algorithm is *not* computationally expensive. For each of the cyclic intervals, one sweep of the graph is required (where the size of the graph is exactly the number of 3-address statements in the program). All of the remaining intervals can be handled by one final left-to-right sweep. Furthermore, since interval graphs that correspond to programs have at most one new interval per time step, the complexity at each point in the sweep is effectively constant.

3.2 Two Other Approaches

In this subsection, we describe a naive coloring algorithm based on Chaitin's original *Interference Graph* approach and a *Greedy* algorithm.

3.2.1 An Interference Graph Algorithm

Since a cyclic interval graph contains information about interference (or overlap) among intervals, we can use a Chaitin-style reduction algorithm to discover a k which guarantees that a cyclic interval graph can be colored in k colors. At the ith step of this algorithm one removes, from among the remaining intervals, the interval with the fewest number of overlapping intervals (this corresponds to removing the node with least degree from an interference graph). For a graph with n intervals, there will be n steps.

Call the interval that is removed at step i, I_i, and let d_i denote the number of overlapping intervals (degree) of I_i at step i. Now pick $k = max(d_1, d_2, \ldots, d_n) + 1$. We can color the original graph in k colors by coloring the intervals in the order I_n, $I_{(n-1)}$, ..., I_1. At each step we color an interval that has at most $k - 1$ previously colored overlapping intervals.

3.2.2 A Greedy Algorithm

Another approach to coloring a cyclic interval graph is to first color the cyclical intervals, and then use a greedy algorithm to color the remaining intervals. At each step in the greedy algorithm the following three steps are performed :

1. From among the uncolored intervals, choose the "best" one to color next, call it I_{next}. Some possible criteria for choosing the "best" interval include: (a) the leftmost uncolored interval (the interval with the lowest starting time), (b) the longest uncolored interval, (c) the interval which overlaps with the most uncolored intervals, or (d) the interval which has the fewest number of available colors (where a color C is *available* for interval I, only if C has not been used for any interval that overlaps with I).

2. From among the colors available for I_{next}, choose the "best" color, call it C_{next}. If no color is available for this interval, then allocate a new color. Some possible criteria for choosing the "best" color 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 I_{next}), (b) worst-fit, and (c) the color which can be used for the fewest number of unallocated intervals.

3. Assign color C_{next} to interval I_{next}.

The experimental results reported in Figure 5 used a greedy algorithm with option (d) for choosing the interval to color, and option (c) for for choosing which color to use (these were found to be the best heuristics).

3.3 A Hybrid Algorithm

A hybrid algorithm can combine the best points of the interference graph approach with either the fat cover method (presented in 3.1) or the greedy method (presented in 3.2.2). Given a graph G, this algorithm finds the coloring in three phases. The first phase applies a reduction step based on interference information. This phase repeatedly removes all intervals that have fewer than $W_{max}(G)$ overlapping intervals. Let us call the intervals removed $I_1, I_2, \ldots, I_m$, and the graph remaining G'. Phase 2 applies either a greedy algorithm or the fat cover algorithm to color the intervals in G', and finally phase 3 colors the intervals removed by phase 1 in the order $I_m, I_{(m-1)}, \ldots, I_1$.

Let us illustrate the *hybrid fatcover* algorithm with a step-by-step example given in Figure 4.

Figure 4(a): In this picture we give the input interval graph. Notice that this graph has a maximum width of 3, and therefore the fat cover algorithm will try to color it with k=3 colors.

Figure 4(b): In this figure we illustrate the graph after the initial reduction phase. The reduction phase simplifies the graph by removing all intervals that overlap with only 2 other intervals. This is equivalent to applying a Chaitin-like algorithm with k=3. Note that only intervals `c` and `f` were removed, and all of the remaining intervals overlap with

3 other intervals. This means that in order to use only 3 colors, a naive interference graph algorithm would have to resort to spilling at this point. However, as illustrated with the next four pictures, when the fat cover algorithm is applied to the remaining intervals, a 3-coloring is found.

Figure 4(c): The first phase of the fat cover algorithm is to find a fat cover for each cyclic interval. As shown by the arrows in this picture, there are four fat spots to be covered. By traversing left from interval `a`, the cover of {`a`,`d`} is found. We can now color `a` and `d` with a new color *red*, and proceed to the next phase.

Figure 4(d): This figure shows the remaining uncolored intervals. Note that the width is now 2, and there are 3 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*.

Figure 4(e): 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*. In general, if the phase that handles all of the m cyclic intervals succeeds, then we are guaranteed to be able to color the remaining intervals with $W_{max} - m$ colors. In this example W_{max} is 3, and m is 2.

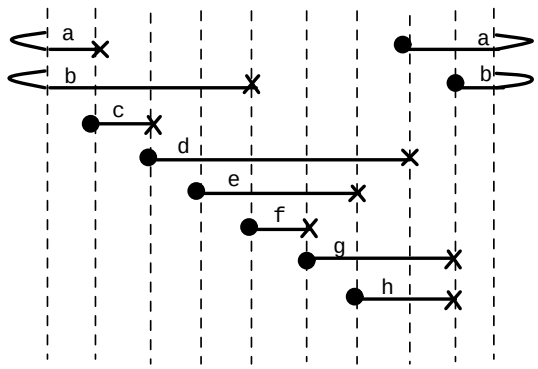
Figure 4(f): The final coloring is shown in this figure. Note that `a` and `d` are *red*, `b` and `g` are *blue*, `e` and `h` are *green*, and the two intervals removed in the first step, `c` and `f` have been colored with colors not used in any overlapping intervals.

It should be noted that this algorithm could be improved further by allowing multiple alternations between the interference graph heuristic and the fat-cover algorithm. For example, after finding the fat cover in Figure 4(c), one could remove all intervals with degree less than 2.

