A Register Allocation Framework Based on Hierarchical Cyclic Interval Graphs

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

May 24, 1993

Abstract

In this paper, we study *cyclic interval graphs* as an alternative representation for register allocation. 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. 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.
- A new heuristic algorithm for minimum register allocation, the *fat cover* algorithm, has been developed, implemented, and studied.
- 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).
- We outline a framework for a compiler to use this representation to handle programs with hierarchical control structure such as nested conditionals and loops.

We demonstrate the effectiveness of our approach by experimenting with a collection of loops. We compare the results of applying our approach, to the results produced by three state-of-the-art optimizing compilers (SUN Sparc C compiler, the MIPS C compiler, and the IBM RS6000 compiler). These experiments indicate that our approach results in significantly better register allocation, and the introduction of considerably less spill code.

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 [HP90]. 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 [HJ91]. Therefore, the benefit of keeping variables in registers is increasing, and thus the impact of good register allocation strategies is also increasing. VLSI technology permits the integration of more and more general-purpose registers on a processor chip, providing both opportunities and more challenges for effective register allocation.

Register allocators in many modern compilers employ the classical graph coloring method originally proposed by Chaitin [CAC$^+$81, Cha82, CH84, BCKT89]. 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 two adjacent nodes are assigned different colors. Thus, a k-coloring

---

*This work supported in part by FCAR, NSERC, and the McGill Faculty of Graduate Studies and Research.

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 contains only the overlapping (intersection) information of any two live ranges in the set. However, it reveals neither the overlapping information for more than two live ranges, nor any notion of the relative times of such overlapping. As a result, some weaknesses have been observed. For example, it is known that Chaitin's heuristic may fail to find the minimum coloring even for some simple cases. Spilling is often quite expensive: both at compile time (due to the need to rebuild and recolor the interference graph after spill code has been introduced) as well as at run time due to the introduction of excessive spill code. The challenge becomes more serious when one considers how to effectively model the live range of a loop variable: its lifetime may cross the boundary of iterations, and be defined and used repetitively at regular intervals.

In this paper, we study *cyclic interval graphs* as an alternative representation for register allocation. We argue that an approach based on such interval graphs can overcome the shortcomings of the traditional interference graph approach. 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.

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. (Sections 2 and 3).
- A new heuristic algorithm for minimum register allocation, the *fat cover* algorithm, has been developed, implemented, and studied. (Section 4)
- 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). (Section 5)
- We outline a framework for a compiler to use this representation to handle programs with hierarchical control structure such as nested conditionals and loops. (Section 6)

The possibility of using interval graphs as a model for register allocation was noted in [Tuc75, Tuc84]. However, to our knowledge, previous research was theoretical in nature and mainly 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.

In section 7, we present experimental results to demonstrate the effectiveness of our approach by comparing our method to the register allocators used in state-of-the-art optimizing compilers (SUN Sparc C compiler, the MIPS C compiler, and the IBM RS6000 compiler). In most of the paper we concentrate on examples that are innermost loops. In section 6, we briefly discuss how our scheme naturally extends to a hierarchical method that can handle nested loops and conditionals. Finally, we present our conclusions in Section 8.

2 Register Allocation: Background and Challenges

In this Section, we present two simple motivating examples to illustrate the weaknesses of the classical interference graph approach. We then outline how these problems can be handled by an alternative representation, cyclic interval graphs.

2.1 Interference Graphs and Chaitin's Heuristics

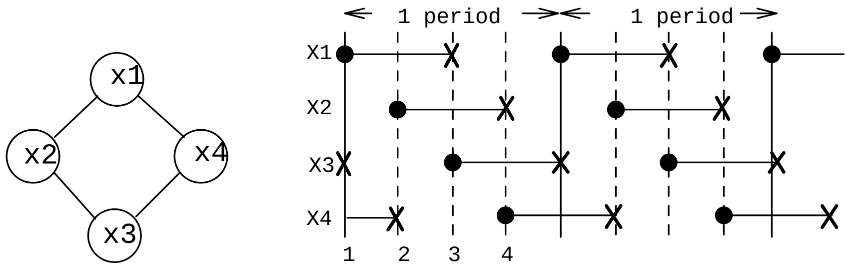
As outlined in Section 1, the traditional approach uses an interference graph representation where nodes represent live ranges of variables, and edges represent interference between two live ranges. To be more precise, according to [Cha82], two live ranges interfere, "if one of them is live at a definition 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 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.