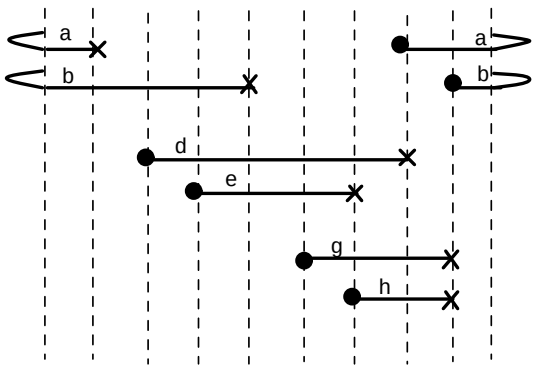
3.4 An Experimental Comparison

In order to experiment with a wide variety of coloring approaches and coloring heuristics, we implemented an experimental platform that supports all four approaches outlined above and also supports a wide variety of heuristics for the greedy approach. The tables given in Figure 5 summarize the experimental results that we collected for the following algorithms: (1) the algorithm based on the naive interference graph (see Section 3.2.1), (2) a greedy algorithm (see Section 3.2.2), (3) the fat cover algorithm (as described in the Section 3.1), (4) a hybrid algorithm that has a first phase based on interference information, and a second phase that applies the greedy method (see Section 3.3), and (5) a hybrid algorithm that uses the fat cover method for the second phase.

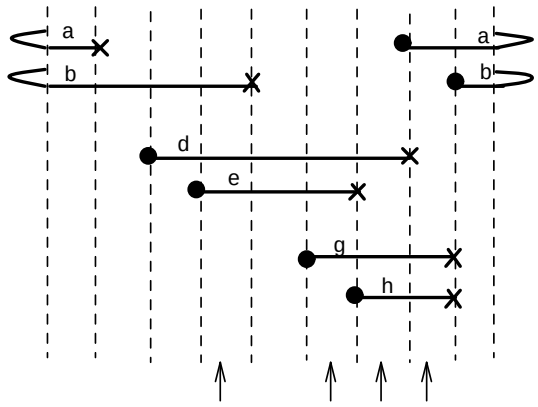
One can think of our experiments as a "heuristic algorithm challenge". We experimented with four graph sizes - 10, 20, 30, and 50 intervals. For each graph size, we studied graphs which



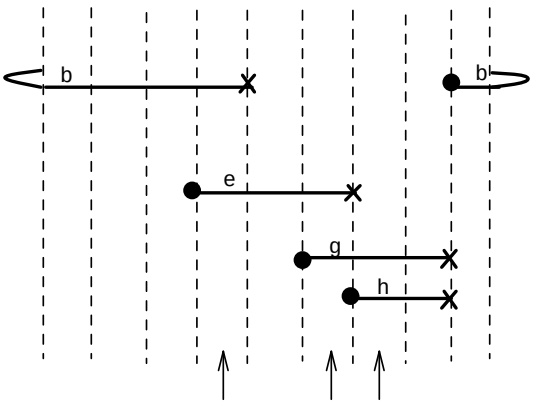
(a) the original interval graph



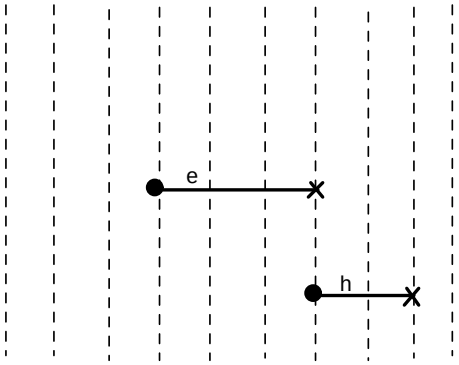
(b) after removing intervals with degree < 3



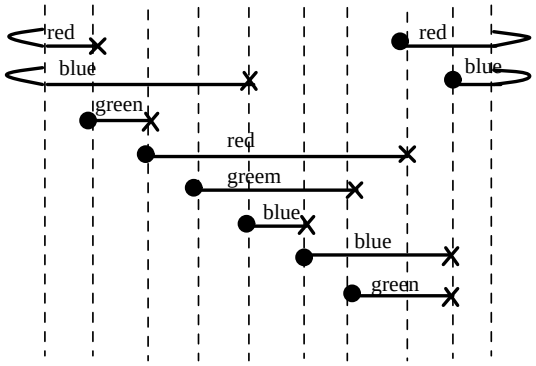
(c) finding a fat cover for `a` - {`a`,`d`}



(d) finding a fat cover for `b` - {`b`,`g`}



(e) coloring the rest



(f) the final coloring

Figure 4: An Example of Applying the Hybrid Fat Cover Algorithm

contained 2,3,4,5, and 10 cyclic intervals. For each of the 20 combinations of graph size and number of cyclic intervals, we ran 1000 experiments on *program graphs* which were generated

10 intervals	# Cyclic Intervals				
	2	3	4	5	10
Naïve Interference	2	18	36	51	0
Greedy	2	6	4	3	0
Fat Cover	0	2	5	3	0
Hybrid Greedy	0	1	0	1	0
Hybrid Fat Cover	0	1	0	0	0

20 intervals	# Cyclic Intervals				
	2	3	4	5	10
Naïve Interference	6	13	17	40	54
Greedy	5	15	22	12	10
Fat Cover	0	5	10	20	25
Hybrid Greedy	0	2	2	1	3
Hybrid Fat Cover	0	0	0	1	2

30 intervals	# Cyclic Intervals				
	2	3	4	5	10
Naïve Interference	2	19	19	30	92
Greedy	9	16	33	42	45
Fat Cover	0	2	7	17	69
Hybrid Greedy	1	4	2	3	5
Hybrid Fat Cover	0	0	1	0	9

50 intervals	# Cyclic Intervals				
	2	3	4	5	10
Naïve Interference	2	13	25	40	114
Greedy	8	16	40	43	82
Fat Cover	0	2	8	11	55
Hybrid Greedy	0	0	1	7	17
Hybrid Fat Cover	0	0	1	1	14

Figure 5: Number of Extra Registers Used in Coloring 1000 Random Graphs

randomly. These program graphs correspond to cyclic interval graphs that could arise from inner loop constructs of a program, i.e. they are not truly random graphs. Each time point in the graph corresponds to one instruction which may have at most one definition, and at most 2 last uses. Non-cyclic intervals correspond to live ranges that are defined and last-used within the same iteration, while cyclic intervals correspond to live ranges that are either live throughout the loop, or defined in one iteration and used in the next. The graph given in Figure 3 is an example of one such graph that has 8 intervals of which 2 are cyclic intervals.

Each experiment was run as follows. Given an input interval graph, each of the five algorithms was applied to the graph, and the number of colors used by each algorithm was reported. If we let k_{min} be the minimum number used by any of the algorithms, then each algorithm was charged with penalty points, one point for each register more than k_{min} that it used. The numbers reported in Figure 5 corresponds to the number of penalty points charged to each algorithm over 1000 experiments. A score of 0 for an algorithm means that it produced the minimal number of registers in all 1000 experiments. The results clearly indicate that the hybrid fat cover algorithm is the winner. In 12 of 20 cases it gave the best result in all 1000 experiments (a score of 0), and in 19 of 20 cases it equaled or beat all of the other algorithms (in the case of 30 intervals with 10 cyclic intervals it was a close second).

These results do not give conclusive answers for how the various algorithms will behave on real application programs, but it does show a general trend for a large number of graphs that correspond to possible programs (any graph we generate corresponds to a possible program). Thus, we can see that the hybrid algorithm is worth implementing in a real compiler, and that

the fat cover is likely a better idea than a wide variety of greedy heuristics. As we will see in the next section, the fat cover algorithm also provides some advantages when we are trying to reduce spill code. Thus, these experiments were important for us to demonstrate that the fat cover algorithm appears to do quite well for a wide variety of graphs. Furthermore, the results motivated us to pursue the next step of integrating the fat cover algorithm with a new spilling strategy.

4 Finding a k-Coloring of Cyclic Interval Graphs

In the previous section we presented the fat cover algorithm that was designed to find a coloring in a minimal number of registers. In this section we present a new approach for allocating registers given the constraint that only k registers are available, and the minimal number of registers required to color the graphs is k', where $k' > k$.

General interference graphs have several drawbacks that our interval graph representation solves naturally:

Separation of the spill phase from the coloring phase: Given that we developed a good algorithm for coloring a graph G with maximum thickness $W_{max}(G)$, we take the approach that register allocation should proceed in two phases. Given $k < W_{max}(G)$, the first phase transforms G to an equivalent graph G' that has maximum thickness $W_{max}(G') = k$. This transformation process introduces register spills and is **guaranteed** to produce a graph G' that can be colored by the second phase without introducing any further register spills. Thus, only one application of each phase is required.

This differs from most approaches based on interference graphs that introduce spilling during or after the register allocation phase. These approaches cannot guarantee that the spilling will result in a k-colorable interference graph in one pass, and it is necessary to iterate the coloring/spilling process until a k-colorable solution is found. It should be noted that the approach given in [13] suggested a means of avoiding this iteration, but it uses a more complex algorithm than that required here.

Choice of spilled quantities: We use the information stored in the cyclic interval graph to make good decisions on which intervals to spill. This information is not available in the interference graph representation, and therefore cannot be exploited in spilling techniques based on that representation.

It should be noted that a similar approach has been used in the context of interference graphs [14]. In this approach the "width" [1] of the interference graph was used for one of the spill heuristics. Similarly our algorithm uses the width of the interval graph as

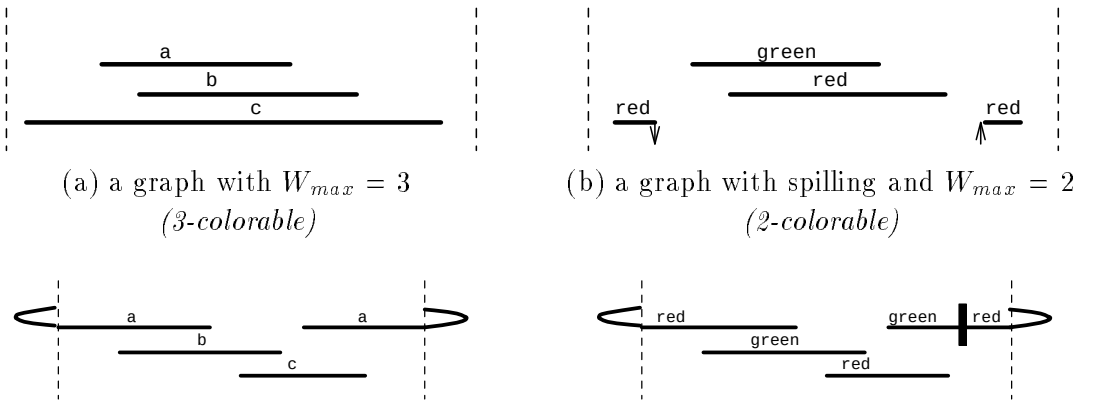
[1] The width of the interference graph at any given point in time is defined to be the number of live variables at that point.

one of the criteria when choosing a node to spill. However our representation captures the width of the graph at every point in the program very naturally and makes it easier to beneficially exploit this information. Furthermore, we can make use of a natural metric for distance, even for cyclic intervals.

Register Floats: Our approach uses a two-level mechanism: (1) floating registers and (2) spilling registers. As fully explained in the next section, a register float corresponds to moving a value from one register to another register, while a register spill corresponds to moving a value from a register to a memory location and back. Clearly a register float is preferred over a register spill.

4.1 Chameleon Intervals, Register Floats, and Register Spills

By carefully studying the structure of cyclic interval graphs, one can see that there are two quite different constraints that make a graph not colorable in k registers. The first is the 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 Figure 6(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 Figure 6(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.[2]



(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)*

Figure 6: An Example of Register Spilling and Register Floating

The second situation is more subtle. Consider the graph given in Figure 6(c). This graph has a maximum width of 2, but is not 2-colorable. In this situation we have not really run out

[2]It was important to choose `c` to spill. Spilling either `a` or `b` would not result in reducing the $W_{max}(G)$ to 2.