2.2 A Motivating Example

In this subsection we provide a small example that motivates our approach.

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



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

Figure 1: An Interference Graph and an Interval Graph

The Optimality Issue

It has been pointed out by some researchers that Chaitin's heuristics are not guaranteed to find the optimal solution, i.e. the minimum coloring of an interference graph. Figure 1 (a) shows a simple section of code while Figure 1 (b) shows its corresponding interference graph.[1]

Even though this graph is clearly 2-colorable, Chaitin's heuristic will fail because there is no node with a degree < 2, and thus spilling has to be introduced.[2]

Live Ranges of Loop Variables

Another challenge is 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 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'.

---

[1] This interference graph is similar to the one presented by [BCKT89].

[2] Since 2-coloring is known to be linear, it is clear that a simple algorithm could handle this case.

2.3 Cyclic Interval Graphs — An Alternative Representation

Our observation is that an interference graph represents only partial information regarding the relationships between the live ranges of variables (pairwise interference). To address this problem, as well as to capture precise information for variables with loop-carried dependencies, we use an alternative representation: *interval graphs*. 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. Since the relation between the intervals is periodic, we can characterize this relation with one period — hence called *cyclic intervals* to be formally defined in the next section.

3 Formulating the Problem

In this section, we formulate the main problems to be studied in this paper and discuss their solutions. In Section 3.1 we formally introduce the concept of cyclic interval graphs. In Section 3.2 we give concrete statements of the two problems to be attacked: the problem of finding a minimum coloring of cyclic interval graphs, and the problem of finding k-coloring with minimal spilling. In Section 3.3 we outline important observations about cyclic interval graphs. These observations will be used in the subsequent sections.

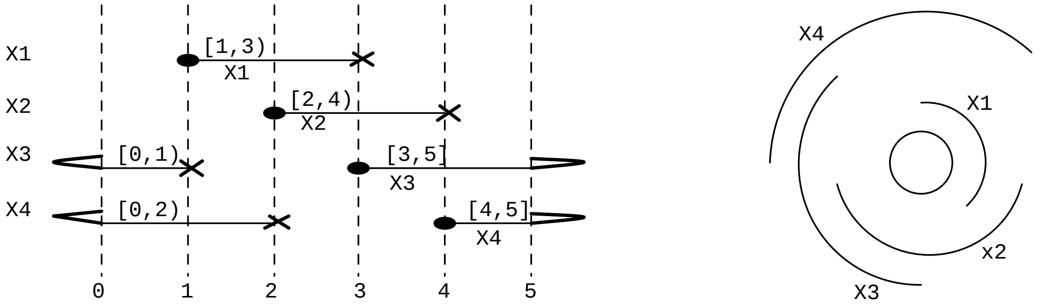
3.1 Lifetime Intervals and Cyclic Interval Graphs

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.

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. 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). We should 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 of greater than 1.

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 in the conditionals. In Section 6, we discuss this process further.



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

Figure 2: A Cyclic Interval Graph

3.2 The Problem Statement

We can now state the main problems to be studied in this paper. First, we introduce the following definitions.

Definition 3.1 *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 3.2 *Two intervals* $I1, I2$ *are* overlapping *if there exists a time t which is covered by both* $I1$ *and* $I2$*.*

Our 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 to the intervals in G such that two intervals should be assigned to two different registers if they are overlapping with each other.

- Problem 2 (Finding a k-coloring of a Cyclic Interval Graph with Minimum Spilling Cost). Given a set of live ranges represented as a cyclic interval graph G and a set of k registers. Find an assignment of the k registers to 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 [SH89].

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 5, this is particularly important for programs in which the register pressure is close to k.

3.3 Some Observations about Cyclic Interval Graphs

Our problems are related to the class of *circular-arc graph coloring* problems [Kle69, GJMP80]. A graph G is called a circular-arc graph if its vertices can be placed in a one-to-one correspondence with a set of circular arcs of a circle in such a way that two vertices of G are joined by an edge if and only if the corresponding two arcs 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 if a k-coloring problem for a circular arc graph with n arcs has a complexity of $O(nk!klogk)$ [GJMP80]. In this paper, we are interested in fast heuristic methods which finds 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 [GJMP80]. For our purpose of register allocation, it is most important to use the information provided in the interval graphs 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 define formally below.

Definition 3.3 *The* width *of a cyclic interval graph G at time t, written as width(G,t), is the number of intervals covering t.*

Definition 3.4 *The maximum* width *of a cyclic interval graph G, written as $W_{max}(G)$, 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 as $W_{min}(G)$, is the minimum width(G,t), for all t which is covered by some interval in G.*

Now, we state the following theorems without proofs. The validity of the theorems are intuitively clear, and were shown elsewhere [HGAM91].

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

Theorem 3.1 *If G is an interval graph containing no cyclic intervals, then G is optimally colorable with $W_{max}(G)$ colors.*

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

4 Finding a Minimal Coloring of Cyclic Interval Graphs

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 3.2), and our experimental observations which indicate that a large majority of graphs that could represent programs can be colored in W_{max} colors, we have developed an algorithm, called *the fat cover algorithm*, that is specifically designed to work well for graphs that can be colored in W_{max} colors.

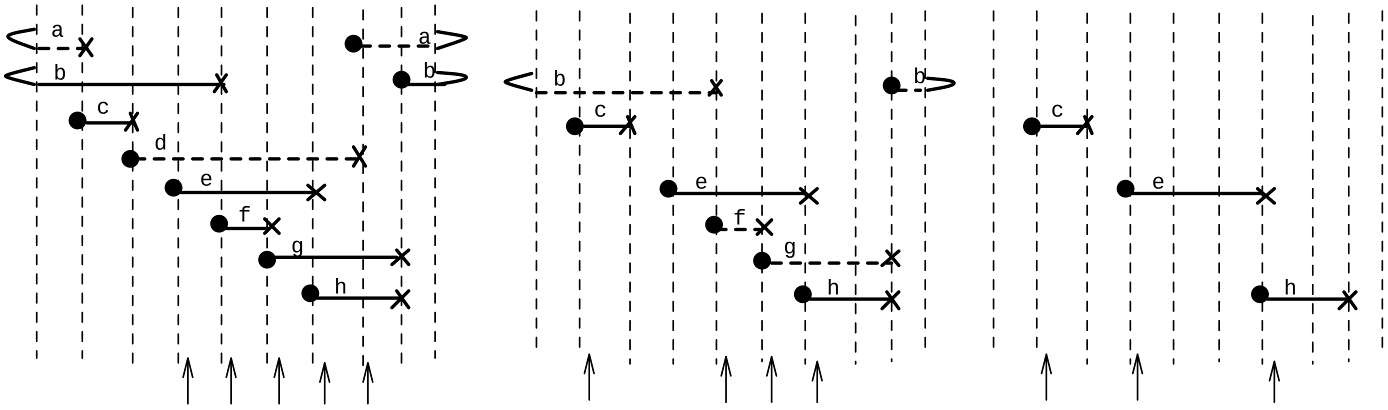
The key to this algorithm is the observation that the *fat spots* of the interval graph are the locations that are most important, and that we can iteratively reduce the maximum width of the uncolored portion of the graph by finding a non-overlapping set of intervals that covers all the fat spots and 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.

4.1 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 covering for `b` is {`b`, `f`, `g`}. We can then reduce the problem to a 1-coloring of the non-cyclic interval graph given in Figure 3(c), which is clearly 1-colorable.