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.

If we allow the interval for variable `a` to change color at the location indicated by the solid bar in Figure 6(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.[3]

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 Figure 6(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 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.

4.2 Reducing the Width of an Interval Graph

Given that we have the coloring algorithm described in the previous section, 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 Figure 6(b)), this transformation must introduce register spills. Therefore we would like an approach which attempts to minimize the number of register spills.

We have developed a new algorithm, the *sweep and split* algorithm that is based on the cyclic interval graph representation.[4] Like the fat cover coloring algorithm, the sweep and split

[3] The idea of register floats is not new, however difficulty in efficiently identifying values to treat as register floats has prevented their widespread use. 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] A similar method was proposed for basic blocks in [5].

algorithm takes advantage of the extra information available in the interval representation. Since this algorithm is straight-forward, we give only an overview.

The central idea of this 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 furthest next use from i. Note we do not split the whole interval, 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. 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.

Let us now give a concrete example of applying the sweep and split algorithm to a cyclic interval graph that corresponds to a small program. In Figure 7(a) we give a small illustrative program, and in Figure 7(b) we give the 3-address code.[5] Assuming that the number of available registers (k) is 3, Figure 7(c) gives the 3-address code program that results from applying the sweep and split algorithm. Figure 7(d) shows the cyclic interval graph for this loop, and Figures 7(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:

Figure 7(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.

Figure 7(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.

[5] Assuming right-associativity, and right to left evaluation.

Figure 7(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.

Figure 7(g) (Time = 4): At time step 4, and all times greater than 4, the width is 3, and so no further spilling is required.

5 Hierarchical Cyclic Interval Graphs

In the previous sections we have concentrated on cyclic interval graphs that represent innermost loops. However, it is very important to note that our techniques are not limited to these cases. In fact, there is a natural hierarchical representation for structured programs that contain nested conditionals and loops.

We should note that our emphasis has been on the application of our methods for innermost loops. We include this discussion of hierarchical structures to show that it is plausible and relatively straight-forward to extend the method to structured programs. For the parts of the program with very irregular control structure, an interference graph approach is probably more effective.

The overall strategy for hierarchical graphs is to allocate pseudo-register numbers in a bottom-up fashion according to the nested structure of the program, and then to assign real register numbers in a top-down pass.

Let us first consider the case of nested loops as illustrated in Figure 8(a). In this example, it is quite clear that this is just a nesting of cyclic interval graphs, with the cyclic interval graph for LOOP 2 nested inside a cyclic interval graph for LOOP 1. Thus, we can apply our spilling and coloring algorithms in a structured manner, starting with the innermost loop and working outwards. For example, let us assume that we find a 4-coloring for LOOP 2 in Figure 8(a). As shown in Figure 8(b), we can proceed to the next outermost loop, LOOP 1, by replacing LOOP 2 with four intervals. Two of these intervals are grafted into the lifetime for variables `a` and `b`, and the other two autonomous intervals represent the other 2 colors required for LOOP 2 (for intervals `c`, `d`, and `e`).

We can now find a coloring for LOOP 1, which is now a non-hierarchical cyclic interval graph. Since this is the topmost level in the hierarchy, we can use the pseudo-register numbers as the real register numbers for all non-autonomous intervals. For the two autonomous intervals, the real register numbers must be assigned to all intervals in LOOP 2 that correspond to the autonomous intervals. For example, consider the situation where autonomous intervals were assigned real registers R_1, and R_5, and the intervals inside LOOP 2 had been assigned pseudo

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

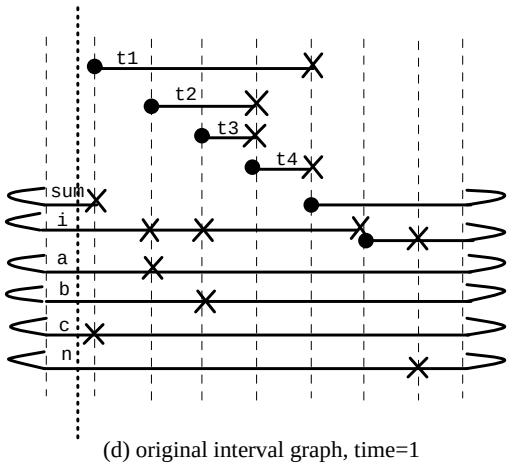
(a) original loop

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

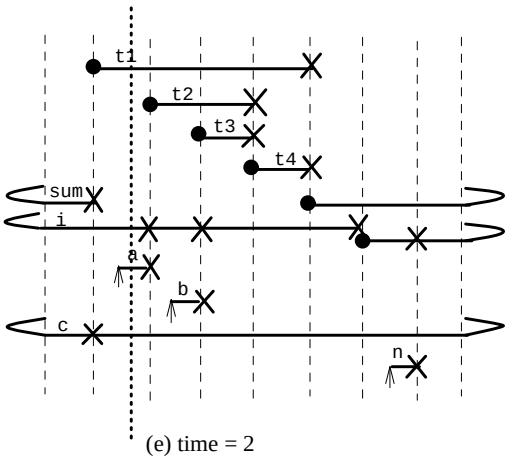
(b) 3-address code

```
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
```