4.2 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.



(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

Definition 4.1 *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 4.2 *The* 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 4.1 *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 4.2. 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 4.2. Secondly, it is clear that by removing F from G that we are left with a graph that is colored with $k - 1$ colors. ∎

Our *fat cover* algorithm is inspired by Theorems 3.1 and 4.1. 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). 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, 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 3.1) 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 does not succeed, then 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.

Our fat cover algorithm can be thought of as a smart way of deciding which subset of intervals should be colored with the same color. In some of the more traditional approaches using interference graphs, a simplification phase is applied to the interference graph in which pairs of nodes are coalesced into one node, thus forcing them to be colored the same color[CAC$^+$81]. In our case we are searching for sets of nodes that have a very specific property, that is they all belong to a fat cover of some cyclic interval. Finding such a set of intervals requires information regarding the location of all the fat spots in the interval graph. This information is explicit in our cyclic interval graph representation, and not available in the interference graph representation. In fact, our example in Figure 3 is not 3-colorable with the interference graph approach.

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.

4.3 A Hybrid Algorithm

A hybrid algorithm can combine the best points of the interference graph approach with the fat cover method. 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 the fat cover algorithm to color all 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 a hybrid 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 only overlap with 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 intervals remaining overlap with 3 other intervals. This means that in order to use only 3 colors, a traditional 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.

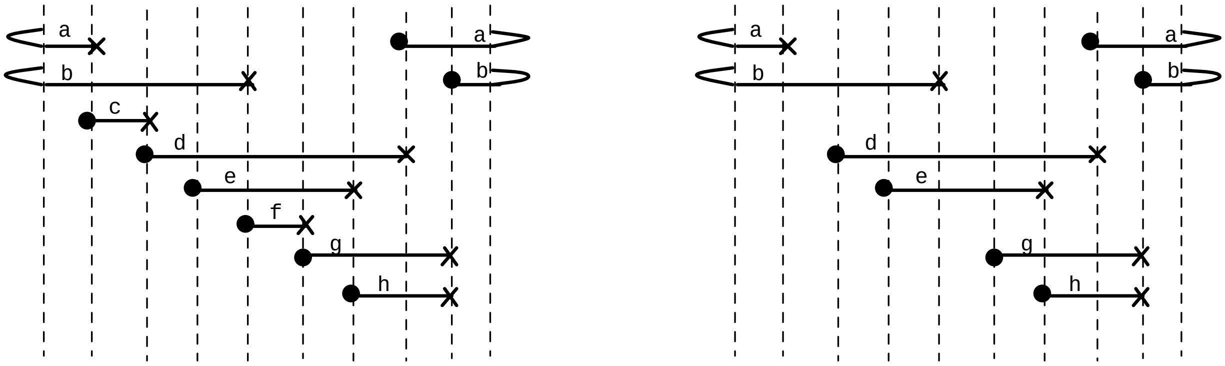
Figure 4(d): This figure shows the remaining uncolored intervals. Note that the width is now 2, and there are 3 fat spots as indicated by the arrows. We find that {`b`,`g`} forms a fat cover for `b`, the only remaining cyclic interval, and we color `b` and `g` with a new color *blue*.

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.

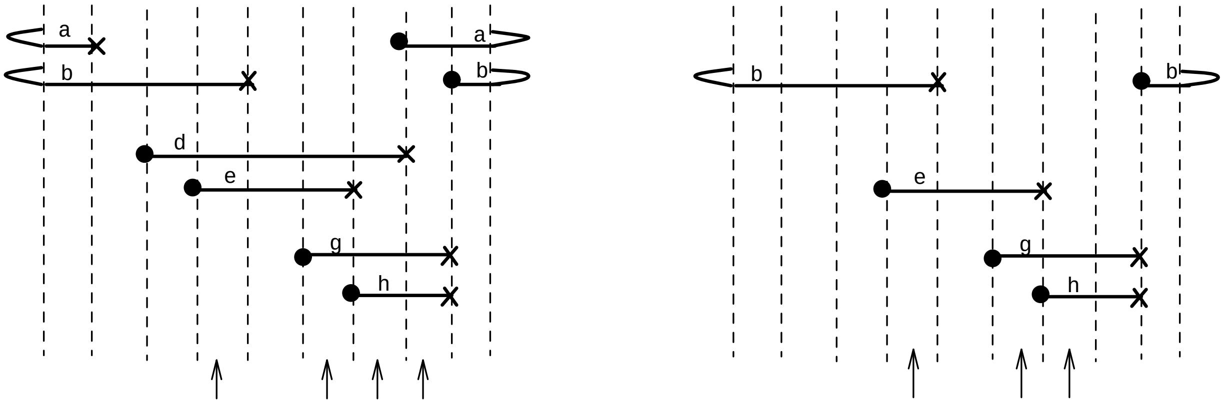
4.4 An Experimental Comparison

In order to experiment with a wide variety of coloring approaches and coloring heuristics, we implemented an experimental platform that supports the traditional interference graph algorithm, the fat cover algorithm, a wide-variety of greedy algorithms, and a hybrid fat cover algorithm. Using this experimental platform we generated random interval graphs that could correspond to real programs, and we compared the performance of the coloring algorithms. As expected, the fat cover and fat cover hybrid algorithms outperformed the traditional and greedy algorithms for the cases in which the optimal colouring was close to W_{max}. Further details of both the experiments and algorithms can be found in [HGAM91].



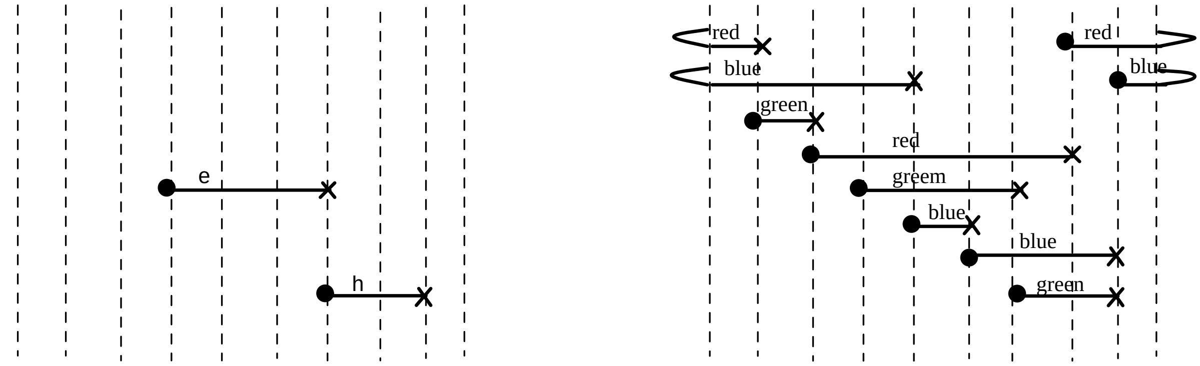
(a) the original interval graph

(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

5 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 the 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 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. ([CH90] suggests 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.

[BGG^{+}89] used the width [3] of the interference graph as one of the heuristics when deciding which node to spill so as to minimize the overall spill code generated. Similarly our algorithm uses the width of the interval graph as 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.

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.

5.1 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 5(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 5(b), where the interval for `c` has been spilled leaving two short intervals representing the definition of `c` followed by a store to memory (↓) and a load from memory (↑) followed by a use.[4]

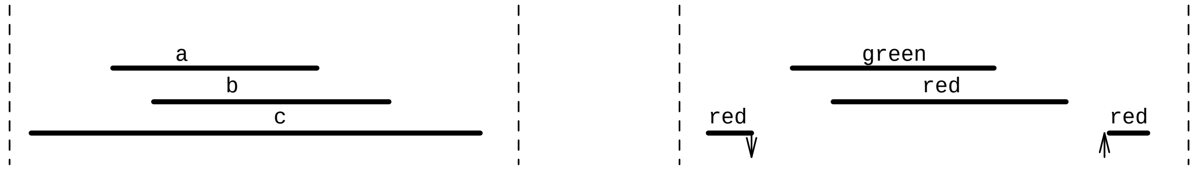
The second situation is more subtle. Consider the graph given in Figure 5(c). This graph has a maximum width of 2, but is not 2-colorable. In this situation we have not really run out of colors, and we need not resort to spilling in order to make this graph 2-colorable. Instead, we use the notion of a *chameleon interval*, an interval that can change color depending on its surroundings.