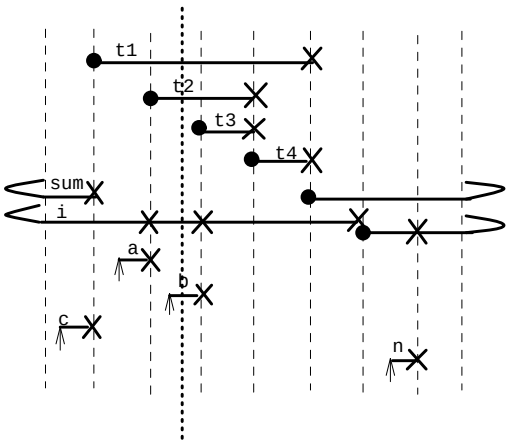
(c) 3-address code after spilling



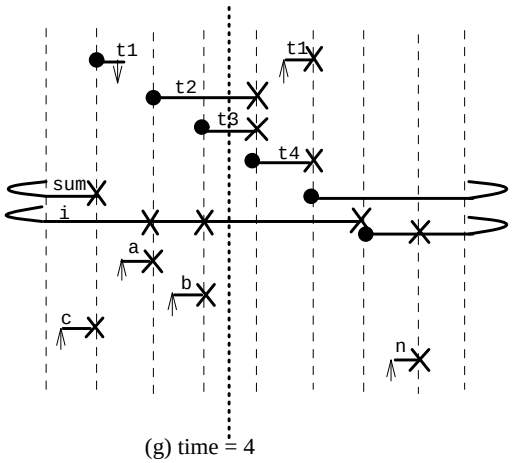
(d) original interval graph, time=1



(e) time = 2

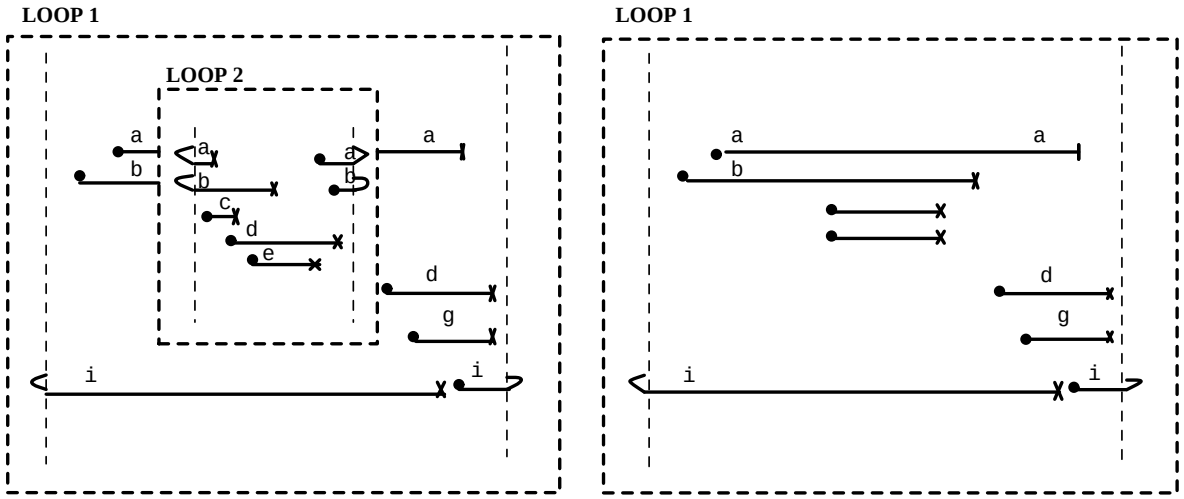


(f) time = 3



(g) time = 4

Figure 7: An Example of Introducing Spill Code



(a) a nested loop (b) after allocating the inner loop

Figure 8: Nested Loops as Hierarchical Structured Interval Graphs

registers P_1 to `c`, P_2 to `d`, and P_1 to `e`. Then, on the top-down pass we must assign R_1 to all intervals assigned to P_1 (`c`,`e`) and R_5 to all occurrences of P_2 (`d`).

In the case of nested conditionals, the hierarchical interval graph nesting is not quite as obvious. There are two basic problems. The first problem is that we must enforce a consistent allocation between intervals of the same name that are live on entry (or exit) to both sides of the conditional. For example, in the program in Figure 9(a), `a` is live on entry to the conditional, and must be allocated to the same register on both sides of the conditional. Similarly, in the program in Figure 9(b), `b` is live on exit from the conditional, and must be allocated to the same register on both sides of the conditional.

The second problem is that there may be constraints that require a consistent allocation between live ranges with the same name that both enter and exit the conditional. Consider the program in Figure 9(c). In this case, the outer loop creates a cyclic interval which imposes the constraint that the the interval for `a` entering the conditional must be allocated to the same register as the intervals for `a` exiting the conditional.

We combine all of these problems in the example of a conditional nested inside a loop as illustrated in Figure 10. Note the constraints on variables live at the start of the conditional (in our example this is variable `a`), and those live at the end of the conditional(variables `a` and `b`). Each such variable must be allocated to the same register on both sides of the conditional. For example, if `b` is allocated to register P_i in the `then` part of the conditional, `b` must also be allocated to P_i in the `else` part. Any variable that is live only within the conditional may be allocated to different registers on each side of the conditional. For example, variable `e`, may be assigned to different registers on each side of the conditional. As illustrated in Figure 10,