If we allow the interval for variable `a` to change color at the location indicated by the solid bar in Figure 5(d), then we can easily color this graph with only two colors. Thus, instead of introducing the loads and

---

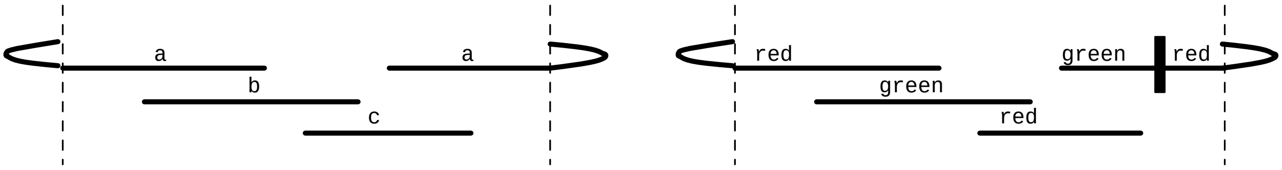
[3] The width of the interference graph at any given point in time is defined to be the number of live variables at that point.

[4] It was important to choose `c` to spill. Spilling either `a` or `b` would not result in reducing the $W_{max}(G)$ to 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 5: An Example of Register Spilling and Register Floating

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.[5]

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 5(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 only introduce chameleon intervals for the cyclic intervals that do not have a fat cover, the number of chameleon intervals introduced is small (at most $W_{min}(G)$).

5.2 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 5(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.[6] Like the fat cover coloring algorithm, the sweep and split algorithm takes advantage of the extra information available in the interval representation. Since this algorithm is straight-forward, we 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 to develop a good heuristic for selecting which intervals to split.

---

[5] 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.

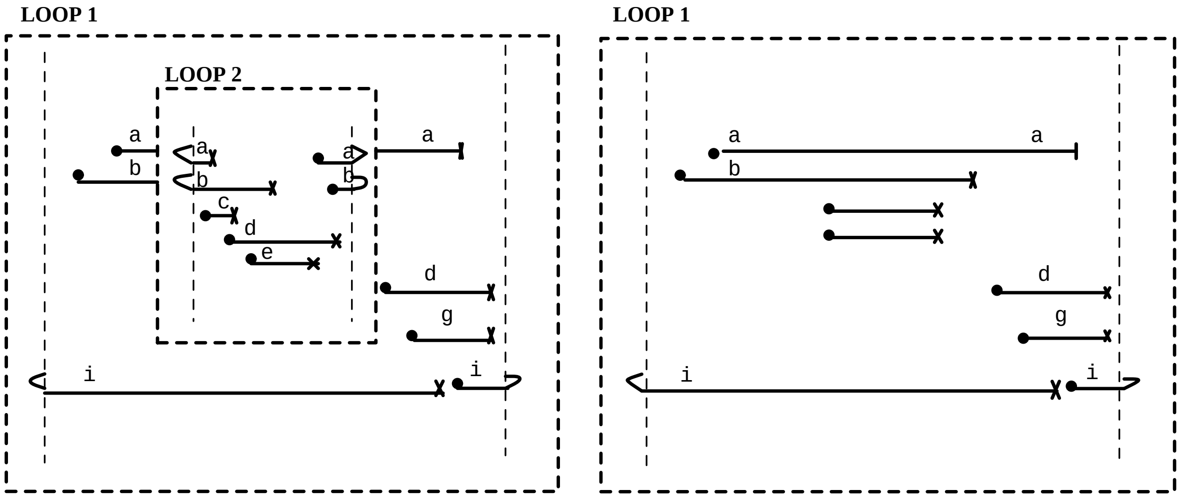
[6] A similar method was proposed for basic blocks in [CH84].

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.[7] The reasoning behind this 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.

6 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.

Let us first consider the case of nested loops as illustrated in Figure 6(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 6(a). As shown in Figure 6(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. We can now find a coloring for LOOP 1, which is now a non-hierarchical cyclic interval graph.



(a) a nested loop

(b) after allocating the inner loop

Figure 6: Nested Loops as Hierarchical Structured Interval Graphs

In the case of nested conditionals, the hierarchical interval graph nesting is not quite as obvious. Let us consider the example of a conditional nested inside a loop as illustrated in Figure 7. In the case of conditionals, the only constraints on register allocation are on the 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 `R1` in the `then` part of the conditional, `b` must also be allocated to `R1` 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 7, we capture exactly these constraints (and no more constraints) by creating a cyclic interval

---

[7] 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'.

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 and we may apply the same mechanism as with nested loops. In this case we first solve the conditional, and then we solve the outer loop.

-bb-error =

Figure 7: A Nested Conditional and its Hierarchical Structured Interval Graph

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

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 6(b).

7 Interval Graph Performance on Benchmark Programs

A spilling algorithm based on the *cyclic interval* representation was implemented. 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[8], the other from the Tomcat SPEC benchmark (Release 1.2).[9]. 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.

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. Instruction scheduling can 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.

---

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

[9] 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.

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*! 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.

	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.

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.

	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.

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. (Both loops were unrolled manually with reused data values explicitly assigned to local scalar variables. Thus no sophisticated analysis of array indices was required by the compilers to match the performance of the interval graph method.) 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 5, 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 on register to another. Using the *fat-cover* algorithm all but the unrolled version of tomcat were successfully colored without using chameleon intervals as can be seen in Table 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).

8 Conclusions

In this paper we have presented a new approach to register allocation that is based on a hierarchical cyclic interval graph representation. We have presented a thorough motivation for choosing the cyclic interval graph representation over the traditional interference graph approach. Furthermore, we have demonstrated how the additional information in such a representation is useful in coloring and spilling algorithms.

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

We have presented two approaches to the minimal coloring problem based on the notion of *fat-cover*. In addition, we have presented a new approach to the k-coloring problem. Our approach introduces the notion of chameleon intervals, register floats, and the concept that some expensive register spills can be avoided through the introduction of less expensive register floats. 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 not available in the traditional interference graph representation.

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 a much superior result.

Finally, we have shown how the cyclic interval graph representation can be extended to accurately capture the register allocation constraints in programs with nested loops and conditionals.

References

[BCKT89] Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Coloring heuristics for register allocation. *Proceedings of the 1989 SIGPLAN Conference on Programming Language*

Design and Implementation, 24(7):275–284, July 1989.

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

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

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

[CH84] F. Chow and J. Hennessy. Register Allocation by priority based coloring. In *Proceedings of the ACM SIGPLAN'84 Symposium on Compiler Construction.* SIGPLAN Notices 19(6), June 1984.

[CH90] F. Chow and J. Hennessy. The Priority-Based Coloring Approach to Register Allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, October 1990.

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

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

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

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

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

[HGAM91] Laurie J. Hendren, Guang R. Gao, Erik Altman, and Chandrika Mukerji. Regsiter allocation using cyclic interval graphs. Technical Report ACAPS-MEMO 33, McGill University, 1991.

[HJ91] J. Hennessy and N. Jouppi. Computer Technology and Architecture: An Evolving Interaction. *IEEE Computer*, 24(9):18–29, September 1991.

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

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

[SH89] P.A. Steenkiste and J. Hennessy. A Simple Interprocedural Register Allocation Algorithm and Its Effectiveness for LISP. *ACM Transactions on Programming Languages and Systems*, 11(1):1–32, January 1989.

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

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