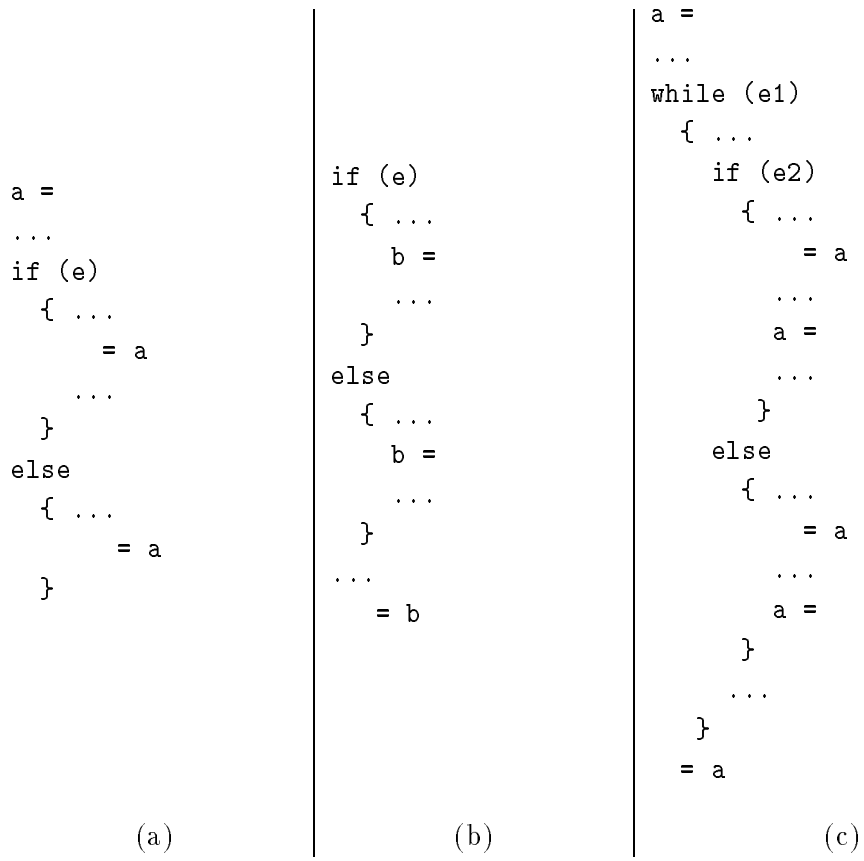


Figure 9: Examples of Consistent Allocation Required for Conditionals

we capture exactly these constraints (and no more constraints) by creating a cyclic interval graph that connects the input variables of the IF-THEN and ELSE parts together and the output variables of the IF-THEN and ELSE parts together. In our example, this means that we connect `a` on the inputs and `a` and `b` on the outputs. With these connections we have now created a cyclic interval graph. As the execution of the two branches of the conditional are mutually exclusive, we can think of the cyclic interval graph of the nested conditional to be composed of a time line that is "wrapped" from one branch of the conditional to the other. Now, we apply the spilling and coloring heuristics to the cyclic interval graph of the embedded conditional, and then the outer loop.

This encoding of the problem will address the problem of consistent allocation of intervals entering or exiting both sides of the conditional. In our example it will force the two intervals for `a` entering the conditionals to be allocated the same color, say P_i, and the two intervals allocated to `a` exiting the conditional to be allocated the same color, say P_j. However, by looking at only the conditional, and ignoring the presence of the outer loop, there is no reason that P_i and P_j must be the same color. This leads to the second problem with conditionals. Note that `a` is actually a cyclic interval in the next hierarchical level, LOOP 1. This means that

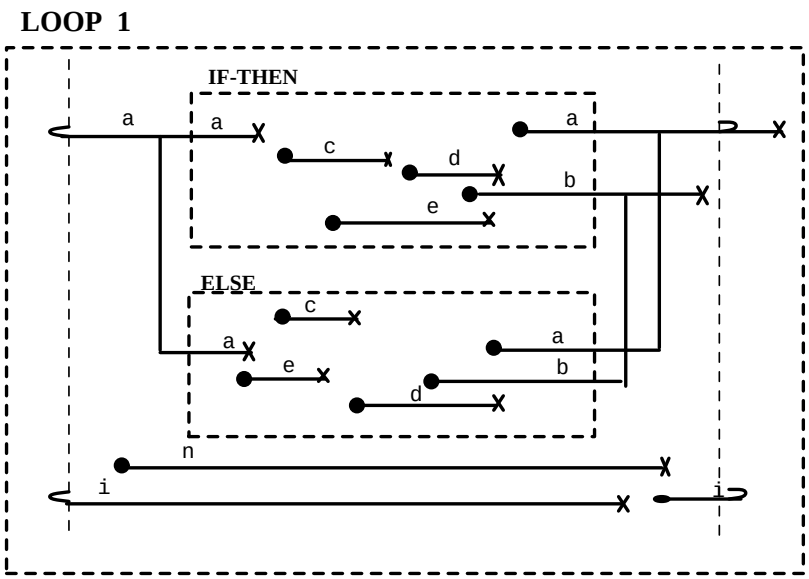


Figure 10: A Nested Conditional and its Hierarchical Structured Interval Graph

at the LOOP 1 level we should really allocate the *same* color to `a` on entry to the conditional as on exit to the conditional.

There are several methods for handling this second problem. One solution is to introduce chameleon intervals for cyclic intervals that have been allocated different registers in a nested structure. This, of course, may introduce extra register move instructions. Another approach is to force the two related ends of a cyclic interval to be pre-allocated to the same color. In our example, this means that we would pre-allocate both the entry and exit intervals for `a` to one pseudo register, say P1. A third method is to reallocate registers on the top down pass. This has the disadvantage of requiring two allocations, but has the advantage of propagating exactly the constraints that are imposed by allocations at a higher level. For example, it might be the case that cyclic interval `a` had already been allocated to a chameleon interval in the processing of LOOP 1, and there is no extra overhead introduced by assigning two different registers to the `a` in the conditional.

In summary, we can apply the cyclic interval graph coloring and spilling algorithms to structured programs by repeatedly applying the following two steps in a bottom-up phase:

Step 1: Solve the innermost nested construct (either a loop or a conditional). In the case of a loop, it is already a cyclic interval graph. In the case of a conditional, we create the proper cyclic interval graph by joining the input and output variables that are common to both the IF-THEN and ELSE parts of the conditional.

Step 2: Given the solution from Step 1, replace the nested structure with simple intervals as illustrated in Figure 8(b).

The bottom-up phase is then followed by a top-down phase that propagates real register numbers to the pseudo allocation done on the bottom-up phase. On the top-down phase, extra constraints due to cyclic intervals entering and exiting conditionals must be resolved by introducing chameleon intervals or recoloring the lower level with some extra initial coloring constraints.

As we shall briefly survey in Section 7, interesting work has been conducted on hierarchical methods for register allocation [15, 16, 17]. However, it is our belief that the clear and simple representation of the cyclic interval graph provides a good basis for representing live ranges that cross the boundaries of nested program structures. This of course helps in the hierarchical register allocation process.

6 Interval Graph Performance on Benchmark Programs

Standalone versions of our spilling and coloring algorithms were implemented [6]. In this section we compare the performance of our interval graph method of spilling and register allocation to the performance of three advanced production *C* compilers for the *IBM RS6000* (Version 1.01.003.0013), 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.

We focus on two inner loop bodies, one taken from Livermore Loop 8[7], the other from the Tomcat SPEC89 benchmark (Release 1.2).[8]. 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.

In order to give all compilers approximately equal input, we first transformed the input source code so that all of the complex transformations had been made explicit. Both loops were manually unrolled and software pipelined. Common subexpression elimination was performed and reused data values were explicitly assigned to local scalar variables. 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

[6] We are currently integrating the spiller and the register allocator, as well as an interference graph approach into our McCAT (McGill Compiler Architecture Testbed) research compiler.

[7] To be precise, the Tomcat loop is the "I-LOOP" beginning with "DO 250 I= I1P,I2M".

[8] 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.

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.

Originally, we implemented an interval graph representation on the *gcc* (Version 1.37.1). However, optimizations, like common subexpression elimination and alias analysis, were poor. Hence variable lifetimes were too short to receive meaningful benefit from our register allocation scheme. We decided that for meaningful results, we had to take an approach where we could make the complex transformations explicit and rely on observing the output of compilers for which we could not modify the source. This is, of course, only a preliminary step that we used to demonstrate that our ideas are worth pursuing and studying further by others in their compilers. When the McCAT compiler supports all of the complex transformations automatically, we will be able to make a more thorough study of different register allocation strategies.

We make several assumptions when generating the interval graph results. As in the rest of the paper we assume that 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. We also assume that instructions are executed in their source code order. If instruction scheduling and register allocation were done together, code can be reordered so as to reduce the live range of certain variables and thus reduce register pressure. 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.

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 only 16 registers are available, our method requires the introduction of spills for these benchmarks. However as can be seen in Tables 1 and 2, 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* to 73 loads *per iteration*! These are proportional to the dynamic loads and stores performed. The reduction in *store* spills is slightly smaller, ranging from 0 to 71 stores *per iteration*. 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.

Table 3 gives analogous results when 32 registers are available on the RS6000. The increased number of registers alleviates the need for many of the spills. The interval graph method allows Loop 8 and the rolled version of tomcat to execute with no load or store spills.

	Rolled			Unrolled − 3×		
	Int Gr	SPARC	MIPS	Int Gr	SPARC	MIPS
LOADS(total)	27	41	34	71	144	131
LOADS(spill)	9	23	16	53	126	113
STORES(total)	4	8	17	20	66	91
STORES(spill)	0	4	13	8	54	79

Table 1: Number of Double Precision Loads and Stores with 16 registers for Tomcat.

	Rolled			Unrolled − 6×		
	Int Gr	SPARC	MIPS	Int Gr	SPARC	MIPS
LOADS(total)	16	23	22	107	158	147
LOADS(spill)	1	8	7	59	110	99
STORES(total)	6	6	6	24	34	40
STORES(spill)	0	0	0	6	16	22

Table 2: Number of Double Precision Loads and Stores with 16 registers for Loop 8.

Most interesting however, is the performance with an unrolled version of tomcat. In this case the interval graph method still required no spills, while the RS6000 had 48 load spills and 22 store spills. Loop 8 also had 48 fewer load spills and 32 fewer store spills. Similarly when only 16 registers were available, the largest (absolute) reduction in spills came with the unrolled tomcat loop.

After registers are spilled, the interval graph must still be colored. As was discussed in Section 4, it is always possible to color an interval graph of maximum width W_{max} with the use of chameleon registers, i.e. moving values from one register to another. Using the *fat-*

	Tomcat				Loop 8			
	Rolled		Unrolled − 3×		Rolled		Unrolled − 6×	
	Int Gr	R6000	Int Gr	R6000	Int Gr	R6000	Int Gr	R6000
LOADS(total)	18	24	18	66	15	16	33	81
LOADS(spill)	0	6	0	48	0	1	8	56
STORES(total)	4	4	12	34	6	6	19	51
STORES(spill)	0	0	0	22	0	0	1	33

Table 3: Number of Double Precision Loads and Stores with 32 registers.

cover algorithm all but the unrolled version of tomcat were successfully colored without using chameleon intervals as can be seen in Table 4.

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 4: Interval Graph Statistics (after Spilling).

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.

This evidence indicates that further work on the cyclic interval graph approach is worth while, and that further tests will show the kinds of programs for which the cyclic interval graph method is superior.

7 Related Work

In this section, we present a survey of work related to register allocation, graph coloring and, interval graphs.

In a number of recent publications, researchers have been trying to improve Chaitin's method for register allocation. Briggs et. al. recognized the fact that Chaitin's original heuristic is not guaranteed to find the minimum coloring [6]. 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 [14]. 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 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. 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 [18]. 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 [19].

Interprocedural register allocation has been studied by a number of people [20, 10, 21]. 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 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 [22]. 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 [23, 24]. 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 [22]. 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 [15]. 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* [17]. 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 Chaitan's interference graph as their input.

Gupta et. al. reported their work in the area of global register allocation using *clique separators* [16]. 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.

As we pointed out before, our problems are related to the class of *circular-arc graph coloring* problems [11, 12]. 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 [7, 8]. 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 [25], and to perform channel routing in VLSI layouts [26, 27]. 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 [14]. A great deal of theoretical work has been done, a good summary of which may be found in [28, 29].

Using circular arc graphs for register allocation has recently been proposed in high-level dataflow synthesis for digital systems [30, 31, 32]. In this application domain, the computation is represented by *data flow graphs*. Data flow graphs with loops can be modeled by *cyclic dataflow graphs* [33, 34], and the corresponding register allocation problem can be modeled by circular arc graphs [34]. 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 [9]. 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.

Finally, let us reiterate our view of the relation between this work and the other related work in register allocation. We believe that our approach based on interval graphs appear well suited to certain structured programs, and in particular large inner-loops and loop nests. However, for less structured programs, perhaps the interference graph is the best representation. We expect that these two representations can complement each other in a compiler, and further experiments are clearly required to determine how to combine them in a most effective fashion. We are currently implementing a combined approach in our McCAT compiler.

8 Conclusions and Future Work

In this paper we have presented a new approach to register allocation that is based on a hierarchical cyclic interval graph representation. Our representation can effectively characterize the overlap between live ranges of variables at different times in a program execution. Furthermore, we have demonstrated how the additional information in such a representation is useful in coloring and spilling algorithms. We believe our method is particularly suitable to handle structured program segments such as structured loops and conditionals.

Based on our interval graph framework, we have presented two approaches to the minimal coloring problem based on the notion of a *fat-cover*. In addition, we have presented a new approach to the k-coloring problem. Our approach introduces the notion of chameleon intervals and, register floats which help avoid expensive register spills by introducing less expensive register moves. We presented a new sweep-and-split algorithm that is used to transform graphs that are not k-colorable into graphs that are *guaranteed* to be k-colorable. This transformation minimizes spills by using a powerful heuristic that is guided by information available in the interval graph representation, but is not available in the traditional interference graph representation.

In addition to illustrating how these algorithms work on inner loops, we have also shown how the cyclic interval graph representation can be extended to accurately capture the register allocation constraints in programs with nested loops and conditionals.

We have implemented our spilling algorithm and applied it to a collection of challenging loops. By comparing our results to those produced by the MIPS, SPARC, and R6000 production compilers, we have demonstrated that the cyclic interval graph representation, combined with our new algorithms, produces very encouraging results.

Based on our experimental coloring testbed and our prototype spilling program, our register allocation framework is being integrated into the low-level structured intermediate representation supported by the McCAT compiler [35]. In addition to this implementation effort, we are continuing to exploit the extra information available from the interval graph representation. For example, one potential use of such information is the combination of instruction scheduling and register allocation.

Acknowledgements

We would like to thank all of the referees for their insightful comments that helped to improve the presentation of our ideas. We would also like to acknowledge the suggestion from one referee who pointed out a natural improvement to the hybrid coloring scheme in which the interference phases and fat cover phases alternate.

References

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

[2] John L. Hennessy and Norman P. Jouppi. Computer technology and architecture: An evolving interaction. *Computer*, 24(9):18–29, September 1991.

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

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

[5] Frederick Chow and John Hennessy. Register allocation by priority-based coloring. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 222–232, Montréal, Québec, June 17–22, 1984. ACM SIGPLAN. *SIGPLAN Notices,* 19(6), June 1984.

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

[7] Alan Tucker. Coloring a family of circular arcs. *SIAM Journal on Applied Mathematics*, 29(3):493–502, November 1975.

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

[9] Q. Ning and G. R. Gao. A novel framework of register allocation for software pipelining. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 29–42, Charleston, South Carolina, January 10–13, 1993.

[10] Peter A. Steenkiste and John L. 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.

[11] V. Klee. What are the intersection graphs of arcs in a circle? *American Mathematics Monthly*, 76:810–813, 1969.

[12] 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.

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

[14] David Bernstein, Dina Q. Goldin, Martin C. Golumbic, Hugo Krawczyk, Yishay Mansour, Itai Nahshon, and Ron Y. Pinter. Spill code minimization techniques for optimizing compilers. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 258–263, Portland, Oregon, June 21–23, 1989. *SIGPLAN Notices,* 24(7), July 1989.

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

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

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

[18] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Register pipelining: An integrated approach to register allocation for scalar and subscripted variables. In U. Kastens and P. Pfahler, editors, *Proceedings of the 4th International Conference on Compiler Construction, CC '92*, number 641 in Lecture Notes in Computer Science, pages 192–206, Paderborn, Germany, October 5–7, 1992. Springer-Verlag.

[19] Christine Eisenbeis, William Jalby, and Alain Lichnewsky. Compiler techniques for optimizing memory and register usage on the CRAY2. *International Journal on High Speed Computing*, 2(2):193–222, 1990. Also published as INRIA Research Report 1302, October 1990.

[20] Fred C. Chow. Minimizing register usage penalty at procedure calls. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 85–94, Atlanta, Georgia, June 22–24, 1988. *SIGPLAN Notices,* 23(7), July 1988.

[21] George S. Taylor, Paul N. Hilfinger, James R. Larus, David A. Patterson, and Benjamin G. Zorn. Evaluation of the SPUR Lisp architecture. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 444–452, Tokyo, Japan, June 2–5, 1986. IEEE Computer Society and ACM SIGARCH. *Computer Architecture News,* 14(2), June 1986.

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

[23] D. J. Kuck, R. H. Kuhn, D. A. 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 26–28, 1981. ACM SIGACT and SIGPLAN.

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

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

[26] Michael Burstein. Channel routing. In T. Ohtsuki, editor, *Layout Design and Verification*, volume 4 of *Advances in CAD for VLSI*, pages 133–167. North-Holland Publishing Company, Amsterdam, 1986.

[27] Ido Dagan, Martin Charles Golumbic, and Ron Yair Pinter. Trapezoid graphs and their coloring. *Discrete Applied Mathematics*, 21(1):35–46, September 1988.

[28] Martin Charles Golumbic. Interval graphs and related topics. *Discrete Mathematics*, 55(2):113–121, 1985.

[29] Wei-Kuan Shih and Wen-Lian Hsu. An $O(n^{1.5})$ algorithm to color proper circular arcs. *Discrete Applied Mathematics*, 25(3):321–323, November 1989.

[30] 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.

[31] Fadi J. Kurdahi and Alice C. Parker. REAL: A program for REgister ALlocation. In *Proceedings of the 24th ACM/IEEE Design Automation Conference*, pages 210–215, Miami Beach, Florida, June 28–July 1, 1987.

[32] L. Stok and R. van den Born. Synthesis of concurrent hardware structures. In *Proceedings of the International Symposium on Circuits and Systems*, pages 2756–2760, June 1988.

[33] Pierre G. Paulin and John 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, Nevada, June 25–29, 1989.

[34] 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.

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